Procedural Planet Generation

Champlain Valley Union High School

Milo Cress

Advisor: Anna Couperthwait

February 20, 2019

Abstract

In this article, we present a system for the online rendering of realistic terrain at varying spatial and temporal scales using concurrent functional programming, and leveraging modern multicore computers.

1 Maps

Foundational to the concept of terrain generation is its representation in 3-dimensional space. In cartography, maps provide information about an area of land by translating a given 2-dimensional point to a value. Heightmaps translate latitudes and longitudes to locations in 3-dimensional space by supplying a z coordinate to a given (x, y) coordinate pair. In climatology, "wetmaps" track rainfall over given areas, and predict, given an (x, y) coordinate pair, the inches or centimeters of rain that land on or near than point in a given amount of time. In more complex simulations, vector fields which simulate wind and weather are used by meteorologists to predict the spread of weather events across an area. By encapsulating a tract of land and the forces that act on it as a spatial function, the large-scale calculation of terrain becomes trivial.

1.1 Functional programming

When speaking about maps, it's easy to imagine them as they are presented in most cartographical texts – a layering of lines and colors on top of a page. However, attempting to use this model for procedural generation presents two immediate challenges:

1. We don't know how these maps were created. All we're given is a data dump and assurance of its accuracy.

2. We are working in a limited resolution, and therefore must trade scale for level of detail. Since this representation of a map forces the value of every point to be know all at once, one can either know the values of a small range of points with a great degree of detail, or a vast range of points with a limited degree of detail.

In short, traditional lookup-style maps are limited by the fact that in order to know the value of a map at a single point or collection of points, one must have pre-determined the value of that map at every conceivable point that could be requested. A functional approach to the map problem is to delay the calculation of the value of a given map until that value is absolutely necessary. While this is process requires more computing power, it vastly reduces the memory required to complete the operation, and it allows a map to encapsulate the processes that create it, rather than merely storing the data it creates.

Adopting a functional style allows the application several tools to the development of complex maps.

1.1.1 Monoids

Monoids consist of a binary operation and an identity element. For example, one can say that addition is a monoid whose binary operation is the function (+) and whose identity element is the number 0. Similarly, lists are a monoid whose binary operation is concatentation (++) and whose identity element is the empty list ([]). Describing maps as an instance of monoids enables the combination and manipulation of maps in a more natural, mathematical way.

1.1.2 Functors

Since maps are a parameterized type, they are essentially capable of encapsulating any imaginable value. This allows a map to be described as a function from a point to a value of any type. Functors describe a set of operations for manipulating map values before a map is fully evaluated, or in other words operating on a value encapsulated by a map. This enables complex transformations to be elegantly specified using category theory.

1.1.3 Monads

Since monads are merely monoids in the category of endofunctors, envisioning maps as instances of monads unlocks the potential for maps to be described more expressively, and possibly as a composition of other monadic values using Kleisli arrows (>>=).

1.1.4 Typeclasses

All of the constructs listed above are examples of `typeclasses`, which specify an interface for a type to implement. When we say that a map is an instance of the typeclass `monoid`, we specify its behavior in certain conditions, and allow the typeclass itself to generalize our definition, and specialize polymorphic functions to our specific type.

2 Engine

Many of the performance and accuracy advantages of procedural geometry generation are negated by "baking" this geometry into the polygonal meshes and rectangular textures required by traditional game engines. During the development of the procedural planet generation software, the author realized that a functional-style engine was necessary to render functional-style maps.

2.1 Raytracing

Many modern engines employ ray-tracing algorithms to simulate the paths of light in a virtual medium.

2.1.1 Distance estimators

In order to detect collision between a ray of light and a surface, distance estimators are employed. These algorithms estimate the minimum distance from a given point in 3D space to an object. Distance estimators can be combined and composed to create complex scenes from simple foundational objects.

2.1.2 Hacking procedural generation

Using gradient descent optimization, the minimum distance from a point to a map can be calculated, allowing a 2D map to be placed and rendered in 3D space by our raymarching engine.

2.2 Lighting

Light rays interact with surfaces by reflecting off of them. These reflections can be simulated by tracing rays from a viewer to an object, checking for a collision as described above, and coloring each pixel based on the amount of light that pixel reflects. Estimating the reflection of a specific point on an object depends on its *normal* vector, a unit vector pointing directly away from the object itself:

$$l = r \cdot (p' - p)$$

Where l is the lighting of a point p, r is the view vector, and p' is the location of the light.

2.3 Shadows

Realistic scenes require more complex interactions between light and objects, as the above shading system doesn't account for the objects themselves obstructing the path of light rays.

2.3.1 Hard shadows

Hard shadows can be calculated by raymarching a point on an object's surface in the direction of its normal, and checking if that ray intersects with an object in the scene.

2.3.2 Soft shadows

Soft shadows are more complex, as they require area lights, and area lights require a more complex *path tracing* algorithm to work effectively. They can, however, be

estimated, by checking the number of steps required to raymarch to the light source, and shading an area as less lit for each step.

2.4 Reflections

Reflections occur when a ray bounces off a surface, and scatters light in a specific direction.

2.4.1 Specular

Specular reflections give a metallic sheen to an object, and can be combined with diffuse light to create a glossy tint. They are created by raymarching a point in the direction of the incidence vector of a light collision reflected over the point's normal vector using the equation:

$$r = d - 2(d \cdot n)n$$

2.4.2 Diffuse

Diffuse reflections are more complex, but can be obtained by calculating the mean of a distribution of randomly scattered specular reflection rays from a given point. This process is prone to noise, so a powerful denoising algorithm is necessary.

2.4.3 Caustics

Caustics are the reflection of bright reflections, commonly caused by water or mirrors bouncing off of a diffuse surface. Though these are difficult to simulate without path-tracing, The author is exploring less computationally costly alternatives.

3 Optimization

The processes described trade predefined geometry for dynamically calculated geometry, giving the programmer greater power in dynamically modifying the level of detail

of a sector at runtime. However, these runtime calculations greatly increase the complexity of the algorithm, which decreases the number of frames that can be rendered in a given unit of time.

Optimizations that reduce the amount of time needed to render a frame can be developed through several methods:

1. Eliminating redundant calculations
2. Simplifying calculations
3. Parallelizing calculations

3.1 Functional programming

Fortunately, the functional model allows compilers to make these kind of optimizations automatically. In this section we'll discuss how these optimizations can be triggered and how they work together to reduce overall computational complexity.

3.1.1 Deforestation (fusion)

Recursive processes that build and destroy intermediate data structures, such as *hylomorphisms* (which are compositions of a anamorphisms and catamorphisms), can often be reduced to a single loop, and controlled by simple rules. For example, the expression:

```
let result = map (+ 4) $ map (/ 2) $ [1..10]
```

which would require two iterations over the input list if interpreted literally, can be reduced using the rule:

```
map f .  map g = map (f .  g)
```

to:

```
let result = map (\x -> x / 2 + 4) [1..10].
```

This kind of reduction is said to *fuse* the composition of two resource intensive loops, eliminating nodes in the evaluation tree (this is also referred to as deforestation). While these optimizations occur automatically for instances of the `foldable` typeclass, it is

often necessary to specify custom rules when working with custom datatypes, such as Maps.

The Haskell Community was clear in its recommendation of the Glasgow Haskell Compiler (GHC) for optimization of functional code [2] over other Haskell compilers.

3.1.2 Graph reduction

The Haskell compiler can also intelligently apply other rules to the optimization of compiled code. One key optimization comes from its ability to recognize and prune unnecessary complexity from an evaluation graph. For example, the complex mathematical expression $(x^2)^{3/2}$ could be simplified at compile-time to x^3.

The lazy evaluation paradigm of Haskell is integral in detecting whether a block of code will be superfluous because it was multiplied by zero, or subtracted from itself. The mathematical guarantee of correctness in these optimizations, coupled with the forward referential transparency ensured by effect-free pure functions enable powerful, high-level optimizations both at runtime and compile time.

3.1.3 Parallelism

Even code that cannot be optimized can often be easily parallized by using Haskell's robust `parallel` library for concurrent and sequential evaluation.

3.2 Automatic differentiation

The engine code base, as well as the geographical simulations, make heavy use of normal and gradient vectors, which require the algorithm to determine partial derivatives of maps either numerically or analytically.

Automatic differentiation enables the automatic derivation of partials, allowing the equation:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

To be satisfied for any function f whose arguments are dual numbers.

The relative simplicity of the above code is evident when compared to the numerical approach, which employs an epsilon value [4]:

$$\vec{n} = \begin{bmatrix} f(x+\varepsilon, y, z) - f(x-\varepsilon, y, z) \\ f(x, y+\varepsilon, z) - f(x, y-\varepsilon, z) \\ f(x, y, z+\varepsilon) - f(x, y, z-\varepsilon) \end{bmatrix}$$

3.2.1 Reworking code to parameterize types

Type flexibility enables the polymorphism that allows automatic differentiation to work hand-in-hand with extensible and optimizable typeclasses, such as `Accelerate`'s `Num` and `Fractional` classes.

3.3 GPU - parallelism

In order to implement powerful parallelism, specialized hardware is necessary. IBM/MIT fellow John Cohn has had several conversations with the author about the promise of graphics processors as general purpose computers.

"GPUs are increasingly capable of massively parallel computation" [1]

3.3.1 Haskell Parallel DSL

The `Accelerate` library for Haskell enables polymorphic functions to be compiled to a GPU-optimized Domain-Specific Language (DSL) during runtime.

3.3.2 Cuda library

Mark Engelhardt, a systems engineer and geospatial programmer, stressed the importance of low-level optimization, even in high-level languages such as Haskell. [3]

The `Cuda` library allows Haskell code to call low-level, handcrafted cuda code to achieve high-performance output.

4 TODO To do

The project, as its original goals were expressed, is nearing completion. However, some key features remain to be implemented.

4.1 Simulation

4.1.1 Perlin Noise

The base of the simulated terrain will be layered octaves of Perlin noise, which is guaranteed to have smooth first and second partial derivatives, greatly simplifying the calculation of gradients and normals.

4.1.2 Continental Drift

The simulation of continental drift requires force-mapping over time. Developing vector fields over heightmaps requires solving a differential equation for each point. This can be accomplished numerically, or analytically, with the aid of automatic differentiation.

4.1.3 Erosion

Simulated rainfall maps can be used to calculate the erosion coefficient for a given area. Simulated forces are then applied to each point in the heightmap to distort it, simulating the percussive force of water on stone and soil.

4.2 Material system

4.2.1 Node-system (blender)

4.2.2 Generative adversarial networks for textures

4.3 Typeclasses

4.3.1 UV mapping typeclass

4.3.2 Random sampling typeclass for soft shadows and diffuse shading

5 Works Cited

References

[1] John Cohn. *Massively Parallel Computations*. Jan. 2019.

[2] Haskell Community. *Triggering Stream-Fusion in Haskell*. Jan. 2019.

[3] Mark Engelhardt. *Optimizing Low-Level Code*. Sept. 2018.

[4] Jamie Wong. *Raymarching Distance Fields*. July 2015.