Salt: An infinite, real-time, procedurally-generated island world in Unity

CS490 Term Report
Advisor: Prof. Holly Rushmeier
Lining Wang

May 2017

1 Abstract

The goal of my project was to develop a performant approach to the procedural generation of islands in the context of real-time world creation in Unity, realized through a video game which utilizes these methods.

The base island terrain was generated through combining traditional gradient noise methods, including Perlin, Value, and exponential noise, with island masks. However, the main novel contribution of the project was its world navigation and creation mechanism, consisting of a pipeline in which islands are seeded, assigned to tiles (which locate the player within the larger global map), and ultimately rendered through meshes as the player moves through a tile neighborhood. That is, the world expands naturally in response to player exploration. Furthermore, islands are compactly represented conceptually through a series of attributes to facilitate flexibility in rendering the final mesh. With multi-threading, this approach becomes performant as the computational work of generating each island is more efficiently distributed over multiple frames, resulting in the islands appearing realistically on the horizon with minimal lag. My approach thus successfully achieves the visual novelty produced by large-scale, macro procedural generation while navigating resource limitations.

I also wanted to achieve an aesthetic vision of a post-industrial, desolate world through procedural methods. To this end, I implemented Worley cellular texturing and solid texturing, which also benefited from the multi-threading optimization approach, in addition to an assortment of post-apocalyptic artifacts and props.

Through interacting with the game in narrative mode, I hope that all of these technical and aesthetic components will come together to form a conceptual whole – one in which players experience a fruitful exploration of a mysterious world.
2 Design overview and completed deliverables

I decided to complete my project in four major phases, from a simple terrain prototype to optimizing the performance of an infinite world of procedurally generated islands.

The first allowed me to gain familiarity with commonly used procedural generation methods, such as gradient noise, and generate some base terrain. In the second phase, I designed a navigation pipeline which uses a tiling abstraction to enable the “exploration” aspect of the game, as well as a seeding mechanism to distribute islands. In the third phase, I focused on the project’s aesthetic and narrative vision by experimenting with various texturing approaches, props, and initial gameplay mechanisms. Finally, to make the game performant, I used profiling and multi-threading programming techniques to optimize and evenly distribute the generation pipeline’s computationally-heavy components.

2.1 Base island terrain

Goal

Generate realistic island terrain using some combination of noise functions and contour shaping.

Gradient noise functions

Gradient noise functions are widely used for fast generation of mountainous, realistic terrain. I define such a function $f$ as one that takes a n-dimensional coordinate as input and returns a float $\in [0, 1]$, interpreted here as the height corresponding to that coordinate. Iterating $f$ over a $nxn$ two-dimensional grid yields a heightmap for that grid, where $n$ is the resolution. From there, I created a Mesh by 1) associating the height-map values with three-dimensional vectors, which form the vertices of the terrain; and 2) calculating the triangles connecting those vertices.

I implemented three forms of gradient noise, each producing a different type of terrain:

1. **Perlin noise.** [2] The industry standard. I implemented the updated, 2002 version of the algorithm which increases the number of reference unit gradients to 16 and uses a different cubic interpolant function $(6t^5 - 15t^4 + 10t^3)$ to remove artifacts.

2. **Exponential noise.** [3] A variation on Perlin noise which results in a more realistic landscape (that is, one with fewer prominent landmarks) by using an exponentially decreasing gradient distribution, based on elevation and gradient data from Utah’s mountain ranges. This is slightly less efficient than Perlin noise because it requires an additional hash function evaluation and scalar multiplication by a value in the (exponentially-decreasing) gradient magnitude table.

3. **Value noise.** [5] Primarily a learning exercise, this method is faster than Perlin noise but results in jagged, uneven terrain.
In addition, I implemented a form of **Voronoi noise**, developed by Steven Worley [4]. Similar to Perlin noise, it was originally intended for use in cellular textures. Worley’s algorithm $f(x)$ assigns feature points to a cubic unit in space through an RNG seeded with the cube’s global position, $\lfloor x \rfloor$. The number of points is determined by inputting the RNG’s first value into a Poisson lookup function, and the location of each point is also determined by the RNG. At the same time, it keeps track of the $n$th closest points to $x$. Using coefficients described in [1], I use a linear combination of the first and second closest points, specifically $-c_1 + c_2$, to calculate the final output value.

I also implemented the spectral synthesis approach described in [1], in which various layers or octaves of noise are combined to create a more realistic, detailed effect (see figure 1). Specifically, for $n$ octaves, the final noise value is $\sum_{x=1}^{n} f(p(\lambda x^t))A^x$, where the point $p$ is multiplied by the product of the frequency $r$ and the lacunarity $t$ per iteration, and $A$ is initialized to the **persistence**, usually 0.5.

**Masking functions**

The primary characteristic distinguishing a **island** from a **terrain** is the presence of a contour, where the edges of terrain tapers off to at or below sea level. To achieve this, I implemented two masking functions $m$ described in [6] to manipulate the height values.
Figure 2: Terrain before and after application of the circular mask.

into an island-like shape. The final height value is $h = m(f(p))$.

1. The **square** mask $\max(|p.x|,|p.y|)$ applied to all points $p$ transforms the terrain into a treasure island-like shape.

2. The **circular** mask $\sqrt{|p.x|^2 + |p.y|^2}$ applied to all points $p$ imposes a circular shape on top of a square patch of terrain. See figure 2.

The future development, I hope to experiment with a greater variety of masks, in particular using the composition of patches of terrain to create irregular, non-symmetrical, or oblong shapes.

Environment set-up and aesthetic refinements

Figure 3: Base island terrain after masking and environment effects.
This step was concurrently completed with the noise and masking implementations. In short, I experimented with the addition of a gradient to the island mesh’s Colors array, effectively creating a colormap. I also found and placed the lighthouse prefab /prop to anchor the game’s story element. Although I was not able to write a shader to make the parts of the island underwater transparent, I sufficiently lowered it so that the visual disturbance was minimal. Finally, I experimented with several metallic textures found through Unity’s web store to begin the process of executing the post-apocalyptic aesthetic vision I had for the game (see figure 3).

2.2 World generation

Goal

The creation of an expanding global map in which 1) island locations are distributed randomly and isomorphically; 2) islands are generated on the horizon from the player’s FOV in response to player exploration, 3) previously encountered islands retain the same appearance when returned to, and 4) the data structures needed to enable these features are represently compactly through Classes.

My approach was to construct a set of abstraction layers in which Seeders assign Island locations to Tiles, while the Nav script keeps track of the player’s current Tile and Tile neighborhood, allocating new Tiles where necessary as the player moves around.

Tile abstraction

The outermost abstraction layer is the Tile, aka a square plane defined by a unique (Vector3) Coordinate, or its location on the global map (corresponding to the lower-left corner of the square to make calculations easier). In Unity, a Tile is represented by a Plane, a GameObject primitive. It also has a (Vector2) Size property which helps determine the Scale attribute of the Tile’s Transform component. To link it to the next level of the abstraction hierarchy, Tiles also contain a list of Islands with which it is associated.

Island and Terrain abstractions

The next abstraction layer is the Island, which is a specific type of Terrain. That is, Island = Terrain + Mask. An Island is uniquely identified by its global (Vector3) Location corresponding to the center of the Island.

Other useful attributes associated with an Island inherited through its Terrain super-class are its (Vector3) Size, specific shape mask, (Texture2D) texture, the noise function needed to create the final Mesh, gradient Coloring, resolution, Material etc. The terrain and Island classes are useful because they neatly package these properties together, enabling a higher-level script to create the final GameObject deterministically, complete with MeshFilter, MeshRenderer, Material, etc, as needed. This has performance benefits because it is more efficient to store these properties in memory than an
Island seeding

Finally, the locations of the Islands are determined by the Seeder class, which returns a Vector3[] list given an initial position and position range. I decided to re-use the functionality of Perlin noise by repeatedly testing the output of Perlin(p) against a certain threshold, and if it is greater, adding the position as an Island location, generating a random point within the given range, and continuing this process until the output is less than the threshold. The threshold itself is determined by the Island density, which is a player or script-tweakable attribute.

Figure 4: Navigation testing mode featuring a complete tiling neighborhood, with spheres as island stand-ins.

Player navigation, or putting it all together

The Nav script links these layers together, and is the only script required to be attached to the GameObject containing the player’s location. There are only two data structures needed to enable navigation: 1) A dictionary allTiles mapping a Tile’s Coordinate identifier to its object instance, representing all the Tiles that are currently active in the GameObject hierarchy; 2) A dictionary activeTiles which is a subset of the above, but only containing the Coordinate identifiers of Tiles that have been physically visited by
the user.

It is important to keep these sets of Tiles distinct because the mechanism of player navigation relies on keeping a Tile neighborhood around the player at any given time. For example, suppose the player’s position $p$ is in a Tile $t$ which has not been visited previously. Then, $Nav$ knows to preemptively prepare $t$’s surrounding Tiles for the player’s potential arrival. (There are two types of neighborhoods I implemented: vonNeumann, which adds a Tile in each cardinal direction for a total of 4, and Complete, which adds all 8 adjacent Tiles to the player’s current Tile.) The specific set of events is as follows:

1. For each Tile in $t$’s neighborhood added not already in allTiles, the Seeder determine its list of Islands.
2. A combination of user and default settings determines each Island’s unique attributes, which is then queued to be displayed.
3. $t$ is added to the activeTiles dictionary.
4. The newly created Tiles and Islands are rendered as Unity GameObjects.

The Tile neighborhood around the player’s current position mainly serves as visual padding, enabling Islands to appear on the horizon when a Tile’s size is large enough. However, edge cases can occur when a new Island’s position is closer to the edge of its Tile that the player is oriented towards. In the future, I hope to spatially differentiate between the neighboring, unvisited Tiles, and a player’s current Tile, as the sizes of those are currently the same, to address these edge cases.

**Design analysis:** The simple skeleton of the navigation system enabled me to quickly build features by assigning them to the appropriate tier in the abstraction hierarchy, corresponding to the scope. For example, Story-related event catchers such as the first island visited are placed in the $Nav$ script itself, while operations such as destroying all GameObjects associated with a Tile’s Islands belong to the Tile class.

In the future, I imagine that the skeleton could be even more simplified by cutting out the Tile abstraction entirely and having the $Nav$ dictionaries map directly to lists of Islands, but currently, Tiles are a useful conceptual abstraction.

### 2.3 Aesthetics and story

**Goal**

The aesthetic feel of Salt should be post-apocalyptic, in that it feels desolate, post-industrial yet otherworldly, and infinite. It should hint at a deeper story while revealing the minimum, much like icebergs, or islands. Specifically, the lighthouse is a conceptual and emotional motif which should appear at locations of importance.
Procedural textures

To create a post-industrial, otherworldly effect, I decided to use two forms of procedural textures. Cellular textures, as discussed previously, were developed by Worley as the textural equivalent of Voronoi diagrams. As seen in Figure 5, the resulting shapes create a scale-like effect which balances between organic (sand cracks, scales of various animals, etc) and industrial (rigid, geometric plates). Perlin noise, on the other hand, creates a solid texture and a feeling of spottiness, decay, and disease. Technical details follow:

1. **Cellular textures** were generated by applying and tiling Voronoi noise with coefficients specified in [1] over a 2D grid, where the float returned corresponds to a color value.

2. **Solid textures** were generated by applying and tiling Perlin noise. See Figure 5
Post-apocalyptic artifacts and effects

As seen in Story Mode (in the demos section), I included props and aesthetic motifs to add a bit of direction to player exploration. Currently, the appearance and disappearance of motifs are aligned along Tile borders, but I hope to make this more flexible in the future. Specifically, my idea was to associate each island with a story or memory, and thus associate time with a place instead of vice versa.

1. To further assert a post-apocalyptic aesthetic, the color of the water in each tile is assigned randomly to hint at corrosion or chemical contamination if there is at least one island in the tile.

2. If a Tile contains an island, the lighting for the Tile will be randomly chosen (currently not retained in memory) out of a list of Skybox prefabs I found through the Unit web store. The lighting corresponds to a certain time of day, ranging from sunrise to sunset.

3. For each Island, an array of props will be instantiated on and around the island such as trash cans, boxes, and other aged items. [9]

4. I gathered industrial Materials from a free pack [10] in the Unity Web Store, such as gold, glass, metal, and copper, to enhance the aesthetic quality of the experience. The effect of these, in particular compounded with the procedural textures, is shown in the Demos section.

5. The lighthouse prefab [7] is instantiated at the first island the player visits. Though I didn’t complete the first chapter of the story, the idea is that the player will be given clues to locate the lighthouse (such as the water changing), which will serve as their home base for future exploration.

2.4 Performance

Goal

Islands are rendered on the horizon in real-time with acceptable resolution and imperceptible frame delay.

Profiling, manual tweaking

Initially, there was a noticeable lag during the frame when the player stepped onto an unexplored Tile, and the navigation algorithm attempted to calculate the mesh and render the Islands for the surrounding Tiles. Each island took about 300 ms to generate, which was aggregately unacceptable when each Tile contained an average of two to three islands. By using the Deep Profile feature of Unity, I was able to zero in on the specific functions which took up the most computation time.

As suspected, those functions were the noise functions in the various Noise classes, primarily Perlin and ExpNoise. These functions are called at least $r \times r$ times during the
Table 1: Max CPU usage time per frame for the gradient noise operation (exponential noise), with resolution=150, where $i$ is the number of islands being generated in the neighborhood; pre- and post-multi-threading

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>i=1</th>
<th>i=2</th>
<th>i=3</th>
<th>i=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-opt</td>
<td>128.34</td>
<td>249.48</td>
<td>380.18</td>
<td>809.36</td>
</tr>
<tr>
<td>Post-opt</td>
<td>141.43</td>
<td>132.46</td>
<td>141.54</td>
<td>142.97</td>
</tr>
</tbody>
</table>

Table 2: Max CPU usage time per frame for one texture, with resolution=64, density = 10, 2000 ops p/second with 10 threads; pre- and post- multi-threading

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>solid</th>
<th>cellular</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre-opt</td>
<td>437.36</td>
<td>2130.36</td>
</tr>
<tr>
<td>post-opt, w/o tiling</td>
<td>183.11</td>
<td>196.90</td>
</tr>
<tr>
<td>post-opt, with tiling</td>
<td>171.16</td>
<td>170.95</td>
</tr>
</tbody>
</table>

frame, where $r$ is the resolution, so any improvement to their performance would impact the overall computation time. I used the following optimization techniques to reduce the CPU computation time of each island to approximately 150 ms for those specific noise functions:

1. Reduced the use of data structures such as Vector2’s, replacing them with const-sized float and int arrays. Previously, there were about four Vector2 allocation calls per each function call. This also required custom implementations of the Dot function.

2. Changed the method of computation of the final noise value. Previously, I was using a scaled fraction combination or ratio of the output of the functions (for example, assigning a weight of 0.7 to the exponential noise generate height value, and 0.3 to the perils noise generated height value), but I decided the aesthetic experimentation was not worth the tradeoff of doubling or tripling the computation time.

3. Replaced my custom wrapper implementations of specific math functions such as Floor or Lerp with the standard library functions, thus reducing function-call overhead.

4. Similarly, caching some constants in another class within the individual noise class resulted in a small improvement.

Despite these improvements, an average lag time of 300 ms for two Islands was still unacceptable, and disrupted the game experience. Furthermore, the cellular texturing noise function did not respond at all to these small optimizations, as it contained a triple-nested for-loop, and each texture’s size was vastly larger than a terrain’s due to the necessity of tiling. Thus, I decided to develop a more sustainable approach.

Multi-threading

The fundamental problem underlying the game’s unsatisfying performance was that the Update function in the Nav script attempted to do too many operations during the first
frame when the player navigate to a new Tile. That is, it attempted to execute all function calls sequentially. Thus, I devised a multi-threading approach which spreads out the work over several frames.

First, I identified the functions which could benefit from being executed in parallel with Unity’s main thread. These were the RenderTerrain and fillTexture functions, corresponding to the macro terrain and texture generation, respectively.

Then I implemented a threadpool attached to the OptController class which exposed an interface for other classes to register tasks to a atomic queue. The number of threads in the pool was determined at initialization time through a user setting. By using C#’s native threading capabilities, each thread was assigned a certain number of tasks from the queue. When they completed these tasks, they automatically add themselves to an idle queue and would be reassigned tasks on the next frame update, as long as there were tasks in the queue.

I identified the specific tasks that would be added to the OptController’s queue as individual, atomic instances of the noise functions being called. I decided to do this rather than the macro function in general because I was concerned about the evenness of the tasks being spread out. For example, if all \( n \) threads were pre-occupied with rendering islands, then a call to fillTexture might not be addressed for a long time.
time. By increasing the granularity of the tasks assigned to threads, I hoped to balance out this work more evenly and fairly.

Finally, I modified the player navigation script to add seeded Islands to a List within the Nav script itself. This is because the process of rendering the Island by creating a GameObject with its corresponding mesh is a separate operation from generating the mesh itself, and all island rendering was handled by the Nav script for organizational simplicity. Though this is not the most efficient method, I checked per frame if any Islands were available to be rendered, and removed them from the List accordingly. I implemented the analogous data structures for textures, although this was slightly more complex since I had to create a data structure for each of the two possible cases: textures completed after the island’s mesh was completed, and textures completed before.

The performance improvements of this approach is shown clearly in Figure 6 and 7. In the first, we see that rather than concentrating the compute time in the first frame, as in the top graph, the computation spikes as each Island is generated separately. The tradeoff here is that the Islands take longer to generate, but I believe this is a tradeoff worth making, as the player is likely to explore around each island before moving on to the next one; that is, the approach takes advantage of the player’s natural curiosity as “free time” in which to prepare for future tasks. This is corroborated by Table 1, which clearly shows that the max lag per frame increases linearly with the number of islands requested to be generated, while it stays relatively constant for the multi-threading approach. In Figure 7, a similar dynamic occurs. Note that on average each frame takes
longer to generate, 16ms vs about 5ms in the texture generation, and 66 ms vs 5ms in the worst case in the terrain generation; however, those still correspond to mostly acceptable frame rates overall (60 and 15, respectively).

**Design analysis:** The granularity of each task, consisting of a function call that takes less than a hundredth of a millisecond, may be too little compared to the cost of its overhead, and the locking of the atomic queue data structure. Furthermore, though I spent a fair amount of time implementing a tile-based approach to texture generation (vs doing it all in one go, albeit via a thread pool), the overhead caused by the additional tile composition operations resulted in a negligible performance improvement (see Table 2, in the “with tiling” row). Also, the design would benefit from incorporating a more asynchronous programming paradigm, such as notifying the Nav script when a task has been completed rather than checking for it every frame.

### 3 Notes on implementation

**Technology used**

Salt was created in Unity and implemented in C#. The code for the game can be found at Github and is completely open-source: [http://github.com/linii/in-valence](http://github.com/linii/in-valence). To import into your own Unity project, simply duplicate the contents of the Assets package.

The resources/assets listed below are not included, but most of them are free and available to download through the Unity Web Store.

**Code structure**

The majority of my code is contained in two modules: the _Lib folder contains implementations of gradient noise functions, masks, math functions, mesh util functions, etc; while the _IslandGen folder uses the exposed functions classes in _Lib in user-facing scripts. It also holds the navigation Pipeline. The distinction between these two modules is demonstrated by the _Lib/Terrain/GenericTerrain and _IslandGen/Island classes, in which the latter inherits from the former; that is, conceptually, an Island is a specific instance of a conceptually generic Terrain to which a Mask has been applied.

**Extensibility**

A goal mentioned in the proposal was the creation of a Unity assets package containing the scripts.

- **Tile** testing mode. When enabled, a plane filled in a random color will be shown instead of the computationally-heavy water prefab.

- **Island** testing mode. When enabled, a white sphere with coordinates at each island’s seeded origin and scale corresponding to the island’s Gaussian-distributed size will be shown, bypassing height-map calculation.
• **Story** testing mode. When enabled, `Ui.Text` representing the main character and their interactions, as well as various aesthetic props mentioned previously, will be displayed. If not enabled, the game automatically runs in **Exploration** mode.

• **TerrainCreator** testing script. Useful for gauging the effects and performance impacts of tweaking variables associated with noise functions (such as resolution, frequency, persistence, and lacunarity) without the player navigation component.

• **TextureCreator** testing script. Useful for gauging the effects and performance impacts of tweaking various associated with procedural texture functions (such as texture density and resolution) without player navigation. When paired with the Opt scripts, also demonstrates the performance savings of using asynchronous processing over per-frame calculation.

**Demos**

Demo videos are available on YouTube for a wide variety of noise and texture settings. Click here for the entire playlist.

- Exploration mode - Perlin noise, no textures [4:25]
- Exploration mode - Perlin noise, solid textures [6:17]
- Exploration mode - Perlin noise, aesthetic materials, no textures [0:27]
- Exploration mode - Exponential noise, solid textures [1:36]
- Exploration mode - Exponential noise, aesthetic materials, cellular textures [6:28]
- Story mode - Perlin noise, no textures [0:55]

**Assets used**

To facilitate the aesthetic development of the game, I downloaded the following models and materials from the Unity Web Store:

- The lighthouse prefab / model, based on the Port Canaveral lighthouse. [7] ($10)
- Skyboxes, from the Classic Skybox package. [8] (free)
- The trash prefabs, from the Low Poly Props Pack. [9] (free)
- Metal textures, from the Basic Metal Textures Pack. [10] (free)

In addition, I used the Water4Advanced prefab from Unity’s Standard Assets pack [11] for the ocean, though I manually tweaked the settings on the attached WaterBase script for aesthetic purposes.
4 Future work

I plan to continue development on Salt in the future. Some features I didn’t get to include:

- Implementing various aging algorithms, such as mesh denting and bumpmaps, to add to the industrial post-apocalyptic effect.
- Adding a terrain mode to increase the extensibility of the main Nav script.
- Using Voronoi noise as a mask for the islands, resulting in more unpredictable shapes. In that same vein, fixing a current error where the Voronoi noise is unable to be integrated to the async optimization module due to its use of the Random.InitState function outside the main thread.
- Experimenting with a different form of async processing, in which the tasks are not atomic but continuous.
- Refining the main gameplay mechanism of recovering memories from islands, that is, tying time to a place.

5 Acknowledgements

Thank you to Professor Rushmeier for guiding me through the process and organizing group meetings with others working on similar projects. Also thanks to Bella Cat for her emotional support.

References

Papers and web resources

Unity Assets