Project Khaki:

Incorporating Minimal and Regional Pre-generation for River Systems in Procedurally Generated Terrain

Mekal Covic

Abstract

Research was done on the possibility of incorporating minimal and fast pre-generation of river systems with procedurally generated terrain as a means of achieving realistic and believable rivers in this terrain.

Table of Contents

Introduction

In the field of World Generation, particularly in Minecraft, particular interest is at play in the generation of believable, flowing river systems. Purely procedural river systems, in which an algorithm that knows nothing of its surroundings must decide whether to place a river at a position, and at what height, fight the challenge of not being able to have a continuous flow from a high start to a low finish at the ocean. Many existing solutions sacrifice one aspect that makes river systems believable, typically the flow of the river from high to low. It is generally assumed that the generation of longer flowing river systems is impractical. However, this paper suggests it may be a practical possibility, and provides a system to be integrated in future terrain. We will investigate the potential of incorporating fast and minimal pre-generation for regions of procedurally generated terrain in order to overlay believable, flowing river systems that can travel long distances without interruption.

1. Design

The goal presented a major design challenge in developing the system. An early prototype, the original 'Khaki,' attempted to achieve this goal via backwards flow from the ocean across a scaled down grid of terrain. However, it encountered major issues. No pregeneration was used, instead requesting data from the river search as it generated the terrain, and thus a lot of caching was required due to the number of terrain samples it was doing. Frequently, it would get stuck on top of a hill, and river carving, especially blending at the intersection of rivers, proved to be very buggy. This approach was scrapped in favour of designing a new approach to river generation.

1.1 Concept Selection

The first concept was to generate terrain around rivers to fit their path. This way, rivers could be guaranteed to flow from the mountain to the ocean, as they would have control over the terrain itself. This was ultimately decided against, in favour of a second, improved candidate concept for generation.

The second concept drew inspiration from the formation of rivers in real life: generate a continent, place a mountain range along the continent, and propagate rivers to the ocean. Tributaries could then be back-propagated from rivers to the mountains, and sampling at a lower resolution when calculating river paths, like in Khaki, could yield faster pregeneration times.

This was ultimately the design chosen and improved upon in the final design implemented and discussed in this paper. Many changes continued to be developed through the many redraftings of the design and the implementation of the prototype. Notable changes include replacing lower scale sampling with taking larger steps in river propagation, and scrapping back-propagation once it was evident that rivers propagated from multiple sources that travelled in a given direction would merge on their own, just as in real life.

1.2 Design Specification

The specification which was worked off for the prototype was as following. Note that all steps aside from what is directly specified as pregenerated are done procedurally.

Firstly, split the world into continent and ocean. The terrain of ocean regions is purely procedural and may generate islands, or be entirely ocean, depending on the exact generation desired. For this prototype, it will be purely ocean. Initially proposed was a grid of continents and ocean, however that was scrapped in favour of a more organic placement of continental regions jittered in the ocean.

Secondly, the base terrain is generated. For this prototype, radial noise and OpenSimplex (Spencer, K., 2014) noise were combined to create hilly terrain that lowered into the ocean as it got further from the continent origin.

In steps 3 and 4, pregeneration of continent data is done. This is designed to be minimal so that it may run in the background, unnoticeable by the average player.

Step 3 is to place a mountain range in the terrain, from which the river will generate. In the prototype, we found it was better to run this stage before step 2 in order to have the mountain range affect the base terrain as well. For this prototype, we will have one mountain range.

Step 4 is to propagate rivers from the mountain down to the ocean. Rivers are placed as pathways across the map. For each river, we start at a mountain, take a centre difference in the X and Y axes (treating Z as height) and normalise to get the direction of flow. Steps must be taken to ensure they go to the ocean rather than a pit. This is addressed in section 2: implementation. Proposed ways to achieve this were clever heightmap manipulation in river sampling, or having methods for unsticking, like carving through terrain.

Step 5, the final step, is to merge the rivers into the base terrain. As the terrain is generated, rivers are given priority over the base terrain, flattening and carving into it to forge their path to the ocean. The region of terrain with the river itself has a U-shape, in which the river's water generates slightly lower, ensuring the river cannot overflow. In the case that multiple rivers are nearby, the closest river dictates the way rivers affect the terrain.

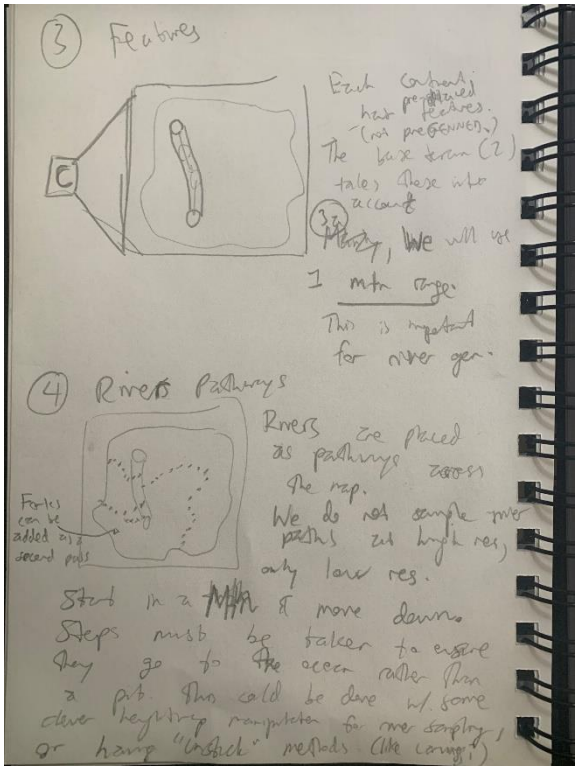

Figure 1.

Mountain-River Pregeneration Draft

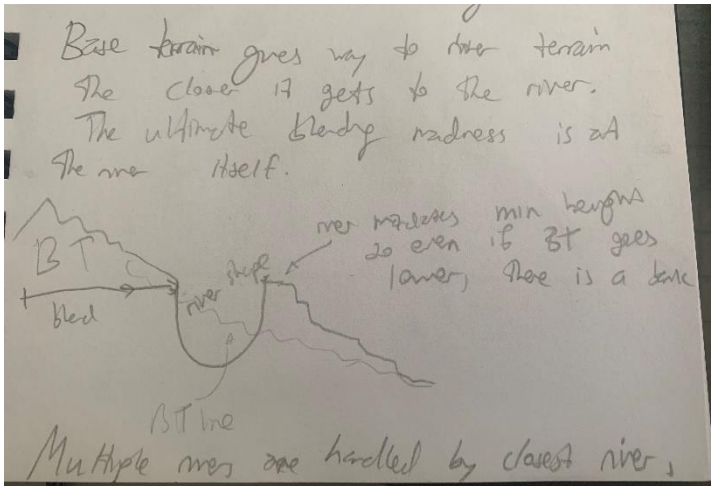

Figure 2

Concept sketch of river-terrain blending.

2. Implementation

The design was first implemented in java, with a viewer written in Swing based off of earlier code for another project. We implemented each stage in the order given in the design. The design was adapted as required during the implementation, especially the river generation itself, which became one of the most complex parts of the whole project.

Distances were measured in 'blocks,' a somewhat arbitrary distance which can be interpreted depending on the application, but designed to work with the game Minecraft™, where this prototype was eventually implemented.

Firstly, the code for identifying continent and ocean was written. A jittered grid was used to select continent positions. A 100-block buffer region to blend each continent into the ocean generation was also established. To ensure continents do not collide with each other and cause unnatural borders, the points needed to be relaxed into less jittered positions. There were two contending methods for achieving this: weighted averaging with a stationary point on the grid, and a spreading algorithm to move points away from close neighbours. The spreading algorithm and averaging both proved equally effective, however the former required each iteration to be cached to achieve relatively fast speeds. Therefore, the less computationally expensive weighted averaging method was chosen, with a relaxation factor of 0.6.

As we wanted a player which spawned near the origin to land on a continent, it was decided to fix the central point to the origin. Therefore, the weighted average constrained all points on the jittered grid closer to the relative 'origins' of their grid boxes.

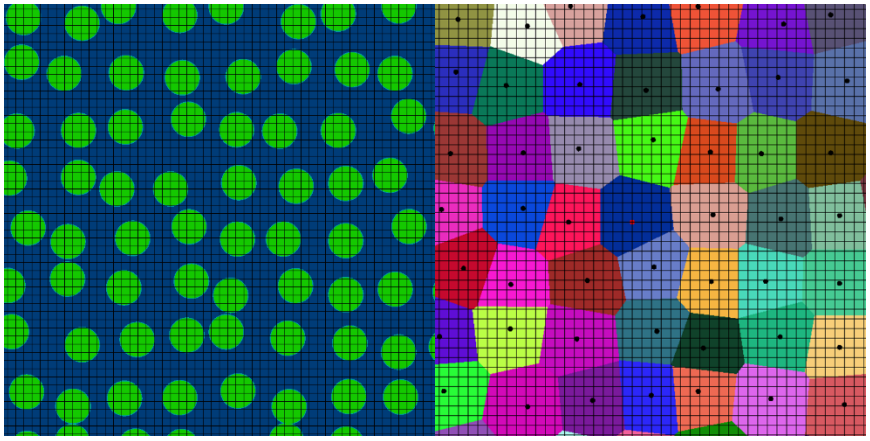

Figure 3

Jittered grid for determining continent and ocean

Secondly, the base terrain was implemented. This was implemented rather simply, as the prototype did not require much. The formula used combined radial falloff with two simplex noise instances, one with low frequency to alter the shape, and the other with a higher frequency, amplitude tempered by the radial falloff, to add hills. The output was then clamped between a minimum and maximum value. y=0 was chosen as sea level, chosen slightly arbitrarily due to its position as the most 'central' real number.

$$height = 60 * (falloff - 0.2) + 30 * shape\ noise + 30 * falloff * hills\ noise$$

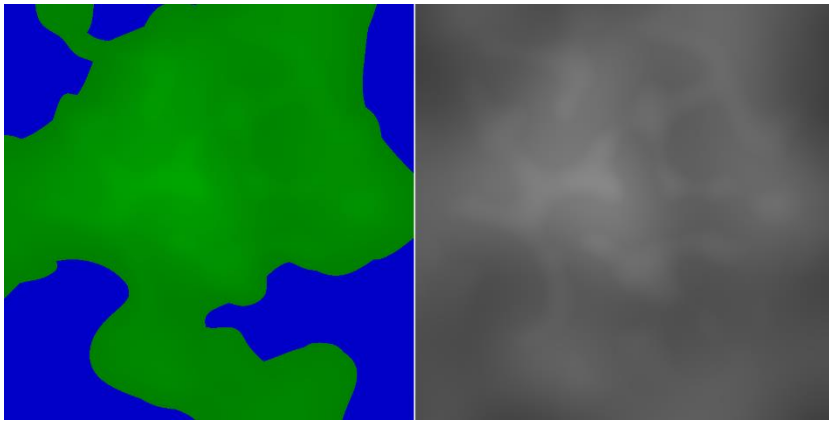

Figure 4

Terrain generated by the base height algorithm for a single continent.

2.1 Mountain Pre-generation

Mountains, being the first pre-generation step laid out in the design, were implemented next. A range of mountain peaks were placed in a line along the continent, their positions offset from the line by a small value. Code to handle them is then added to the end of the base algorithm.

Each point raises the terrain by the 1 minus square root of its distance, up to a maximum distance of 750 blocks, as this creates a curve that escalates to a peak at the mountain centre. **&**In future iterations of this terrain, I would opt to define a range in which terrain is raised and ridged open simplex noise determines peaks instead of using this method of mountain generation.

Care had to be taken to ensure the mountains did not generate too far from the centre of the continent, as the terrain could then be raised such that the continent extends well past the edge of the continent region. The central point of the range was, though randomly placed, forced close to the continent, and the length of the mountain range was constrained between a minimum and maximum length.

Figure 5, Mountain Generation and its effect on the terrain

2.2 River Pre-generation

Rivers were pre-generated by plotting the general paths of rivers, connected as river nodes, via a source-drain method. For each river to generate, a point was chosen within the mountains to propagate from, a central difference was performed of the base height, and a step of 16 blocks was taken in that direction. This was repeated until the river reached below sea level, where it can be guaranteed it has made it out to sea. This was enhanced by methods to merge rivers, smooth river paths, and prevent rivers getting stuck in ditches. Finally, river points were stored as nodes containing two points, in addition to additional data only for pre-generation. River nodes were stored in cells which represented 64x64 regions of the region to pre-generate in, so that only nodes near a given point are iterated over when looking for the closest node to a point.

The first major enhancement to the river algorithm was merging. A merge threshold was decided upon, and any rivers which came within the given threshold to another river while pathfinding was merged into the river, if it could flow horizontally or down into the river. /merge, divert, having to account for not redirecting old nodes of self (? wait how did this even happen – check commits), and the 1->0, 0->2->1 conundrum

A 'frame' of the node was used for smoothing which contained only nodes added for the river currently being generated, to further ensure the smoothing algorithm only handled notes

Another major issue that had to be contended with was the tendencies of propagating rivers to get stuck in a pit. The solution to this issue was to give rivers a wider field of view if the step they could not find a downwards path. Twice this was done: firstly, as a general measure in the central difference calculator: if none of the four sampled points were lower or equal in height to the current position, the river increased its view and took a larger stride accordingly. Secondly, if a river took an upwards step at any point, it would force it to take a larger field of view and a larger stride, as well as correcting that upwards step to staying at the same height, in order to prevent rivers flowing upwards. It would sometimes take a few steps for a stuck river to truly find a way out, and thus a spaghetti mess of river nodes would be created. Thus, a smoothing algorithm was created to merge river nodes if they got too close, cutting out the original longer and likely spaghetti connection. This was implemented on the nodes rather than on the points to take advantage of the performance optimisations offered by the grid-box system of storing nodes.

Notes: [[Lorem ipsum source, flow, merge, redirect strategy, smooth, unstuck strategies of increasing view, 'carving' vs 'tunneling' in combining. This took a most time in the project, and has a lot of steps.

Merge/Redirect/Smooth interactions and Pathfinding Issues. 2 solutions for first, main solution for second being giving the pathfinding a wider view over which to compute the gradient, taking viewsize/riverstepsize steps in that direction to account for the wider view]]

References

Spencer, K. (2014, September 19). Visually isotropic coherent noise algorithm based on the simplectic honeycomb. Retrieved from https://gist.github.com/KdotJPG/b1270127455a94ac5d19/df72ca5e708ebcb1d3c401a0617e1e288c76da82