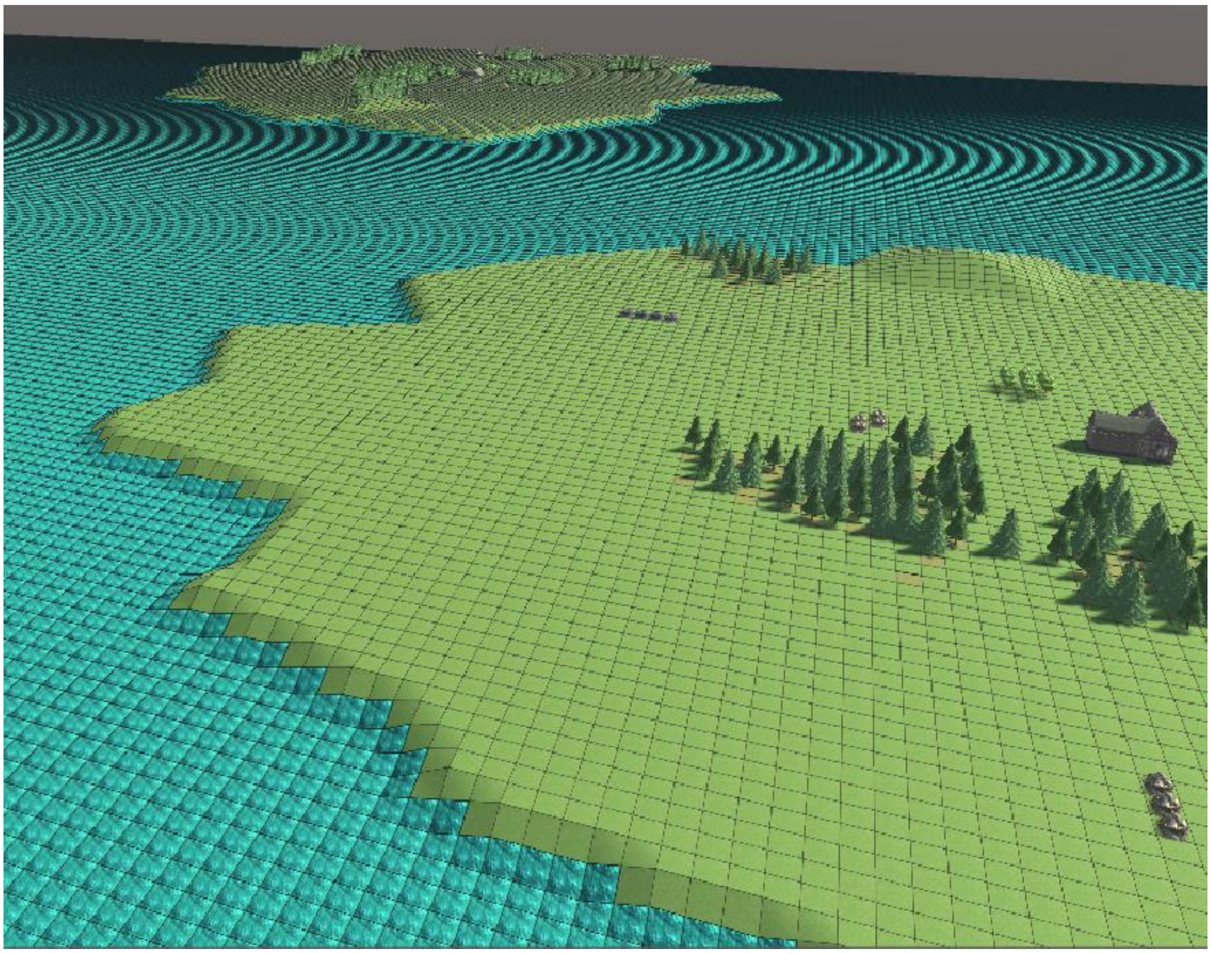
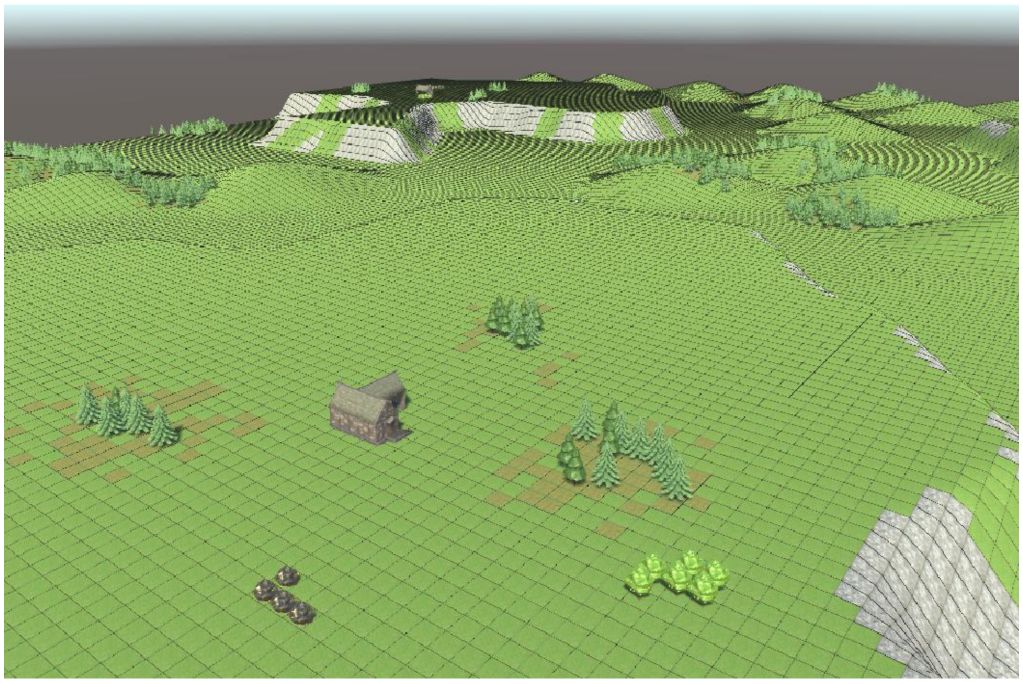
Random Map Script

framework for Unity





Michael Marchesan – 945887

Table of Contents

Objective and features

The objective of the project is to produce a framework to allow the creation of RTS-style random maps through game-specific Random Map Scripts (from now on RMS).

The framework itself tries to be as much game-agnostic as possible, supplying classes which can be inherited from in order to add game-specific features to the system.

Inspiration

The main sources of inspiration for both how the RMS is laid out and what results it produces, were games out of the Age of Empires franchise; in particular Age of Empires II and Age of Mythology. Some more complex features can still be added by game-specific code.

Gameplay considerations have been made based on recent E-Sports tournaments (Red Bull Wololo), order to add instruments that allow the creation of balanced maps, while still trying to not be too much limiting

Difference in approach

While I was taking inspiration from the Age of Empires franchise, I've also willingly decided to take a hugely different step early on. AoE's random map scripts rely on the script defining reference points, and then generating "regions" from those reference points. This allows for a lot of flexibility, but leaves most of the load to the script-writer. Almost nothing is automatized. As a result, something simple like "make all the islands contain the same amount of forests" becomes extremely difficult.

In my RMS Framework, the map is automatically split into more or less even regions, through a Voronoi diagram generated starting from points which are generated by Poisson disc sampling.

The map script defines the density of the sampling, and then has to work with the regions generated by the Voronoi diagram.

This makes achieving the mentioned "make all the islands contain the same amount of forests" extremely simple. However as a side effect, it becomes harder to have jagged borders on shorelines, or long and thin hills and forests. Moreover, the larger the cells, the more noticeable their straight edges will be, especially when manipulating the terrain.

For that reason, it is possible to split the terrain in multiple layers of regions, each with its own sampling density. However, with too many and too dense layers, map generation becomes really slow.

Built-in features

The current framework handles terrain creation through a custom terrain class. It has RMS elements that allow to modify the terrain's altitudes and textures (the textures side is fully working but still WIP for improvements), and RMS elements for create game objects in the world.

There are various degrees of randomness, both in splitting the world in regions through a Voronoi diagram, and in filling those regions, by using different random distributions and smoothing functions.

Potential for game-specific expansions

The classes that manage terrain's altitudes, textures and objects generation all inherit from a common parent. That parent can be inherited to add different kinds of "generators" for game-specific needs.

Underlying structures

These are the core classes that are used by the system in order to work.

Terrain and tiles

Scripts/Map

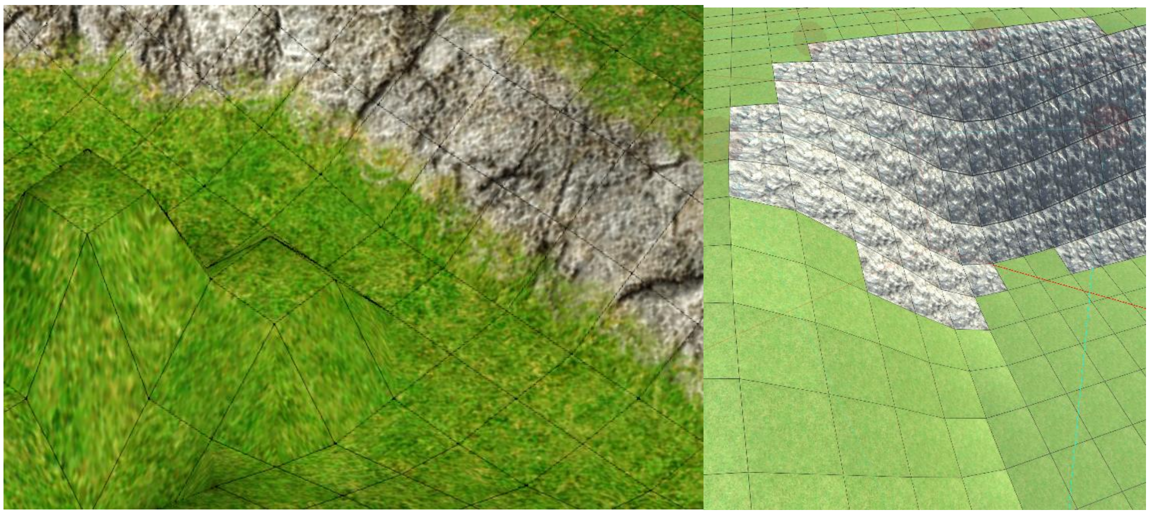
A terrain is created and is accessed through scripts as a matrix of tiles. The actual vertices of the mesh are managed internally by the Map_terrain class.

Each tile has 4 vertices in its corners, those vertices are identical but not shared to the appropriate vertices in the neighbouring tiles. This is to allow each tile to use different UV values for terrain texturing.

The alternative (and previous version) had each tile share the ownership of vertices with its neighbouring tiles. That version is still present in the code and commented out. It allows to save memory and will be used in later editions.

The current version causes a natural grid to appear around each tile. I didn't try to solve that, as future versions will ditch the current version and rollback to the previous one (see reasons later).

Terrain management and vertices handling mimics the one from Age of Mythology, with each tile having an height value. However, whereas in Age of Mythology the vertices are modified based on a delta on the tiles, here the vertices are generated depending on the height of the tiles they share. This avoids sudden spikes, like the ones that can be seen on the bottom-left of Age of Mythology's picture:



(Age of Mythology on the left, RMS Framework on the right)

The core methods of the Map_tererain class are "update_heights" and "update_uvs" which will update the underlying unity mesh to use the values from the tiles matrix.

The current approach makes it impossible to smooth the variation of texture across tiles. A further upgrade would require switching to Unity's HDRP and use Unity's multi-layered material to accomplish the desired result. For time reasons, I decided to complete the uv-based solution, as it's enough to illustrate terrain variation regardless of fancy looks, and it freed up time to work on the actual RMS. The only real negative effect is the noisy black grid.

Geometry

Scripts/Utils/Geometry

This is a collection of classes to manage 2d geometry made by me. The entire library works following the more usual conventions in 2d graphics of having the y coordinate going downwards. A “Point” class bridges this library with unity, by taking care of dealing with the difference in coordinates system.

All the classes in Geometry are designed to be stand-alone, and in no way tied to the scope of this project.

> Point’s x is Vector3’s x
> Point’s y is Vector3’s -z

Then come Segment and Polygon (with its own subclasses Concave, Convex, AxisAlignedBoundingBox, Rectangle, Triangle).

The geometry classes are used to support a Voronoi diagram and checks for intersections during map generation.

Voronoi

The Voronoi diagram construction starts from a library found here (https://github.com/Zalgo2462/VoronoiLib). The library is partially incomplete, and doesn’t support adding user data do the diagram’s cells.

Starting from that library, a Voronoi diagram can be built with user-defined types for the diagram’s cells (in this case, the Cells type is defined by the RMS Framework, which this class would otherwise be unaware of).

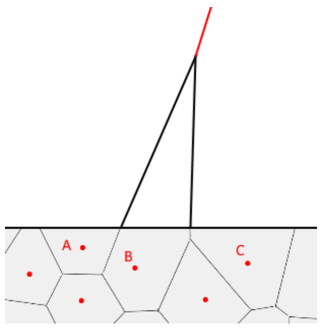
Some problems arose with trying different libraries:

There were unpredictable behaviours on the outside of the diagram, and none of them produced fully enclosed “cells”. The cells in the borders were open to the outside, not closed polygons.

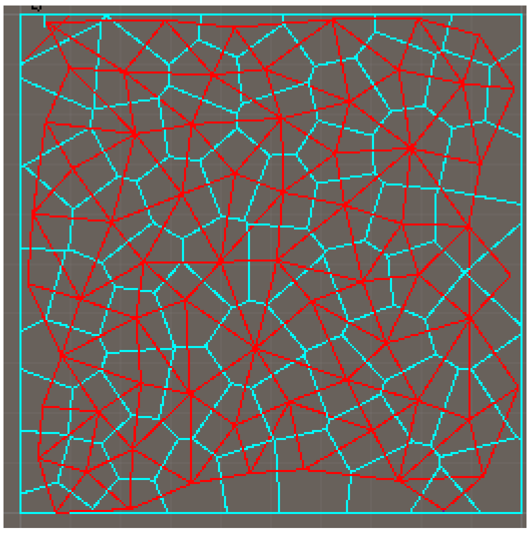
So, Cell’s construction through my wrapper to VoronoiLib also takes care about filling the gaps and making sure each Cell is enclosed by a polygon.

Another problem with all the Voronoi implementations I found is that the Delaunay triangulation used to define each cell’s neighbours also included other cells which had shared edges outside of the wanted region. Those were removed from each cell’s neighbouring in my wrapper.

Example: A and C would have been reported as neighbours:



The final result is the following, where blue lines are the edges and red lines connect neighbours:



Randomness and smoothing

In order to achieve an even distribution of points to generate the Voronoi diagram, I used a Poisson Disc Sampler implementation, taken from here (http://gregschlom.com/devlog/2014/06/29/Poisson-disc-sampling-Unity.html), and further adapted it in order to allow starting the sampling with some predefined points, which will be the individual players spawns. That way we can enforce additional constraints in the definition of some points.

For smoothing, the following library has been used (https://github.com/lordofduct/spacepuppy-unity-framework-3.0). It allows to use various smoothing functions on a 0-1 range.

Testing

Most of the functionality mentioned in this section is independent form map generation. A specific scene “Support_test” was made to test these features independently.

The Random Map Script

Scripts/Generator

Management Classes

Some management classes are required in order to use an RMS. The built-in managers deal with terrain textures (Textures_manager) and game resources (Resources_manager).

They are meant to be the same through the whole game, and the RMSs will poll data from those managers.

They both share the same structure. The user through the inspector can add new entries, give them an unique name, and assign data to each entry.

Resources Manager

The resources manager defines mapping between a string and a resource, where a resource is defined as a list of Unity prefabs. When the map is populated, the resource string will be used, and a random prefab will be instantiated. This lets the map creation take care about offering visual variety. For example, an “Oak tree” resource can have 10 different prefabs with 10 different oak tree models.

Textures Manager

The textures manager creates a “super texture”, and exposes the UV values of each subtexture, so that the map tiles can use the correct UV values to draw the appropriate texture.

While the interface and inspector won’t need any change, the internal functioning will need to be updated when switching to HDRP’s multi-layer textures.

RMS Manager (and RMS Generator)

The RMS_manager component is the one that must be added to a game object in order to initiate map generation.

It automatically includes all the required classes (terrain and managers), and RMS_generator, which is the script that actually gives inspector options. The Generator is a separate class because Unity can’t manage adding components to objects via editor if those components are classes defined within another class. Because of that limitation, and since the RMS Generator needs to access private data of the RMS class, the Generator component must be added via code, and that’s what the RMS_manager class does. This allows the RMS class itself to stay clean and not expose methods or fields that are not supposed to be directly accessed from user-defined random map scripts.

All of this because C# has no way to declare friendship besides class declaration nesting, and Unity’s Editor doesn’t see nested classes when trying to add a Component.

The Generator also retrieves via reflection all the RMS scripts present in the project (See the Script dropdown), in order to start generating a map accordingly to a given script through the Generate button).

Note that this way the RMS class doesn’t need to inherit from MonoBehavoir or ScriptableObject, nor does it need to be somehow instantiated somewhere in the editor. The RMS Generator class takes care of everything. Additionally, the Generator a parameter “Tile_size” which is meant to be the same for the whole game. For that reason, it’s not specific to individual scripts. It determines how big a tile is in unity’s coordinate system. By default its value is 1.

RMS

This is the core class from which users must inherit to write their own Random Map Scripts.

It prepares the environment by generating the terrain, generating spawn positions for the players, and splitting the map in different regions through a Voronoi diagram, which points are defined by my variation of the Poisson Disc Sampling library, passing it the spawn positions first, so the sampling is aware of those points being taken.

When RMS is inherited, the users can define certain properties which are used to setup the environment:

```
protected virtual int   tiles_width
protected virtual int   tiles_height
protected virtual float minimum_region_radius
protected virtual float minimum_player_starting_space
```

and can then implement the actual script inside the run() method.

Random Map Script Classes

From here onwards, we will talk about classes which are meant to be used inside the user-defined RMS scripts. Those are used to actually define the map itself. All these classes are designed around having an intuitive interface to use inside scripts. All the ugliness is ideally hidden within the RMS parent class, which exposes clean properties and methods that bridge between the script and the behind-the-scenes.

Selectors

Selectors come in 2 forms: region selectors, which deal with Voronoi cells, and tile selectors, which deal with terrain tiles. They allow to determine a set of regions or tiles on which to perform different actions. They work through chaining different functions until the desired set is obtained.

Region Selector

Is used to select a set of regions. The RMS parent class exposes the default Voronoi set as `regions`, through which the script can start selecting. However, a script can also produce new layers of regions via the `create_new_regions_layer` and `create_new_regions_layer_with_spawn_positions` functions. Those Voronoi diagrams can have sizes which vary regardless of the default regions. This feature is used to achieve more variety. For instance, Cliffs based on a layer can cut through forests based on a different layer.

Each region layer exposes some properties to access predefined sets of regions from which a selector chain of methods can start. Those are `all`, `border`, `inner`, `filled`, `empty` and `spawn`.

`Example:`

```
// Create a new layer
var layer = create_new_regions_layer_with_spawn_positions(10);
// Take only the regions that contain spawn points
var test_on_layer = layer.spawn;
```

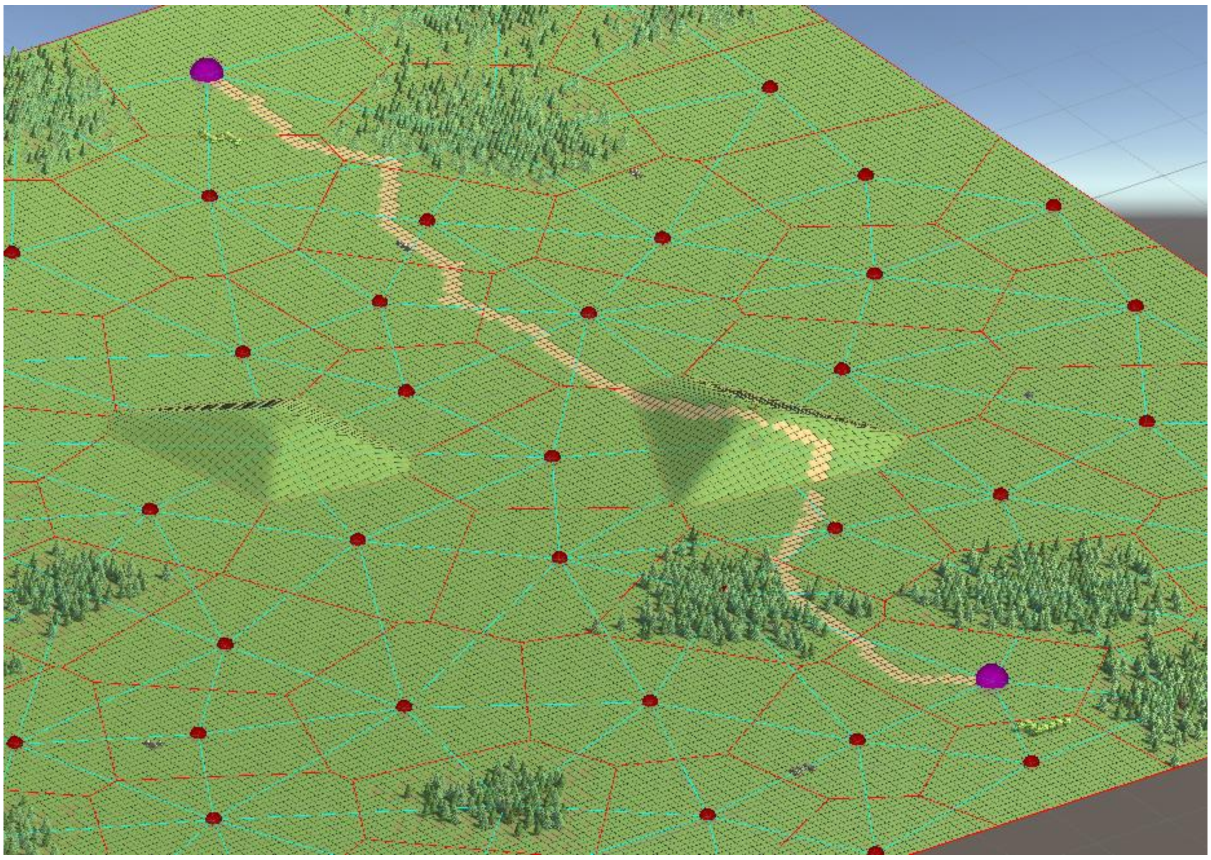
From there on, it is possible to modify the set by including, excluding other regions, selecting adjacent regions, selecting n random regions amongst the currently selected ones, or selecing n closest regions (difference with adjacent is that if n is greater than the amount of adjacent regions, it will keep looking in the next neighbourhood). Additional selecting methods like "containing" and "touching" are meant to allow selecting regions of a layer based on information from another layer.

```
var myregions = regions.border.include(regions.spawn).random(5).adjacent()
.except(regions.spawn).touching(test_on_layer).containing(test_on_layer);
```

Tile Selector

The tile based selector is similar to the region selector, but it deals with tiles. Tiles are unique for the map, so there's no concept of layers, nor randomness involved. Besides the obvious include and exclude methods, tile selectors allow for checks on their content: `height` filters tiles of a certain height, or between certain heights, `slope` filters tiles with a certain slope or within a certain slope, `terrain` filters tiles to which a certain terrain texture is assigned, and `contains` filters tiles on which a certain resource was instantiated.

Finally, the `in_line` method filters all the tiles which lay on a line that connects the Voronoi points of two given sets of regions. This allows for much freedom of action. A thick line can be used to have passable "ramps" that cut a rocky cliff (see the Acropolis.cs example), but a more complex usage could be creating a regions layer with very small regions, and generating a path that connects two players spawns (see the Arabia.cs example).



Modifiers

Modifiers act on the world based on the Selectors filtering. The generic class Modifier is meant to be inherited from to make any game-specific tool that should be available in scripting.

A Modifier allows to act on the selected cells or tiles, according to some randomness parameters.

Inheriting for game-specific code

A Modifier class must implement the action method

```
public abstract void action(ref map_gen.Tile tile, float weight = 1f);
```

with the action to be performed on each tile. Weight can either be interpreted as a chance or an actual weight. For example, the terrain altitude modifier (Terraformer) uses the weight parameter to determine the delta height to apply, whereas the resources generating modifier (Filler) uses the weight parameter as a chance to spawn a resource in a given tile. Ideally when using multi-layer materials for terrain texturing, the weight parameter would determine the opacity of a newly applied texture mask.

Besides implementing the action method, for each Modifier there should also be methods in the RMS parent class to act with it transparently from within scripts (The idea is that scripts will never contain “new”, being only limited to very simple and easy code).

This is possible thanks to C#’s support for partial classes definitions, which lets the user wanting to extend scripting features, define his new Modifier class inside the RMS class.

RMS-side methods

There are few methods to determine how a modifier acts on the selected regions.

The simplest is `all`, which performs the action in all the tiles of the selected regions, with a weight of 1. It is used to fill entire regions (Example: texture the whole world with a base terrain).

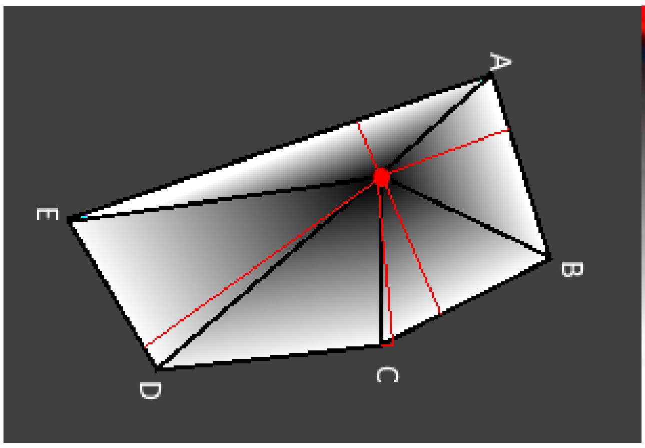
`uniform` will affect the tiles of selected regions each with a different random value.

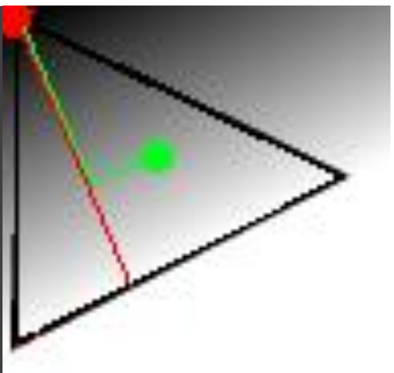
`grouped` picks a random tile of the region, and starts navigating from there, until the desired amount is reached (if possible). It always has a weight of 1

> Sometimes not working properly and producing less than the desired amount, under investigation.

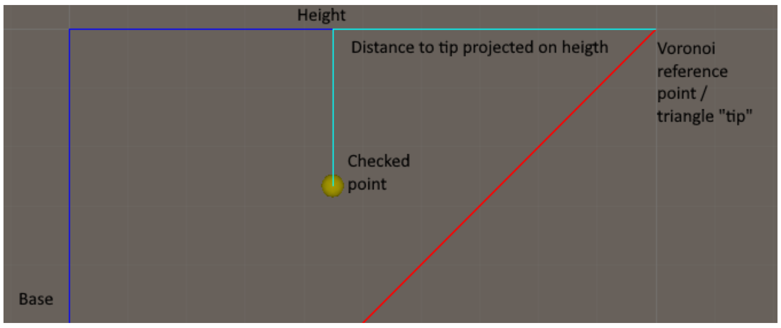
`centre` picks the tile which is the closest to the Voronoi reference point of a region.

And finally the most complex and useful, `radial`. Radial by default defines a weight of 1 by the centre of each tile, and 0 by its perimeter, decreasing linearly with distance. Since not all the vertices of a Voronoi cell are equidistant from the “reference point”, a big part of the geometry module comes into play. The polygon that defines a Voronoi cell is split into triangles.





Then for each tile taken into account, the triangle that contains that tile is selected. Finally, the distance between that tile and the reference point, along the height of the triangle is calculated and normalized relative to the full height.

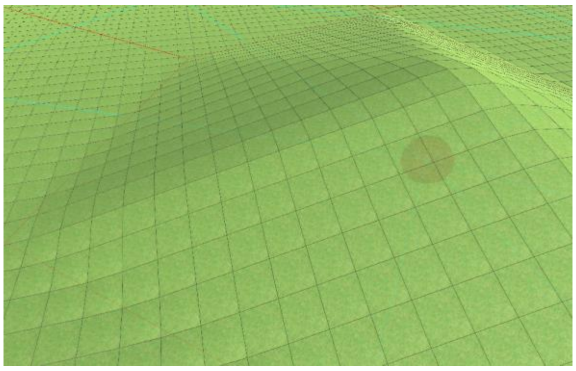

Height
Distance to tip projected on heigth
Voronoi reference point / triangle "tip"
Checked point
Base

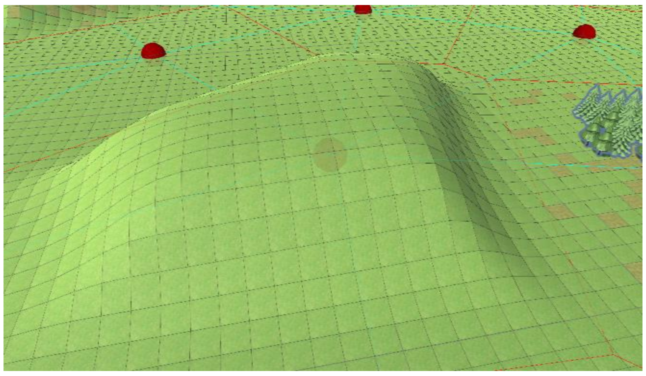
An example of this calculation can be tested in the geometry example scene.

Finally the smoothing library comes into play, by allowing the user to define different easing styles. The easing functions are explained in further detail here: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/graphics-multimedia/easing-functions?view=netframeworkdesktop-4.8

Two additional parameters are from_centre and from_border, which allow to determine an area of constant weight before the easing kicks in.

Here we can see hills generated with different parameters:


Linear "easing", no spacing


Cubing easing, spacing from_centre

Mirroring regions selectors and tile selectors, there is also a tile-based variant of the modifier. However, since it takes tile selectors and has no notion about regions, it only has access to `all` and `uniform`.

Built-in Modifiers

Filler

This modifier takes care of spawning resources; it is started via the `populate` method. It takes a string that indicates a resource name, and internally takes care of retrieving that resource from the resources manager. Obviously, the resource must exist.

The weight parameter is used to determine the chance a resource will spawn in each tile.

```
populate(forests).with("Tree").radial();
```

Terraformer

The terraformer takes care of terrain height; it is started via the `rise` method. It takes a float which indicates how much the terrain must be risen. The actual delta applied is that value multiplied by the weight on each action call.

As seen earlier, a radial terraforming modifier would create a spiky hill, whereas using easing functions and adding flat spacing from the centre, can create smoother results

```
rise(hill).by(4).radial(EaseStyle.CubicEaseInOut, from_centre: 0.2f);
```

Painter

Finally the painter is responsible for applying different textures to the terrain; it is started via the `paint` method. In the current version the weight parameter is used as a chance, like for the Filler modifier when used in region-mode; the weight is ignored when used in tile-mode. With layered materials, the weight would determine the opacity of the newly applied material. It takes a string that indicates a texture name, and internally takes care of retrieving that texture from the textures manager. Obviously, the texture must exist.

> There are some issues with Unity's ability of serializing dynamically generated texture packs. If errors appear trying to generate a map, try to click the generate button on the texture manager first, then re-run the RMS.
> Upon restarting Unity, the terrain material becomes white. This is due to issues serializing the texture pack inside a Unity Mesh. I do not have access to the internal code of any of this. Generating a new map will solve this issue.

```
paint(forests).with("Underbrush").radial();
```

Examples

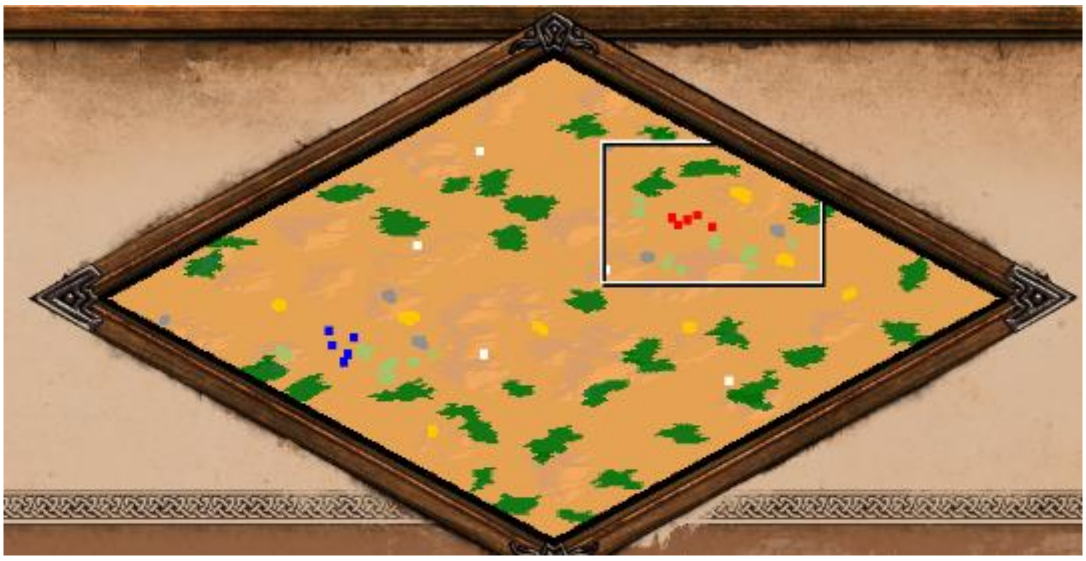
Example/RMS

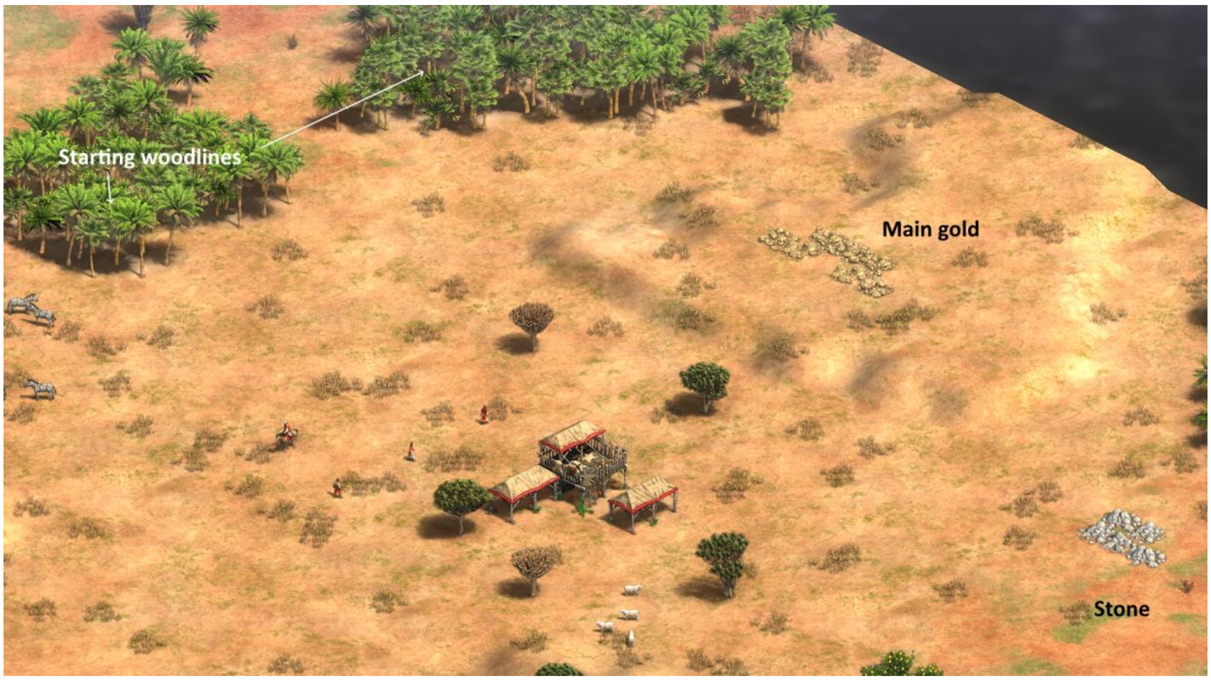
The examples in the repository try to recreate 3 popular maps from the Age of Empires franchise, making use of all the described features.

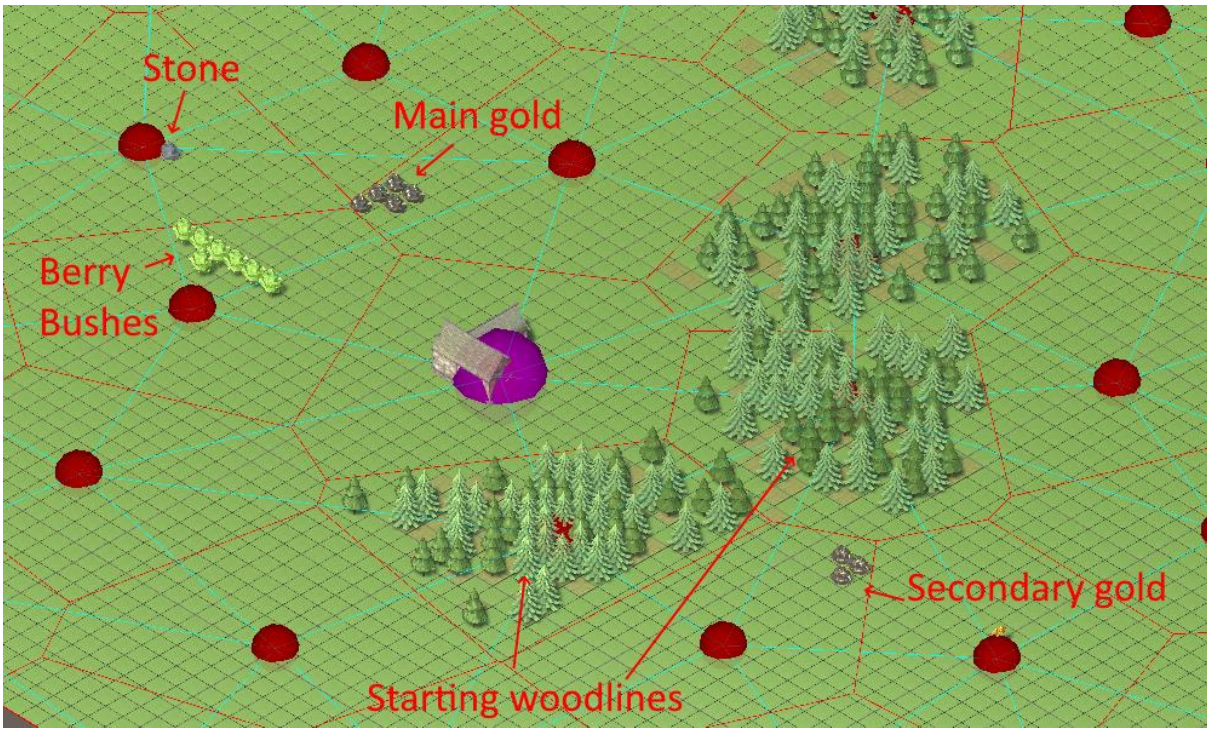
Arabia

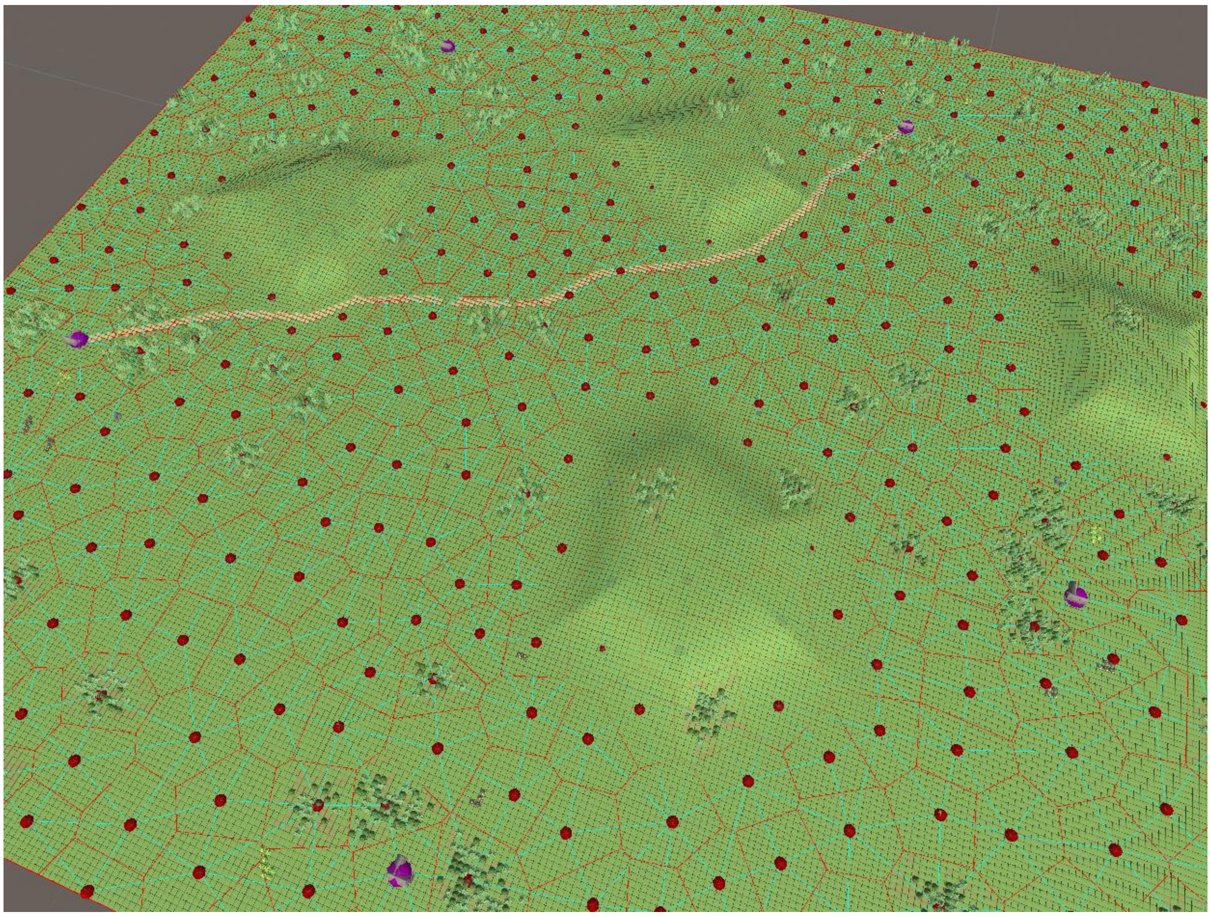
Extremely popular in Age of Empires 2, it's main feature is being… extremely simple and empty. The only relevant part is starting resources locations. It is common opinion that having uneven resource safety amongst the involved players will result in greater gameplay variety; hence there was no effort in trying to evenly secure some resources across all players.

In this example, the red player has a really safe gold spot and back wood lines, while the blue player has more exposed resources:



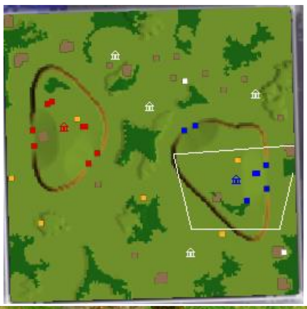


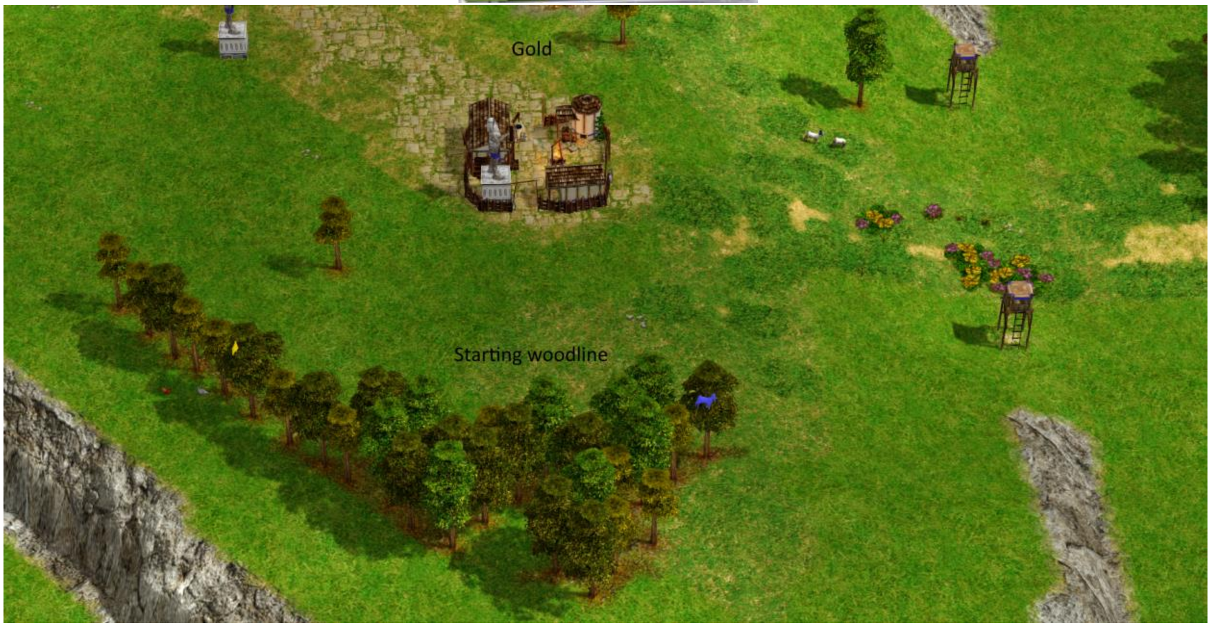

Stone
Main gold
Berry Bushes
Secondary gold
Starting woodlines



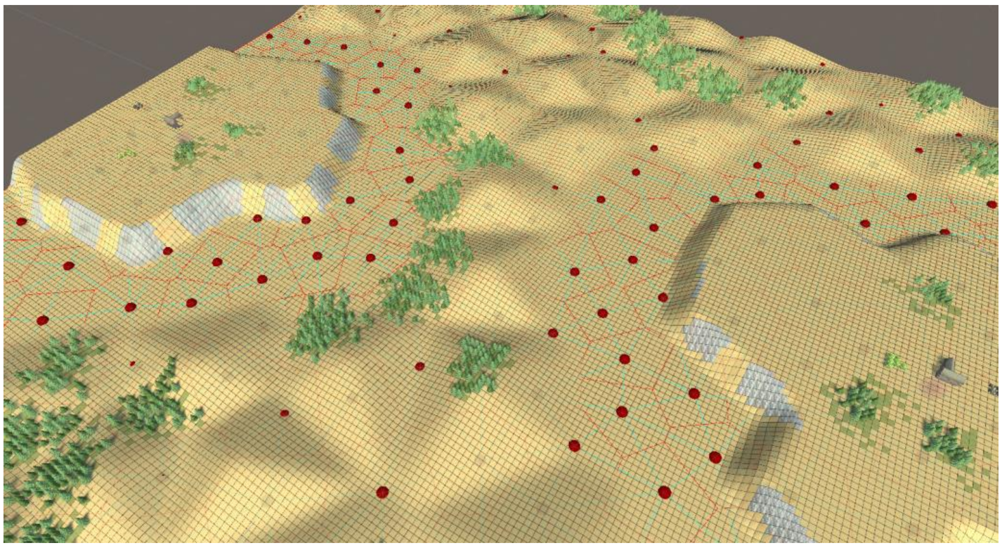
Acropolis

This map is strictly inspired by the Age of Mythology version, with steep cliffs around players starting points, predefined accesses to the hills, and an overall scarcity of wood around the spawns.



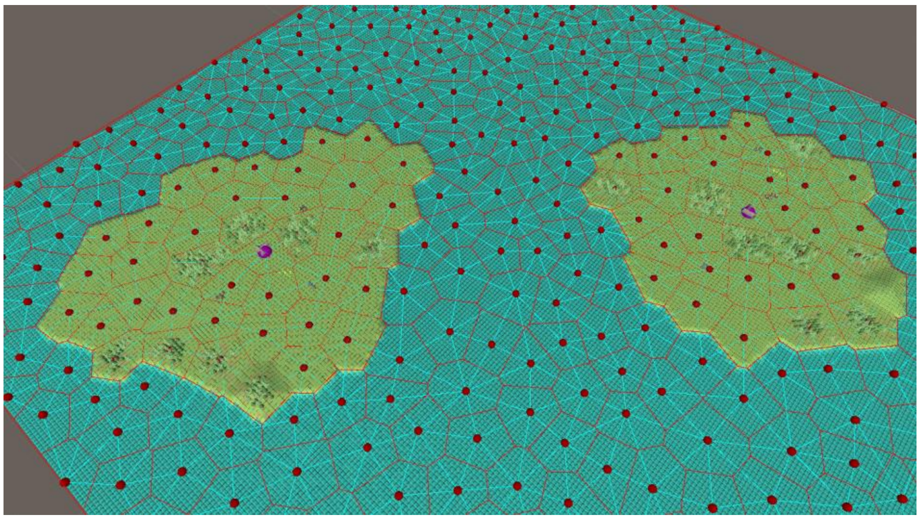


(In this picture I used the sandy terrain in order to make trees more visible)



Islands

This one is common to all the games in the Age of Empires franchise. Each player has his own island, without necessarily a balanced distribution of wood. The island with more wood will determine a definite advantage in the late game, so the players with less wood are pushed towards being more aggressive with early landings. In my version however, thanks to the Voronoi regions approach, I can easily let each island have the same amount of forests. There is more balance, but not strict equality, as not all forests necessarily have the same amount of trees.



Afterthoughts

All in all, I'm satisfied with how the Voronoi regions approach turned out.

There are a couple things that need more refining, but there are some huge things that would need further working, and could result in extremely better results.

Terrain texturing through layered materials

As mentioned multiple times, this would be an extreme change, and is really needed. However, the difference would just be visual, and the project in its current state is enough to try generating actual maps.

Regions merging

Merging regions would open the doors to making long and thin things. Hills, forests, rivers. It would be easy to implement "merging" of adjacent polygons, and most of the code would work fine.

However the most important features, radial mode for modifiers and any selector option that depends on a region's "centre" would need a severe rework. The radial modifier weight wouldn't be as simple as calculating the proportional distance from the vertex of a triangle. It would instead require working with polygons defined by inner lines. A region's "centre" wouldn't be a point anymore, but a contiguous line, which could potentially have even some branching. The complexity of that cases rises steeply.

Boost's library for Voronoi diagrams (https://www.boost.org/doc/libs/1_75_0/libs/polygon/doc/voronoi_main.htm) already supports generating regions from contiguous curves (note, non-branching) instead of points, and there's a C# port of it. However, all the rest of the geometry utilities would also need to be made from scratch.

Regions splitting

Another useful feature would be generating regions from within a given region. Splitting it into an inner Voronoi diagram. This would be an advanced version of the current layers generation, but for a specific part of the map. The this is more easily doable than the previous point, but would still have required more time to work on it. The difficulty here is in completing the non-closed polygon of the outer regions. Currently, it is known that the diagram resides within an axis aligned rectangle, so completing polygons only requires snapping open regions to the corners and borders and completing the missing lines. However, if the regions are inside a non-axis-aligned polygon, snapping becomes more complicated, some edges need to be cut short, some others may need to be prolonged, and even the bounding polygon edges would need to be further split and assigned to different cells.

Once regions splitting is supported, things like non-rectangular maps would become easy to achieve.