Yet Another Map Generator library

Abstract :

This library provides a set of function helpers to create OAD maps. It focuses on map morphing and structuring much more than textures painting and objects placing. Its main goal is map realism and cares not much about playability.
In OAD, maps are a set of square tiles, each corner having their own height, which defines the relief of the ground. Next, those tiles can be painted with structures and populated with actors (structures or units). To achieve a realistic landscape, one must deal first with the relief, and next must identify some regions to be managed differently: for instance, lakes and pools will be painted and populated differently.

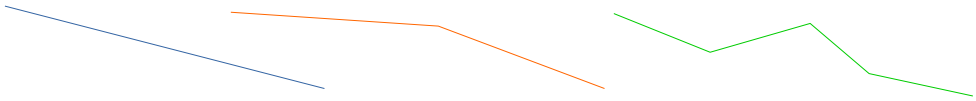
Table:

Map relief.

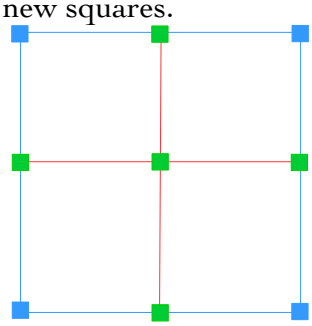
To create a realistic map relief, two set of functions are provided. The first one operates on the whole map and the other on a region only. They use the same algorithm which shall be explained so the reader can understand better the parameters influence.

Fractal generation.

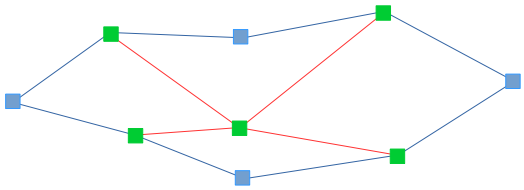
Fractal generation is a very general and simple operation. The idea is to replace each individual shape with a more complex one, and repeat the process on the parts of the shape. For instance:



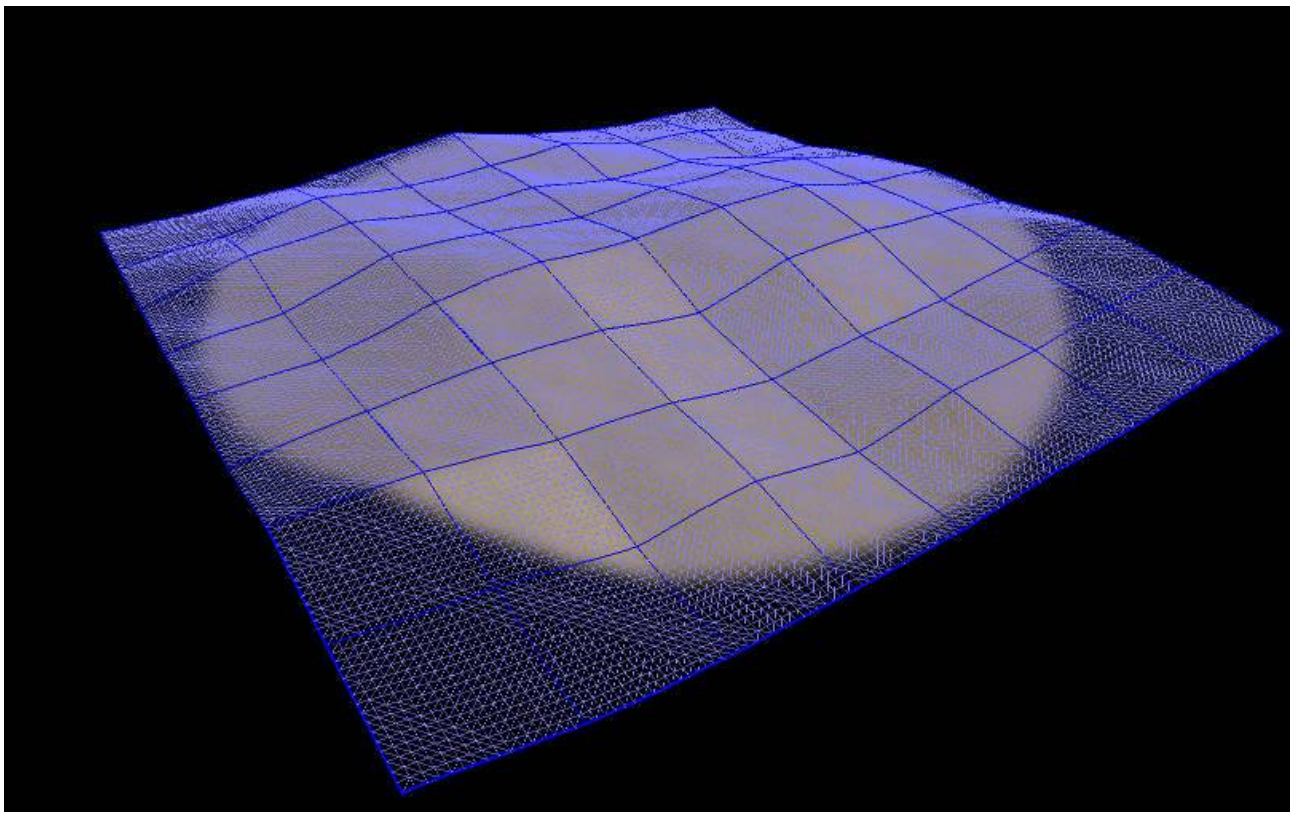
The straight blue line is replaced with a broken line shown in red. Next both segments of this line can be replaced too with a broken line, which gives the green line. The process can be repeated to any level of detail or scale, provided the replacing pattern must be sized to the part to replace. At each scale, the pattern must look the same.
The process can be applied to each edges of a square. A small vertical displacement is added to the middle point of each edge. Adding those middle points cuts the square in four squares on which the same process is applied. Looking at it from behind, we can see the five green points added to the grid which delimit four new squares.



From a perspective point of view, it gives this:



So we start with the corners of the whole map, set their height and start the process: divide each edge in the middle and giving this point the mean value of height edge extremities plus a random value whose range is the first parameter of the function (to make the edge a broken line). Height of the square middle is computed in the same way. Then we can repeat the process on each four new squares, but the random displacement range must be scaled according to edge size (fractal principle), in other words halved, since the new edges length are the half of the former ones. If not, it would be only a fully random map. The process is repeated until we can't divide squares anymore, i.e. their edge size is 1. It finally gives this:



RNG influence.

RNG stands for Random Number Generator. What is important to us is to know RNGs are not random at all but fully deterministic. They mimic the behavior of real random number sources implementing functions which return numbers equally distributed in the range and exhibiting no repetitive patterns[1]. It's just as if they were reading in a very large numbers array, picking the next value each time they are called. The array is cyclic, so one can start reading at any index. The 'seed' can be understood like the index of the first number to read, and then it's clear that:

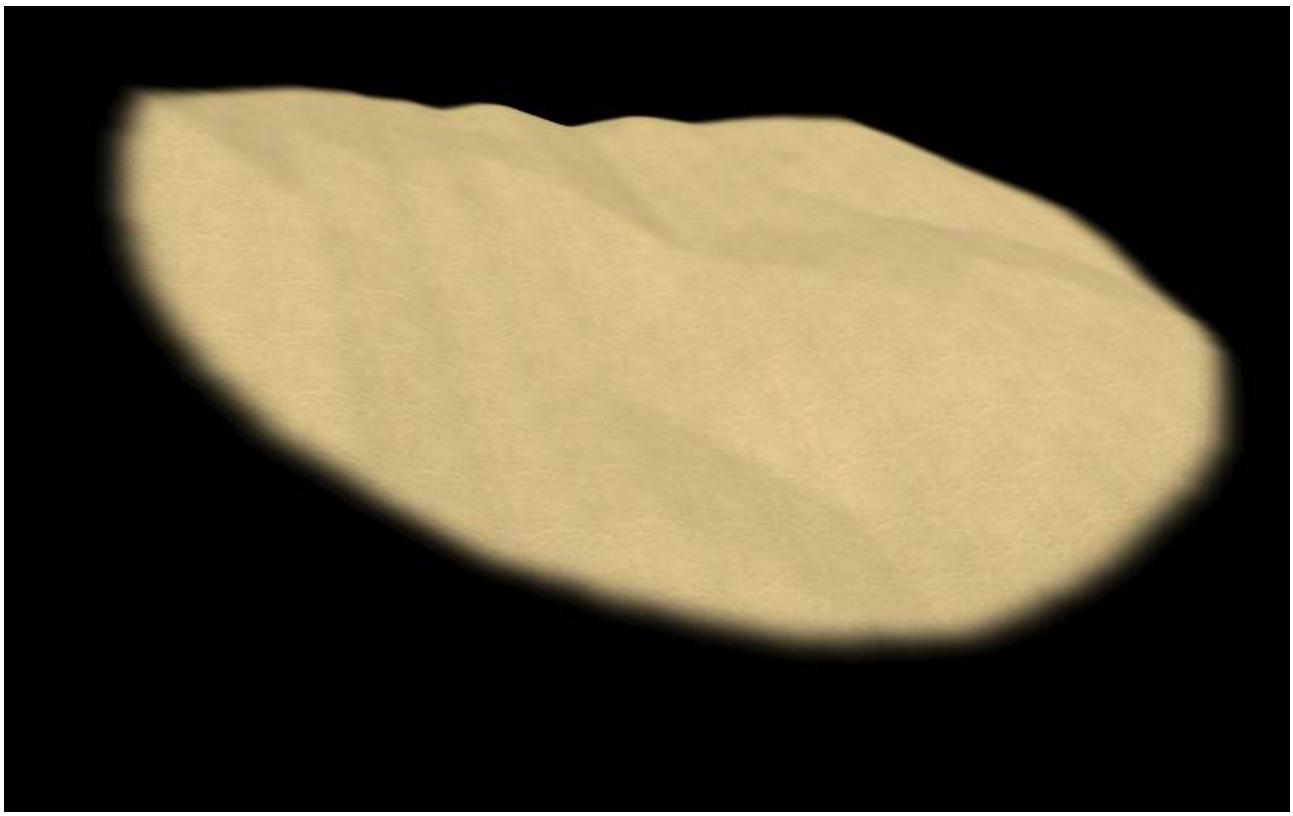
- Starting with two different seeds will return different numbers sequences.
- Starting with the same seed more than once will return exactly the same numbers sequence.

The 'seed' field in Atlas is just that. Now, if you use fractal generation which uses the RNG to compute the offset value, it will reproduce exactly the same map for each seed and size value (which wouldn't be the case if the the number generation was really random). Now, it's true if and only if you don't insert new code calling the RNG before the fractal use. If so, the random number sequence shall be different and produce a different map.

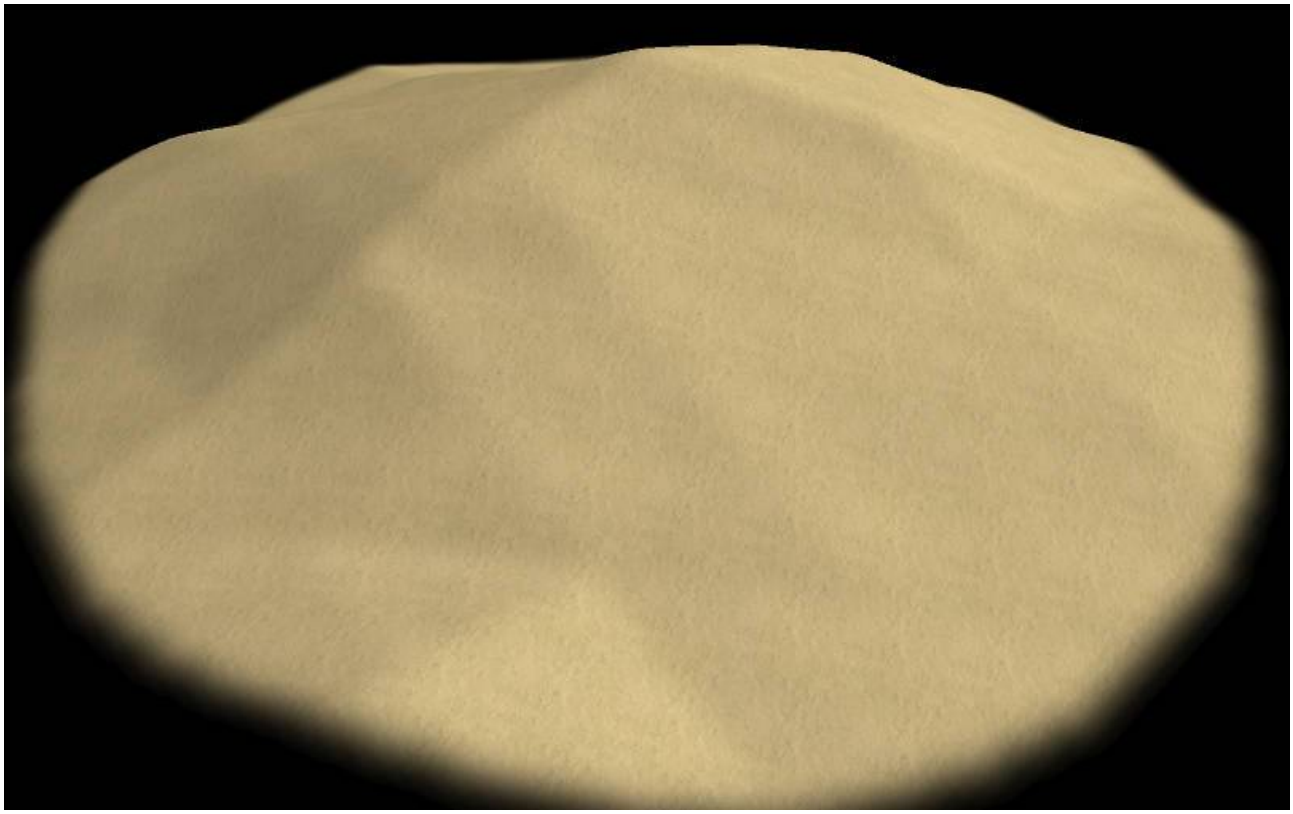
Parameters and tuning.

As already said, we have only to define the corners height (and optionally the middle height) and the random factor range (the noise). For instance, we set here one of the corners height to a higher value than others. This gives the map some inclination:

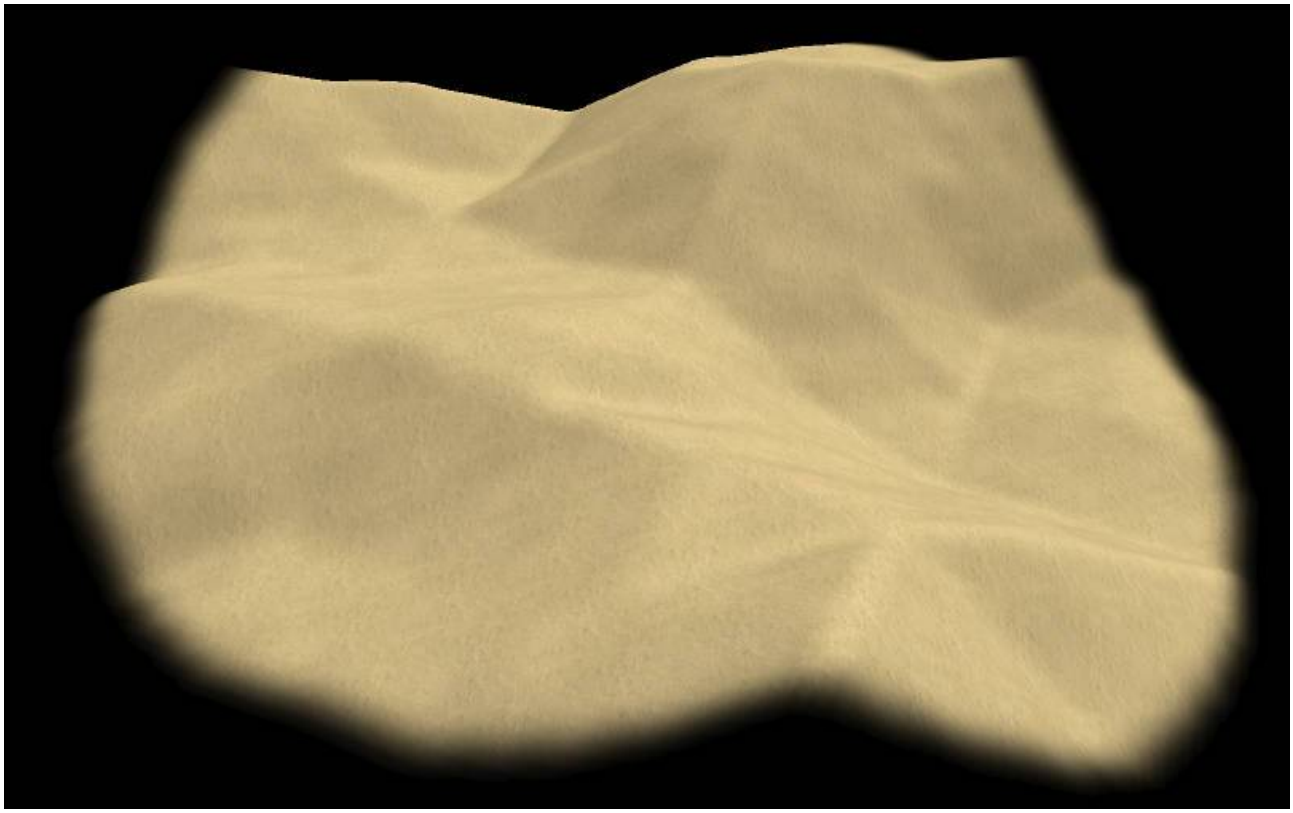
1- … and many other less obvious properties, but we don't care about since we don't deal with Monte Carlo method nor cryptography.



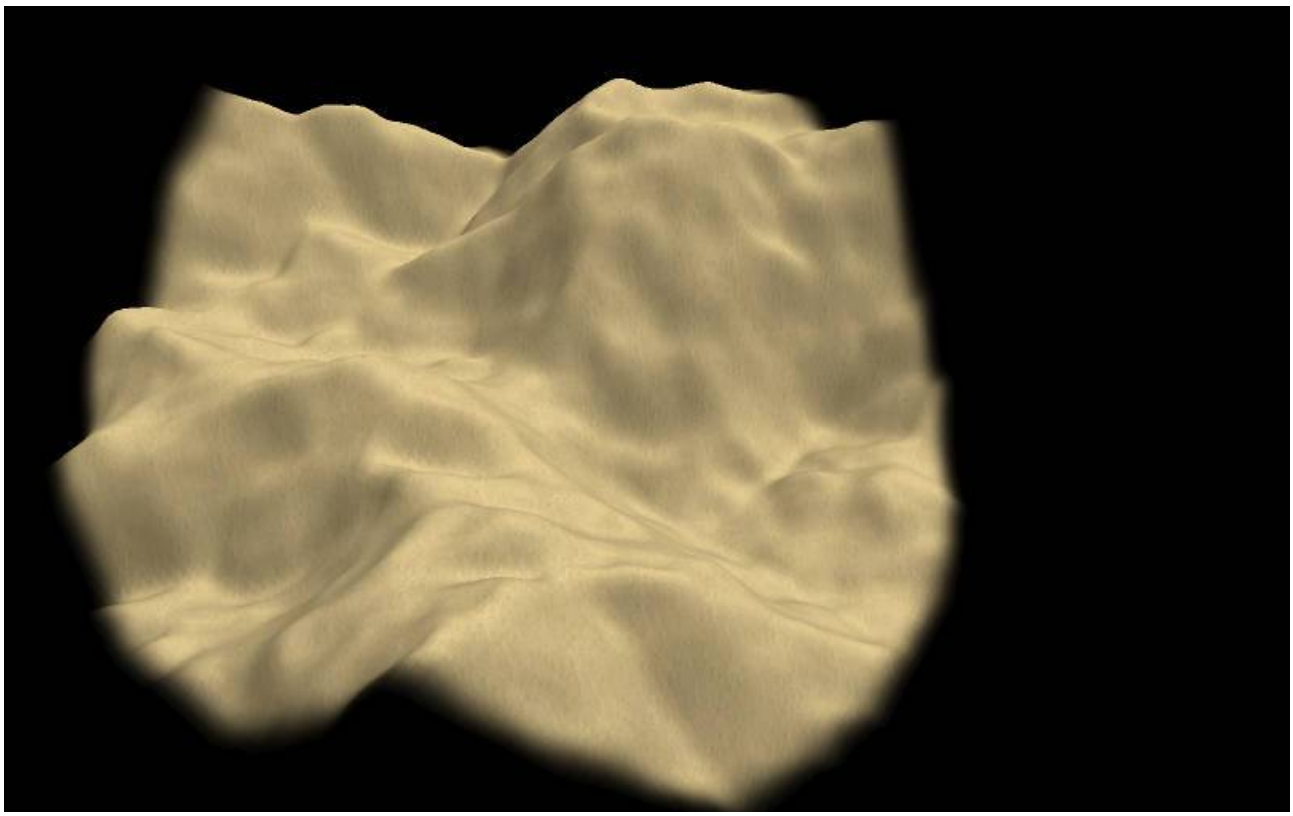
Here we set the middle point higher which creates a hill at the center:



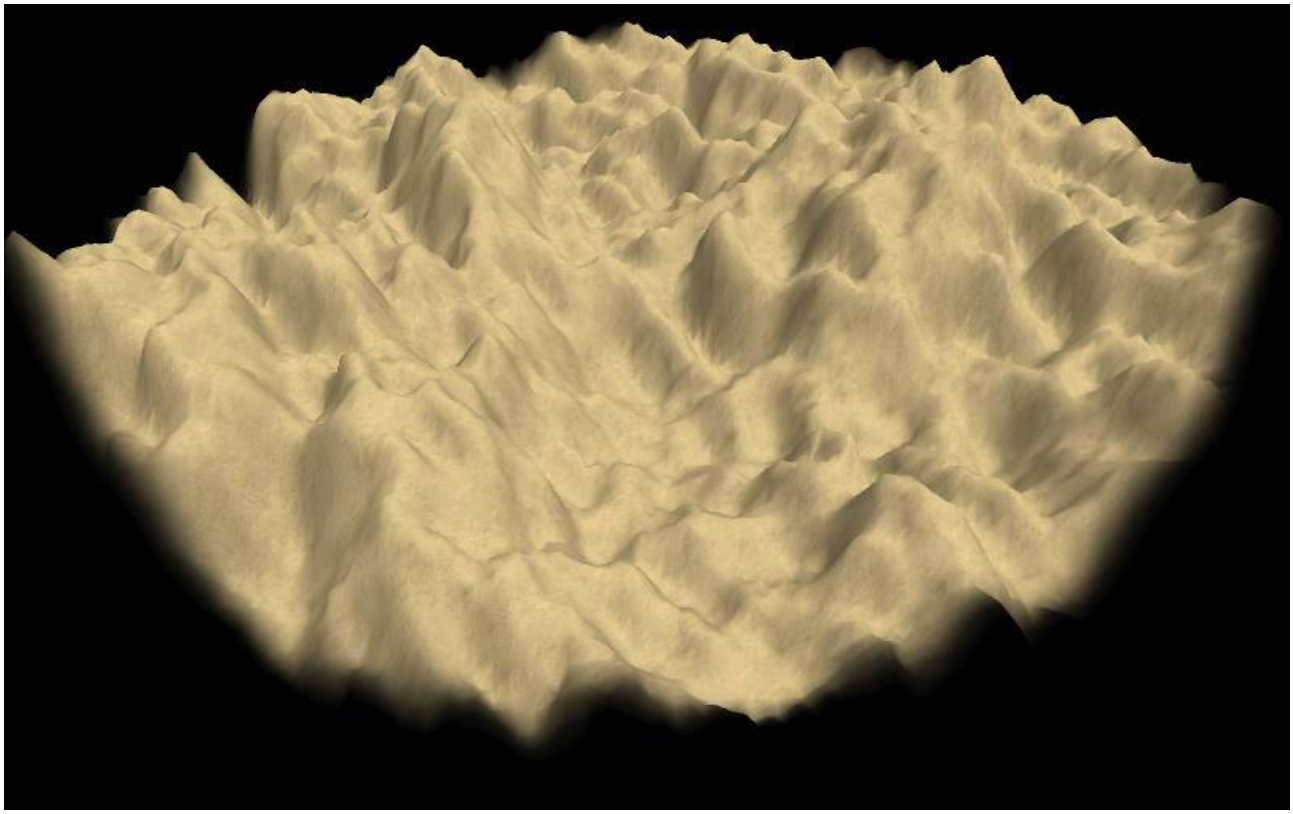
Now the noise factor is obviously very important. Here, we set it to a rather low value which means the relief will be rather smooth, even if some places can be very high. The next picture uses the same corners height, but a higher noise factor:



The map is now much more irregular. Actually the noise factor is not exactly halved on each recursive step as fractal principle requires, but is scaled differently at large size and detail. This gives more control on the map, because you may want large height range and smooth terrain as in the picture above, or the contrary. So there is a parameter named **tProgress** which delimits the boundary on which we swap from one scaling method to the other one. The parameter **tRelief** defines the height range on larger scales, and another one, **tRough**, defines the noise at the detail level. Let's give examples. Here we only raised **tRough**:



Here is an example of a very flat map with very chaotic terrain:



Of course, these examples are extreme, just to show what can be done.
Parameters values:

- **tRelief** is a number up to MAX_HEIGHT * 2. Since it's better to adapt it to map size, I often use the formula: `(MAX_HEIGHT * mapSize) / 200 (high relief) < value < 800 (low relief.`
- **tRough** is a positive decimal. Suggested values are from 0,1(very smooth) to 1,2 (very rough), 0,5 being the standard value. Higher values can be tested but generally don't have good effect.
- **tProgress** is a positive integer from 4 to map size or more. Usual values are 16, 32, 64.

Use cases

Most often, we will not satisfied with rather homogeneous maps. We want some flat and plain regions and more chaotic ones. This can be done merging height maps created with different parameters. There are two ways to do this:

- On a map region only with the fractal painters
- On the whole map.

Fractal painters.

Fractal painters are objects which have a constructor and a method **paint(area)**. You give the parameters in the constructor and the **paint(area)** method do the job (it paints nothing, only modify the height map). In this way, you can use them exactly like other painters from **rmgen**. Here is an example of code:

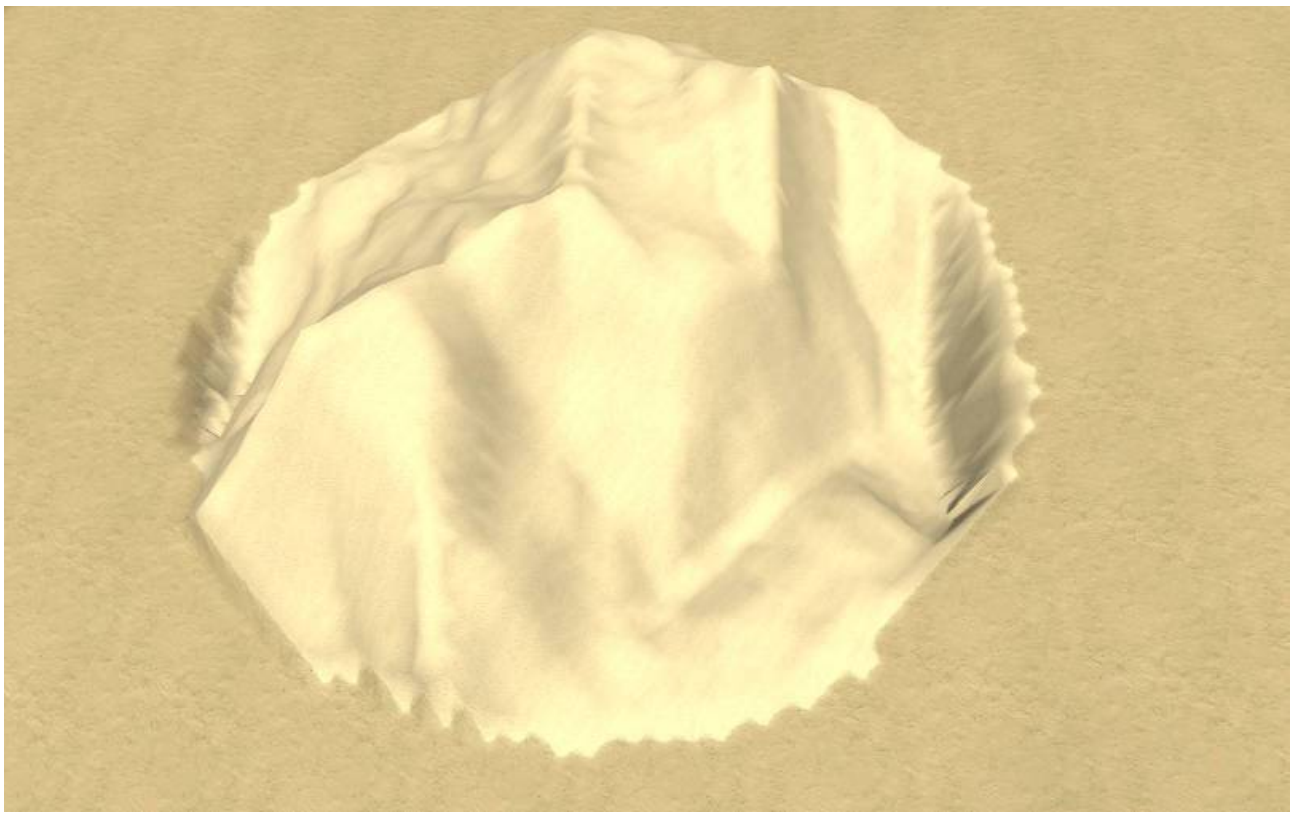
```
let fp = YfractalPainter(centerHeight,tRelief,tRough,tProgress);

fp.paint(area);
```

The parameters are those we have seen before and rules the kind of terrain you want. Area is the region on which we apply the modification. It's an **Area** object or anything having a **getPoints()** method returning an array of x,y pairs (a **Vector2D** most often). **CenterHeight** is the desired height of the center of the region. Actually, the program creates a separate square height map whose size is at least the size of the region bounding box. The center heigth of this square is set to the parameter value. Next the height map is computed and merged into the **g_Map** height map, but only the points belonging to the **area** region are modified. The painters comes in various flavors to help creating:

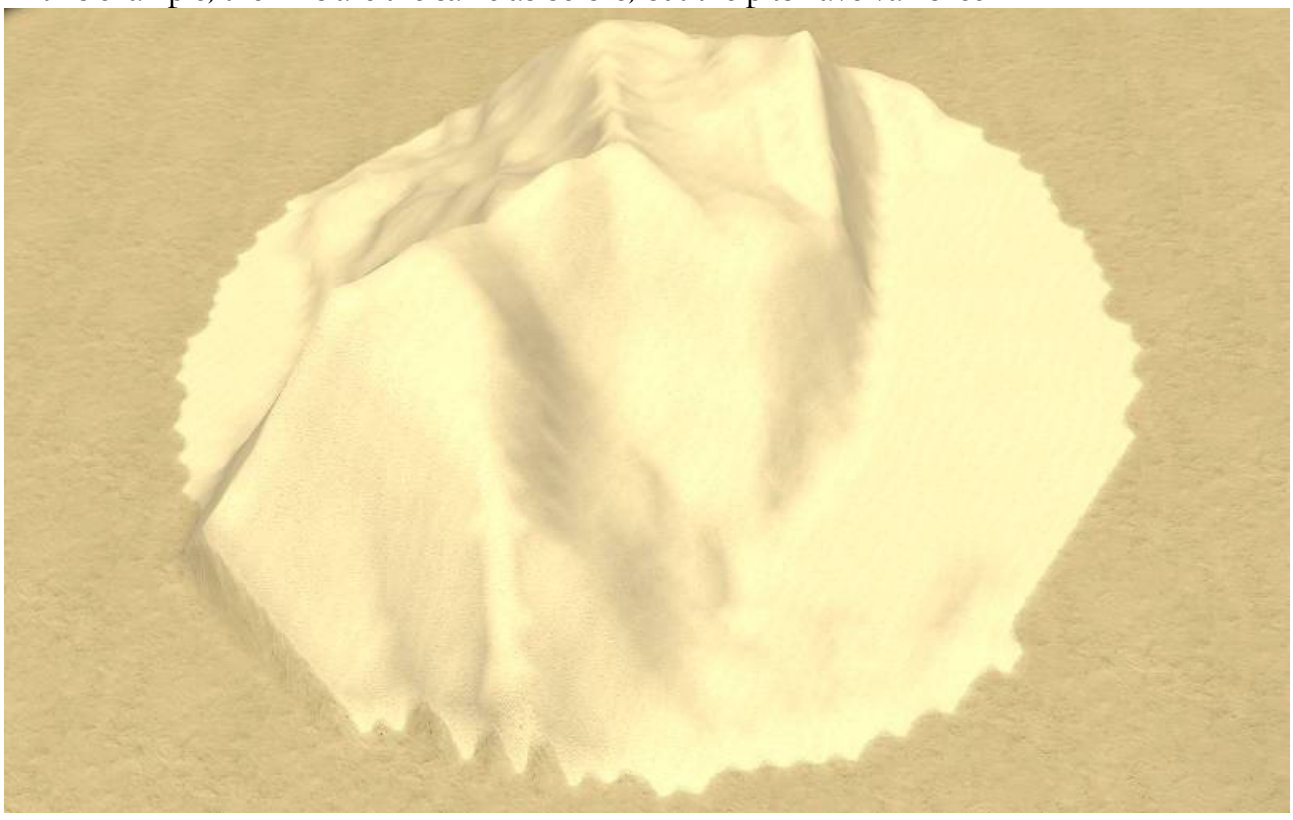
- Mountains or hills regions above the unmodified map.
- Cracks, valleys and holes under the unmodified map.
- Mesas above the unmodified map.
- Flat cracks under the unmodified map.

YfractalPainter

This painter replaces the whole area with the map it computes. This is fine, but rather difficult to manage, because it can equally create peaks, cracks or both as in this example:

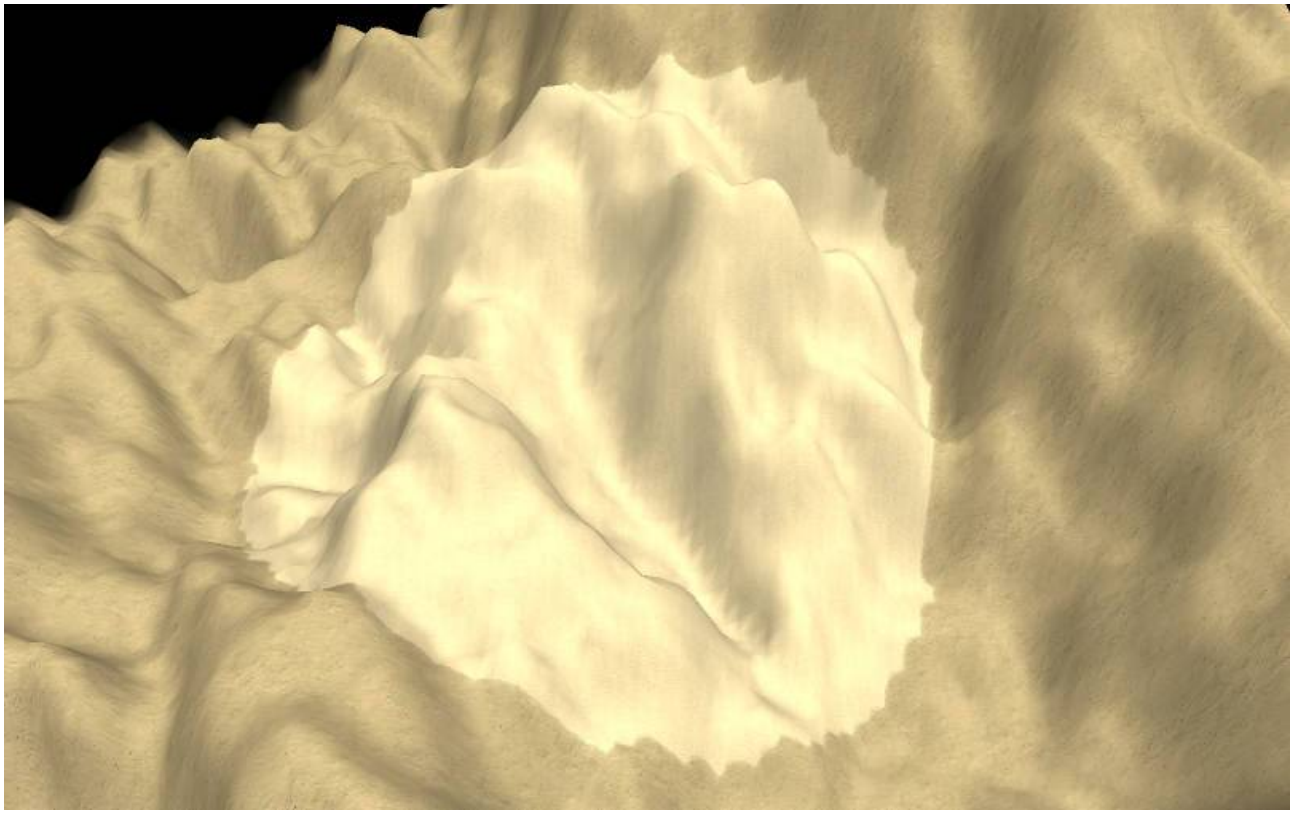


This is why the library provides another painter, **YHillsPainter** which includes only parts higher than the original map, which allows a better merge.
In this example, the hills are the same as before, but the pits have vanished.

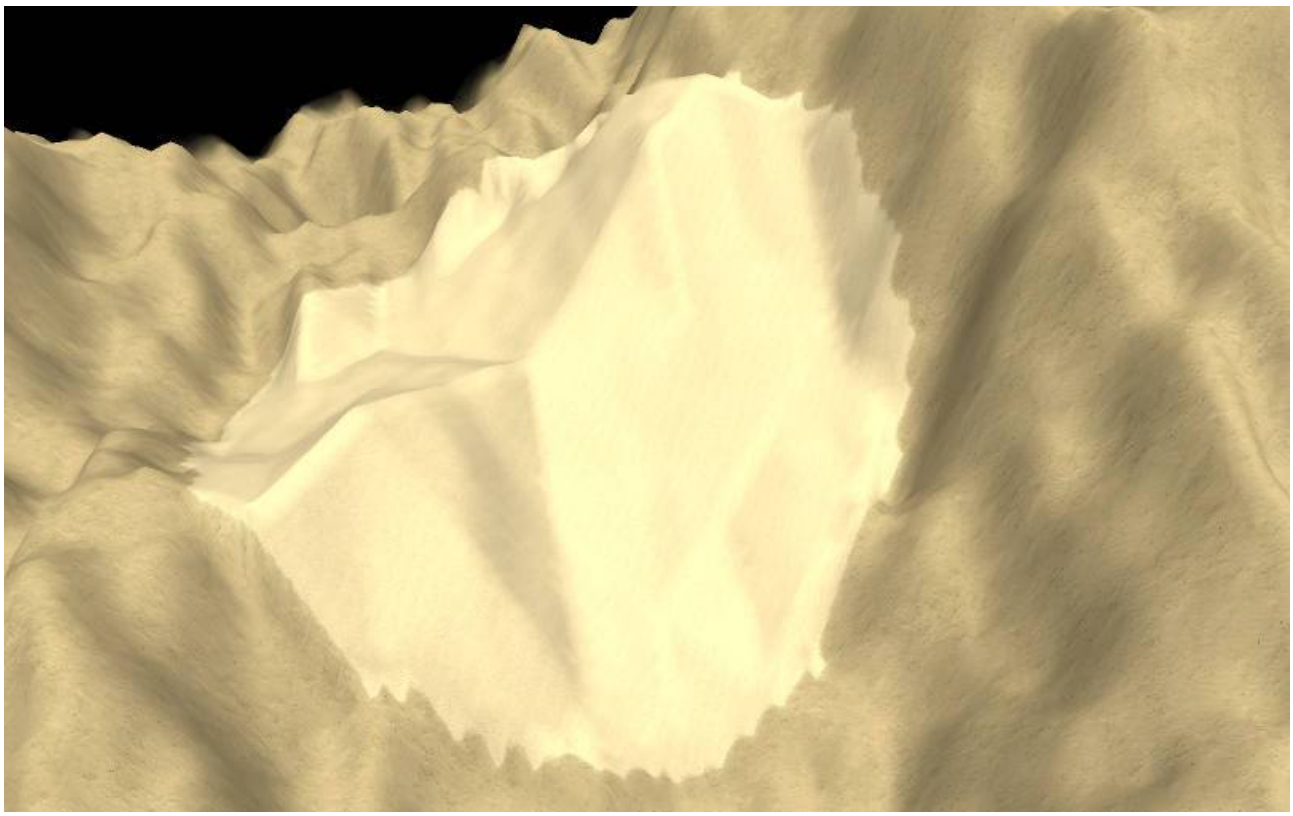


It uses exactly the same parameters in constructor.

For this painter giving normal results, you should set the **centerHeight** higher than the original map.
Here, the parameters are: `(MAX_HEIGHT * mapSize) / 300,0.7,64`
The YFractalPainter can be used to smooth a rough terrain too:



is turned into:



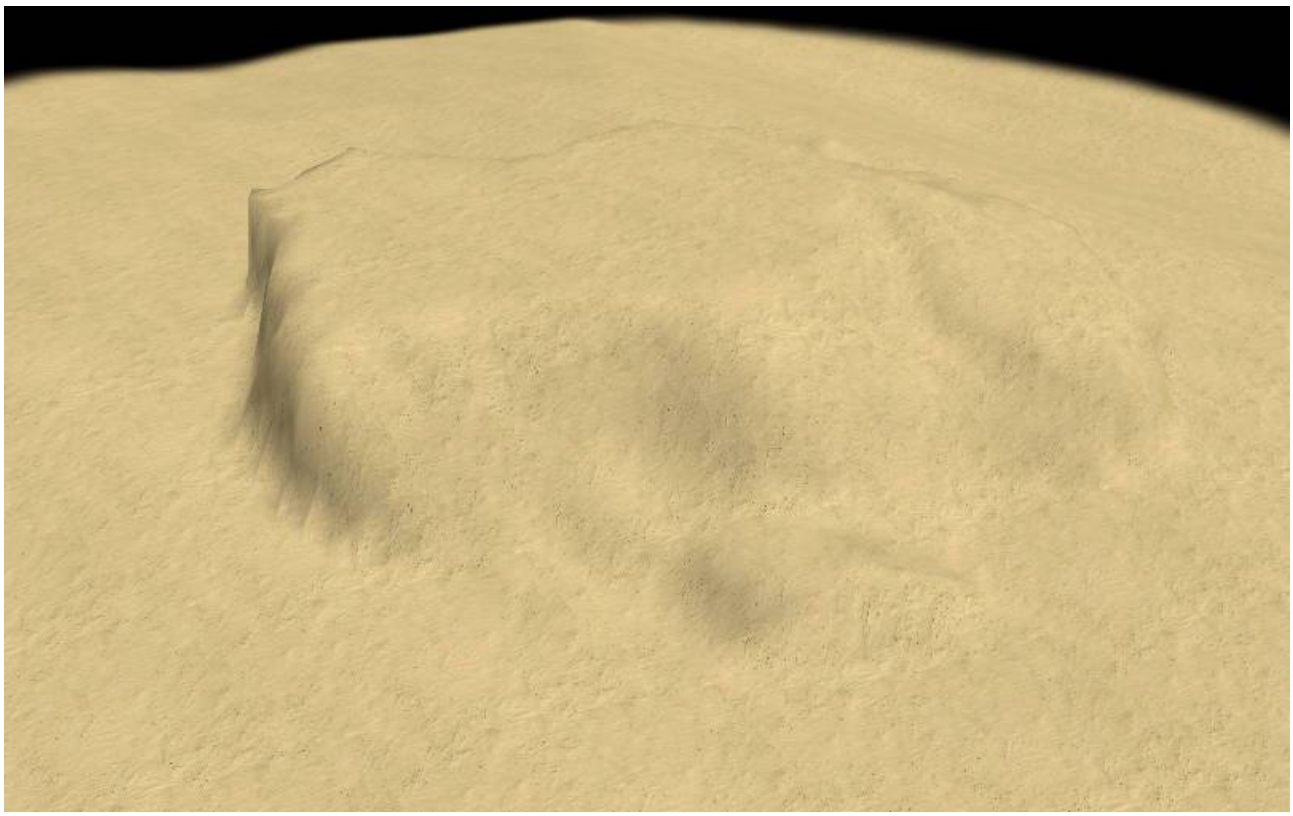
Parameters used are: `(MAX_HEIGHT * mapSize) / 600,0.6,32`

YCracksPainter.

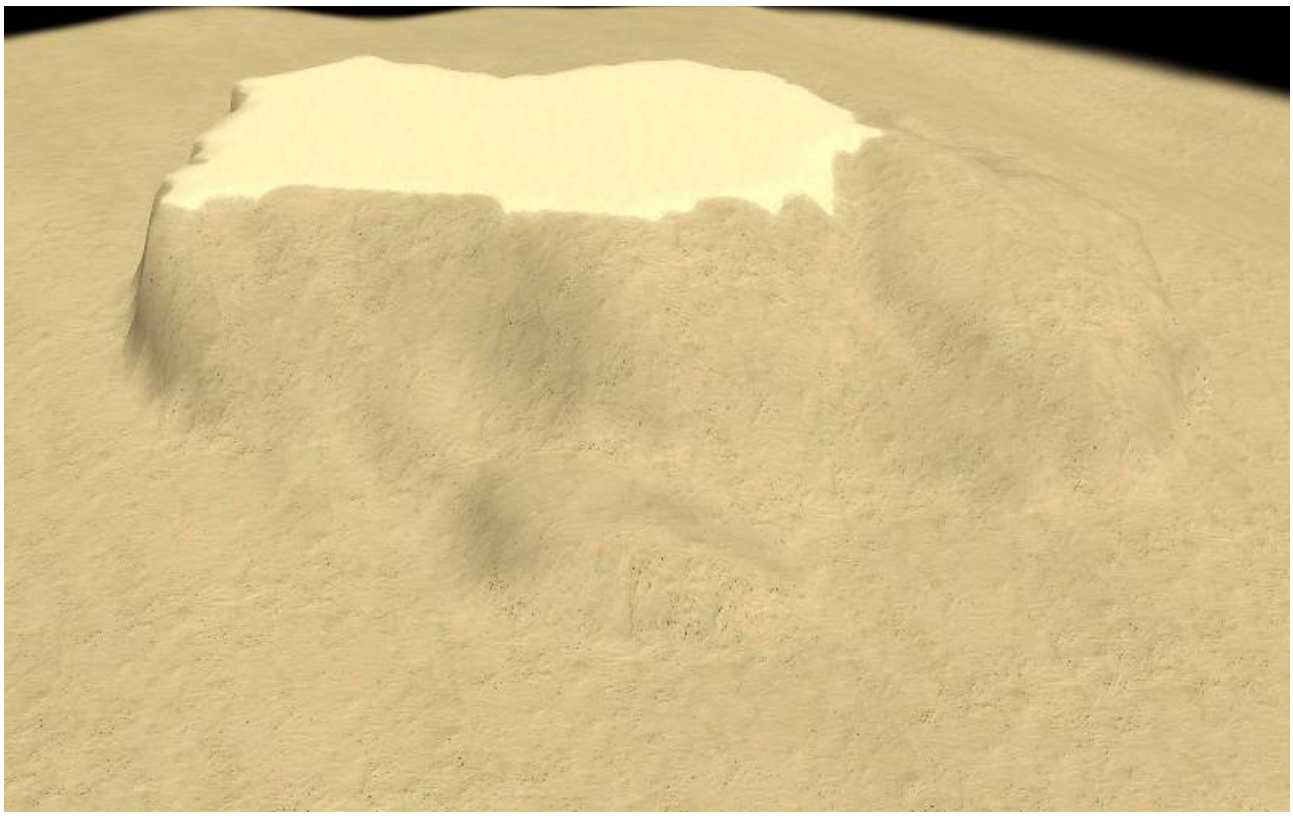
The **YCracksPainter** works in the same way to create only cracks and depressions. For this painter giving normal results, you should set **centerHeight** lower than the original map.

YMesaPainter

This painter is like **YFractalPainter**, but limits the produced hills to a specified height (additional parameter). This creates a mesa or a cliff:

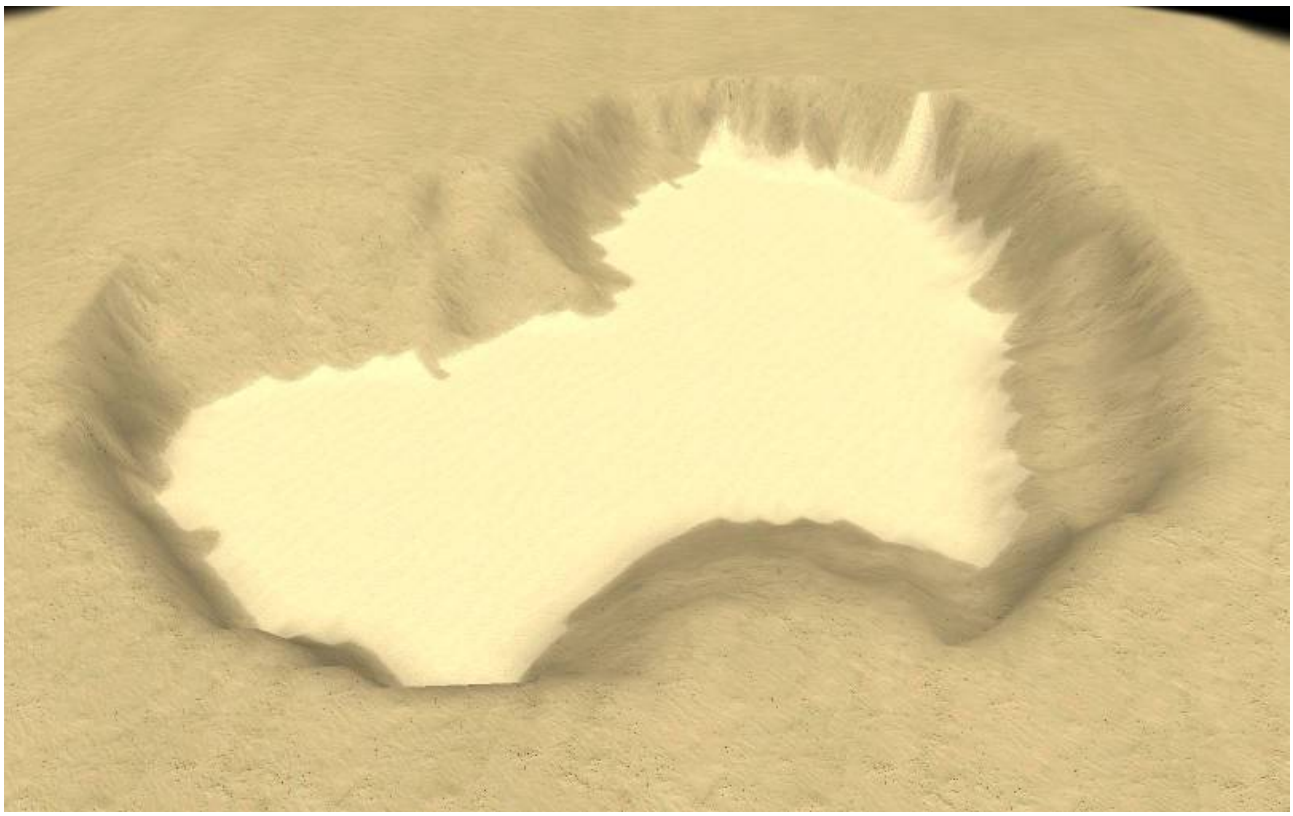


Of course, best results are obtained when **centerHeight** is substantially higher than the region to modify, and the mesa limit set to something lower.
The flat part of the region is available as a result of the **paint** method. It is the **flat** member of the painter: **painter.flat** and is, as usual, a points array. Here, we used it to color the top of the mesa, but it can be used for anything else:



YDepressionPainter

This is the counterpart of the previous: create depressions in the map. In the same way, the **flat** member is a points array holding only the flat part of the created depression.
In the same way as above, **centerHeight** should be lower than the region to modify, and the limit too.



We'll see in the third part how to create your own version of this kind of painter if you want to merge the maps in another way.

Global map generation.

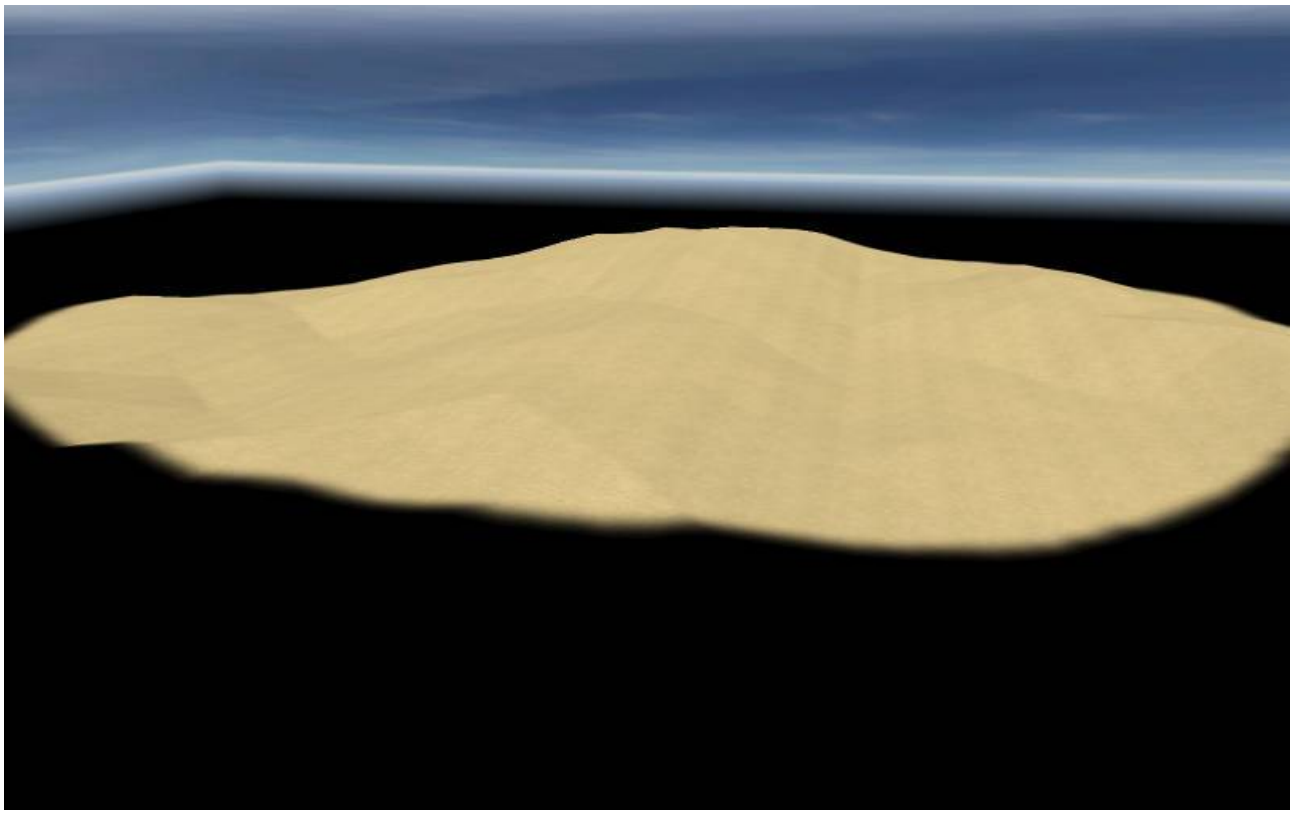
Here we describe a procedure to create a complete height map at the beginning of a script. The idea is to use the algorithm to create maps of different kinds and merge them in order to have finally rather precise ratio of flat terrain, cliffs, underwater and so on.
To make things easier, we shall use a special object:

```
var hMap = new HeightArray(mapSize);
let mSize = hMap.mSize;
```

The reason is the algorithm works only on square maps whose size is a power of two. This explains why the map created is mSize, sometimes larger than mapSize which is the size of g_Map. Then we shall first create a rather flat map which shall the base of our design.

```
let tRough = 0.4;
let tProgress = 64;
let tRelief = (MAX_HEIGHT * mapSize) / 600;
let baselevel = 50;

hMap.tMap[0][0] = baselevel;
hMap.tMap[hMap.mSize][0] =  baselevel;
hMap.tMap[0][hMap.mSize] =  baselevel;
hMap.tMap[hMap.mSize][hMap.mSize] =  baselevel;
hMap.tMap[hMap.mSize/2][hMap.mSize/2] =  baselevel;

hMap.createAltitudes(0, this.mSize, 0, this.mSize,tRelief,tRough,tProgress);
```

This call uses exactly the same parameters as fractal painters above. The only difference is it takes the corners of the map to create as parameters. When done, we transfer this map to g_Map. The map is cropped if necessary to fit into g_Map height. That's why the method `hMap.finishMorphing();` exists. Then we have a moderately bumped map:



Now we want to cover 15% of this map with water and 20% with mountains. To do that we shall first compute a height histogram, i.e. an array holding the number of points having the same height. The code here counts only the points in the circular part of the map if needed:

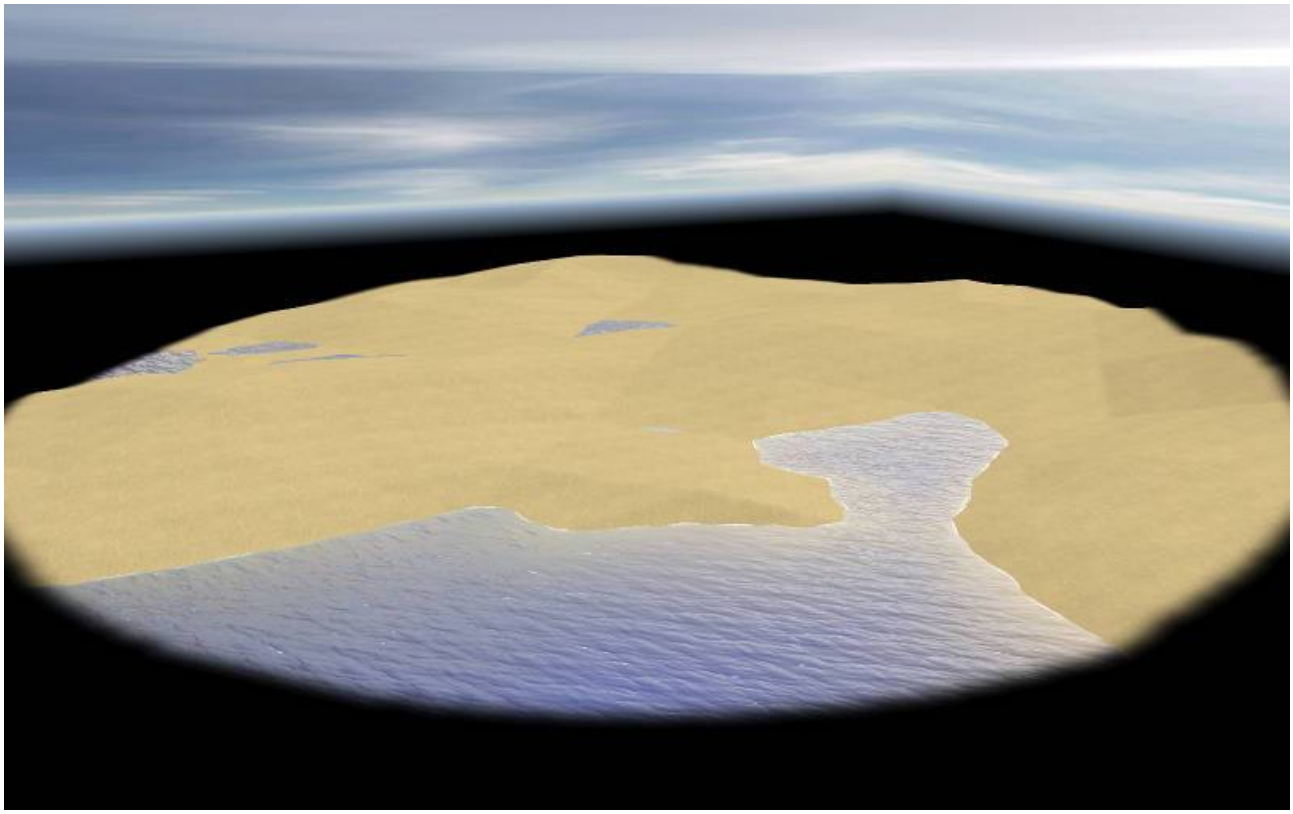
```
for (let i = 0; i < yHISTOSIZE; i++) // histogram initialization
	hMap.histo[i] = 0;

if (g_MapSettings.CircularMap) {

	var rad2 = mapSize * mapSize / 4;
	for (let i = 0; i < mapSize; i++)
	{
		let d1 = i - (mapSize / 2);
		d1 = d1 * d1;
		for (let j = 0; j < mapSize; j++)
		{
			let d2 = j - (mapSize / 2);
			d2 = d2 * d2;
			if((d1 + d2) <= rad2) {
				let idx = Math.round(g_Map.height[i][j] * 10);
				hMap.histo[idx]++;
			}
		}
	}
} else {
	for (let i = 0; i < mapSize; i++)
	{
		for (let j = 0; j < mapSize; j++)
		{
			let idx = Math.round(g_Map.height[i][j] * 10);
```

```
                        hMap.histo[idx]++;
                    }
                }
        }
```

When done, we sum the content of the array from height 0 until we have the desired number of points: **limit**. The index will give us the water height to use.

```
let count = 0;
let i = 0;
if (g_MapSettings.CircularMap) {
        var limit = rad2 * Math.PI * (15 / 100);
} else {
        var limit = mapSize * mapSize * (15 / 100);
}
while ((count < limit) && (i < yHISTOSIZE))
        count += hMap.histo[i++];

let waterHeight = i / 10;
setWaterHeight(waterHeight - MIN_HEIGHT);
```

Finally we get this:



Setting the mountains now. First we create a new height map with rough terrain parameters. We need first to reinitialize the **hMap** object first. An important feature of the algorithm is it doesn't modify any already defined height, or in other words, any height != UNDEFALT which is a negative constant set to an impossible value. That's why we need to reinitialize the **tMap**. We don't use the feature here, but you can call **createAltitudes** one more time (with different parameters) on the same map, after setting some points of the map to UNDEFALT. Only those points will be modified by the new call. Back now to our work:

```
hMap.tMap = [];

for (let i = 0; i <= hMap.mSize; i++)
{
        hMap.tMap[i] = new Float32Array(hMap.mSize + 1);
        for (let j = 0; j <= hMap.mSize; j++)
        {
                hMap.tMap[i][j] = UNDEFALT;
        }
}

let tRough = 0.8;
let tProgress = 64;
let tRelief = (MAX_HEIGHT * mapSize) / 450;

let hmin = baselevel;
let hmax = baselevel * 1.5;

hMap.tMap[0][0] = randIntExclusive(hmin,hmax);
hMap.tMap[hMap.mSize][0] =  randIntExclusive(hmin,hmax);
hMap.tMap[0][hMap.mSize] =  randIntExclusive(hmin,hmax);
hMap.tMap[hMap.mSize][hMap.mSize] =  randIntExclusive(hmin,hmax);
hMap.tMap[hMap.mSize/2][hMap.mSize/2] =  baselevel * 1.5;

hMap.createAltitudes(0, hMap.mSize, 0, hMap.mSize,tRelief,tRough,tProgress);
```

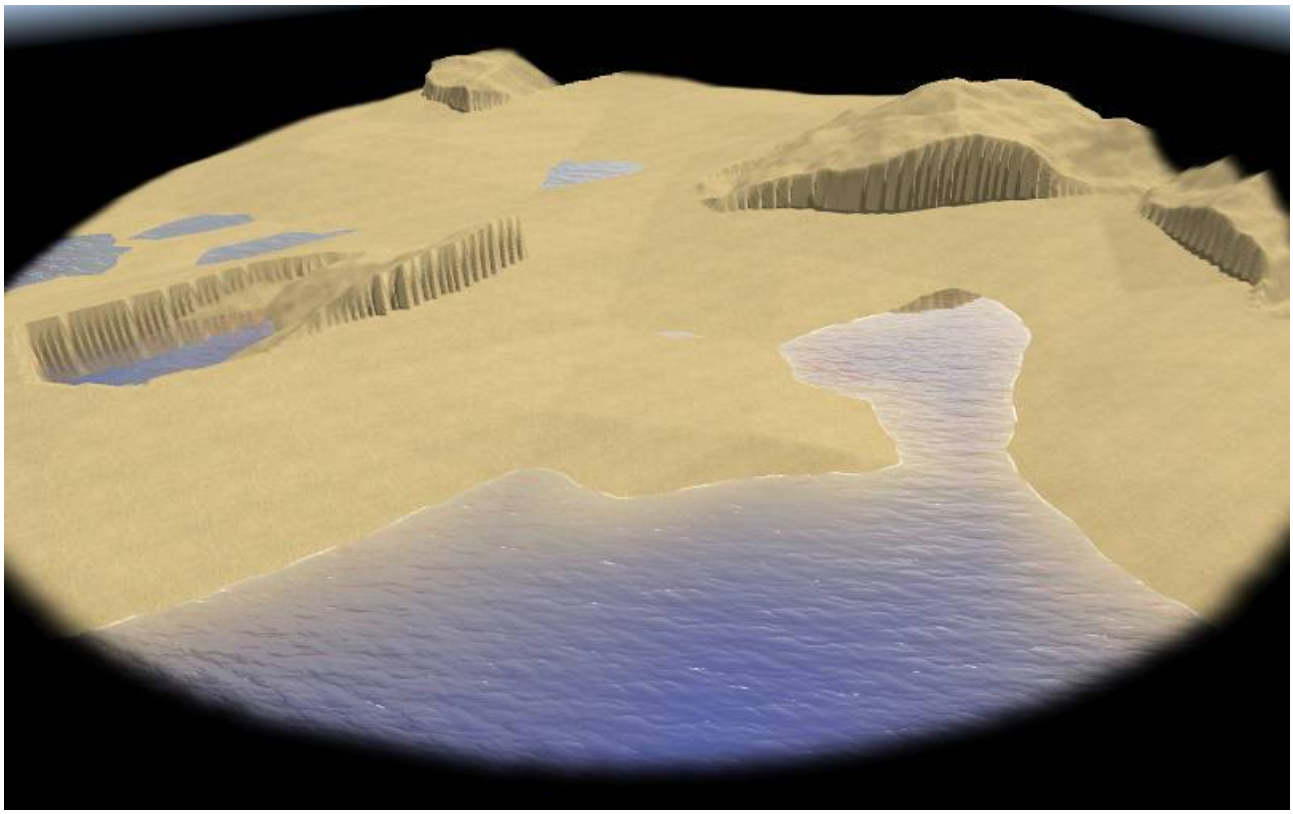
Now we have to compute the limit of the mountains. We must leave untouched 80% of the map so the limit is:

```
if (g_MapSettings.CircularMap) {
        limit = rad2 * Math.PI * (80 / 100);
} else {
        limit = mapSize * mapSize * (80 / 100);
}
while ((count < limit) && (i < yHISTOSIZE))
        count += hMap.histo[i++];
var peakHeigth = i / 10;
```

Now we can merge the two maps:

```
let off = (hMap.mSize - mapSize) / 2;

for (let i = 0; i < mapSize; i++)
{
        for (let j = 0; j < mapSize; j++)
        {
                if(g_Map.height[i][j] > peakHeigth) {
                        g_Map.height[i][j] = hMap.tMap[i+off][j+off];
                }
        }
}
```

which gives this:



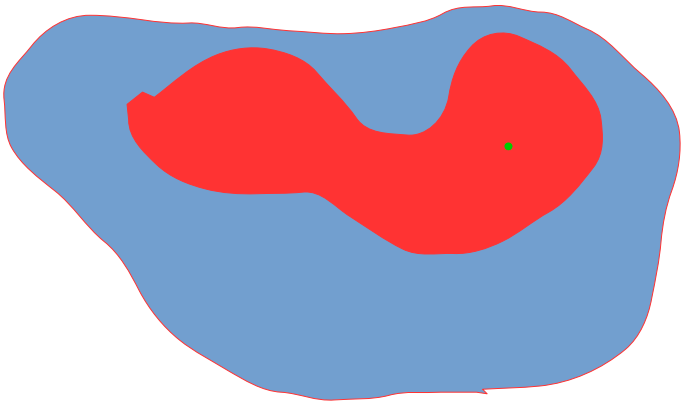
There are many ways to merge the two maps (this one looks a bit strange). You can find two different ones in the library: **fullMap** and **newMap**, but you can create your, using those as a template, and even merging more than two maps if needed.

Regions.

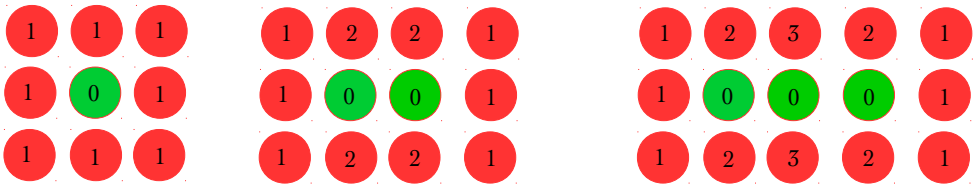
Regions are an important part of map making. We need them to define area to paint or where to install buildings or scatter units. There are many ways to define them and of course, it's more realistic when they are aware of relief and terrain types. Realistic here means the result looks realistic, but the functions are not based upon physics or geometry. It's only a crude approximation which works rather well.
Here we'll describe some functions based upon a filling algorithm, in other words a function originally designed to fill a picture region with a color. We'll describe the algorithm first and next the functions.

Filling a region by connectivity.

So we want to fill this red area only with green color, starting from the green point.



A way to do that can be to look at the points connected to the start position, and to change their color if and only if they are red. Next, we could repeat the process on this neighbors and crawl on the region until we find the border (points which are not red). Since we have to manage those points one by one, we must keep the others in a stack to retrieve them later. Let's take a closer look. In the picture below, we figure successive steps of the algorithm. The number in the points show how many time they are stored in the stack. One step consists in picking a point in the stack, color it to green, and store in the stack all its red neighbors. When the stack is empty, the work is done.



The picture shows the painting progresses straight to the border storing the same point more than once which is not efficient and could cause memory overflow. It would be good to make a provision to avoid storing a point more than once. More of it, running straight to the border works well if we want to fill the whole region. Then it matters not which points are colored first. But if we want to define regions, we have to find a way to control the expansion, or in other words in which order the points are colored.
First we shall use a point (or tile or cell) object holding not only its coordinates, but a boolean flag **done** too. The flag is set when a tile is stored in the array, and tiles whose flag is set will not be stored or visited again.

Next we can store the waiting points in a heap, in other words a dynamically sorted array. We have to assign every point some value enforcing the order in which they are colored or selected.

Cell object.

The Cell object in the library is an object holding not only the coordinates of the tile in the **g_Map** object. It has the **done** flag we explained earlier and some other values:

- **Slope**. An approximation of the slope of the tile (since it is generally not plane, it's rather difficult to define it rigorously).
- **Alt**: the mean value of the four corners heights.
- **terrain**: holds a string value describing roughly what kind of terrain it is: water, cliffs, flat fields etc… It's rather handy to paint the full map but not used by the filling algorithm.
- **Lock**: it's a bit field stating if the tiles are available for modifications or not. It's not always a full lock: some tiles can be forbidden to receive trees, but open to other actors for instance.
- **Iterator**: this is a function returning the neighbors of this cell (checking map boundaries). It's obvious we'll need this feature very often and it makes the code clearer: to fetch the neighbors of a tile, this is enough:

```
for(let neighbor of tile)
```

You shouldn't set directly **slope** and **alt** members since they're computed from tile corners heights (in other languages, we would have declared them private). The `update()` method does this for you.

If you want to modify the heights in **g_Map** you should consider using the mutator `setHeight(h)` which does the change and update the four cells impacted by this change. We'll see later this can be done globally, on the whole map or only a region.

If you don't want to bother with cells updating, you can use accessors `getSlope()` and `getAlt()` which always return correct values, but the features described later work correctly only if cells are correctly updated.

These object are created and stored in a two dimensional array in the same way **g_Map.height** stores heights. This cell map must be created before using the functions described here.

```
var g_TOMap = createTileMap(waterHeight + 0.5,snowLimit);
```

Uses **waterHeight** you probably defined earlier. **snowLimit** can be set to any value greater than MAX_HEIGHT if you don't use it. There's no need to fill or update the cell array, `createTileMap()` does all this for you, but it would be better to create it only when the height map is near finished to avoid too many updates.

PatchPlacer.

The place method.

The **PatchPlacer** is a placer using the filling technique. The value (or key) allowing to sort tiles is a combination of four factors:

- tile slope, flat tiles are preferred to steep ones.

- height difference with the starting tile. Little differences are preferred.
- noise, a random factor can be added.
- distance from the starting tile. Actually it's not exactly a distance. It's adding to current tile key the key of it's previous tile, so it's in some ways the number of tiles visited to hit the current tile.

These four values are given a weight specified in the parameters, allowing to give them more or less influence (or even none). Alone, this factors have the following effect:

- **slopeRatio**: the expansion avoid steep slopes as much as possible.
- **altRatio**: the expansion avoid tiles too high or too low from the starting cell, which is not exactly the same as above.
- **noise**: the expansion is random. This affects mainly the border of the region.
- **compact**: this one favors tiles close to the starting cell.

Additionally, its possible to set boundaries for slope (**slopemin/slopemax**) and height variation (**altmin, altmax**), and test the **lock** member of the cell against **mask** (AND). This allow to filter candidates tiles. The full parameters list is:

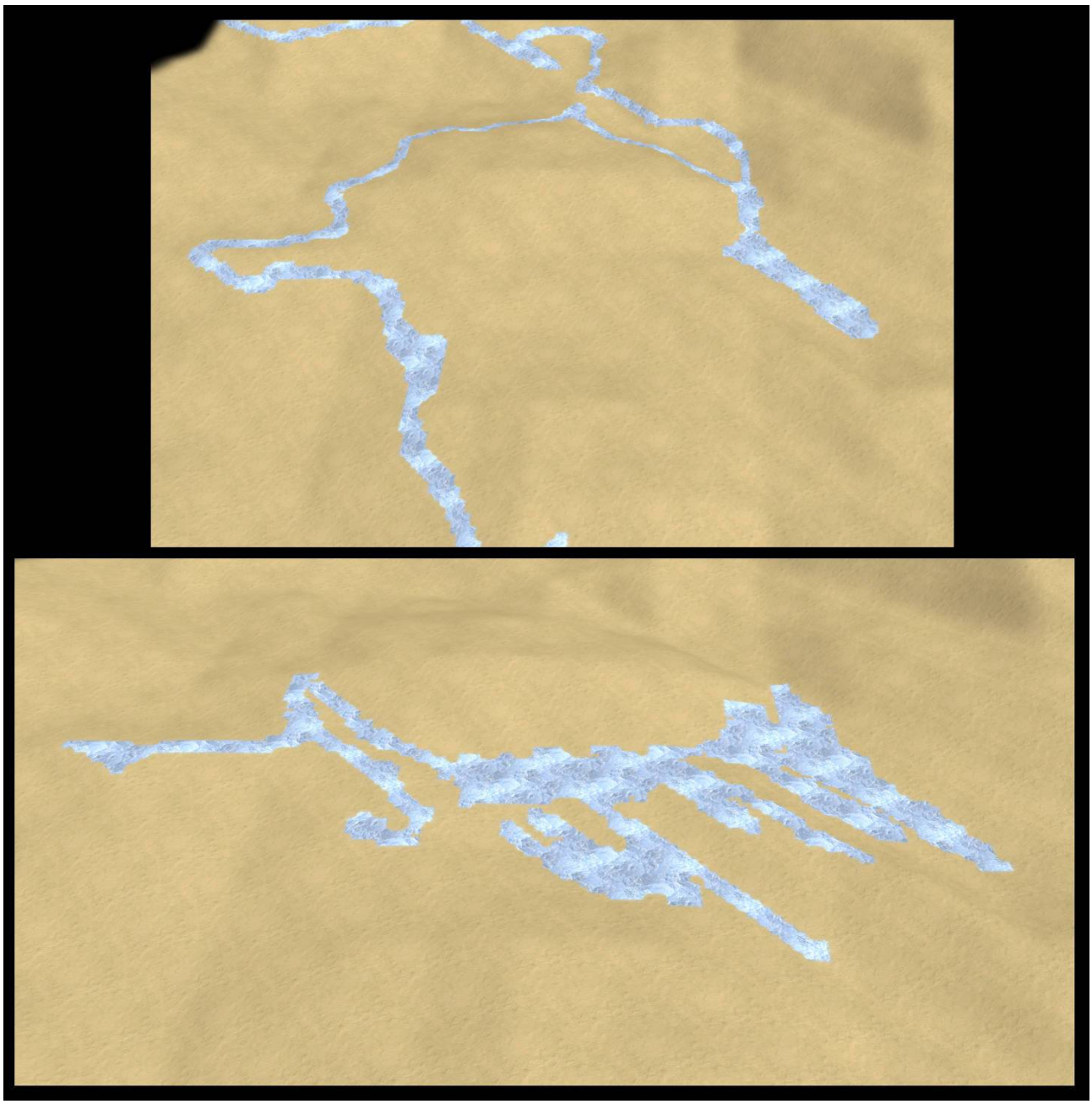
```
(startx,starty,count,slopemin,slopemax,altmin,altmax,compact,slopeRatio,altRatio,noise,mask,lock)
```

The **lock** parameter is a bit field. The selected tiles **lock** members are OR-ed with this bit field. Set it to 0 for no effect.
In general, the placer will work well if the child tiles have a key larger than their father, or more accurately if the condition:

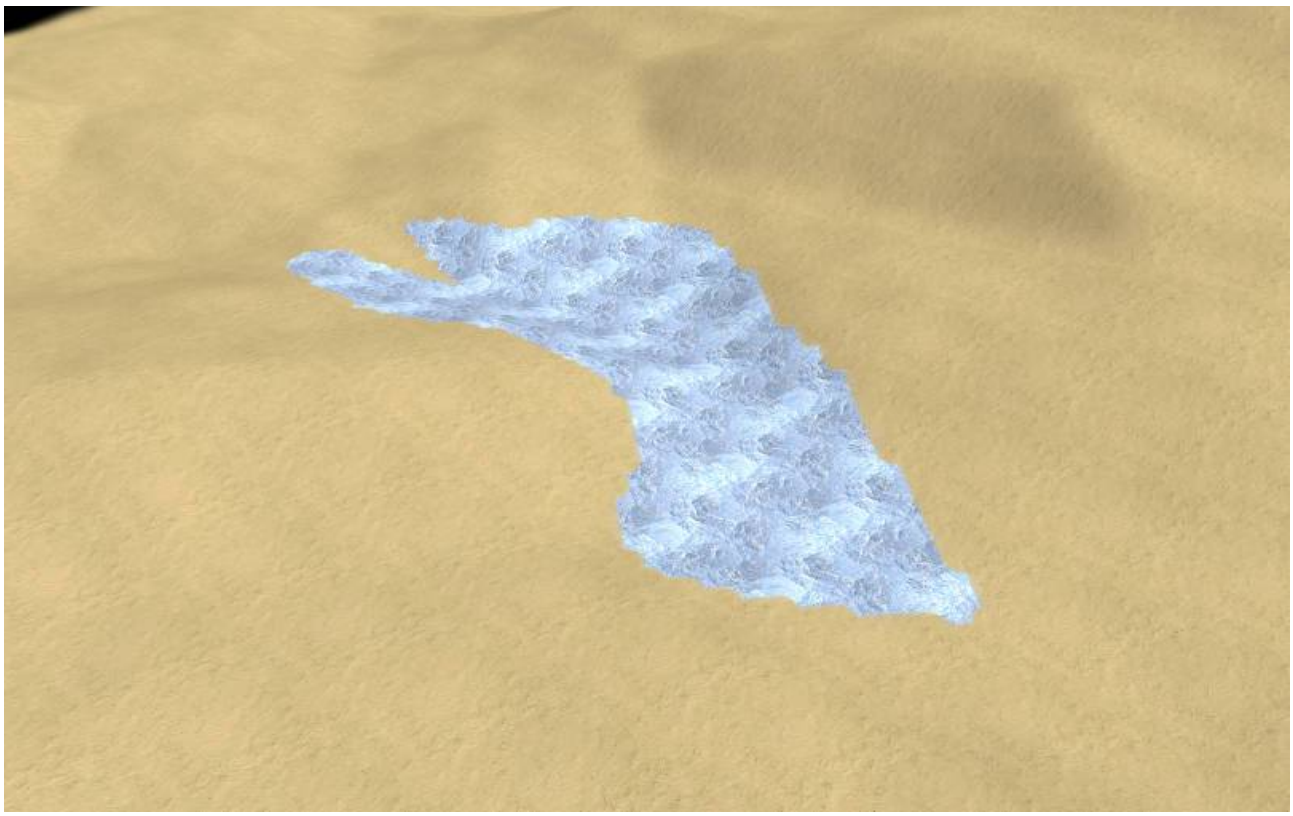
```
father_key < (father_key * compact) + (slope * slopeRatio) + (height_difference
* altRatio) + random(0,noise)
```

is always met. You can set **slopeRatio** or **altRatio** to negative values if you want to inverse their effect, but then, you should increase other settings accordingly.

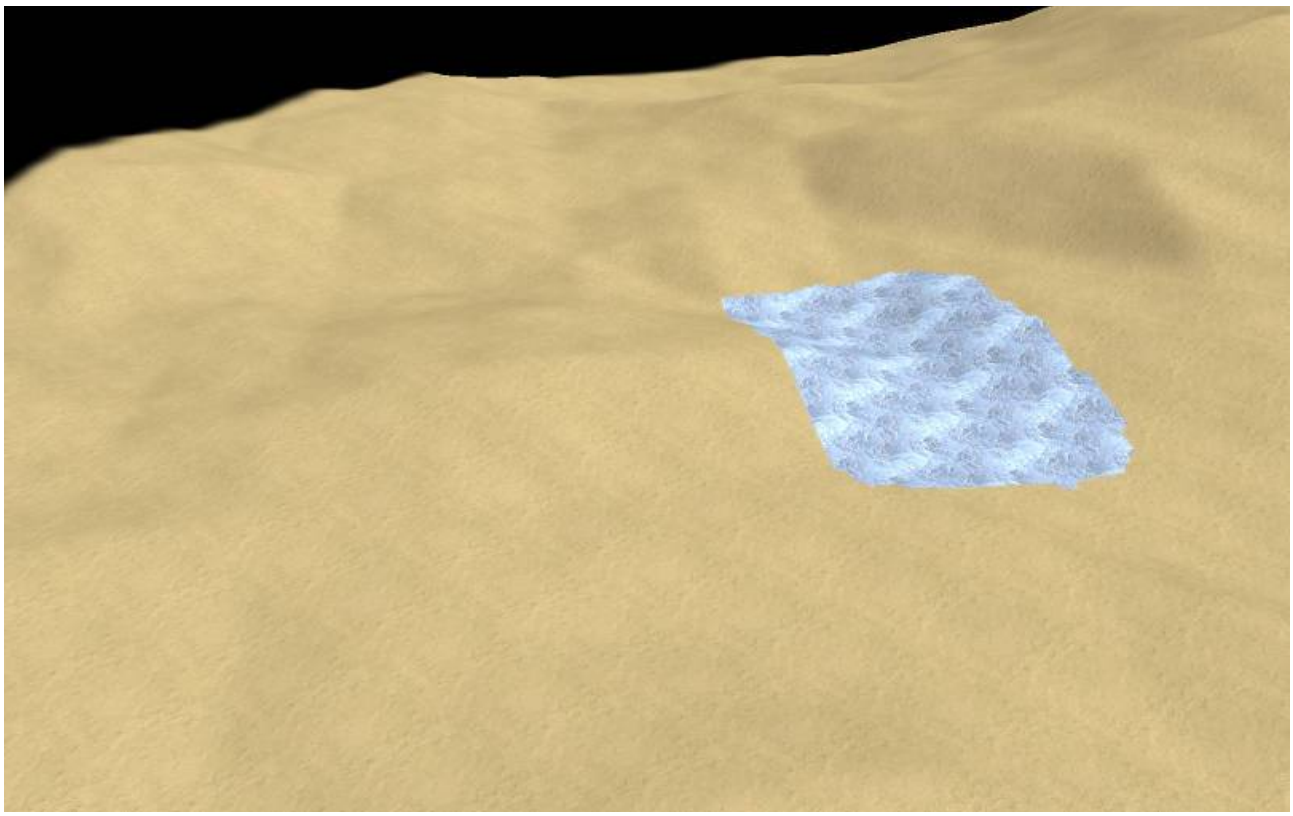
The placer works well when all tiles are given a different **key** so the sort can be effective. It's not the case on perfectly flat terrain where all tiles have slope = 0 and the same height. It's not the case if you set **compact** to something and all other control settings are 0. Then all the tiles will share the same key. This will work only if you want to fill a whole region within boundaries (then set compact to 1 and the count to a very large value).
These pictures show the region created with slope and height influence alone. Both follow the relief, but not in the same way. This show too why the compact parameter is often needed.



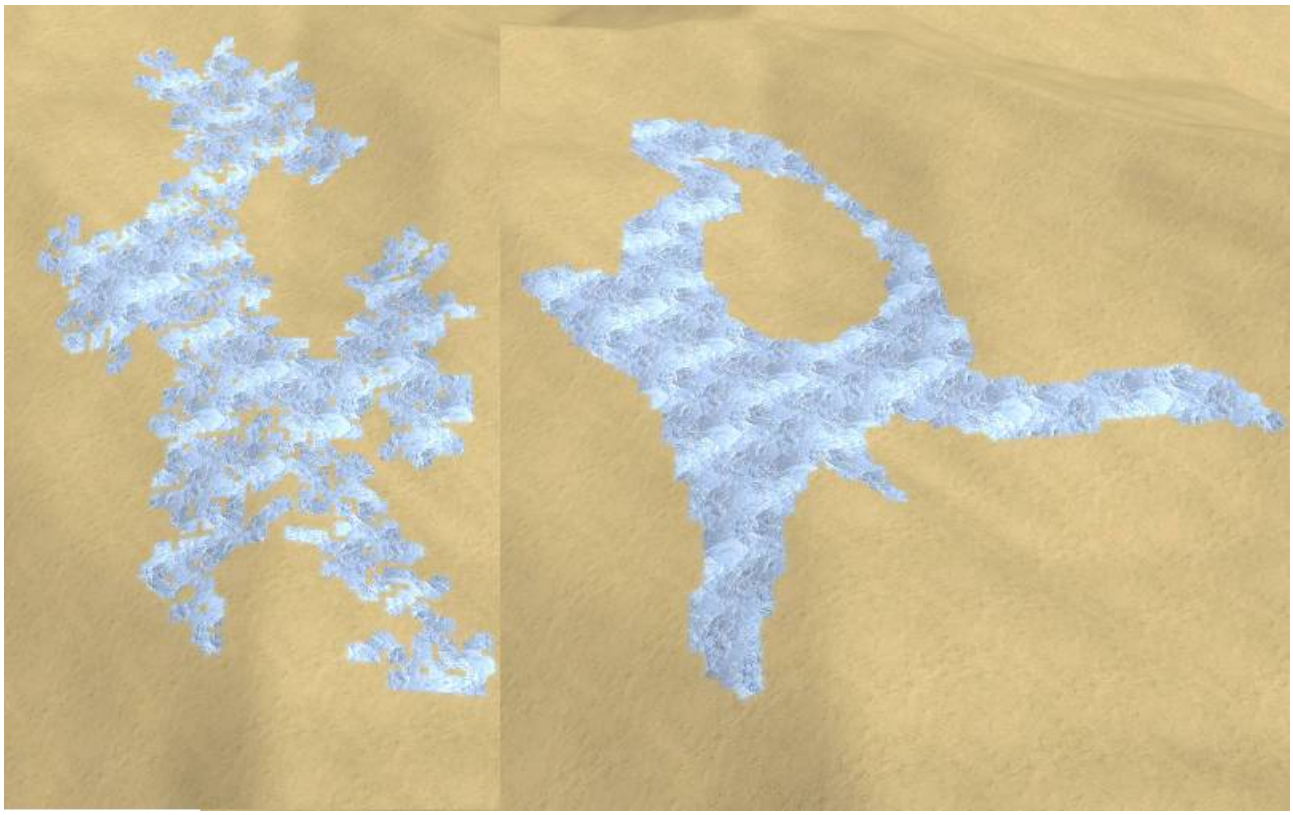
Most of time we want something like that:



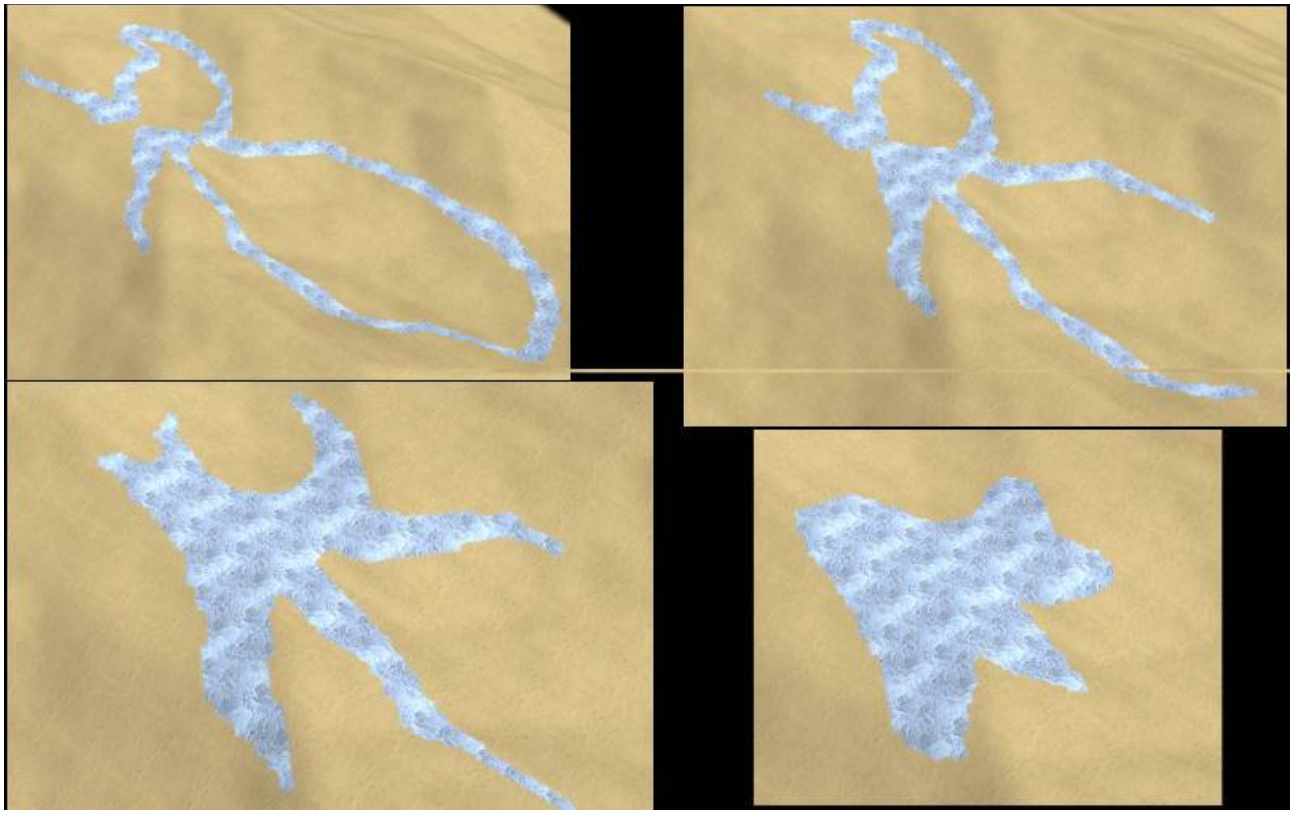
which is obtained setting **compact** to some value between 0,7 and 1,2. Large values will give regular regions like this one, which is not very interesting.



Here we can see the effect of **noise** (on the left).



Here we can see the effect of **compact** on a height driven placer. With compact = 0,8, the region follows more or less level curves. Successive values of 0,95 0,98 and 1,1 break more and more the loops and the fingers, regrouping more tiles around the starting point.



Results.

The **place()** method returns a region, as one could expect, as an array of points (coordinates pairs). If you want a Cell array, call the **find()** method instead. But results are available as well in the placer object.

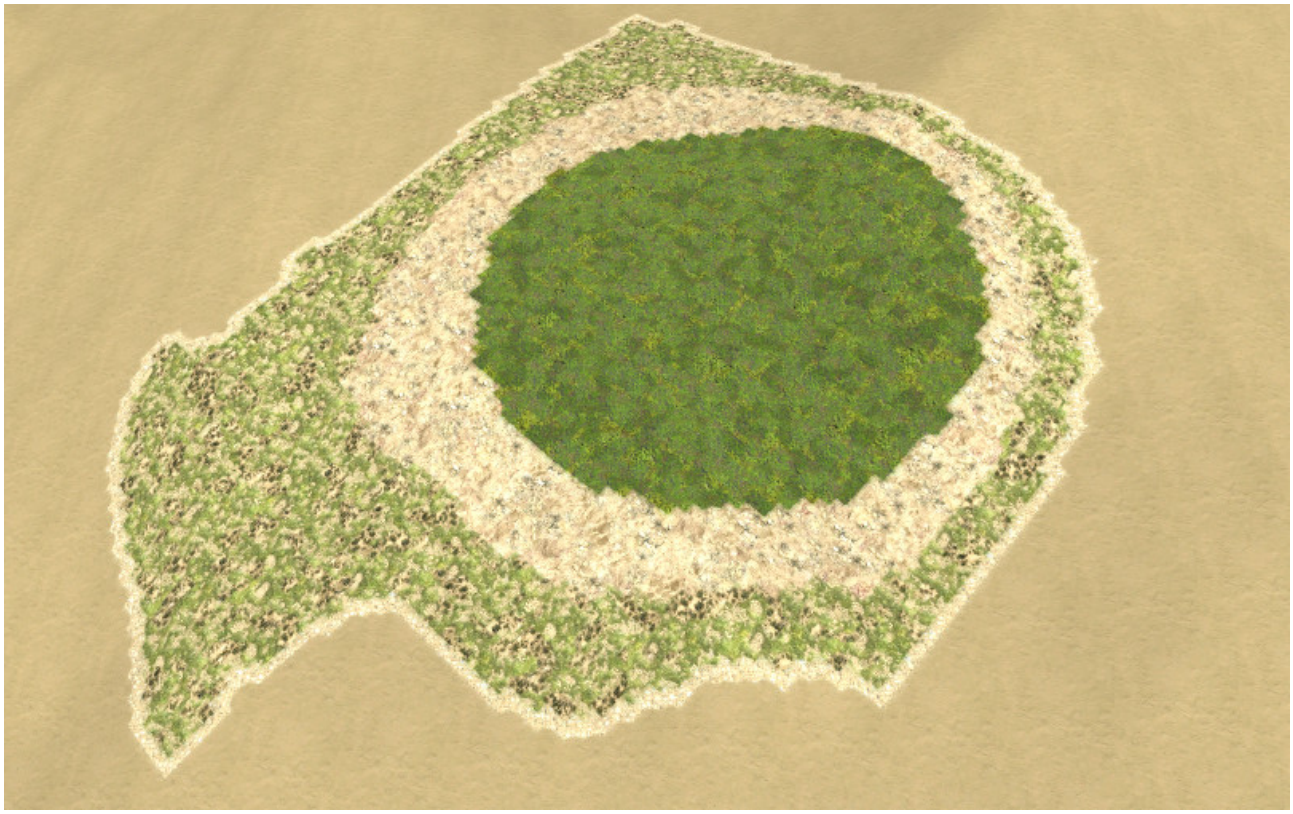
- **count**: holds the difference between the number of tiles requested and actually found.
- **card**: holds the cardinal number of the region produced.
- **barx, bary**, holds the mean value of x,y coordinates. Since the region can expand in any direction from the starting point, this one is more like the "center" of the region (not very reliable if the region looks like an octopus).
- **zone**: stores the region produced, so you need not to call **place()** again if you need it. (Cell array)
- **border**: the border of the region produced (one tile wide, Cell array).
- sets the lock bit to every tile of the region.

Note this placer will not always succeed and return the desired number of tiles, particularly if you specify too narrow constraints, and if the starting point itself doesn't match those constraints.

Region expansion.

A region defined with the patch placer can be expanded calling the method `expand()` taking two parameters: **count** = number of tiles requested, and **addFlag** which is true if you want the new tiles to be added to the current region. If not, the region returned is the ring around the former one. Border is computed as well, et and you can use other settings of the placer, modifying them directly in the object. This can be used in many way, particularly to have more control on the final region shape: you could for instance start with a rather compact region and add some "fingers" when expanding it with a **compact** lower value.
You can call **expand()** more than once and play freely with the regions and the borders created.

Here for instance, we expanded the inner region twice and raised the rings to a fixed height. Ring borders have been raised some more as well to emphasize the limits.
You can too expand a region defined in another way. The method **expandZone()** does that. It takes a region (an array of x,y coordinates or of Cell objects) and a **count** as parameter. **Count** is the number of tiles to add to the region. If 0, you get the region and it's border as a result, and you need not to set placer parameters to correct values (except **startx**, **starty** which must be 0, unless you set them to a point located in the region). They're not used if you don't want to expand the region. But you can call **expand()** next any time you want. Now note the placer will not expand correctly regions holding unconnected patches: only one patch will be expanded.
Here we started with a **Clump Placer** to create the inner region and called **expandZone()** to initialize the placer. It is next expanded twice and finally we painted the border:



Managing the 'done' flag.

For this placer to work as expected, and particularly the border feature, it is important to manage the **done** flag in the cells correctly. Cells marked as **done** are NEVER visited again by this or other placers using the heap (roads and players base search as well).
The flag should be reset with `g_TOMap.clearDoneFlag()` each time you're finished with one region and it's expansions.
If you omit this, the successive regions will not overlap which can be desired, BUT the borders will not be correct and the **expand()** method which rely on them will expand only from some parts of the border. If you want both borders and not overlapping regions, you should use a **mask** and a **lock** set to **yPATCHLOCK**, which is a constant reserved for this use. To clear both this **lock** and the **done** flag, you can call `g_TOMap.clearDoneAndLock(yPATCHLOCK)`.

Cell map methods

To update the whole map after modifying some heights you can use `recalcAll()`. If only on a region, calling `update()` on all region cells is enough.

`clearDoneFlag()`

`clearDoneAndLock(lock)`

As we have seen earlier, these two clear the done flag and eventually a bit mask on the whole map.

`putRandFlock(zone,count,card,retry,entity,placer)`

Uses a patchplacer to create little flocks of card entities on zone. Count is the total number of desired entities.

`putForestChunks(zone,count,card,retry,trees,floor,placer)`

This one is more dedicated to set up forests. It uses a placer to put count entities in packs of max 'card'. Entities are chosen randomly in the 'trees' array. Floor is the forest floor terrain. Retry is a

counter to avoid endless loop if placement is impossible (too much entities on a zone for instance).

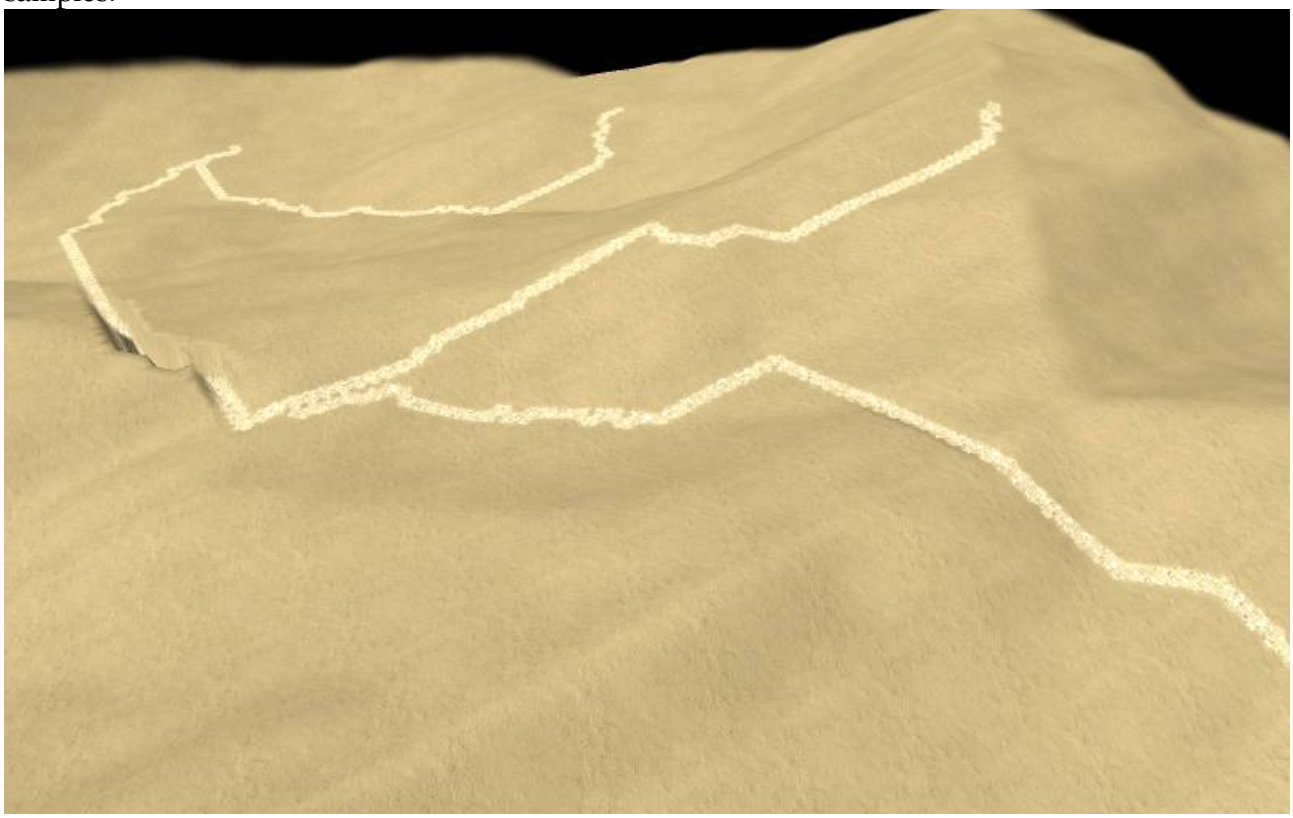
More features

Make roads.

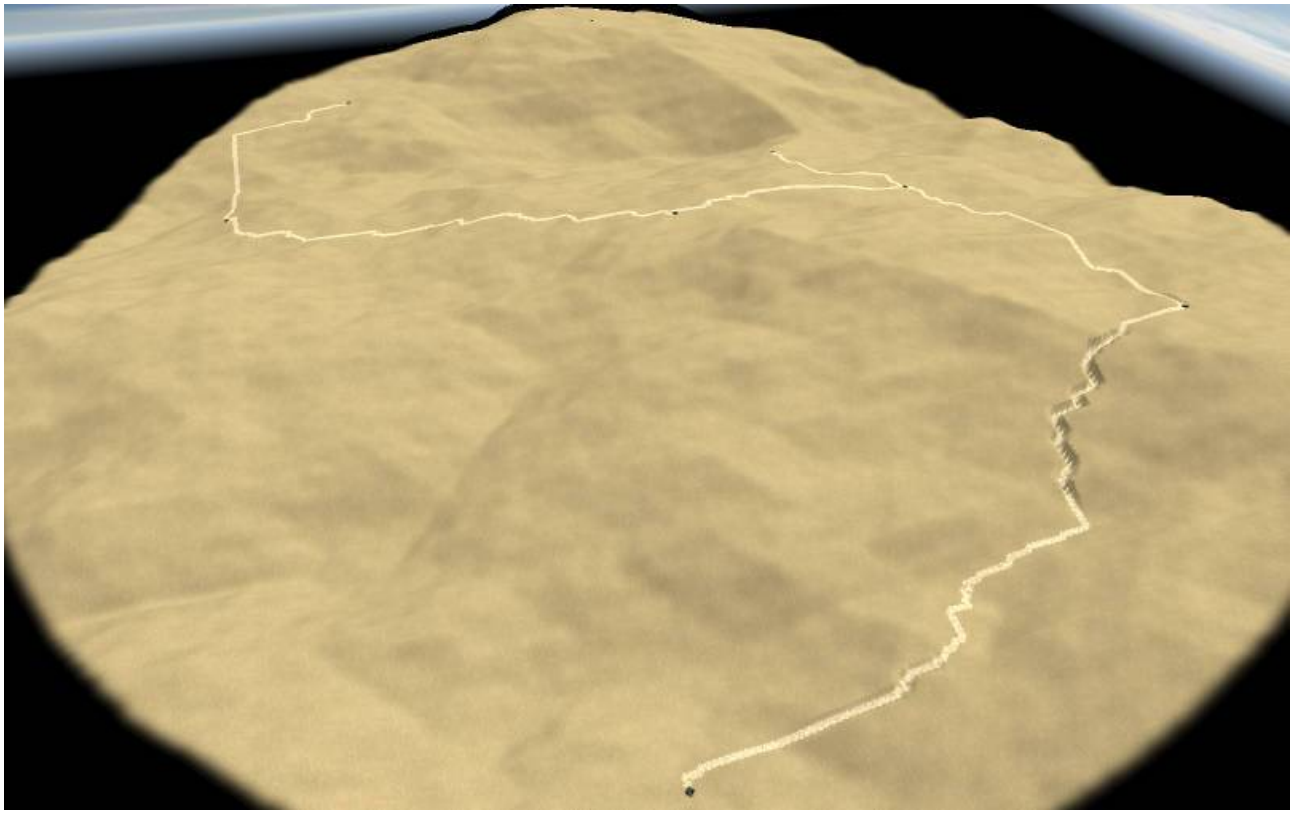
This one is a Cell map method creating roads between all the cells given in the parameter **endpoints**. It uses the filling algorithm we described earlier but with some modifications. It will explore all the possible paths from the first endpoint, but the faster ones will be fetched first. Additionally, all visited tiles are given a reference to the tile from which they are visited. When another endpoint is reached, this references chain is used to create the road.
What means faster ones ? This is computed in the key member of the cell. As with real roads, we shall prefer smooth roads even if they're longer than more straight ones. The cell crossing cost is a function of slope, height variation, and suffer a special penalty when going underwater or in the mountains. More of it, too deep water, too high mountains, too steep slopes are eliminated.
You can tune all this with the function parameters:

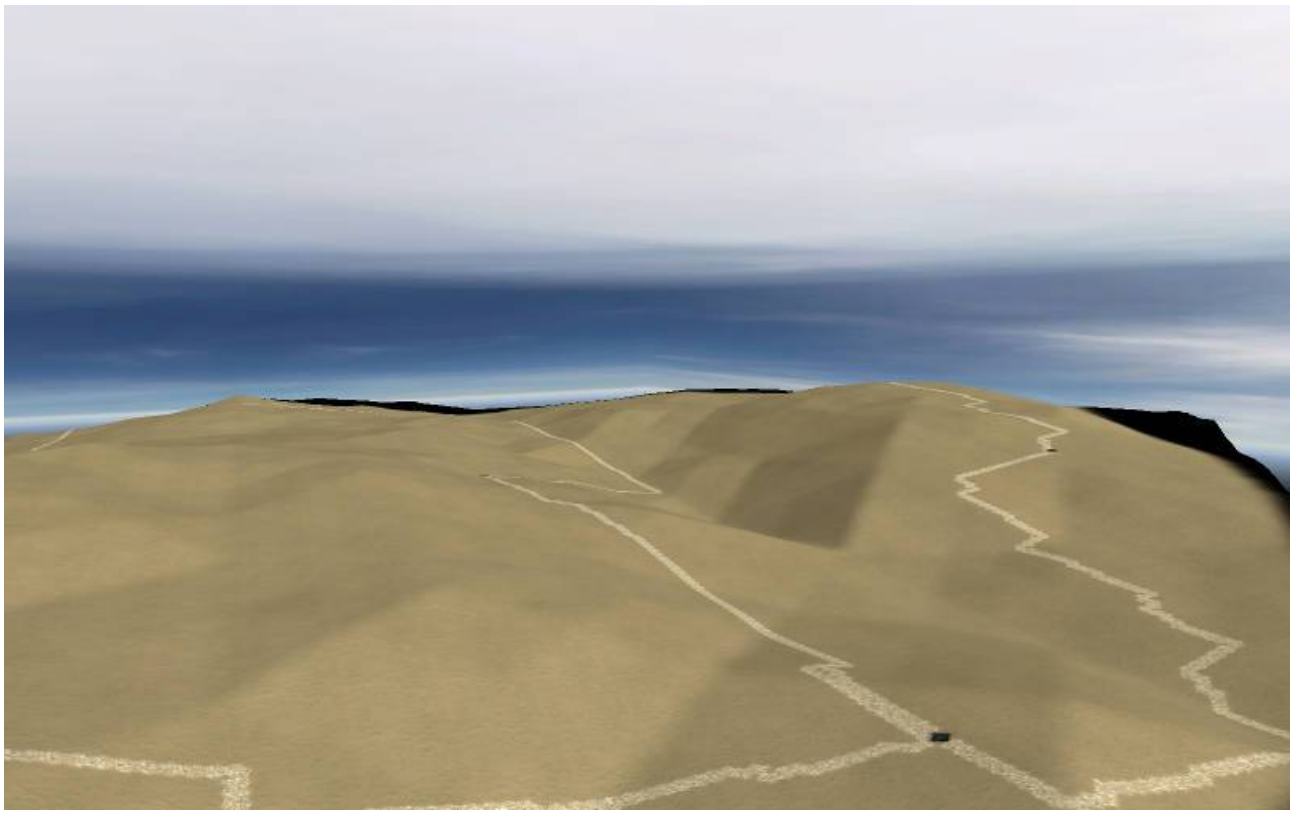
- **endpoints**: you can put any number of points, not only players bases which are obvious candidates.
- Heights: **hWater** = roads can't go lower, **hMini** = limit between flat land and mountains, **hMiddle** = upper limit, the roads can't go higher.
- **mPenalty**, a positive number, the penalty to add to cells higher than **hMini**. The greater it is, the more roads will avoid mountains.
- **wPenalty**, a positive number, the penalty to add to cells lower than **waterHeight** (fords).
- **noise**, a positive number to make the roads more irregular.

Roads can dig in the mountains to make their way through a too steep terrain. Here are some samples:



In this one, the endpoints are painted in black to show more clearly how it works:



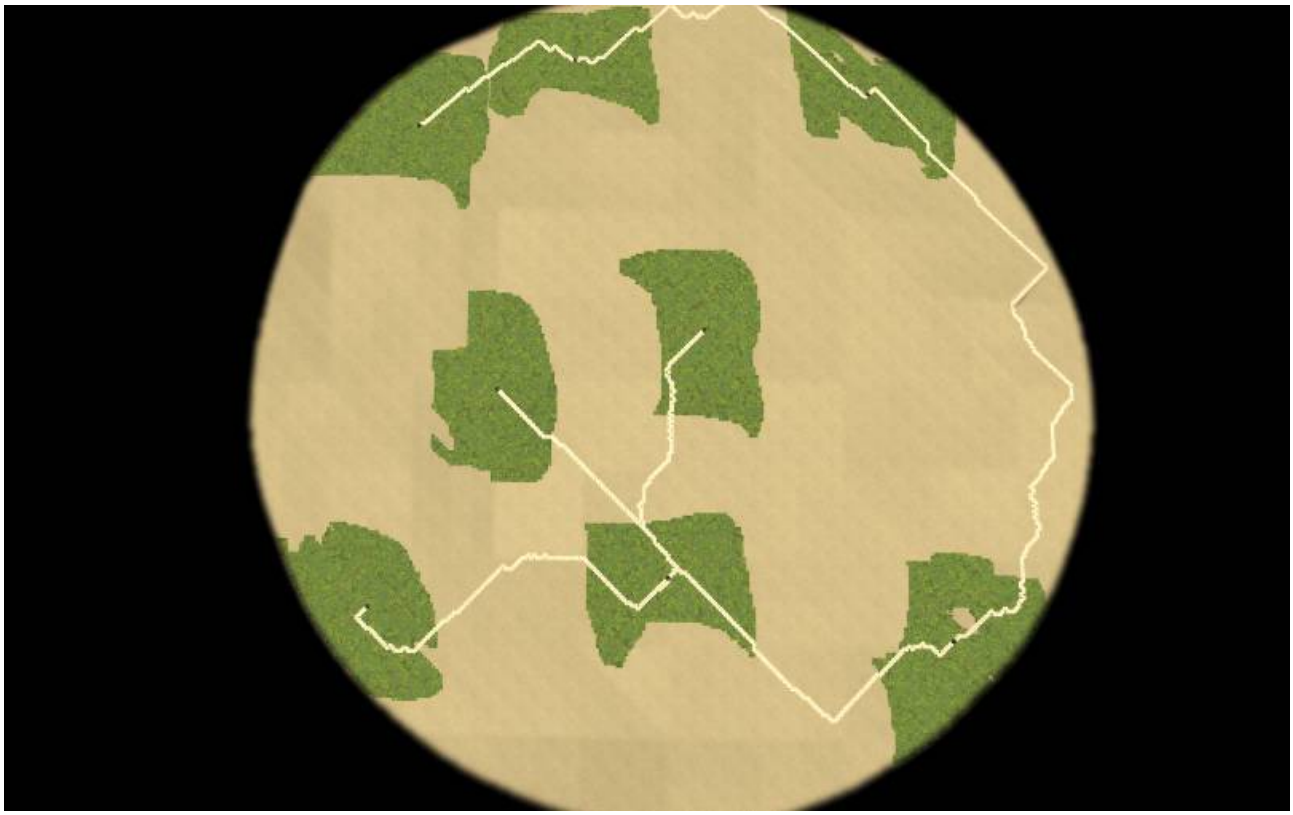


Players bases.

This function tries to find players bases or more accurately, rather flat regions of 500 + (mapSize * 6) tiles as far as possible from other bases. It uses patch placers to do that. Not optimal maybe…
It returns two things:

- **fields**: a patchplacers array holding all the regions satisfying to the 'rather flat' condition but not retained at last.
- **patches**: It's not a region array but a patchplacers array. Of course you can get the region in the placer zone member. But you may want to reuse the placer, and its **barx,bary** coordinates is most probably a good place for the CC.

The sample below shows the result. Green patches are the selected regions for each player, and CC position is painted in black. Players always can build in those regions. If the terrain is chaotic, bases may be teared, which can be desired or not.



Custom painters and placers.

Here we show how to create different placers and painters to better fit your needs using inheritance. It avoid replicating code and understand the whole thing.

Fractal painter.

The part which can be modified is the two maps merging. It is done in the **paint()** method. It calls first the **maps()** method of the abstract painter which creates the temporary map and compute the offsets:

```
let res = this.maps();
let depx = res[0];
let depy = res[1];
```

When done, merging the two maps is done in a loop:

```
	for (let pt of this.region)
	{
		let xg = pt.x - depx;
		let yg = pt.y - depy;
		g_Map.height[pt.x][pt.y] = this.tMap[xg][yg];
	}
```

Here, we replace each height of the region with the height in the temporary map. But we could wish something else, for instance use instead the mean value of the two height or anything else:

```
g_Map.height[pt.x][pt.y] = (g_Map.height[pt.x][pt.y] + this.tMap[xg][yg]) / 2;
```

To do this, we define a new fractal painter. The blue parts are Javascript voodoo and need not to be changed except the new painter name (in red):

```
function MyFractalPainter(area,centerHeight,bump,rough,progress) {
	AbstractFractalPainter.call(this,area,centerHeight,bump,rough,progress);
}
MyFractalPainter.prototype = Object.create(AbstractFractalPainter.prototype);

MyFractalPainter.prototype.paint = function() {
	let res = this.maps();
	let depx = res[0];
	let depy = res[1];

	for (let pt of this.region)
	{
		let xg = pt.x - depx;
		let yg = pt.y - depy;
		g_Map.height[pt.x][pt.y] = (g_Map.height[pt.x][pt.y] + this.tMap[xg][yg]) / 2;	}
}
```

Patch Placer.

Same with the Patch Placer. Here, we can play with the filling algorithm using different filters and key computing:

```
		for(let neigh of it) {
			if((neigh.slope < this.slopemax) && (neigh.slope >= this.slopemin) &&
(neigh.alt < this.altmax) && (neigh.alt >= this.altmin) && !(this.mask & neigh.lock)) // test if the
neigh cell is a good candidate.
{
				if(!neigh.done) {
					neigh.key = (it.key * this.compact) + (neigh.slope *
this.slopeRatio) + this.altRatio * Math.abs(neigh.alt - this.base) + randIntInclusive(0,this.noise);
// compute the key value
				    neigh.lock |= this.lock;
					g_TOMap.heap.addCell(neigh);
				}
			} else if(!neigh.done)
				cc = true;
		}
		if(cc)
			this.border.push(it);
```

The **if** line tests if the cell can be included in the region or not. You can change this to anything you want, but you must keep the `!(this.mask & neigh.lock)` test to ensure correct behavior of **expand()** method.
The value of **neigh.key** can be anything, provided it will finally have a value > **it.key**, his ancestor. If not, the algorithm will not crash but give lines more than regions.

Here is the code to inherit from the abstract Patch Placer, defining the **explore()** method of the placer. The green parts are those which can be changed safely:

```
function MyPatchPlacer
(startx,starty,count,slopemin,slopemax,altmin,altmax,compact,slopeRatio,altRatio,noise,mask,lock) {

YAbstractPatchPlacer.call(this,startx,starty,count,slopemin,slopemax,altmin,altmax,compact,slopeRati
o,altRatio,noise,mask,lock);
}
MyPatchPlacer.prototype = Object.create(YAbstractPatchPlacer.prototype);

MyPatchPlacer.prototype.explore = function() {
	while((g_TOMap.heap.getSize() > 0) && (this.count-- > 0)) {
		let it = g_TOMap.heap.pickCell();
		if(it == undefined)
			break;
		this.zone.push(it);
		this.barx += it.x; this.bary += it.y;
		this.card++;
		let cc = false;
		for(let neigh of it) {
			if((neigh.slope < this.slopemax) && (neigh.slope >= this.slopemin) &&
(neigh.alt < this.altmax) && (neigh.alt >= this.altmin) && !(this.mask & neigh.lock)) {
				if(!neigh.done) {
					neigh.key = (it.key * this.compact) + (neigh.slope *
this.slopeRatio) + this.altRatio * Math.abs(neigh.alt - this.base) + randIntInclusive(0,this.noise);
				    neigh.lock |= this.lock;
					g_TOMap.heap.addCell(neigh);
				}
			} else if(!neigh.done)
				cc = true;
		}
		if(cc)
			this.border.push(it);
	}
}
```

Have fun !