Parallelized Procedural Terrain Generation

Divjot Bedi, dsb2177

Overview

This project plans to deal with procedural terrain generation. In the realm of computer graphics and algorithmic design, the creation of realistic and varied digital landscapes is captivating and challenging. Taking on this problem, I aim to develop a system capable of generating large-scale, complex terrains through procedural algorithms, leveraging the power of parallel computing. At its core, the problem involves generating random numbers, taking their dot products, smoothing them out, and then combining this with some sort of heatmap rendering function (thresholding, further smoothing via kernel convolution, etc). What really constitutes a better terrain is sort of "subjective" but different methods yield unique results. At a glance, my project stands at the intersection of art and technology--combining aesthetics with advanced computational techniques.

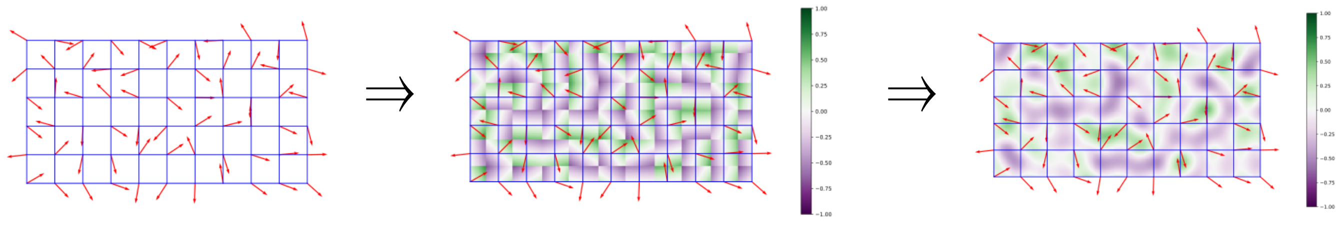
Approach

The core of the project revolves around the implementation of procedural generation algorithms. These algorithms, based on mathematical formulas and noise functions like Perlin noise and Voronoi noise, will be used to craft various terrain features. The procedural nature of these algorithms makes them inherently suitable for parallelization, as different sections of the terrain can be generated independently yet cohesively.

Perlin Noise

Perlin noise, a method often used in computer graphics, can be implemented in various dimensions, typically two, three, or four. The process generally involves three key steps:

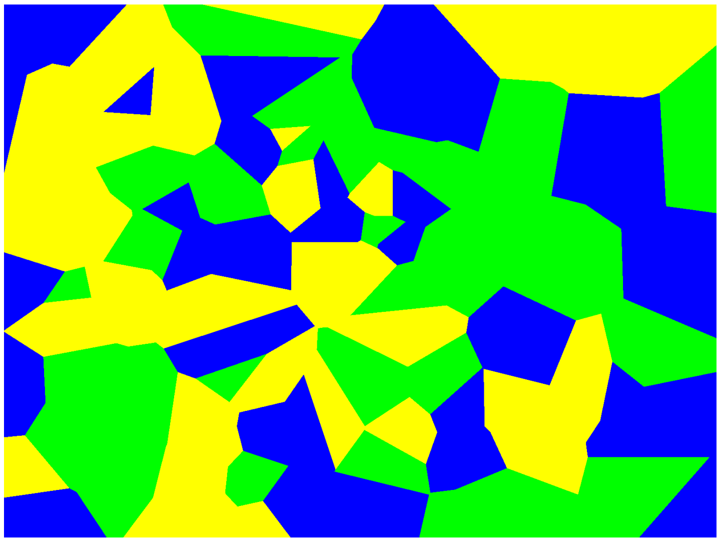
1. Defining a Grid of Random Gradient Vectors: This step involves setting up a grid where each point on the grid is assigned a random gradient vector.
2. Computing the Dot Product: For each point where noise needs to be calculated, the dot product is computed between the gradient vectors and their respective offsets (differences between the grid points and the point of interest).
3. Interpolation: Finally, the values obtained from the dot products are interpolated to smooth out the noise, creating a more natural and continuous variation in the noise pattern.

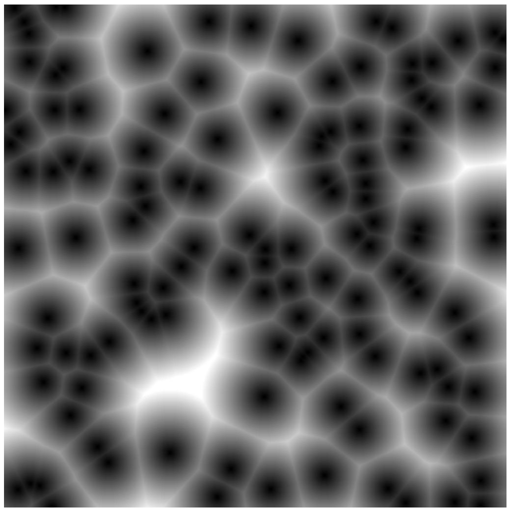


Voronoi Biomes

Another method commonly used in generating landscapes is the Voronoi algorithm/Worley Noise. Voronoi noise, a form of procedural noise, is generated by first dividing space into cells and assigning each cell a random point. The pattern is based on the proximity to the closest point. This process involves flooring input values to define cells, generating random positions within each cell, and calculating distances from these points. The shader-based implementation involves two key steps: finding the closest cell point and determining the nearest edge distance. This technique is used to create varied visual effects, including colorful patterns and cell borders. For more in-depth information and code examples, please refer to Ronja's tutorial on Voronoi Noise here.

Voronoi noise can be used to generate biomes for terrain, and allows one to randomly split the terrain into triangles. These triangles can then be enumerated and/or colored to indicate the chosen biome.


Example voronoi_biomes.png output from my code


Wikipedia

Implementation

Perlin Noise Generation

In the sequential approach for Perlin noise generation, the Haskell code defines a function that iterates over a two-dimensional grid. Each grid point computes its noise value using the perlin function, which generates smooth, coherent noise patterns. This process is inherently linear, with each point's noise value calculated independently and sequentially.

Parallelizing Perlin noise generation in Haskell involves leveraging concurrency primitives like *parMap* and *rdeepseq*. The parMap function allows simultaneous computation across different segments of the grid, significantly accelerating the process. The use of *rdeepseq* ensures complete evaluation of noise values in parallel threads, avoiding lazy evaluation pitfalls common in Haskell.

Code snippet

```
module Terrain where

import Data.List (find)
import Numeric.Noise.Perlin
import Codec.Picture
import Control.Parallel.Strategies (parMap, rdeepseq)

-- Heatmap rendering function
heatmap :: [(Double, PixelRGB8)] -> Double -> PixelRGB8
heatmap thresholds value = case find match thresholds of
 Just (_, colour) -> colour
 Nothing          -> PixelRGB8 0 0 0
 where
   match (threshold, _) = value > threshold

-- Generate Perlin noise
-- generateNoise :: Int -> Int -> Perlin -> [[Double]]
-- generateNoise width height noise = parMap rdeepseq (map noiseFn) coords
--   where
--     coords = [[(x, y) | x <- [0..width-1]] | y <- [0..height-1]]
--     noiseFn (x, y) = noiseValue noise (fromIntegral x, fromIntegral y, 0)
-- Generate Perlin noise

-- Splitting into chunks to parallelize even better?
generateNoise :: Int -> Int -> Perlin -> [[Double]]
generateNoise width height noise = parMap rdeepseq processChunk chunks
 where
   coords = [[(x, y) | x <- [0..width-1]] | y <- [0..height-1]]
   chunks = chunkList numChunks coords
   numChunks = 4 -- or any number depending on your parallelization needs
   processChunk = concatMap (map noiseFn)
   noiseFn (x, y) = noiseValue noise (fromIntegral x, fromIntegral y, 0)

-- Function to split a list into n chunks
chunkList :: Int -> [a] -> [[a]]
chunkList n xs = go n (length xs) xs
 where
   go _ _ [] = []
   go k l ys = let size = (l + k - 1) `div` k
               in take size ys : go (k-1) (l-size) (drop size ys)


-- Parallel version with convoluted smoothing
```

```haskell
generateSmoothNoise :: Int -> Int -> Int -> Perlin -> [[Double]]
generateSmoothNoise width height iterations noise = iterateSmooth iterations initialNoise
 where
   initialNoise = generateNoiseSeq width height noise
   iterateSmooth 0 noiseData = noiseData
   iterateSmooth n noiseData = iterateSmooth (n - 1) (parMap rdeepseq (map (smoothNoise
width height)) coords)
     where
       coords = [[(x, y) | x <- [0..width-1]] | y <- [0..height-1]]
       smoothNoise w h (x, y) = averageSurroundingNoise x y noiseData width height

averageSurroundingNoise :: Int -> Int -> [[Double]] -> Int -> Int -> Double
averageSurroundingNoise x y noiseData width height = let
   points = [(dx, dy) | dx <- [-1..1], dy <- [-1..1], inBounds (x + dx) (y + dy) width
height]
   total = sum [noiseData !! (y + dy) !! (x + dx) | (dx, dy) <- points]
   avg = total / fromIntegral (length points)
 in avg

inBounds :: Int -> Int -> Int -> Int -> Bool
inBounds x y width height = x >= 0 && y >= 0 && x < width && y < height

generateSmoothNoiseSeq :: Int -> Int -> Int -> Perlin -> [[Double]]
generateSmoothNoiseSeq width height iterations noise = iterateSmooth iterations initialNoise
 where
   initialNoise = generateNoiseSeq width height noise
   iterateSmooth 0 noiseData = noiseData
   iterateSmooth n noiseData = iterateSmooth (n - 1) (map (map (smoothNoise width height))
coords)
     where
       coords = [[(x, y) | x <- [0..width-1]] | y <- [0..height-1]]
       smoothNoise w h (x, y) = averageSurroundingNoise x y noiseData width height


generateNoiseSeq :: Int -> Int -> Perlin -> [[Double]]
generateNoiseSeq width height noise = map (map noiseFn) coords
 where
   coords = [[(x, y) | x <- [0..width-1]] | y <- [0..height-1]]
   noiseFn (x, y) = noiseValue noise (fromIntegral x, fromIntegral y, 0)

-- Render heatmap from noise
renderHeatMap :: Int -> Int -> [[Double]] -> Image PixelRGB8
renderHeatMap width height noiseData = generateImage pixelRenderer width height
 where
   pixelRenderer x y = heatmap thresholds (noiseData !! y !! x)
   thresholds = [snow, mountains, forest, land, sand, shallowWater, depths]
   snow         = (0.85, PixelRGB8 255 255 255)
   mountains    = ( 0.5, PixelRGB8 200 200 200)
   forest       = ( 0.1, PixelRGB8 116 151  62)
   land         = ( 0, PixelRGB8 139 181  74)
   sand         = ( -0.1, PixelRGB8 227 221 188)
   shallowWater = ( -2, PixelRGB8 156 213 226)
```

```
  depths       = ( -25, PixelRGB8  74 138 125)
  -- Add other thresholds here
```

Voronoi Biome Generation

For Voronoi biome generation, the sequential version calculates the closest seed point for each pixel to assign a biome, a process that can be computationally intensive for large images. The parallel version uses similar concurrency techniques to distribute the computation across multiple cores.

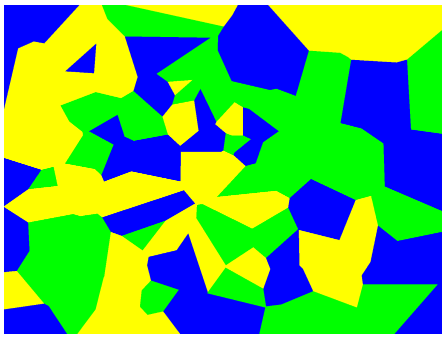
The choice of *parMap* and *rdeepseq* in Haskell's parallel strategies is crucial for achieving efficiency in computation. parMap enables dividing the problem into sub-problems that can be solved in parallel, while *rdeepseq* ensures that these computations are fully evaluated in parallel, leading to a more efficient use of system resources and faster execution times. This approach significantly enhances performance, especially for high-resolution terrain and biome generation tasks.

Code snippet

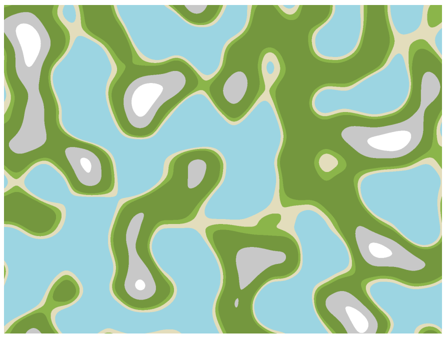
```haskell
module Biome where

import Codec.Picture
import System.Random
import Data.List (minimumBy)
import Data.Ord (comparing)
import Control.Parallel.Strategies (parMap, rdeepseq, rpar, rseq, using, parList)

-- Define a biome and a seed data type
data Biome = Forest | Desert | Ocean deriving (Show, Eq, Enum, Bounded)
data Seed = Seed { seedX :: Int, seedY :: Int, seedBiome :: Biome } deriving (Show, Eq)

-- Generate a random list of seeds
generateSeeds :: Int -> Int -> Int -> IO [Seed]
generateSeeds numSeeds width height = do
 gen <- newStdGen
 let biomes = [minBound .. maxBound] :: [Biome]
     biomeGen = randomRs (0, length biomes - 1) gen
     (xGen, yGen) = split gen
     xs = take numSeeds $ randomRs (0, width - 1) xGen
     ys = take numSeeds $ randomRs (0, height - 1) yGen
     biomeIndexes = take numSeeds biomeGen
 return [Seed x y (biomes !! index) | (x, y, index) <- zip3 xs ys biomeIndexes]

-- Euclidean distance function
distance :: Seed -> Int -> Int -> Double
distance seed x y = sqrt $ fromIntegral ((x - seedX seed) ^ 2 + (y - seedY seed) ^ 2)

-- Function to render the Voronoi diagram
```

```haskell
renderVoronoiSeq :: Int -> Int -> [Seed] -> Image PixelRGB8
renderVoronoiSeq width height seeds =
 generateImage pixelRenderer width height
 where
   pixelRenderer x y = biomeToColor $ seedBiome $ closestSeed x y
   closestSeed x y = minimumBy (comparing (\seed -> distance seed x y)) seeds
   biomeToColor biome = case biome of
     Forest -> PixelRGB8 0 255 0     -- Green
     Desert -> PixelRGB8 255 255 0   -- Yellow
     Ocean -> PixelRGB8 0 0 255      -- Blue


renderVoronoiPar :: Int -> Int -> [Seed] -> Image PixelRGB8
renderVoronoiPar width height seeds =
 generateImageParallel pixelRenderer width height
 where
   pixelRenderer x y = biomeToColor $ seedBiome $ closestSeed x y
   closestSeed x y = minimumBy (comparing (\seed -> distance seed x y)) seeds
   biomeToColor biome = case biome of
     Forest -> PixelRGB8 0 255 0     -- Green
     Desert -> PixelRGB8 255 255 0   -- Yellow
     Ocean -> PixelRGB8 0 0 255      -- Blue

generateImageParallel :: (Int -> Int -> PixelRGB8) -> Int -> Int -> Image PixelRGB8
generateImageParallel pixelFunc width height =
 let rows = [ [ pixelFunc x y | x <- [0 .. width - 1] ] | y <- [0 .. height - 1] ]
     parallelRows = rows `using` parList rseq
 in generateImage (\x y -> (parallelRows !! y) !! x) width height


-- generateImageParallel :: Int -> Int -> (Int -> Int -> PixelRGB8) -> Image PixelRGB8
-- generateImageParallel width height pixelFunc =
--   let rows = [ [ pixelFunc x y | x <- [0 .. width - 1] ] | y <- [0 .. height - 1] ]
--       parallelRows = parMap rdeepseq id rows
--   in generateImage (\x y -> (parallelRows !! y) !! x) width height
```

Example PNG outputs:



Voronoi Noise→ 3 biomes: water, forest, desert

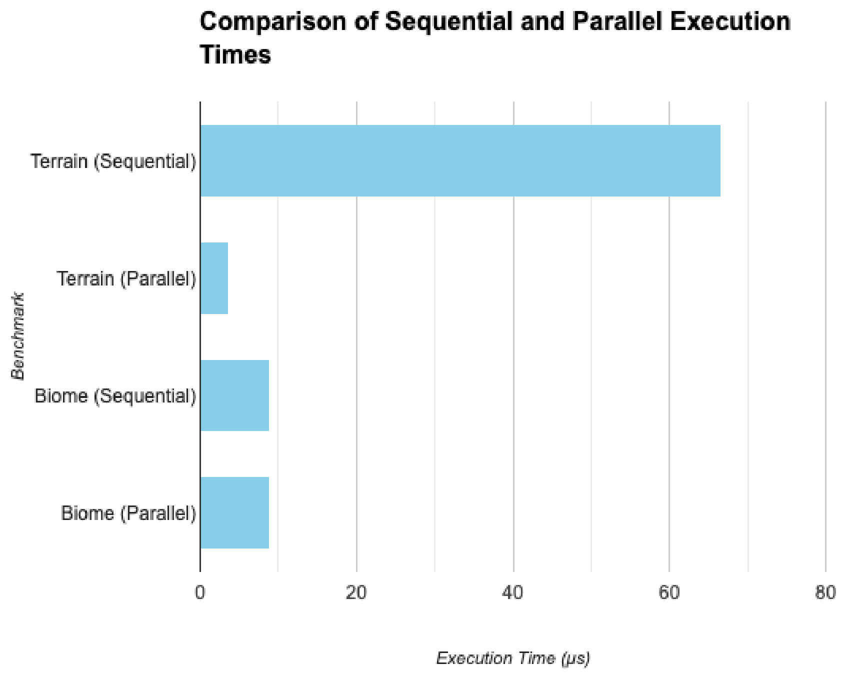


Perlin noise→ 5 elevations: snow, mountain, grass, sand, water, deep water.

Benchmarking

The Criterion Haskell library is employed to evaluate the performance of both sequential and parallel implementations of terrain and Voronoi biome generation. Criterion provides accurate and statistically robust measurements by running each benchmark multiple times and calculating the mean and standard deviation of run times. This method helps in mitigating the impact of anomalies and system-related noise, ensuring the reliability of the results.

The methodology for benchmarking involves running each implementation (sequential and parallel) under the same conditions and comparing their execution times. The key metrics used for evaluation are the mean execution time and the standard deviation of these times, providing insights into both the efficiency and consistency of the implementations. As seen in the graph, the parallel implementation of terrain generation shows a significant decrease in execution time compared to the sequential version, demonstrating the effectiveness of parallelization. However, for Voronoi biome generation, the execution times are similar for both implementations, suggesting that the computational complexity or the data size may not sufficiently benefit from parallel processing. This graph effectively illustrates how parallel processing can dramatically enhance performance in certain scenarios, but its impact can vary depending on the nature of the task



Benchmark Graph A

Appendix

Terminal output (used for Benchmark Graph A):

```
benchmarking Terrain Generation/Sequential
time                 66.18 ns   (65.70 ns .. 66.79 ns)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 66.73 ns   (66.34 ns .. 67.33 ns)
std dev              1.607 ns   (1.129 ns .. 2.597 ns)
variance introduced by outliers: 36% (moderately inflated)

benchmarking Terrain Generation/Parallel
time                 3.674 μs   (3.568 μs .. 3.909 μs)
                     0.974 R²   (0.922 R² .. 1.000 R²)
mean                 3.666 μs   (3.574 μs .. 4.034 μs)
std dev              568.7 ns   (24.85 ns .. 1.205 μs)
variance introduced by outliers: 95% (severely inflated)

benchmarking Voronoi Biome Generation/Sequential
time                 8.859 μs   (8.851 μs .. 8.870 μs)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 8.854 μs   (8.839 μs .. 8.889 μs)
std dev              67.34 ns   (46.06 ns .. 109.4 ns)

benchmarking Voronoi Biome Generation/Parallel
time                 8.865 μs   (8.859 μs .. 8.871 μs)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 8.857 μs   (8.845 μs .. 8.878 μs)
std dev              48.33 ns   (29.66 ns .. 82.78 ns)
```

```
Profiling information:
  62,855,450,080 bytes allocated in the heap
     127,307,744 bytes copied during GC
       1,672,408 bytes maximum residency (142 sample(s))
         728,640 bytes maximum slop
              71 MiB total memory in use (0 MB lost due to
fragmentation)
```

```
                                     Tot time (elapsed)  Avg pause
Max pause
  Gen  0     15917 colls, 15917 par    1.171s   1.023s     0.0001s
0.0027s
  Gen  1       142 colls,   141 par    0.203s   0.125s     0.0009s
0.0173s

  Parallel GC work balance: 35.47% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 8 (8 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.001s  (  0.017s elapsed)
  MUT     time   21.594s  ( 19.518s elapsed)
  GC      time    1.357s  (  1.111s elapsed)
  RP      time    0.000s  (  0.000s elapsed)
  PROF    time    0.017s  (  0.037s elapsed)
  EXIT    time    0.001s  (  0.001s elapsed)
  Total   time   22.970s  ( 20.684s elapsed)

  Alloc rate    2,910,774,195 bytes per MUT second

  Productivity  94.1% of total user, 94.5% of total elapsed
```

References:

Here are the citations for the provided sources in APA format:

1. Wikipedia contributors. (n.d.). Perlin noise. Wikipedia. Retrieved December 20, 2023, from https://en.wikipedia.org/wiki/Perlin_noise
2. Hackage. (n.d.). hsnoise. Retrieved December 20, 2023, from https://hackage.haskell.org/package/hsnoise
3. Wikipedia contributors. (n.d.). Simplex noise. Wikipedia. Retrieved December 20, 2023, from https://en.wikipedia.org/wiki/Simplex_noise#:~:text=Simplex%20noise%20is%20useful%20for,in%20large%20portions%20of%20space
4. Carnegie Mellon University. (n.d.). Terrain Generation. Retrieved December 20, 2023, from https://www.cs.cmu.edu/~112/notes/student-tp-guides/Terrain.pdf
5. Volkov, N. (n.d.). Profiling Cabal Projects. Retrieved December 20, 2023, from https://nikita-volkov.github.io/profiling-cabal-projects/

6. Wikipedia contributors. (n.d.). Worley noise. Wikipedia. Retrieved December 20, 2023, from https://en.wikipedia.org/wiki/Worley_noise#:~:text=Worley%20noise%20is%20used%20to%20create%20procedural%20textures.&text=Worley%20noise%20of%20Euclidean%20distance,the%20location%20of%20the%20seeds.
7. Hackage. (n.d.). Criterion: A powerful but simple library for measuring software performance. Retrieved December 20, 2023, from https://hackage.haskell.org/package/criterion-1.6.3.0/docs/Criterion-Main.html
8. Ronja. (n.d.). Voronoi Noise. Ronja's Tutorials. Retrieved December 20, 2023, from https://www.ronja-tutorials.com/post/028-voronoi-noise/
9. Jasper, J. (n.d.). Voronoi Noise. Catlike Coding. Retrieved December 20, 2023, from https://catlikecoding.com/unity/tutorials/pseudorandom-noise/voronoi-noise/