McGill University

COMP 521

Modern Computer Games

TerraGen: Procedural Terrain Generation and Resource Analysis

Benjamin San Souci
Maude Lemaire

April 14, 2015

1 Introduction

TerraGen aims to generate realistic island terrains using a relaxed polygonal map and assigning biomes based on simulated moisture and elevation. Following the map generation process, the city generation algorithm analyzes the terrain and produces a set of cities with unique statistics directly related to their location and access to resources.

Island generation was modeled after Amit Patel's "Polygonal Map Generation for Games"[12], where Chrisophe Le Besnerais's JavaScript version, "Island.js"[7] was extended to accommodate for additional features. First a graph structure of polygons is generated, then annotated with elevation, moisture, and labeled as either land or water. In order to generate a more natural-looking island, Voronoi polygons were generated and relaxed using an implementation by Raymond Hill [6]. Extensions to the implementation included adding fishing regions, minerals, volcanoes, tectonic plates, forest density, and fauna abundance.

The TerraGen city statistic generation addresses a problem that has been little discussed in the scope of games beyond the intersection of geography information systems (GIS) and 3D game engines. Using the generated plot, the city generation algorithm finds a suitable location at which to place a city. From this location, resources in close proximity are identified and used in determining the city's population size, army size, happiness level and wealth, among other features.

The result is a randomly generated island with over two dozen biomes, oceanic regions, dispered minerals and a handful of cities. A sample map at different stages of generation is included in the methodology portion of the report.

The project was separated into two major components: the terrain generation and extensions, and the city statistics generation. Maude was mainly responsible for the terrain generation portion of the project and the majority of the final report. Benjamin produced the city generation algorithm. Both team members collaborated on the methods by which terrain analysis and terrain generation were extended.

2 Background

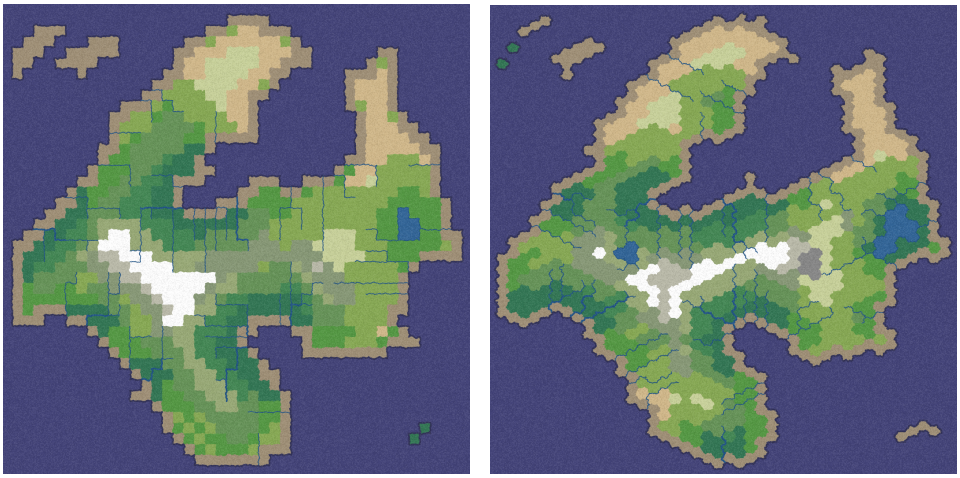
Map generation has been a important focus within the gaming industry in recent years as a means of promoting re-playability without having to manually generate a large number of maps. By randomly generating a map with new instances of a game, the players must face new challenges and adapt their game-play to suit the new terrain. A wide range of approaches already exist, with varying levels of complexity and randomness.

2.1 Grids

Terrains are typically built off of a grid system; this can be a simple tile-based square grid, a hexagonal grid, or a polygonal grid either composed of randomly sized convex shapes or relaxed polygons. Figure 1 displays the difference between each of these grid types and how "natural" each of these appear in island generation.

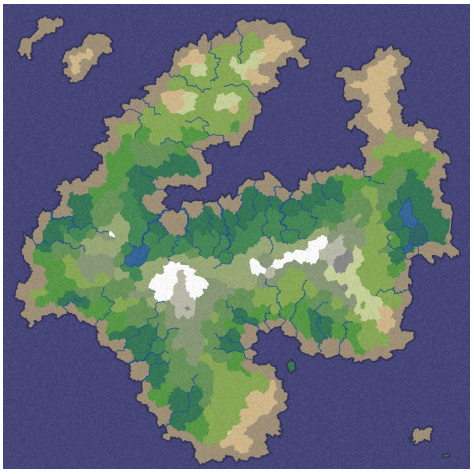
Hexagonal grids and polygonal grids typically provide the most natural-looking islands. Hexagonal grids have been extensively developed by academics such as Clark Verbrugge[14], and can be perturbed to add irregularity. Polygonal grid generation has mainly revolved around randomly generating a series of Voronoi polygons. A number of algorithms generate Voronoi diagrams based on a set of random points (Fortune's algorithm[4]), a set area (Lloyd's algorithm[2]) or Delaunay triangulations (Bowyer-Watson algorithm[17]). In particular, the set of points upon which the polygons are generated can be mapped to known natural phenomena. For example, researchers at the Université de Lyon,generated Voronoi diagrams based on hydrology [5]. As an extension, Lloyd relaxation can be used to distribute points more evenly for diagram generation.

Following Patel's polygonal map generation methods, once a Voronoi diagram is generated, the internal representation consists of nodes and edges; the first is used for identifying adjacent polygons. The second represents the Delaunay triangulation which can be used for pathfinding. Conceptually, every corner in the triangulation corresponds to a polygon center and every edge to an edge in the diagram. As polygons in the map are irregular, it is difficult to make assumptions about neighboring cells and paths from one point to another; this representation ensures that the map is traversed in a meaningful way throughout the generation process.

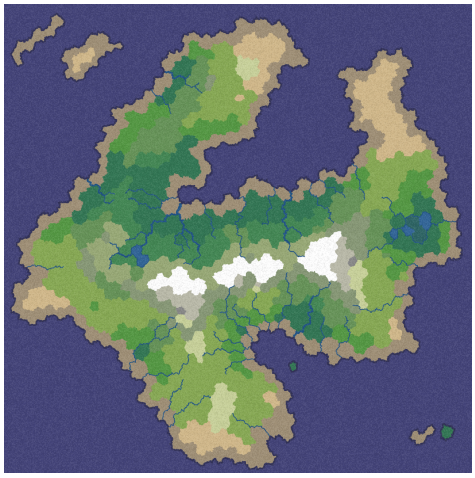


(a) Square Grid

(b) Hexagonal Grid



(c) Random Polygon Grid



(d) Relaxed Polygon Grid

Figure 1: Islands generated using Amit Patel's demo. Each is generated using Perlin noise, displays biomes, and demonstrates the use of different grid types.

2.2 Islands

Realistic islands must maintain a rather circular shape, with some random variation. A radial function is suggested, using sine waves to produce a round island. The island is generated according to polar coordinates stemming from the center of the canvas. Depending on the sine waves, the island can resemble an octopus, with thin strips of land extended from the center of the island.

Alternatively, Perlin noise can be generated and used to control the shape of the island given some constraints. This method provides more realistic island shapes due to the fact that Perlin noise consists of visual details of equal size; this gives a more controlled random appearance, closely imitating natural textures.

Once land polygons have been identified, ocean is filled in from the edges and any remaining "water" polygons unreachable from the fringe of the map (landlocked) are marked as lakes.

2.2.1 Elevation

In generating terrains, in order to provide a more realistic map, a natural-looking elevation must be assigned to each grid unit. Some typical approaches include using Perlin noise, fractal landscapes, or real-world data. Simpler and less computation-intensive attempts at generating elevation focus on assigning a maximum elevation to a central point on the island and gradually decreasing the elevation according to a cumulative distribution until the lowest elevation point (the ocean) has been reached[12].

Perlin noise is a popular technique due to the fact that it successfully generates realistic textures (as previously discussed under island generation). One of the drawbacks of using Perlin noise in elevation generation is that there tends to be a rather uniform distribution to the terrain height and few realistic peaks and valleys typical of natural landscapes. A number of approaches go beyond using strict Perlin noise generation and instead focus on value noise, a less mathematically-involved method which may be calibrated to take actual geographical data into account [11].

Fractal landscapes can provide a pseudo-random terrain elevation; these landscapes focus on self-similarity, where subsets of the object resemble (or are identical to) the whole. Recursive algorithms such as the diamond-square algorithm use midpoint-displacement in order to generate heightmaps. The

resulting meshes can be masked over a terrain to provide elevation information [9].

Each of these methods can be further improved by simulating natural phenomena such as erosion and plate tectonics. In particular, terrain (and specifically elevation) can be simulated entirely according to plate tectonics; although computation-heavy, this approach results in the most realistic elevation map, and can be used to add features such as earthquake risk zones and volcanoes [15].

Whatever approach is used, the resulting elevation map must be applied to the map grid; this can be done by assigning an elevation to each corner, or center of a grid unit. Given a polygonal map (or hexagonal map with more than four corners per grid unit), assigning elevations to corners yields a more complex elevation model with a greater number of discrete steps. Greater detail is key as elevation is then used to derive biomes and rivers later in the terrain generation process.

2.2.2 Rivers and Moisture

Based on the previously determined elevation model, rivers can be traced from a point of high elevation to the ocean. As rivers tend to be thinner than a single grid unit, a more realistic visualization approach involves tracing rivers alongside the edges of grid units, whether they be tiles, hexagons or polygons.

Water-flow models can be further developed by taking into consideration the amount of moisture available at a given starting point. For instance, it is more probable for a river to stem from a snowy mountaintop than a tundra or bare terrain. Instead of drawing strictly uniform-width rivers, rivers can be widened according to surrounding elevation and water availability from a given source; tributaries can combine to form a larger river emptying into the ocean [5].

Depending on the order of the approach taken (rivers generated first or moisture generated first), areas of high moisture could be assigned to those in closer proximity to lakes and rivers, or more lakes and rivers could form in areas of higher moisture. Typically, moisture is mapped to distance from the ocean and elevation. In either means of terrain generation, moisture is modeled after known hydrology models such as those discussed by Génevaux and his team [5].

In summary, moist, fertile soil occurs in areas near rivers and lakes. This

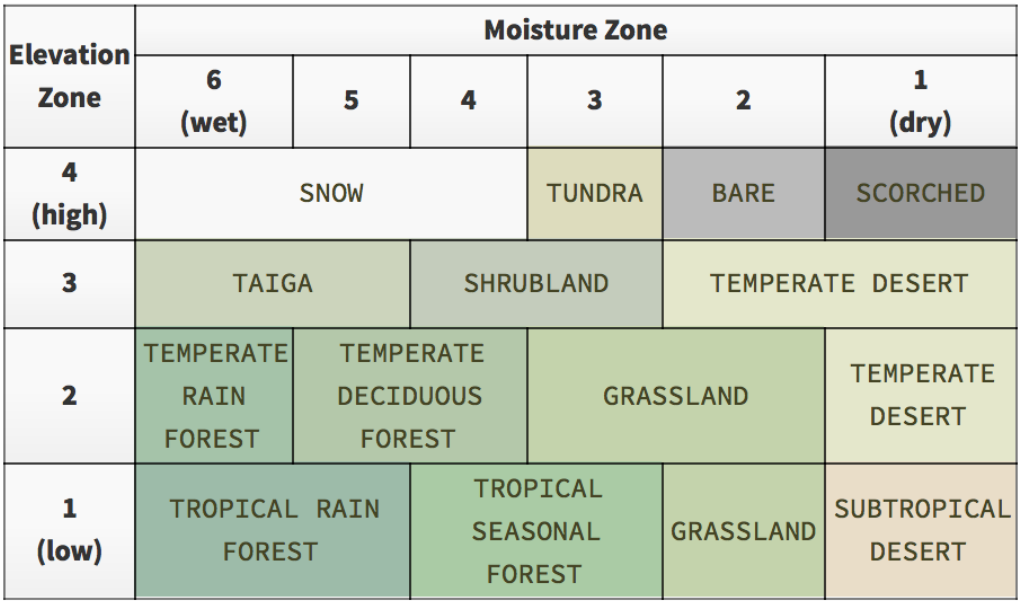


Figure 2: Biomes According to Elevation and Moisture

soil leads to thicker vegetation growth, which in turn slows water flow. Rainfall may vary according to area, but flows downhill in all instances, gaining momentum and pooling in springs, rivers and lakes as it travels towards the ocean.

2.2.3 Biomes and Minerals

Biomes are generated according to moisture and elevation. Depending on the range of values given to each of these features, a wide range of biomes can be simulated. Assuming the elevation and moisture maps demonstrate gradual changes Due to gradual change in moisture and elevation, biomes will be assigned to groups of grid units and rarely result in isolated single-unit regions. Patel proposes a model consisting of fifteen biomes as shown in Figure 2.

Each biome can be associated with a set of flora of fauna, distinct to its features. For example, although tropical rain forests cover only 7% of the earth's surface area, they are home to over 50% of the earth's species. A multi-layered forest directly contributes to the spread of nutrients and perpetuation of the nitrogen, oxygen and carbon cycles [13].

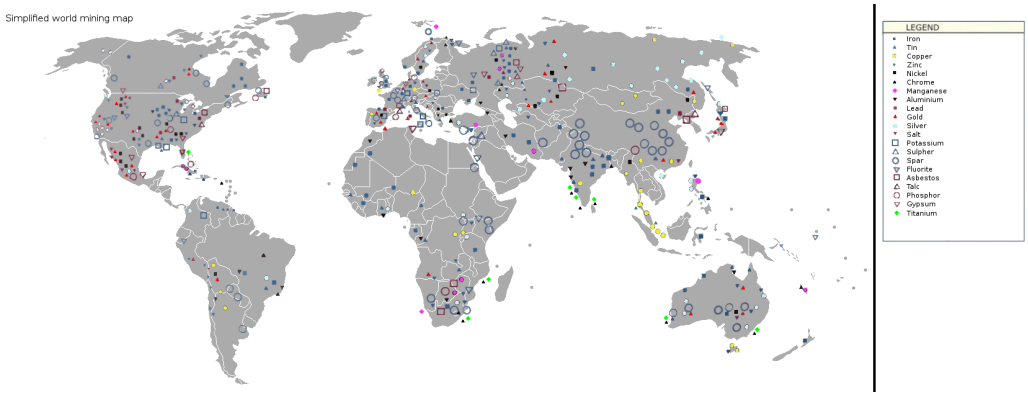


Figure 3: Mines across the world

With every terrain comes resource distribution, including important minerals. Minerals form according to the heating and cooling of molten materials, pressure within the ground, and evaporation of liquids [3]. As such, minerals form within set biomes with different levels of rarity depending on the precise conditions under which they are produced. For instance, gold is mainly found near the earth's core, although veins of gold are gradually being exposed to the surface and eroding away. Due to its weight, gold can become concentrated in hollows and trapped in beds of rivers. A range of mines and corresponding minerals across the globe can be seen in Figure 3.

2.3 Terrain Analysis

Terrain analysis has been little studied beyond urban development in the scope of video games. In recent years, there has been a push towards combining geography information systems (GIS) and gaming. GIS are mainly used for scientific research purposes in the area of geoinformatics; the systems are aimed at designed to capture, store, manipulate, analyze, and present spatial and geographical data. By combining these analytical features with gaming, the industry can produce more realistic scenarios. Understanding how players can interact with geographical features and resources found on the terrain could be a natural extension of including GIS in gaming [1].

Procedural city growth and urban development within the gaming world has mainly revolved around urban, modern settings dealing with multi-story buildings and complex grids of roads and highways. In fact, companies have focused efforts towards building software capable of generating cities. Elec-

tronic Arts has released a tool known as CityEngine, specifically used for city generation in their Need for Speed game series[16]. In regards to medieval cities and little-developed civilizations, although many games revolve around this theme, there has been fewer attempts at procedurally generating such environments.

3 Methodology

TerraGen was built as an extension to an existing JavaScript implementation of Patel's polygonal map generation method, written by Christophe Le Besnerais [7]. This implementation provided a relaxed polygonal grid, with basic elevation and moisture mapping, and a simple waterflow model. The terrain analysis portion of the implementation was generated from scratch, and driven by the provided island map.

Le Besnerais's implementation uses client-side JavaScript to render a map on an HTML5 canvas using Paper.js[8], a JavaScript implementation of Voronoi diagrams by Raymond Hill [6] and a Perlin noise generator by Sean McCullough[10]. Built-in functionality included basic island generation and biome assignment.

3.1 Island Generation

As provided with the existing implementation, terrains are generated on top of a relaxed polygonal grid. Island shape and size is determined by a randomly generated Perlin noise patch of given height and width (which can be seen as a separate `canvas` item within the rendered web page). Islands are highly configurable; a list of customizable parameters are shown in Listing 1. For the remainder of the report, it will be assumed that the island generated has a Perlin width and height of 350, canvas width and height of 1,000 and 50,000 polygons.

Listing 1: Island Parameters

```
Island.init({
        width: 1000,
        height: 1000,
    perlinWidth: 350,
    perlinHeight: 350,
```

```
    allowDebug: false,
    nbSites:50000,
    maxRiversSize: 1000,
    shading:0,
    nbGraphRelaxation: 0,
    shadeOcean:false
});
```

3.2 Elevation

A simple elevation mapping was included in the base Island.js code. In this implementation, elevation is set at each corner of every polygon as the distance from the coast where water polygons do not contribute to the distance. A secondary pass is performed to redistribute the elevations to match the following cumulative distribution:

$$y(x) = 1 - (1 - x)^2$$

.

This further ensures that elevation the ocean will always be reached when traveling downhill.

It chose a central point on the island and gradually decreased the elevation Mapped according to a central location and decreases as the polygons near the ocean. Additional features such as plate tectonics and volcanoes were simulated on top of the generated elevation.

Used to map to fishing areas and randomized throughout the ocean.

3.3 Rivers

Allowed to pool into lakes and lake were allowed to empty into the ocean. Lake colors are distributed according to depth.

3.4 Biomes

Ice was added as an additional biome

3.5 Minerals

Unique extension beyond the basic features provided in the starter code.

3.6 Terrain Analysis

The intent behind TerraGen's terrain analysis is to generate initial statistics for cities. After the island is generated, the terrain analysis portion of the implementation receives the grid. Six cities are randomly distributed on the island, with randomly assigned curiosity and aggressiveness attributes. Listing 2 demonstrates a sample city JSON item.

Listing 2: JSON City Item

```
{
  center: center1,
  resources: {
    vegetables: 0,
    meat: 0,
    wood: 0,
    minerals: {
      copper: 0,
      iron: 0,
      gold: 0,
      titanium: 0,
      silver: 0,
      coal: 0,
      gems: 0,
      aluminium: 0,
      lead: 0,
      nickel:0,
      uranium:0,
      zinc:0
    }
  },
  populationSize: 100,
  armySize: 0,
  location: [center1],
  curiosity: 1.5,
  aggressiveness: 0.2,
  technology: 0,
  artistNumber: 0,
  economy: 0
```

};

The above statistics and features are calculated using a simple update loop, which consists of the following steps:

1. Randomly generate city aggressiveness (value between 0 and 0.5) and city curiosity (value between 0 and 2).
2. Expand the territory of the city.
3. Increase city resources based on controlled territory.
4. Increase population.
5. Calculate economical power of city.
6. Calculate number of artists within the city.
7. Calculate technological advancement.
8. Calculate army size.

3.6.1 City Expansion

In order to properly expand the city, we attempt to double the number of cells controlled with each time-step. To find a newly conquerable cell, the fringe surrounding the city is calculated where unconquerable cells are ignored. Unconquerable cells are determined by current army size and accessibility. Cell accessibility is calculated using Dijkstra's algorithm where weights are defined as the difficulty of moving through cells. Difficulty is a simple vector distance (in a space where x is elevation and y is moisture) between the most prominent biome inside the city and the biome of the target cell. Before adding the cell with minimal move difficulty, the total move difficulty over the whole path from the center of the city to that cell is calculated and compared to the city's curiosity level. If it fits within the threshold, the cell is added to the city's controlled area. Finally, the cell is set to belong to the city, making it much more difficult to conquer.

3.6.2 City Resources

Now that the city has an updated territory, each cell it occupies is viewed in order to determine how much the city can do with that given land. Listing 3 demonstrates a sample cell JSON item.

Listing 3: JSON Cell Item

```
{
  elevation: rand(),
  moisture: rand(),
  minerals: {
    copper: 0,
    iron: 0,
    gold: 0,
    titanium: 0,
    silver: 0,
    coal: 0,
    gems: 0,
    aluminium: 0,
    lead: 0,
    nickel: 0,
    uranium: 0,
    zinc: 0
  },
  conquerability: 0,
  fertility: 0,
  wildlife: 0,
  trees: 0
}
```

The only additional attributes assigned to the cells post island-creation is the conquerability and fertility. Conquerability is calculated according to elevation and the quantity of trees found at the target cell. Fertility is chosen randomly with bounds dependent on the biomes.

To calculate accessible resources, the city is given a collecting power, determined using the following formula:

$$CITY\_POWER = \frac{(1 + city.technology)}{MAX\_TECH} \times (city.populationSize - city.artistNumber - city.armySize) \times WORK\_EFFICIENCY + city.armySize \times ARMY\_WORK\_EFFICIENCY \quad (1)$$

Global constants are used to define how much technology is needed to mine each mineral. The amount of each mineral collected is then calculated according to the following formula:

$$MINERAL\_QUANTITY = Math.floor(Math.min(cell.mineralQuantity, COLLECT\_POWER \times (1 - \frac{MINERAL\_MODIF}{MAX\_MODIF})) \quad (2)$$

3.6.3 Population Size

Population size is determined by looking at the life conditions available at each cell controlled by the city and the cell's distance from the city center. For simplicity, life conditions are deemed to be the same as previously discussed move difficulty.

$$\sum (1 - getLifeConditions(city.center, c)) * 100 \quad (3)$$

Beyond population, a closer look is taken at the proportion of citizens that are artists or enlisted in the army. Due to the fact that zinc is an important mineral in paint, the number of artists is directly proportional to the amount of zinc controlled by the city.

3.6.4 Overall Economical Power and Technology

Economical power is determined according to the amount of gold controlled by the city over the total available amount of gold contained on the map; the quantity of gems acts as an additive component to the economical power but is not affected by its total availability on the generated island.

4 Results

Sample map with cities.

Run ten times with six cities in each simulation, the average generated city has the following statistics:

5 Conclusions and Future Improvements

Speed could be optimized.

References

[1] Elizabeth Borneman. Gis and gaming, 2011.

[2] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Review*, 41(4):637–676, 1999.

[3] Melinda Dyar and Mickey E. Gunter. *Mineralogy And Optical Mineralogy*. Mineralogical Society of Amer, 2007.

[4] S Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM.

[5] Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedřich Beneš. Terrain generation using procedural models based on hydrology. *ACM Trans. Graph.*, 32(4):143:1–143:13, July 2013.

[6] Raymond Hill. Javascript-voronoi, March 2015.

[7] Christophe Le Besnerais. Island.js, March 2015.

[8] Jurg Lehni and Jonathan Puckey. Paper.js, March 2015.

[9] Paul Martz. Generating random fractal terrain.

[10] Sean McCullough. Perlin noise classical js, February 2010.

[11] Ian Parberry. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques*, 2014.

[12] Amit Patel. "polygonal map generation for games", September 2010.

[13] Mike Robinson. The tropical rain forest, October 2008.

[14] Clark Verbrugge. Hex grids, 1996.

[15] Lauri Viitanen. Physically based terrain generation: Procedural heightmap generation using plate tectonics. Master's thesis, Helsinki Metropolia University of Applied Sciences, 2012.

[16] Benjamin Watson, Pascal M?, Oleg Veryovka, Andy Fuller, Peter Wonka, and Chris Sexton. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, 28(3):18–26, 2008.

[17] D. F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.