Generating organic game worlds with multi-level spatial partitioning techniques

Nico Andrew Glas
Amsterdam University of Applied Sciences
Duivendrectsekade 36-38
1096 AH Amsterdam, The Netherlands
nico.glas@hva.nl

February 24, 2013

Abstract

This paper frames the process of generating an organic game world with the use of spatial partitioning techniques. These techniques use sample points to divide a plane into multiple spaces and then using another set of sample points to divide the original spaces. These nested in each other can create even more organic shapes than just one division.

1 Introduction

When discussing procedural game-content generation, research usually focusses on the generation of missions and levels and such. This in itself is not a real shocker, designing missions and/or levels are one of the most time consuming tasks that game developers face. Many games that use a form of content generation use templates (e.g. Blizzard's *Diablo* series) or, ambiguously named, "Dungeon Generation" as done by *Rogue*. These methods create rather static and monotonous levels as the former could be recognized by keen-minded players, and the latter generating only sterile rectangular shapes that can be seen as rather dull. Another method for content generation is the use of Voronoi diagrams in level and mission generation. This method is a known technique[1] for the creation of organic game worlds (done by *Realm of the Mad God*), but has its flaws, especially when the details are observed. In this paper I discuss an improved technique using multiple Voronoi diagrams nested in one and other, making it easier to specify details and creating even more organic game worlds.

2 Previous and Related Work

As mentioned in the previous section, Amit Patel did a paper on creating game worlds with use of polygons[1]. These polygons are generated using a Voronoi diagram and Delaunay triangulation represented as a graph. He created a graph system that links all Voronoi and Delaunay properties to each other. For more on this system see section 3.3. To generate the Voronoi diagram I used an algorithm created by Steven Fortune[4][6] and converted to C++ by Shane O'Sullivan[5]. This algorithm uses a

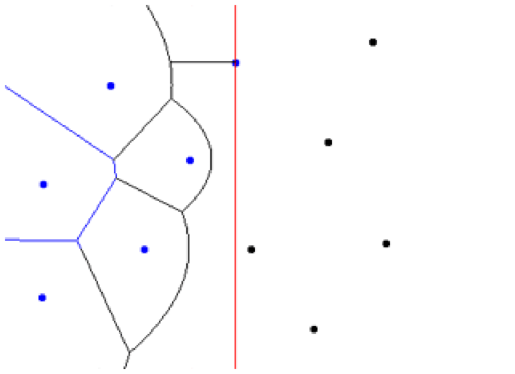


Figure 1: Still frame from the Fortune's Algorithm animated gif [6]

'sweep line', or a straight line (either horizontal or vertical) to determine what points belong to which seed point. See figure 1 for an example of this sweep line and I recommend you read [6] for a more detailed explanation on how it works.

3 Graph structures

3.1 Voronoi Diagrams

Voronoi diagrams are a way to partition space. Given a "random" sample of points an algorithm will create cells containing all points that are closer to the cell's seed point than all other seed points. In figure 2 you see an example of a Voronoi diagram. The small dots in the cells are the sample points from which the cells were defined. With the irregular shapes that this gives us, we need to do some other steps before we get a workable partitioning. There are a few ways to 'unify' the shapes generated, we opted for Lloyd' algorithm, also known as Voronoi iteration or - relaxation. This algorithm calculates the centres of all the cells and uses these as new sample points for a new diagram. In figure 3 there is a visualization of this process.

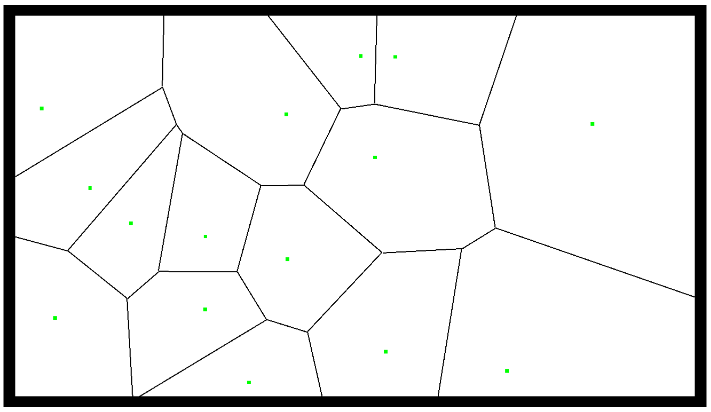
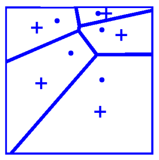
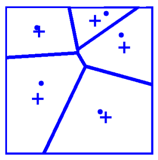


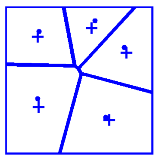
Figure 2: Voronoi diagram with 15 nodes.



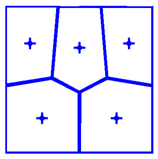
(a) Lloyd's algorithm first iteration



(b) Lloyd's algorithm second iteration



(c) Lloyd's algorithm third iteration



(d) Lloyd's algorithm fifteenth iteration

Figure 3: Lloyd's algorithm [2]

3.2 Delaunay Triangulation

The relation to the different Voronoi cells can be visually represented using a Delaunay triangulation. This results in an undirected graph which specifies neighbouring cells. In spatial partitioning this is of particular note, for with this information we can start to direct a path from one cell to another at any point in the entire game world. Thus, for example, being able to make certain cells inaccessible from specified directions, or make path-finding easier. Figure 4 shows a Voronoi tessellation with the relation of cells represented in a Delaunay triangulation. The red and black lines represent the Voronoi and Delaunay edges respectively. With the red dots being the corners for the Voronoi cells and the black dots the nodes for the Delaunay triangulation (as well as the seed nodes for the tessellation)

3.3 Connecting Graphs

In the computational representation of these we use three object types: *Centre, Corner and Edge*. The centre being the node of a Delaunay triangulation or the seed point of a Voronoi cell. Corners are exactly that, the corners of a Voronoi cell. These two have references to each other and others of their type that they are connected to. They are further connected by the final type; the edges. An edge is nothing more than a connection of corner to corner and centre to centre that convey both the Voronoi and Delaunay relation as shown in figure 5. The *A,B* and *1,2* represent the centres and corners respectively, with the lines representing their edges. The edge object holds a reference to these two groups of two nodes.

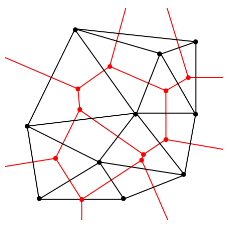


Figure 4: Voronoi diagram with a Delaunay triangulation overlay [3]

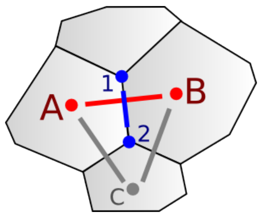


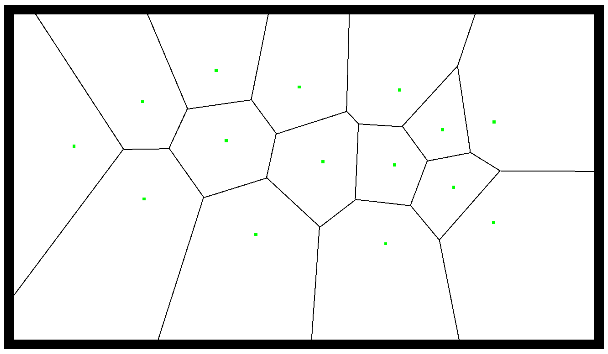
Figure 5: The relational graph[1]

4 Defining a game world

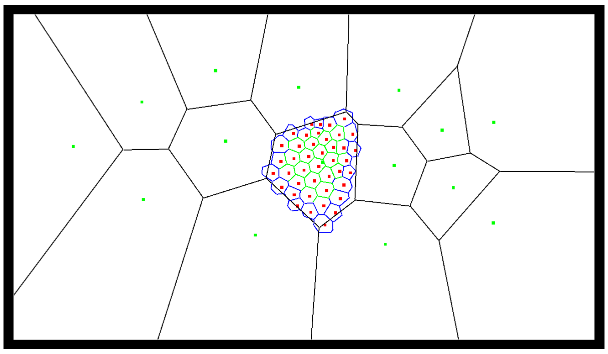
I did not want to use the diagram purely for the generation of landscapes or game-worlds. I wanted to define certain 'mission-spaces' in which players completes objectives and traverses to the next mission space where he/she finds one or more other players. This continues until all players have gathered in the last mission space. To create organic worlds, as well as the mission space in a more organic way I needed to nest two Voronoi diagrams. In figure 6a you see the generated world space. Each cell is one of the aforementioned mission spaces. These mission spaces get partitioned in another Voronoi diagram as seen in figure 6b. The outer cells are used to define borders between each other outer cell, whereas the inner cells define such things as objective location, terrain and vegetation.

4.1 Nesting Voronoi diagrams

There are several methods to nest these diagrams. The most obvious would be adding some random seed points within the outer cells and running the algorithm within the cell's bounds, but this has some drawbacks. As mentioned before, we used a sweep-line algorithm made by Steven Fortune [4]. This algorithm is optimized for rectangular planes as the sweep-line itself is fixed and following the bounds of a polygon requires quite a bit of calculation.

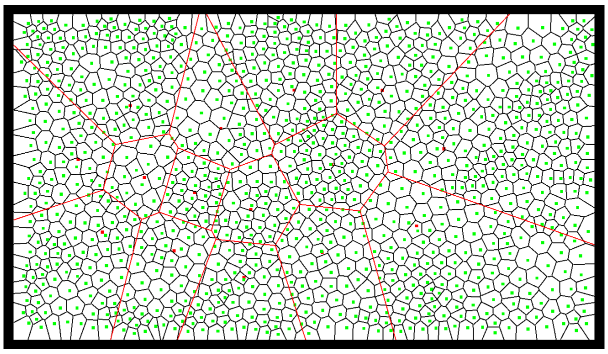


(a) Voronoi diagram

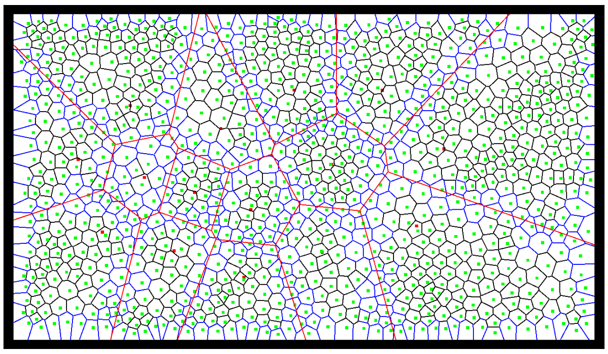


(b) Voronoi cell with children

Figure 6



(a) Voronoi diagram with inner and outer cells



(b) Voronoi diagram with inner and outer cells showing the border cells

Figure 7

We opted for a simpler method that did not require any modification to the algorithms and techniques we were already using. This method uses the same plane as the outer cells and adds more seed points to this plane. After the algorithm did its job, we assign inner cells to outer cells by their very Voronoi property, distance to seed point. In figure 7a you see the outer cells in red and the inner in black (with green seed points). In figure 7b and 6b you can see the results of the assignment. The blue cells indicate that they are on the border of their respective parent cell,

5 Finding possible paths

In our game we wanted different players to start in their own mission spaces and gradually team up after overcoming a challenge in a mission space. For this I decided on a binary structure as shown in figure 8. All nodes in this tree represent a mission space, where the leaf nodes represent the player's starting mission space. For this example I used eight players that team up after every mission. Thus player 1 teams up with 2 (team A) and 3 with 4 (team B) after that team A and B get together for the third node. To realize this structure we needed to find this tree (or similar, depending on the number of players) in the Delaunay graph of the mission spaces. This is a recursive brute

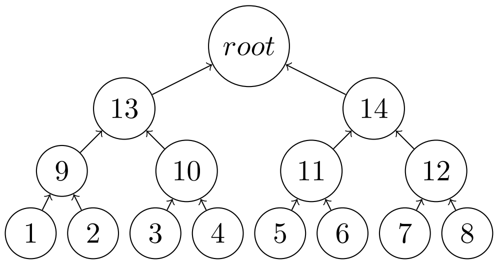
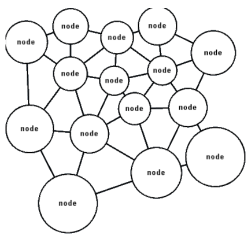


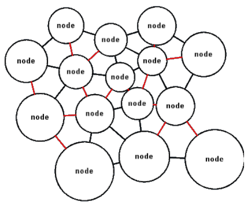
Figure 8: Binary tree where each node represents a mission space. The leaves represent the starting position of players.

force method that searches all possible paths for one that fits the description. See figure 9 for a good example of the outcome.

Due to the structures we used the Delaunay edges are interchangeable for the Voronoi edges. Running the brute force algorithm in figure 10a results in figure 10b.

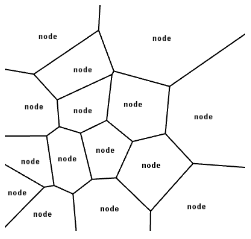


(a) Delaunay triangulation. (made with Ludoscope)

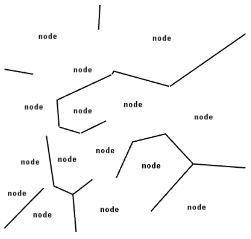


(b) Delaunay triangulation where the red edges are the edges of figure 8 that was found in this graph. (made with Ludoscope)

Figure 9



(a) Voronoi diagram. (made with Ludoscope)



(b) Voronoi diagram where the black lines represent physical borders, forcing the players to travel to the next space as dictated by the algorithm. (made with Ludoscope)

Figure 10

6 Conclusion

A single Voronoi diagram is a powerful tool for spatial partitioning and can create some amazing organic shapes. However, when the devil is in the details, just merely generating a lot of Voronoi cells make a world cluttered an disorganized. it can even give technical limitations with two, or more, seed points being to close to one and other. To solve this problem I tried to nest two diagrams in each other, creating a top and bottom layer. With these two structures put in place. We can define, not only the entire world with a organic feel, but give the different top layer cells different attributes such as its own climate. Creating a feel for depth and complexity that a single Voronoi diagram might lack.

References

[1] Amit Patel,
Polygonal Map Generation for Games,
Stanford University,
http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation

[2] Wikipedia Community,
Lloyd's Algorithm,
http://en.wikipedia.org/wiki/Lloyd's_algorithm

[3] Wikipedia Community,
Delaunay Triangulation,
http://en.wikipedia.org/wiki/Delaunay_triangulation

[4] Steven Fortune,
A sweepline [sic] *algorithm for Voronoi Diagrams*,
1986. ISBN 0-89791-194-6

[5] Shane O'Sullivan,
VoronoiDiagramGenerator,
(Fortune's algorithm in C++) http://mapviewer.skynet.ie/voronoi.html

[6] Wikipedia Community,
Fortune's Algorithm http://en.wikipedia.org/wiki/Fortune's_algorithm