Procedural Generation of 3D Maps
A Study of Polygon Graph based Map Generation

Marc Dangschat
FH Münster University of Applied Sciences
dangschat@fh-muenster.de

Yves-Noel Weweler
FH Münster University of Applied Sciences
y.weweler@fh-muenster.de

Abstract

The procedural generation of new content in video games is booming in the last years. There where even games whose biggest selling point was the amount of procedurally generated data. In this paper we elaborate about our use of an graph based map generation method that was used to generate archipelago type maps for a seafare and pirates 3D game. The map graph has been computed based on a random point distribution which was used to create a Voronoi diagram, which has been combined with its corresponding Delauny triangulation. We will explain how this graph data can later be used to compute islands and mountainsides and biomes and rivers and lakes. Based on this data, we place vegetation and NPCs and other game objects in appropriate places on the map. Following the detailed description on what methods we used for the different tasks, we provide some insight into what worked well in our approach and what aspects could be improved.

1 Introduction

Procedural generation is a collective term covering methods for algorithmically generating data. Generally meaning that the generated data was not directly specified in its final form by the developer, but rather was produced by the algorithm. Procedural generation is gaining popularity fast within the movie and video game industry. Content generation enabled the video game industry to minimize the work needed to create content. Algorithms support the artists or automate the creation of simple content completely. Thereby developers try to reduce the amount of data to be delivered to the customer as well as reducing development costs. Some modern games like Minecraft [12], No Man's Sky [10] or Dwarf Fortress [1] have little to no traditional content that was created by a designer. A substantial amount of their content is created by algorithms. Technology like this opens new prospects for content variety.

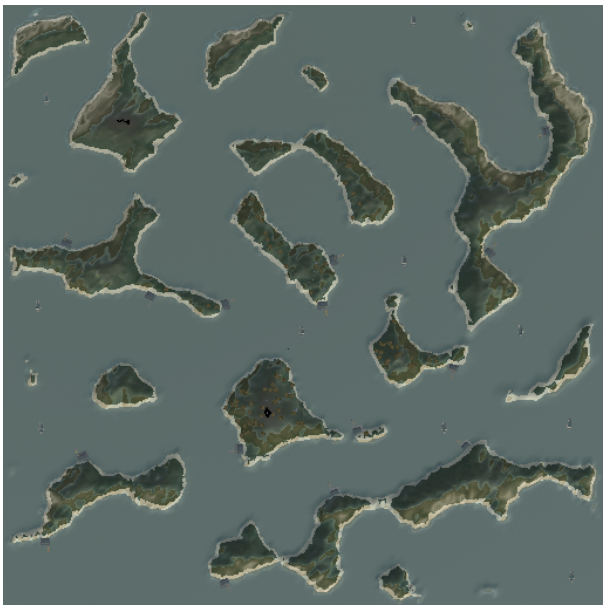


Figure 1: Example of a map that was generated using our implementation. The map is mapped with textures and populated with vegetation, props and interactive game objects.

For the same reasons procedural content generation is becoming more and more popular in the movie industry. Movies like Avatar [2] feature scenery which contents were generated to a substantial amount by computers. Algorithmically modeling hundred of plants and simulating the growth of entire forests [7]. Even the generation of textures, music and the behavior of artificial intelligence is more and more often influenced by procedural algorithms. The algorithms are generally designed to reproducible generate apparently random content. Given the same initial state, the algorithm produces the same content, every time.

We describe our experience and implementation of the 3D map generation technique using polygon graphs introduced by Patel in [11]. Our implementation generates a complete 3D map including automated texture mapping and distribution of vegetation, props and objects used for gameplay interaction, as shown in Figure 1. The map is generated by connecting the graph representation of a Delauny triangulation with the corresponding dual graph of the Voronoi diagram. This graph is refined using Lloyd relaxation and traversed afterwards to assign parameters like elevation, moisture and other information to nodes and edges of the graph. The graph is converted into a polygon mesh, mapped with textures and populated with vegetation, props and objects for the player to interact with. An overview of the implemented approach is described in section 3.

The main contribution of our work is:

- Created an alternative point of view on the polygon graph based map generation approach originally described by Patel in [11].

2 Related Works

In our work we investigate previous work about polygon graph based map generation, in particular procedural content generation for video games. We follow the graph based map generation method described by Patel in [11]. To implement a rough game prototype, we extend the approach by techniques for procedurally assigning terrain textures as well as placing foliage, props, and objects for direct player interaction. Using a graph structure to generate the world representation allows for simultaneous generation of gameplay related contents. Physical world topology and map meta-information used for enforcing gameplay constrains are combined in one representation.

In our implementation, we realize the generation of rivers and watersheds by traversing the graph edges from nodes with low elevation, shoreward up to nodes with higher elevation in proximity to mountains. Other techniques use regular height fields or parameterized curves to describe rivers. Different alternative techniques like [8] employ Voronoi cells like we do, but reverse the generation process. They first generate rivers represented by a directed acyclic graph. After that, they decompose the terrain in into a set of patches by using the river nodes as center points and computing the corresponding Voronoi cells. In our implementation rivers do not flow through the Delaunay nodes, but instead flow along the Voronoi edges.

For map mesh rendering, we transform elevation data from the graph based representation into a heightmap. Alternative methods for generating feasible height fields mostly rely on fractals and noise, for example Brownian surfaces [13], Random Midpoint Displacement Fractals [4] or the popular Simplex Noise method [5]. For realism these methods are sometimes combined with physical erosion simulation to generate realistic-appearing terrain.

An alternative approach used for 3D map generation is the altering of volumetric terrain representations, mostly in the form of voxels [6]. The volumetric generation often allows for representation of more diverse terrain formation, for example overhanging rocks or caves. The approaches employed for generation are based upon the before mentioned fractals and noise functions, but are transferred to higher dimensional spaces.

3 Approach

Figure 2 shows an overview of the pipeline for the generation of 3D maps using polygon graphs. A set of random points in a plane with a Gaussian distribution is produced. A Voronoi polygon is calculated from the random points. Several Lloyd relaxation iterations are made on the Voronoi polygon depending on how evenly distributed the single polygons cells should be for map generation. Note that after a certain amount of relaxations, the polygon cells do not change their appearance anymore and remain fixed. After a Voronoi polygon is obtained, the basic map generation is performed. This includes definition of the basic terrain, distribution of terrain elevation, lakes, rivers, moisture and assignment of a biome. After the graph based map representation is populated, we generate a 3D map mesh and slice it into chunks used for rendering. In a second game specific generation step we apply textures and generate vegetation as well as props based on the terrain profile. Aside from that we generate further game specific content like harbors or enemy encounters.

4 Map Representation

The map is represented using two graphs that are linked together. The first graph represents the Delaunay triangulation of a set of points. It holds one node for each randomly placed point and edges between adjacent points. The second graph represents Voronoi polygons, where each node is a polygon corner and the edges connect the corners. A visual presentation of the graph structure can be seen in Figure 3. Inside every polygon

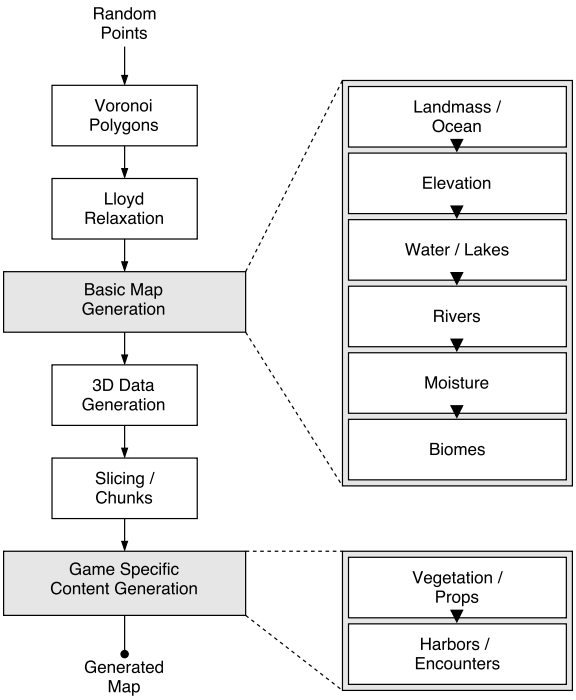


Figure 2: System overview for 3D map generation using polygon graphs.

of the Voronoi graph, there is a corresponding node (polygon center) in the Delaunay graph. Contrary, inside every triangle of the Delaunay graph, there is a corresponding node in the Voronoi graph (polygon corner). As shown in Figure 3 the edges of both graphs are also connected. For every edge connecting two nodes of the Voronoi graph (polygon corners), there is always a corresponding edge connecting two nodes (polygon centers) in the Delaunay triangulation and vice versa.

A larger example of this connected graph structure can be seen in Figure 4. With the Delaunay nodes (polygon centers) in red and the Voronoi nodes (polygon corners) in blue. The Voronoi edges are visualized in blue and the Delaunay edges in red.
When we generate a map, we first build a single connected graph data structure that combines the Delauny and the Voronoi graphs. Within this graph nodes and edges can store different parameters that will be populated in the following steps.

A Delaunay node, for example, contains data about the adequate biome, its elevation and moisture as well as several flags to identify its affiliation to landmass, coast, ocean or water in general.

```
DelaunayNode {
```

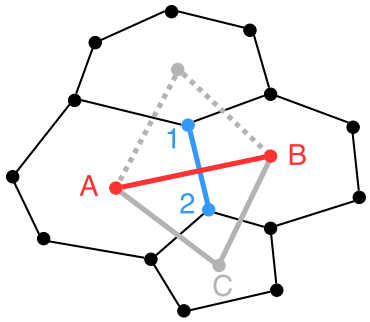


Figure 3: Visualization of the graph interconnection. *A* and *B* are nodes of the Delaunay graph, 1 and 2 are nodes of the Voronoi graph. The Triangle *ABC* represents a Delaunay triangle.

```
  Biome biome,
  float elevation,
  float moisture,
  bool flags: IsBorder, IsCoast, IsOcean, IsWater,
  ...
}
```

Contents of a Voronoi node are similar to a Delaunay node. But since biomes are assigned to a polygon as a whole, single polygon corners do not store biome information. Since river calculation is done on the Voronoi graph, they store adjacent nodes with the highest downslope and the amount of rivers which pass the node.

```
VoronoiNode {
  VoronoiNode downslopeNode,
  float elevation,
  float moisture,
  bool flags: IsBorder, IsCoast, IsOcean, IsWater,
  int rivers,
  ...
}
```

Edges, however, are mainly used for river generation and store only the amount of rivers that flows through them.

```
Edge {
  int rivers,
  ...
}
```

5 Map Generation

5.1 Landmass

The first step in generating the map is a basic binary assignment of either landmass or water to each Voronoi

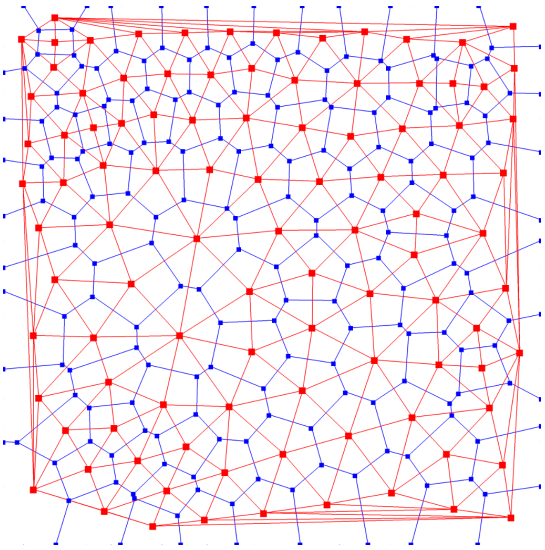


Figure 4: Example of a larger graph. Delaunay nodes (polygon centers) in red, Voronoi nodes (polygon corners) in blue. Voronoi edges are visualized in blue and the Delaunay edges in red.

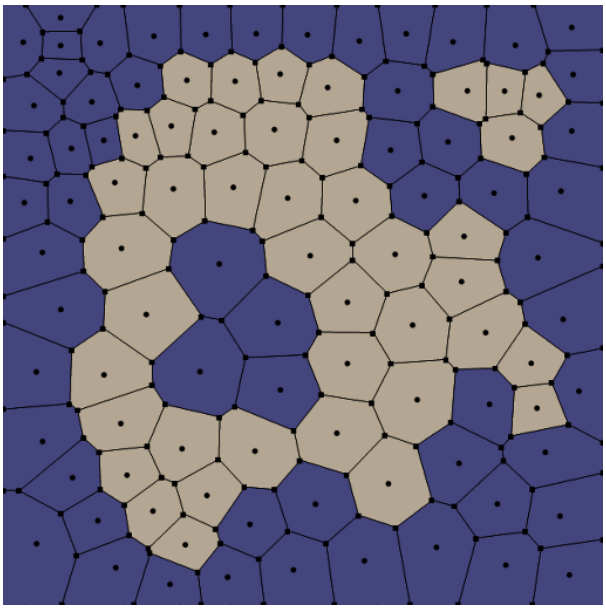


Figure 5: Assignment of each Voronoi polygon to either landmass or water.

polygon. This assignment determines the general shape of a generated world. After this categorization the graph structure is traversed to mark all nodes and edges where land and water meet as coastline. A visualization of this step can be seen in Figure 5.

Determining which polygons should be landmass and which one should be water can be done by evaluating a function L (Equation 1). It takes a polygon corner p as an input and outputs if it's landmass or water.

$$L(p) = \begin{cases} 0 & \text{if polygon is water} \\ 1 & \text{if polygon is landmass} \end{cases} \tag{1}$$

What function to use depends on the desired results. It can for example be based on a user defined mask, noise, fractals or even something as simple as the distance to the maps center point. In our implementation, we wanted to generate a set of small random islands. For that we decided to use Perlin noise, since it offered the possibility to generate two dimensional pseudo random noise whose behavior can be controlled. After a parameter study, we came out with Equation 2, where p_x, p_y are the x and y components of the corner point p and o_x, o_y are a random component displacement. A visual representation of how this function transforms noise into islands can be seen in Figure 6.

All polygons adjacent to the map borders, are marked as water automatically. This ensures, not only a greater freedom of movement for a player on the map, but is also useful in later stages when working with lakes or rivers for example.

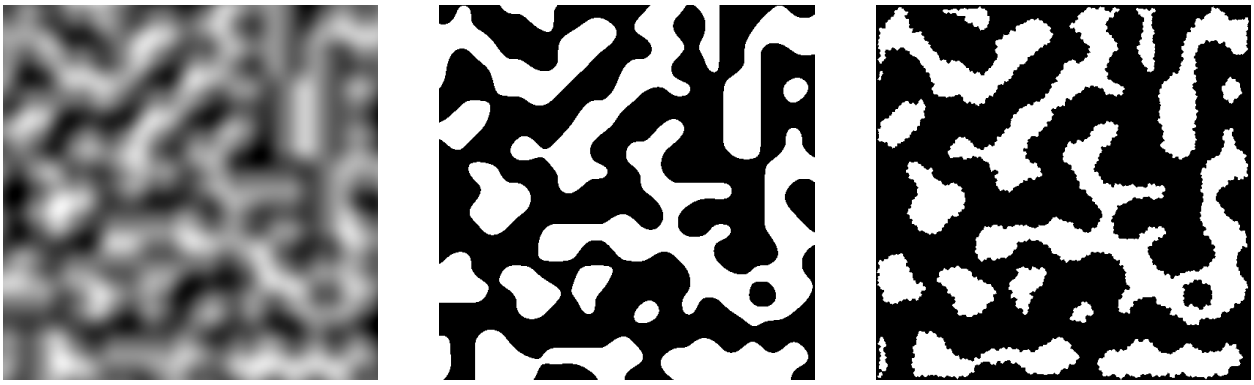
To generate the Perlin noise, we use an octave count of 1 and a frequency of 9 Hz. Perlin noise can be seen as the sum of several noise functions with increasing frequencies and decreasing amplitudes. Each of these added frequency layers is called an octave, since it has double the frequency of the previous layer. Therefore, Perlin noise generated with higher numbers of octaves contains increasingly more high frequency components. The effect of an increasing octave count can be seen in Figure 7. An octave count of one combined with a low frequency of 9 Hz is sufficient for our purposes since the resolution of our Voronoi polygons is very coarse.

$$L(p) = \begin{cases} 0 & \text{if } \mathscr{A}(p) \leq \frac{1}{2} \\ 1 & \text{if } \mathscr{A}(p) > \frac{1}{2} \end{cases} \tag{2}$$

$$\text{with } \mathscr{A}(p) = Perlin(\frac{p_x + o_x}{9}, \frac{p_y + o_y}{9})$$

5.2 Lakes

Lakes can be defined as patches of water polygons that are fully encapsulated by landmass. Ocean on the other hand is water with an adjacent ocean polygon or water connected to the map borders. The lake generation is implemented by traversing the graph structure starting from the map border. Since all polygons adjacent to the map border are marked as ocean, we start with them. We



(a) Unmodified Perlin noise (b) Applyed > 0.5 threshold (c) Function evaluation for all polygons

Figure 6: Evaluation process of the Perlin noise function for landmass assignment.
(a) Output of the Perlin noise function before thresholding.
(b) Perlin noise after thresholding each point with the $^1/_2$ boundary.
(c) Assignment of Voronoi polygons to either landmass or water by evaluating the noise function for each polygon center point.

repetitively mark neighbor polygons as ocean when they are water and not landmass. A visual example of this process can be seen in Figure 8. As one can see water polygons fully encapsulated by landmass result in lakes. Water polygons connected to ocean are marked as ocean themselves.

With our implementation we solely rely on the previously generated landmass to have patches of water polygons encapsulated inside it. This approach can be extended to insert additional lakes into the world. For example, one could build a second noise function with a higher frequency and an appropriate threshold to create small patches. Depending on its representation this function could be subtracted or multiplied to the basic landmass function from Equation 1 to mark the small patches as water.

5.3 Elevation

Just like Patel did in [11], we generate elevation data after shaping the landmass. So instead of creating elevation data and a planar water level to define coastal regions we generate elevation afterwards.

The elevation algorithm has to satisfy the following constraints:

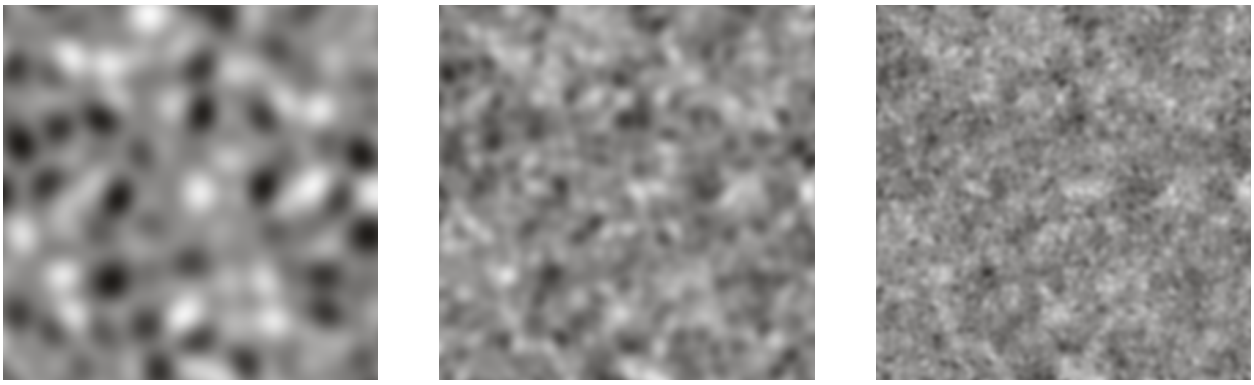
- Strictly monotonically decreasing elevation from the map center to the map borders or in the case of islands, strictly monotonically decreasing elevation from the island center points.
- Near planar elevation within lakes.

With a strictly monotonically decreasing elevation function we circumvent local minima that would make the later river generation much harder. With an monotonically decreasing terrain altitude we ensure that rivers always flow downhill towards the ocean. A nearly planar terrain altitude inside lakes is needed to ensure that a planar water surface can be rendered later. Otherwise, a lake surface could for example end up on a steep hillside.

We calculate terrain altitude for all corners of a polygon. We start at the world borders with the smallest possible elevation and recursively traverse adjacent graph nodes from there to the map center. The elevation of nodes adjacent to the currently processed node is increased by a small factor ε regardless if it is water or landmass. However, if an adjacent node is landmass we increase its elevation by an additional constant factor of 1. After traversing the graph, elevation is redistributed and normalized. Generated terrain altitude can be seen in Figure 9.

Redistribution

The described generation of elevation data tends to result in high elevation in the center of a Pangaea type single landmass or high elevation for each island center in an archipelago type map. This goes along with steep slopes in direction of the surrounding ocean. This method favors islands with high mountains and steep slopes over flat islands. To counteract this, the overall distribution



(a) Octaves: 1 (b) Octaves: 2 (c) Octaves: 3

Figure 7: Visualization of octaves in the generation of two dimensional Perlin noise. The frequency components double with each octave. Noise generated with higher numbers of octaves contains increasingly more high frequency components.

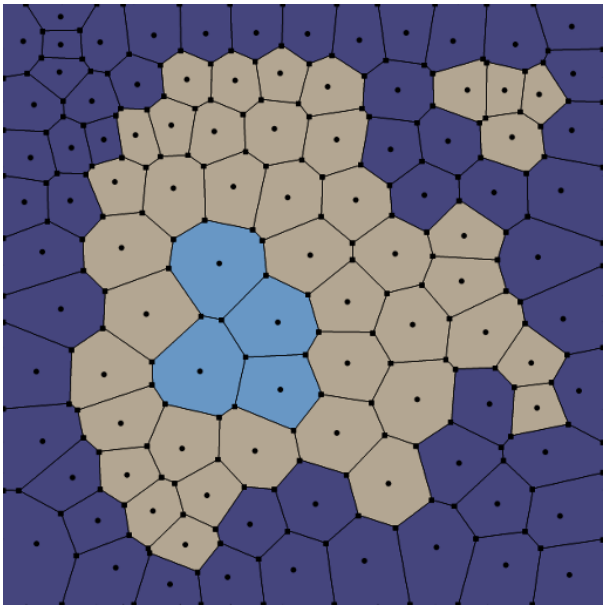


Figure 8: Voronoi polygons encapsulated by landmass are marked as lakes.

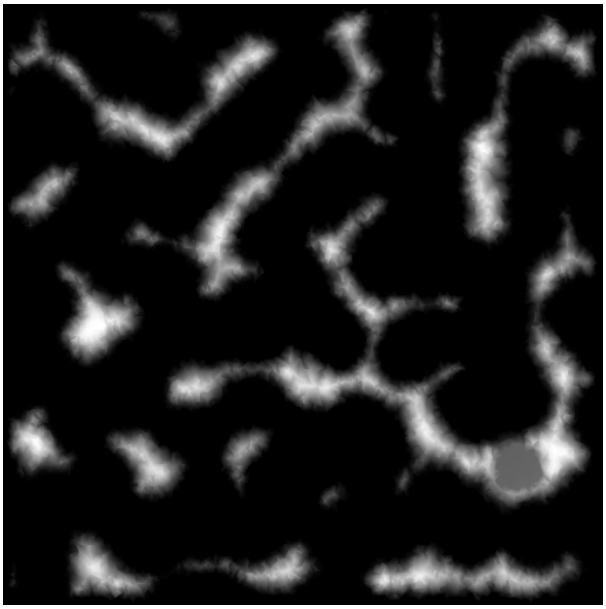


Figure 9: Visualization of elevation data generated by our algorithm. Terrain altitude is strictly monotonically decreasing from the island center to the coast. Water, especially lakes have a near planar elevation.

of elevation can be changed, so that lower elevations are more common than higher ones. Such a redistribution has to preserve the elevation constraints formulated earlier.

Lets assume a terrain with a cumulative elevation distribution function $F_X(x) = x^2$ as shown by the red plot in Figure 10. This cumulative distribution function takes an elevation value x and outputs what amount of the terrain area has an elevation value $\leq x$. With a given terrain, one can easily calculate how much area of the map has an elevation $\leq x$. Assumed one would like to redistribute elevation to follow a cumulative distribution function $Y_X(x) = 1-(1-x)^2$. Then the function can be adapted to calculate new elevation values as shown in Equations 3 – 5.

$$\begin{aligned} Y_X(x) &= 1-(1-x)^2 \\ \Rightarrow Y_X(x) &= 2x-x^2 \\ \Leftrightarrow x^2-2x+Y_X(x) &= 0 \end{aligned} \tag{3}$$

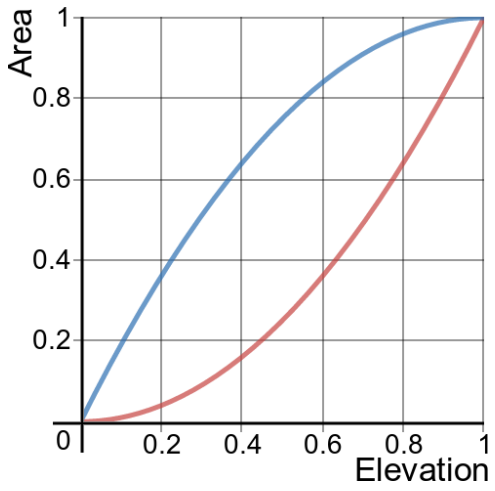


Figure 10: Visualization of elevation redistribution. Elevation data with a cumulative example distribution function $\bullet F_X(x) = x^2$ is redistributed to follow $\bullet Y_X(x) = 1-(1-x)^2$.

Solving Equation 3 for x as show in Equation 4 allows to calculate the new redistributed elevation values based on the existing distribution $F_X(x)$.

$$\begin{aligned} x_{1,2} &= \frac{2 \pm \sqrt{4-4F_X(x)}}{2} \\ &= 1 \pm \sqrt{1-F_X(x)} \end{aligned} \tag{4}$$

Since $x, F_X(x) \in [0,1] \Rightarrow \sqrt{1-F_X(x)} \in [0,1]$. Therefore

$$\begin{aligned} & x_1 = 1+\sqrt{1-F_X(x)} \notin [0,1] \quad \lightning \\ \wedge \quad & x_2 = 1-\sqrt{1-F_X(x)} \in [0,1] \end{aligned} \tag{5}$$

Thus new elevation values for a redistributed terrain can be calculated by $r(x) = 1-\sqrt{1-F_X(x)}$. A graphical representation of the new elevation scaling can be seen in Figure 10.

5.4 Rivers

Generated rivers flow downhill until they reach the ocean. The strictly monotonically decreasing elevation constraint, described in subsection 5.3, ensures that the calculation of the river pathway through the terrain graph is not being complicated by local minima. Rivers are also allowed to enter and leave lakes on their path downhill as well as to merge with other rivers.

Our implementation chooses random polygon corners inside the landmass as source. From there we continue to follow the edges with the steepest slope until we reach the ocean or a maximum river length. There is a counter for each edge, that tracks how many rivers go through. This can be used in the rendering process, to draw different variations of rivers depending on the amount of water flowing through. Calculating rivers on the nodes of the Delaunay triangulation would also be feasible. However, since the river would pass a lot less nodes that can bend its path this would produce irregular rivers.

In our implementation however, we had problems using the generated rivers in a 3D map where they are looked upon from a close distance. Visual errors were particularly noticeable. Rivers can follow the general slope of the terrain, but the water has to be planar. Water would therefore float over the terrain. And unlike in reality where rivers bend through the terrain and have round edges, our rivers look very angular. To generate more realistic looking rivers that can be used in a 3D environment where they can be observed from a close distance, some improvements are necessary. To improve river presentation one could fit a B-spline through all nodes a river passes. Figure 11 shows an angular river compared to a B-spline based river. As indicated by the green circle in Figure 11 even using B-splines can lead to artifacts. This is especially noticeable when using sparsely populated graphs. There the B-spline can divert from the polygon slope resulting in partly ascending rivers.

Furthermore to prevent hovering rivers, riverbeds should be carved out of the terrain mesh. The water effect can then be embedded into the riverbed.

5.5 Moisture

Moisture generation follows the idea of natural evaporation of water from the ocean, lakes and rivers. This influences the biome generation, vegetation placement and also texture mapping. Arid areas can be rendered with less vegetation and different ground textures as opposed to humid regions.

Our general approach for generating moisture is as follows:

1. Initialize every corner with a moisture of 0.
2. Corners touched by rivers emit moisture proportional to the amount of water going through.
3. Lake corners emit a moisture of 1.

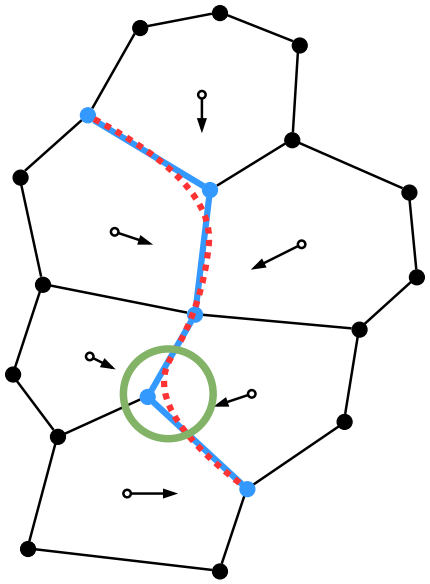


Figure 11: Comparison between angular edge based rivers and B-spline based rivers. The blue edges represent the angular river model. The red dotted line represents the B-spline river model. The green circle marks a position where the B-spline diverts from the polygon slope indicated by the arrows.

4. Distribute moisture with a penalty of 0.9 to all adjacent non water corners.

5. Set moisture for all ocean and coast corners to a constant moisture of 1.

6. Global moisture redistribution to enforce numeric range of $[0,1]$.

7. Apply moisture values to the polygon centers.

Consequently, moisture generation is calculated based on a distance metric. The moisture of a landmass corner is calculated by $m_{new} = m_{dist} 0.9^k$, where m_{dist} is the moisture of a distant corner and k is the distance to that corner. Where k is the amount of nodes passed when traversing between those corners.

Ocean moisture is inserted after moisture from rivers and lakes has been distributed. In case the ocean moisture would be inserted together with moisture from rivers and lakes, distribution to adjacent corners would produce a very homogeneous moisture coverage throughout the landmass. Especially on smaller islands, this would lead to a uniform moisture distribution throughout the islands.

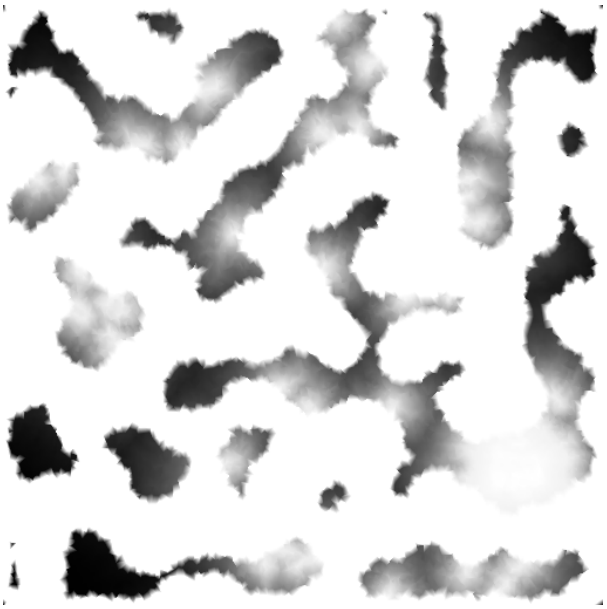


Figure 12: Visualization of moisture data generated by our algorithm. Lakes emit moisture to the surrounding terrain. Oceans do not emit moisture to terrain.

Redistribution

To work with moisture more easily we expect it to always evenly cover a numeric range from 0 to 1. We redistribute them by sorting all corners by moisture and assigning new moisture based on the corners position in the sorted list. Later calculations can then rely on normed moisture values within that range.

5.6 Biomes

We implemented biome generation like the one described in [11]. We use a modified Whittaker diagram [3] shown in Figure 13 to determine a biome for each Voronoi polygon.

Elevation and moisture are used to select biomes from the diagram. To keep selection simple, terrain elevation is used in exchange for temperature. This approach can be used to cover the high mountain peaks far away from the coast in snow or tundra. For each polygon we sample moisture and elevation at its center by taking the average of the polygon corners elevation and moisture. These values are used for the biome lookup. There are a few exceptions to this procedure for oceans, lakes, ice, beach and marsh. Polygons connected to the map borders are always assigned ocean biome. Water not connected to ocean is assigned a lake biome, whereas lakes at high elevations freeze up and get ice. Contrary lakes at very low elevations are assigned marsh. Another special case is the coast biome that is assigned to every polygon

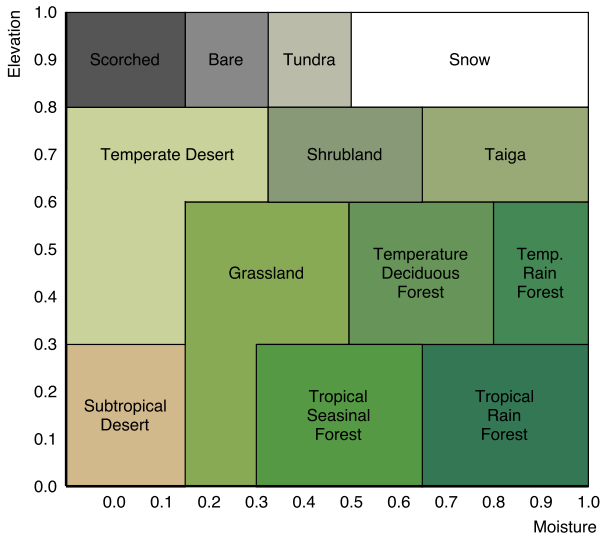


Figure 13: Whittaker diagram used for biome selection. Parameters used to select a biome from the diagram are moisture and terrain elevation acting as temperature.

neighboring ocean polygons. Exemplary biome assignment can be seen in Figure 14 where each polygon is pigmented with a biome from the Whittaker diagram in Figure 13.

More realistic approaches of biome placement could involve further terrain parameters to lookup biomes. A realistic temperature metric instead a of terrain elevation, latitude and solar radiation for example. Further improvements on moisture generation like wind, clouds and rain can also be used to refine biomes.

5.6.1 Textures

Texture generation for the terrain surface is simple. The final terrain texture is a linear combination of sand, grass, dirt and rock textures. Their influence in the final texture depends on elevation, surface normals, slopes and moisture.

Every terrain position below water or below a constant elevation marking the beach level is assigned a sand texture. This forms the beaches as well as the underwater grounds of lakes and oceans. The terrain above the beaches can be made of grass, dirt and rock. The influence of grass grows linear to the amount of moisture. The dirt texture is assigned based on elevation and moisture. The influence of dirt increases linear with growing altitude and decreases with rising moisture, respectively. The contribution of rock texture to the final texture is dependent on the terrains slope and elevation. Very high altitudes and slopes are covered by rock to create mountain peaks and cliff like ground appearance.

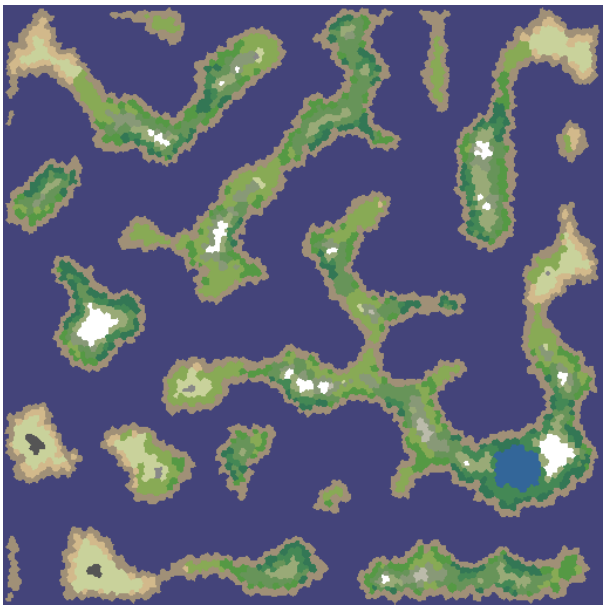


Figure 14: Biomes are assigned to the terrain based on elevation and moisture. Each polygon has a single biome associated.

5.7 Vegetation

With our implementation vegetation placement includes not only trees and grass but also other objects like stones.

5.7.1 Plants

Plant placement is rather simplistic. The general approach can be described as follows:

1. Choose random point on landmass.
2. Choose random plant type for the biome.
3. Based on elevation check if a plant can grow.
4. Based on surface normal check it the terrain is to steep to grow plant.
5. Based on the previous texture mapping check if the underlying soil is suitable to grow a plant.

Our plant placement procedure selects random points on the landmass to place plants at. The tree type is randomly selected according to the biome at that point. We then check certain parameters of the terrain surface that have to be fulfilled in order to grow a plant. Elevation as well as terrain steepness and soil composition have to be suitable for the plant. The slope check prevents trees from growing on terrain formations like cliffs. Soil

composition is checked based on the previously done texture mapping. Textures like rock, sand or gravel that are related to a biome control were plants can grow. Thereby we prevent them from growing on unsuitable surfaces like rock or gravel.

Further improvements to generate a more natural looking plant distribution can be made. Techniques like [9] try to mimic their distribution with seeds. Most plants grow from seeds that spread from other plants. Depending on where the seeds land on the ground, parameters like moisture and soil composition determine if the seeds can grow. At the beginning randomly place the initial population on fitting terrain. After some iterations of new plant growth and the decay of old vegetation, a realistic plant distribution can be achieved.

5.7.2 Props

In our implementation we place stones as non intractable objects for visual enhancement. To position them we select a certain number of random points on the terrain that aren't occupied by other objects and place a stone variation. Placement probability is influenced by the ground texture and terrain slope. Surfaces with a rock or gravel texture are more likely to spawn stones. Larger stones are more likely to spawn on plains whereas smaller ones are more likely to be placed on steeper slopes.

5.8 Objects

In our game, the player needs interactable objects like harbors and enemy ships placed throughout the world.

5.8.1 Harbors

Harbors should be distributed along the coast of islands for the player to interact with. Small islands should host few or even no harbors whereas larger islands are able to support multiple harbors. For that we calculate the area of every island and normalize it by dividing it by the overall landmass area. First, we check if an island is large enough to host harbors by checking if its area is above a threshold. If the island is large enough we calculate how many harbors should be placed. Their quantity is defined by multiplication of the island area with a scalar defining how many harbors per unit of area are suitable. To place the calculated amount of harbors first we get a list of points forming the islands coastline. We then iteratively select a random point from the coast and check if it's not too close to an existing harbor on the island. If it's too close, we select another random point and try again.

5.8.2 Enemies

In our game the player can engage with enemy ships. These encounters should be placed on a free spot in the open sea, so that they don't interfere with harbors or other ships. To locate suitable spawn locations, we randomly select points from the ocean and verify that they are suitable spawning points. To ensure that a selected point isn't too close to the coast we calculate its distance to each island. This is realized by extracting the outline nodes of the polygon meshes forming islands and calculating distance to the selected spawn point. To prevent a new enemy from spawning too close to existing ones, we calculate the distance to all other spawn points and discard it if it's too close.

6 Results

Results of our implementation can be seen on the last page of this paper. Examples for different generated maps from areal perspective are shown in Figure 15. Close-up views of the generated maps are illustrated in Figure 16.

7 Discussion

Over the course of implementation we faced various limitations and encountered the strengths of this approach. The process is particularly versatile when it comes to control over the generated terrain. By manipulating individual stages, a large variety of different map types can be generated. Maps containing continents, islands or only landmass can be created and furthermore can be combined. For example, by exchanging the elevation model the approach can be used to generate underwater worlds. This versatility is also noticeable when it comes to the possible forms of visual representation styles. Just by altering the distribution of the points used in the generation it's possible to enforce different visual styles. Positioning the points on a hexagonal grid leads to a terrain composed of hexagonal shaped cells. Placing them in a raster, leads to a map composed out of square cells. These different visual representations make the approach suitable for a large range of games. All data used and produced during the generation is stored in a graph structure. As a consequence a wide range of generation steps is expressed by general graph traversing techniques making them easy to alter and extend. This is especially true for the creation of rivers.

However, depending on the intended purpose of the generated maps there are some weaknesses. When generating extremely large maps, the approach doesn't scale very well. Counteracting this by generating smaller portions of a large map is also difficult since the genera-

tion requires the graph structure as a whole. In our implementation we therefore generated the map as a single huge structure only slicing it into chunks for later rendering. For larger maps we also ran into problems with the creation and visualization of the map mesh. The generated mesh reached an amount of vertices where it couldn't be rendered as a whole. If super large maps are needed the mesh generation process has to divide the map into several smaller sub-meshes. When using the generated maps in situations where they are looked at from close distances, we recommend additional improvements for rivers. To improve visual quality and realism even further, efforts to excavate actual riverbeds should be taken. In addition to that the course of rivers could be smoothened by using B-Splines rather than flowing down the angular graph edges.

As long as one is aware of these limitations the implemented approach ultimately delivers a highly versatile way of generating maps for various applications.

References

[1] Tarn Adams and Zach Adams. *Dwarf Fortress*. Bay 12 Games. Aug. 8, 2006. URL: http://www.bay12games.com/dwarves/ (visited on 03/10/2017).

[2] James Cameron. *Avatar*. Twentieth Century Fox. Dec. 17, 2009. URL: http://www.avatarmovie.com/index.html (visited on 03/10/2017).

[3] Different. *Biome*. Apr. 15, 2017. URL: https://en.wikipedia.org/wiki/Biome#Classifications (visited on 04/25/2017).

[4] Different. *Diamond-square algorithm*. Mar. 4, 2017. URL: https://en.wikipedia.org/wiki/Diamond-square_algorithm (visited on 03/10/2017).

[5] Different. *Simplex Noise*. Jan. 7, 2017. URL: https://en.wikipedia.org/wiki/Simplex_noise (visited on 03/10/2017).

[6] Different. *Voxel*. Feb. 16, 2017. URL: https://en.wikipedia.org/wiki/Voxel (visited on 03/10/2017).

[7] Renee Dunlop. *Avatar*. Jan. 14, 2010. URL: http://www.cgsociety.org/index.php/CGSFeatures/FeaturePrintable/avatar (visited on 03/10/2017).

[8] Jean-David Génevaux et al. "Terrain generation using procedural models based on hydrology". In: *ACM Transactions on Graphics (TOG)* 32.4 (2013), p. 143.

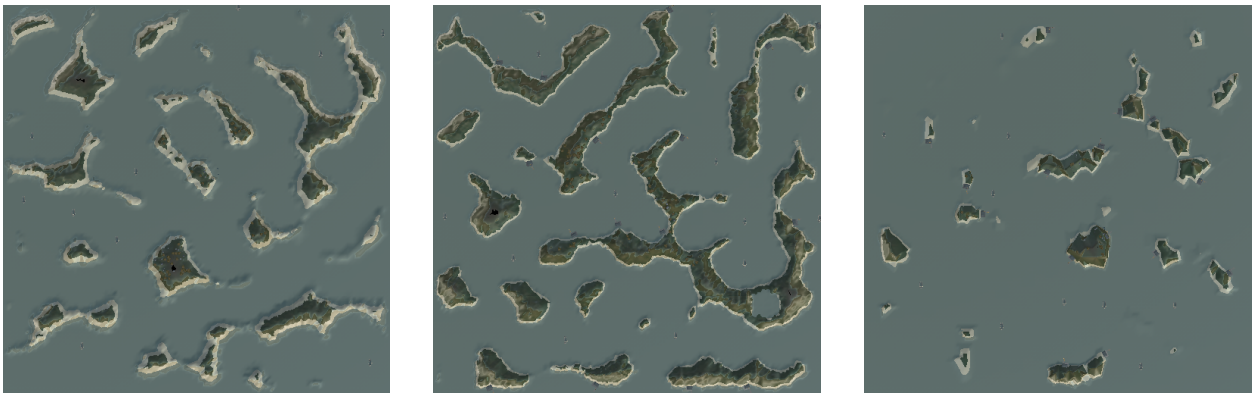
[9] Brendan Lane, Przemyslaw Prusinkiewicz, et al. "Generating spatial distributions for multilevel models of plant communities". In: *Graphics Interface*. Vol. 2002. Citeseer. 2002, pp. 69–87.

[10] Sean Murray et al. *NO MAN'S SKY*. Hello Games. Aug. 9, 2016. URL: http://no-mans-sky.com (visited on 03/10/2017).

[11] Amit Patel. "Polygonal Map Generation for Games". In: (Sept. 4, 2010). URL: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/ (visited on 03/09/2017).

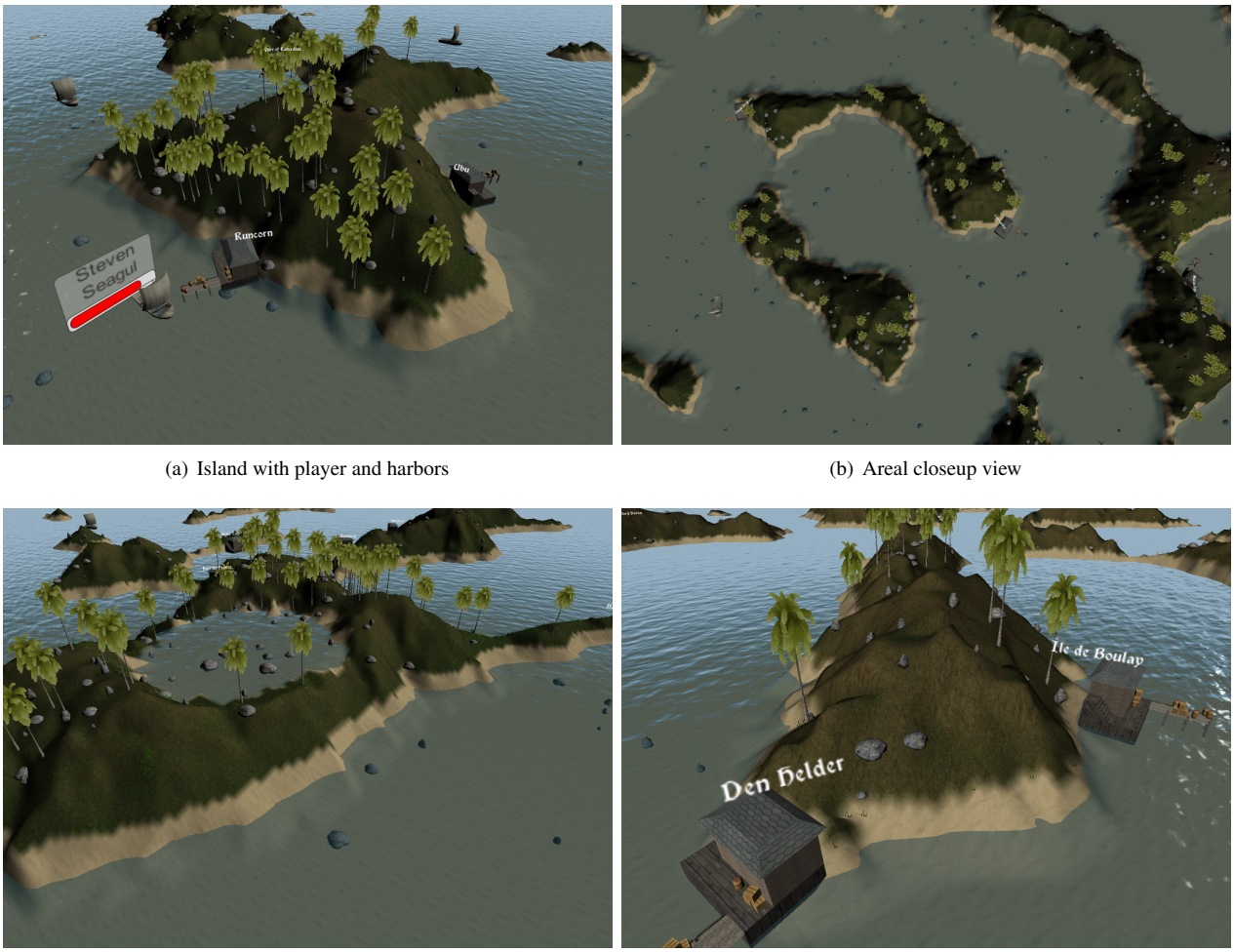
[12] Markus Persson and Jens Bergensten. *Minecraft*. Microsoft Studios. May 17, 2009. URL: https://minecraft.net/en-us/ (visited on 03/10/2017).

[13] Tamas Vicsek. *Fractal Growth Phenomena*. World Scientific Pub. Co. Inc., Oct. 28, 1991. ISBN: 978-9810206680.



(a) World 1 (b) World 2 (c) World 3

Figure 15: Examples of three different generated maps from aerial perspective.



(a) Island with player and harbors (b) Areal closeup view

(c) Island with a lake (d) Dry island with harbors

Figure 16: Examples of different maps from close range when flying through the maps in first person mode.