A map management for games and simulators.

Chapter 1

Presentation

1.1 Need to fulfill

1.1.1 Present constraints

Many applications having on purpose to be realistic in an landscape simulation must, for example, remember a height and some elements about each square, or any-shaped, part of the map. It is then confronted to these three limitations :

Macro-limit Each map must be severely bounded. A twice bigger map demand twice the memory to be stored.

Micro-limit The size of the smallest part of the map must be fixed. The lower it is, the more memory is needed. The higher it is, the more the map let appear repetitive schemes and *textures*.

Human time Maps are often generated with some random algorithms before being completed and perfected by human work.

1.1.2 Present solutions

These limitations didn't avoid simulators and game to be complete and every one did success to make it's work within these bounds. They are not too constraining and let people create enough.

For example of what people do with these restrictions, a flight simulator should use a really big map of, let's say, some square kilometers. Anyway, in this map, every atomic square is of some meters. When viewed from the sky, these are so tiny that the repetitive schemes can be ignored easily. On the other side, a pedestrian simulator should provide more precision for the ground composition. Some hill over here, some hole there, the street defaults, ... So, the way to do it is to have maps that doesn't expand to one kilometer away.

1.1.3 Remaining constraints

So we can see that ground-map generation is widely different for a flight simulator or a pedestrian first-person simulator.

On the other hand, everybody know that a game lifetime[1] depends on the size of the world to explore. Here, to make bigger the lifetime of the game, we're compelled to work more to create new maps. To keep on this idea, players are often frustrated when they run along a street and notice that every street getting away the main are dead ends and they are enclosed in a tiny place.

1.2 Local knowledge

The first point if we want a map of this size to be possible is to introduce the concept of *Terra-Map Local Knowledge*. The point here is to differentiate what *exist* and what is *known*. In extenso, we need a map that *exist* and is immutable[2] and we need to know only one portion, these that is to be drawn, analyzed or whatever.

1.2.1 TMLK usage

The *local knowledge* of any map will be applied the previously described constraints *macro-limit* and *micro-limit*. Indeed, TMLK will depend of the point of view of the player. This will demand a parallel programming that has on purpose to maintain the TMLK as near as possible of the terra-map for the player. This will be implemented with two actions :

sharpening The TMLK will be corrected so that points near to the player will become precise enough in order for him not to be annoyed by map pixelisation problems, we should so have nearest parts kept sharp enough.

fuzzing In order not to bypass the *micro-limit*, we're compelled to forget the part of the map we don't need to be drawn. We should so have further parts fuzzier.

These two actions are complementary and should be run together in order to keep a quite constant memory-usage of the TMLK.

1.2.2 TMLK maintenance

TMLK will then depend of the position of the viewer. While the player is expected to move on the map, two ways are possible to maintain the TMLK

[1] The time people will play before getting fed up

[2] Or can change with time by the way we program, but that doesn't change from white to black because we forgot the colour of a point.

up-to-date :

Triggered When the player pass through some fixed-limit, the simulation is paused for the time the TMLK is re-generated for the new viewer position.

Continuous While the player move, a separate thread fuzzy the parts that aren't need and sharpen these we need.

The best way will be a compromise between these two methods. While the player isn't expected to move continuously straight forward for the entire play time, a continuous method with a weak thread priority for the maintenance thread is enough when the player turn around the same place. On the other side, a long trip through the map will demand more sharpening/fuzzing and, then, will demand either more thread priority to maintain the TMLK properly or, easier, the game to pause a little time for the TMLK to be completely re-generated.

1.3 Solution theory

Terra-maps are introduced here. *Terra* stand for the metric name given to 10^{12}([3]). We already have some maps of earth taken by satellites with a definition of one meter per pixel; these databases take sizes counted in terra-bytes.

Terra also remember that these maps are able to size like an earth or seven if needed.

1.3.1 Random map

We can easily conceive the terra-map as a random map. When we need the height of a point to sharpen it's TMLK, we get it randomly. In order for the map to be immutable, we just have to use correctly the concept of *pseudo-random numbers* that is the only random-like behaviour we can expect from a computer. Pseudo-random algorithms give numbers that don't *seem* correlated from a *seed*. For a seed, the pseudo-random numbers so generated are always the same.

So, we can use pseudo-random generation with position of a point for seed to know its height. This need no memory and the height won't change through time.

[3] *kilo* stands for 10^3, *mega* for 10^6 and *giga* for 10^9

Chapter 2

Deeper TMLK

2.1 Chosen implementation

the Terra map I'm going to show use the *triangle* shape as atomic map part. This will allow it to be either 2D either 3D oriented. Moreover, it's the easier to implement.

A TMLK will be implemented as a set of triangles. A TMLK will so be able to fuzzy itself and will need interaction with the terra-map when it sharpens.

2.2 Abstract

A TMLK will so be, at the beginning, one or several *root triangles*. Triangles will be divided (to sharpen map) or merged (to fuzzy map) like shown in fig. 2.1.

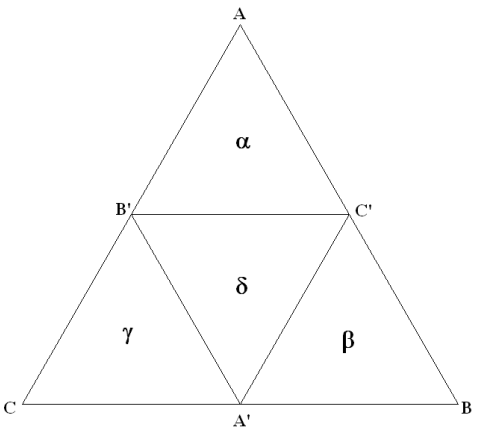


Figure 2.1: Triangle division

There, the triangle ABC is divided into four triangles : $AC'B'$, $BA'C'$, $CB'A'$ and $A'B'C'$ (respectively α, β, γ and δ). The TMLK being a tree of triangles, when this division is done, the triangle ABC doesn't need anymore to be drawn; we can so use a flag to browse the tree and draw only the *leaf triangles* but the quicker way is to maintain a list of *drawable triangles* that is modified on division/merge and used when drawn, the tree can be four times bigger to browse than the drawable triangles list.

I just remember here that division is recursive and show the naming convention I'll use for divided triangles, in the case each of the four *children* triangle is divided again.

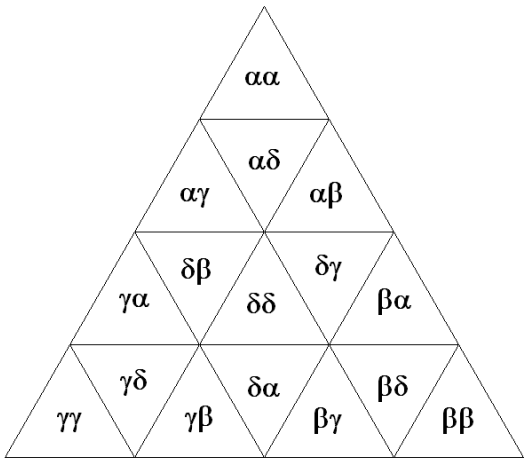


Figure 2.2: Triangle division - two generations

2.3 Sides management

While each side is shared by two triangles, the division of one triangle will lead to some side effect for its neighbours. Let's take the case where β is applied a division (fig. 2.3). The side $A'C'$ is now divided, there is two consequences :

- the triangle δ shouldn't be drawn flat. It should be a difference between the middle of $A'C'$ and the D point.
- When the triangle δ will be applied a division, the D point shouldn't be retrieved from the terra-map, the TMLK already knows it !

Note that we should assume (from the immutability hypothesis) that, if the TMLK make a redundant retrieve of the D point for triangle δ, it should receive the same point with the same height and characteristics than this retrieved for triangle β.

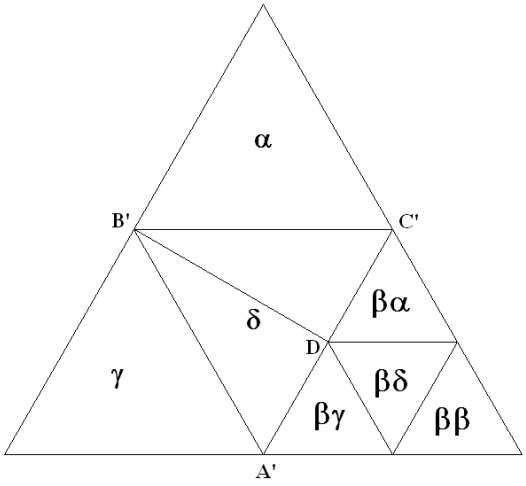


Figure 2.3: Triangle not-regular subdivision

2.4 Sharpening pragmatical view of TMLK

2.4.1 Tree view

We can so consider for efficiency a TMLK as a *tree* of sides and a *list* of drawable triangles. Each side will either be a *leaf side* and be the support of a drawable triangle, either be the parent of two other sides. Each side, except these on the *root triangles*, so the *root sides*, has one and only one brother side it can be merged with.

2.4.2 Set view

If we consider only the leaf sides, we can consider a side to support two triangles. Each triangle supported is retrievable with the corresponding third point. In our example (fig. 2.4a), the *side* AB support triangles with points C and D.

2.5 Deeper tree view

Like sawn on fig. 2.1, the sides $A'B'$, $B'C'$ and $C'A'$ are sons of no other side. This will lead to the use of another structure that will arrange the side trees. This structure should be aware of the *triangle* concept while roots $A'B'$, $B'C'$ and $C'A'$ are defined with the triangle ABC.

2.5.1 Unusual sides

That view will lead to a particular consideration about some sides. As seen in fig. 2.3, the side $B'D$ cannot be used as another common side; sides in

trees are those defining *right-side* triangles. Here, the triangles $B'C'D$ and $B'DA'$ are only component of the only drawable triangle $A'B'C'$.

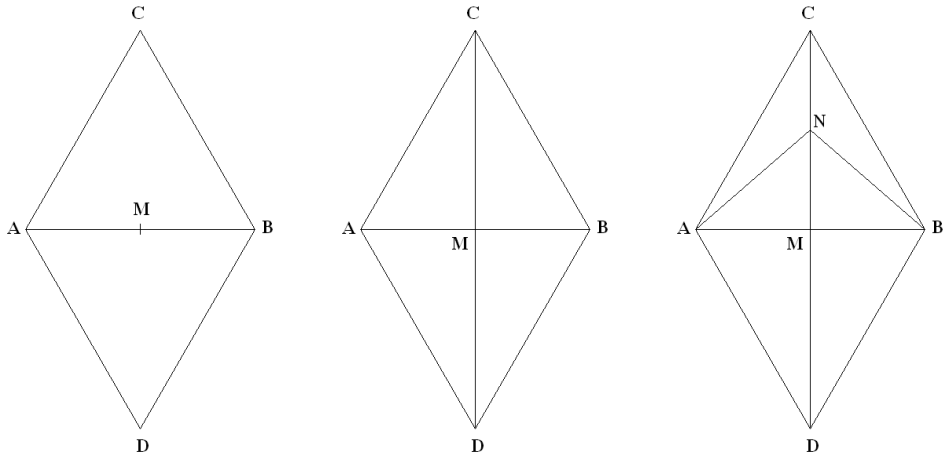
This will lead to two possible management :

- Sides of drawable triangles are supposed to be divided only once. This will demand the division of some sides if they are neighbours of other sides that need to be divided. Also, drawable triangles, when drawn, will expect four cases about their side division (zero to three sides divides)
- A drawable triangle is considered as a list of points to link with a surface and an algorithm must be written to fill triangles like common polygon.

2.5.2 Immutability of tree view

Immutability is here ensured while trees never intersect. Each sides are *right-side* triangles' side; this way, if a point is found from the division of the side defined by points A and B, it will always be found from the same division; we ca then use points A and B as seed for random.

2.6 Deeper set view



(a) Before division (b) After division (c) Second division

Figure 2.4: Side division

Notice here that every triangle is supported three times that way: triangle ABC is retrievable from each of its sides: AB, BC and CA.

What happens in our data when the side AB is divided with the point M ?

First of all, the side AB doesn't exist any more in our *sides set*. Two more sides are created: AM and MB. These sides support the same triangle as AB : with C and D. Sides AC and DA doesn't support any triangle with B but with M now, and on the same idea, sides CB and BD support now triangles with M and no more with A.

The point that'll lead to complexity is the *merge* phase. In this example, two merge ways are possible : AM with MB and CM with MD. We can specify when two sides are mergeable : when they share their supported triangles (CM and MD both support triangles with A and B)

While the search of mergeable side pairs can be exponential, the best way to manage this point will be to maintain a *mergeabilities set*.

2.6.1 Mergeabilities maintenance on division

To specify this, let's consider the case when we divide CM with the point N (fig. 2.4c).

First of all, all *mergeabilities* referring CM should be avoided. Indeed, they will be replaced, some with CN and some with NM. In this example, we only have the mergeability $\{CM, MD\}$ that is replaced with $\{NM, MD\}$. This mergeability[1] is due to the shared support of points A and B.

Added mergeabilities are : $\{CN, NM\}$ and $\{AN, NB\}$. It should sound surprising that $\{AN, NB\}$ should be merged into AB while AM and MB are still separate. This should anyway not happen while are merged sides that are too visible for the viewer. Here, sides AN and NB will always be more visible than AM and MB ! It's why AM and MB should be merged before AN and NB.

2.6.2 Mergeabilities maintenance on merge

To specify this, let's consider the case when we merge NM and MD.

First of all, ND inherit every mergeabilities from NM *and* MD. Here, the only one is $\{CN, NM\}$ that become $\{CN, ND\}$. After it, a mergeability simply disappear : $\{AM, BM\}$. Indeed, every mergeability specifying a side that goes to the point M isn't valid any more.

2.6.3 Immutability of set view

Here, points can be found from several ways; for example in fig. 2.4a, point M can be found as the result of the division of AB or of CD.

We can be sure, if we divide $B'D$ in fig. 2.3 that the middle correspond to a point that should have been found if we divided triangle $A'B'C'$ several

[1] it's the only one here but we should consider several are possible for generality

times. That way, each point can be found a unique *name* describing the path from the root sides, following only *right-side* triangles' side division, that lead to that point. After it, the way to determine point name from its parents name should be determined. There, the name of the point is unique and can be used as seed.

Also, random is used to find the difference between the middle of a side and the real position of the point; not to find the point position directly. That way, dividing in fig. 2.4a from AB or CD will lead to different side middle[2]. Therefore, another immutability process have to be found in order to use the *set view* of sides.

2.7 Division and merge decision

2.7.1 Problem specification

Let's remember the main purpose : to have the sharpest TMLK for the minimum memory usage and the minimum drawing time.

sharpest TMLK That we work on triangles or sides, sharpest TMLK means smallest *apparent size*. Here, apparent size can be only seen as a function of real size and distance from the viewer[3].

minimum memory usage and drawing time This purpose can be short cut with *the minimum number of triangles* if we work on triangles or *the minimum number of sides* if we work on sides.

2.7.2 Decision schemes

While purpose is described as two restrictions, one can be used as a fixed restriction and the other as the one to *maximise*. This will lead to two different decision schemes leading to more or less the same results...

2.7.3 Fixed objects number

We can fix the number of triangles or sides we want generated. This way, the TMLK maintenance algorithm will each time :

- Divide the most fuzzy not-divided object, the one that has the greatest apparent size, if there is enough allowed object for this.

[2]Indeed, the drawing here give the same middles, but triangle will barely not be so regular, this has no interest in natural landscape generation

[3]While distance only talk about 3D generation, this concept pass through the *projection* used. If we want a 2D global map, we use *orthogonal* projection and distance can be seen constant for any point of the map

- Merge the sharpest divided object if the maximum number of object is reached.

This decision scheme lead then to a constant activity and, when the point of view doesn't move on the map, will keep on modifying it. A good way to use it should then be to have the TMLK management process given a priority corresponding to the point of view movement.

2.7.4 Fixed precision

Can be also fixed a purpose precision. This can be done by specifying a precision margin. Here, the algorithm will only divide not-divided object that has an apparent size greater than the maximum margin and merge divided-objects that have an apparent size smaller than the minimum margin.

Margin has to be chosen wisely : larger it is, less maintenance is needed. Indeed, if margin is chosen too small, the maintenance process can enter in an infinite loop when divided objects are too sharp and, when merged give a too fuzzy object that need to be re-divided.

Also, lower is the *minimum apparent size* bound, more the memory will be used; higher is the *maximum apparent size* bound, fuzzier the TMLK will be. So, this will lead to a fixation of numbers needed to achieve some optimality.

2.7.5 Alternatives

Alternatives, or intermediate decision schemes can be chosen. At first, a specified rate between object numbers and average precision can be decided. Either by a number leading to a *"fixed objects number"*-like scheme and its infinite loop in a process for which the priority can be a function of the distance between aimed rate and actual one. Either, by a margin in which the rate is expected to remain, leading to a "boolean" process priority (there is something outside the margin or there is not).

2.7.6 Chosen one

Seeking optimality, we shouldn't search for a minimum each step of TMLK management, a margin-like process should then be used.

I will here work on 3D applications while 2D orthogonal maps doesn't demand that much processing while distance to every triangle is constant, apparent sizes can be sorted the same way than actual sizes.

A margin decision scheme can be seen as a triggered action scheme. Indeed, while an object is to be divided when its apparent size is greater than a fixed bound and while apparent size is the quotient $\frac{actual\ size}{distance\ to\ the\ point\ of\ view}$, for a not-divided object, we can compute a trigger distance under which this

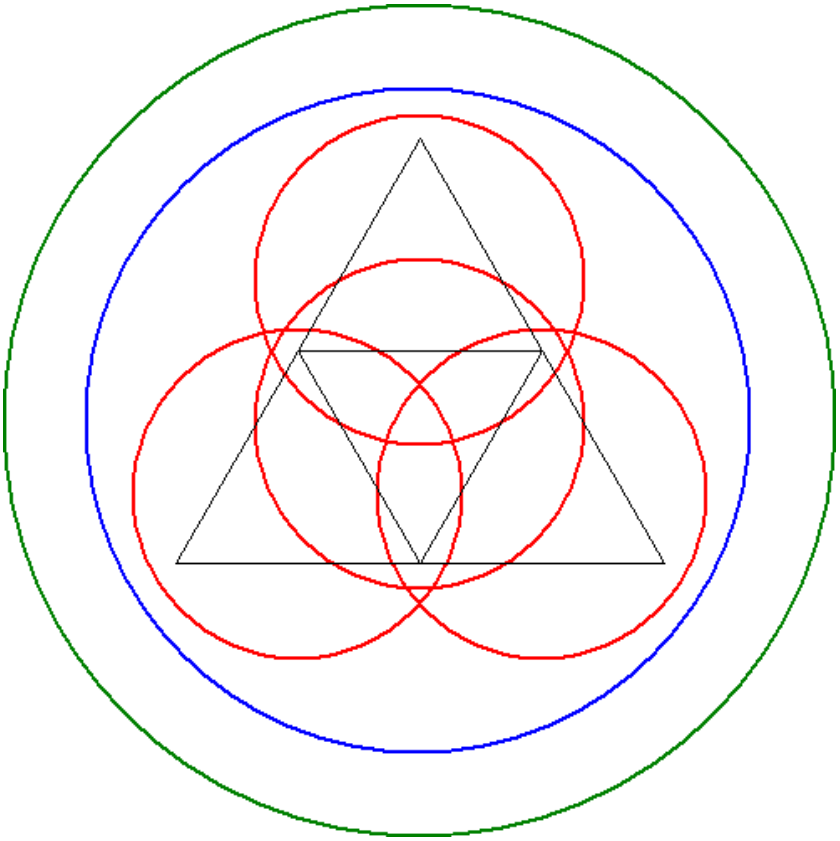


Figure 2.5: Sphere triggers of margin decision scheme

object will have to be divided. The same way, we can compute for a divided object a distance over which this object will have to merge back. In fig. 2.5, objects are triangles, the blue circle is the *division trigger distance* of the main triangle, the green circle is the *merge trigger distance* of this same triangle when this one is divided and the red circles are *division trigger distances* of the sons triangles. This is here that the choice of the margin will be influent. If margin is too small, green circle[4] can become nearer to the blue one[5]. Indeed, if margin is too small, the green circle can even get smaller than the blue one and let a zone between them where the divided triangle will have to be merged and the merged one will have to be divided.

To enhance this algorithm by setting the margin not on a *sharpness* criterion but a *sharpness / number of objects* one can lead the circles to change their size while other triangles are divided or merged; this will then not enhance the rate but will demand much more CPU usage.

[4]fixed by the maximum sharpness allowed
[5]fixed by the minimum sharpness allowed

2.7.7 Bounding boxes

Bounding box : to specify the limits for every axes : x, y and z. That let the work directly be done with comparison on the x, y and z of the point of view before to compute a distance with a point that demand square and square root extraction.

We can here use bounding boxes to ensure a minimal CPU usage. That way can be maintained for each direction[6] a list of sorted trigger bounding boxes. That way, only the first in each direction, the ones for which the point of view is inside the bounding box, have to be checked. This way, each object work : division or merge, lead to the maintenance of every sorted directional trigger lists; the work is then $\mathcal{O}(\log n)$ where n is the number of objects.

Indeed, we can use bounding boxes for spheres that trigger when the point of view *enter*. We should use *inside box* for those that trigger when the point of view leave.

For the process algorithms, entering of a trigger must be detected by a conjunction of axe triggers while leaving a trigger is detected by a disjunction of axe triggers.

Entering-triggered bounding boxes

For these cases, a conjunction of every trigger is needed before to verify if the point is effectively inside of the trigger sphere. Ie., if the point of view is *inside* the bounding box : $x > x_m \wedge x < x_M \wedge y > y_m \wedge \cdots$

This way, if we let in directional lists every *entrant* triggers, these lists will contain a bunch of useless triggers. The way to manage this efficiently is then to have every trigger listed only once : either in one of the two list about a dimension either in a "presently triggered" for that dimension.

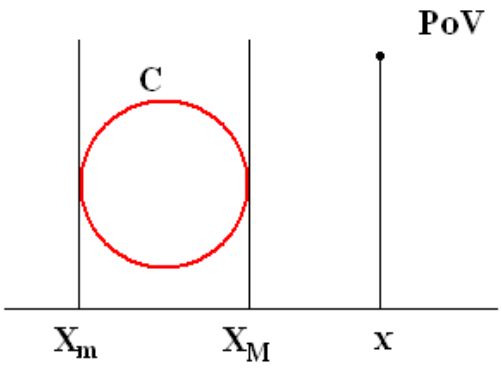


Figure 2.6: Trigger list from point of view position

[6]There are 6 direction : two for each dimension

Referring to fig. 2.6, to trigger the entering of the point of view in the bounding box of C while $x > X_M$, we just have to have X_M placed in the *lower bounds* list for axe x. To have X_m placed in the *higher bounds* list for axe x is useless and will consume CPU time because $x > X_m$ will always be triggered while $x > X_M$.

So, if $x > X_M$, C is specified in the *lower bounds* list for axe x with value X_M. Respectively, if $x < X_m$, C is specified in the *higher bounds* list for axe x. If none of these cases are verified, so if $X_m < x < X_M$, C can be specified in the "presently triggered" list for axe x.

Now axe-triggers only move actual split/merge trigger from one list to another. Here, the two axe-triggers lead the split/merge trigger into the "presently triggered" list. To get it out, should be maintained two more triggers lists for each axe : when C come into the "presently triggered" list, X_m is added to the *lower bounds* un-trigger list and X_M to the *higher bounds* un-trigger list. This way, trigger C will move from one list to another.

This is done for axe x and will have to be done exactly the same way for axes y and z. That way, the algorithm will only have to retrieve the intersection of "presently triggered" lists of each axe, and for every of its element, to check wether the point of view is inside the circle, in which case division should occurs.

Keeping only bounding boxes

We can even more optimise the algorithm by forgetting the circles and by using only the bounding boxes. When the point of view enter or leave a specific bounding box, the corresponding object is respectively divided or merged. This will skip distance computation sometimes still not really necessary.

2.7.8 Size computation

A small but important point about the computation of the actual size of objects, needed to compute apparent size. If sides are used, size is only their length. If triangles are used, the size used can be their surface and then be computed as a simple scalar product. Care should be brought while a really long and thin triangle can have a tiny surface but look really fuzzy to the user.

With some conditions, size can be approximated by the *generation* of the object, where the generation of an object is only the one of its parent incremented of 1. Indeed, a segment will have sons of approximatively an half length and triangles will have sons of approximatively the fourth of their surface.

2.7.9 Conclusion

With the chosen decision scheme, we have three concurrent "resources" that we can spare with the costs of another.

- memory usage
- CPU usage
- fuzziness of the TMLK

while *memory usage* is fixed by the maximum bound of apparent size, *fuzziness of the TMLK* by the minimum bound and *CPU usage* by the difference between the bounds.

2.8 Raw improvement

2.8.1 Generation maximisation

We can imagine a case where point of view "hit" the ground. In this case, distance to some parts can become nearly null. To avoid useless infinite generation, we can fix a "maximum sharpness" over which division is no more applied. For a sample human-sized simulator, we could fix this limit to somewhat like the size of a little rock on the road. Here, scale should be known for decision, that limitation won't fit for an ant simulation, this limitation could be applied to size of really insignificant details.

2.8.2 Multi-maps usage

There are cases where the use of several maps can be useful, let's imagine one specifying height and the other specifying color. Indeed, this shouldn't increase the TMLK maintenance CPU usage : the TMLK are specifying corresponding points on each maps. If the TMLK of a map divide some object, every TMLK do the same.

Therefore, it will consume more memory (by a fixed factor, the number of maps) but not more CPU usage.

Chapter 3

Terra map generalities

3.1 Working hypotheses

We now consider the *immutability hypotheses* ensured. This mean that, for a point, a seed is given. If this point has to be retrieved later, the same seed will be given.

Also, we will work on a *relative* basis : a point is always retrieved as the "middle" of a segment given by two other points. This lead to the fact that the extracted point can be estimated to be the middle of the segment and that random will only bring an offset. *Immutability hypotheses* then ensure also that the *base* of a point will always be the same for the same point.

3.2 Morphing

For each point will then be brought a *deformation*. This lead to three values : height morphing and lateral morphing that takes two directions.

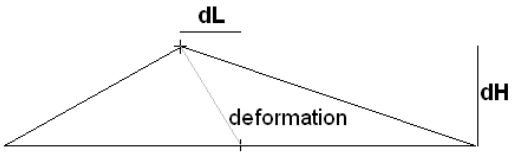


Figure 3.1: Point offset

3.2.1 Lateral morphing

Cares should be taken when using lateral morphing : the points if moved too far away from their basis can lead to inconsistent terrains (fig. 3.4).

Also, if lateral morphing is enabled, size approximation by generation (see 2.7.8) will be less accurate.

(a) Without (b) With, badly-configured

Figure 3.2: Usage case of lateral morphing

Anyway, lateral morphing can bring really interesting shapes impossible to create without it.

3.2.2 Spherical terra-map case

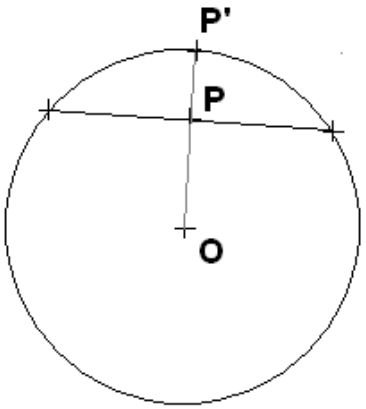


Figure 3.3: Base point correction for sphere

When seeking to generate a spherical terra map, the map can be approximated by a polyhedron; therefore, a point won't be exactly the middle of a segment, a spherical correction should be brought. Also, height offset will become a radius offset (fig. 3.3). Other lateral offsets will have to be corrected as orthogonal ones.

3.3 General morphing

While we only want to generate an empty landscape, the best way remains to pick offsets randomly. Randomly means that we need a *range*, a large range if a sharp landscape and a small range if we want a smooth one. There, actual range will depend of a number specifying the apparent morphology

and the size of triangles. For example, a morphing factor of one will lead to hillocks which slope can get to 45 degrees if lateral morphing is not used. A smaller one will give a flatter landscape and vice-versa.

3.3.1 Landscape shape and Morphing factor function

While we can have rough plains or eroded mountains, we could define a morphing factor depending on the object size we're dividing. While sizes are divided on each object division, we could define this function on the object precision, where the precision is the inverse of the size or, better, the logarithm of this inverse.

The advantage of using the logarithm of the inverse of the size is that it can be approximated by the *generation number* of the object. While sons size can be approximated by size of parents divided by some factor k (2 for segments or 4 for triangles), size can be approximatively computed by $S_{g+1} = S_g \times \frac{1}{k}$.

Landscape shape and morphing factor range

Using generation as argument to find the morphing factor also allow us to use a discrete function : its definition domain is in $\mathbb{N}$ and we can assume generation doesn't get down infinitely.

The formula used here is that, height deformation is a number in $[-\alpha, \alpha]$ where $\alpha = K \times S \times M(g)$ and where :

K is a global scaling constant

S is the size of the object we're dividing

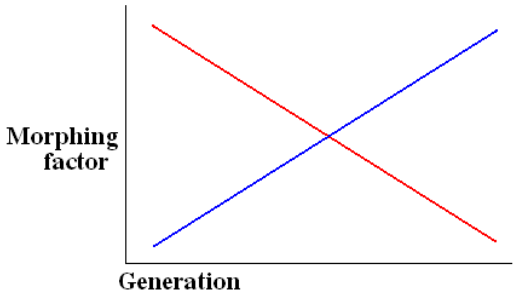
$M(g)$ is the morphing factor for generation g

With these formula, a slope will have a direction vector in $[-K \times M(g), K \times M(g)]$, so that $M(g)$ specify the shape at a certain scale. Size can be approximated with the generation of the segment. Here, segments are used and size is divided by two at every new generation, so that $S \simeq K' \times 2^{-g}$ where g specify the generation number and K' stand for the main segment size. In the formula, K and K' are constants so that $K \times K'$ can be replaced by K'' so that the formula becomes $\alpha = K'' \times 2^{-g} \times M(g)$

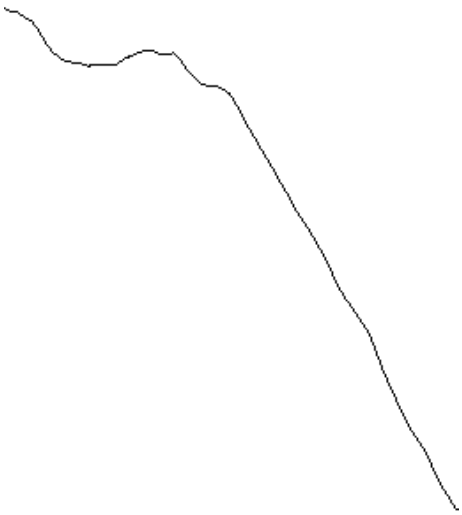
I chose in the example a K'' to make drawings pretty but changing the K'' global factor is equivalent to change every M local factor, it could be removed for generality.

Distribution as factor of landscape shaping

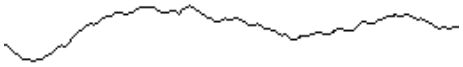
Another factor significant is the *distribution function* used to shoot numbers in $[-\alpha, \alpha]$ is the distribution function covering this range. For example,



(a) *Morphing factor* functions



(b) Red function : eroded mountains



(c) Blue function : rough plains

Figure 3.4: Different landscape shape based on *Morphing factor* function

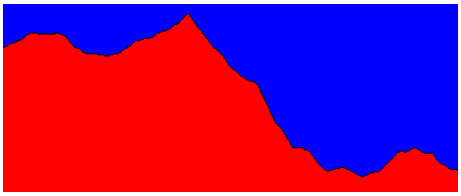
giving a strong probability to small offsets with a great morphing factor[1], can give smooth landscape with few abruptness. For another example, if height 0 is the level of sea, strongest probabilities must be given to negative offsets for the sea to cover - generally - more than the half of the surface.

3.3.2 Variable factors

Now come the fact that every landscapes on earth are not the same; there are sometimes rough plains and sometimes eroded mountains. The point is that there are continuous transition from one landscape to another. In other words : landscape shape vary the same way that land height.

For a point to be more like in a rough plain or in an eroded mountain

[1]So, a great α



A possibility to vary landscape shapes should be to use this generated map like this : left side of the generated shape should be more "eroded mountains"-like while right side should be more "rough plains"-like. For a point to know which morphing factor range to use, it just need to look at this map, on the specific X position to see the height and the amount of red or blue function to use.

Figure 3.5: Landscape shape local decision

can be specified by it's own *morphing factor function*[2] and it's own *morphing distribution function*. The example on fig. 3.5 show an auxiliary map allowing to choose between two morphing factor range function but we can imagine as many auxiliary map as needed. The point should be to use one for range, one for mean and one for variance so that distribution can be described. This will give more than three maps while we need a range, a mean and a variance for every beneath generation.

If for a point,

$R(g)$ specify the morphing range for a generation,

$M(g)$ specify the morphing mean for a generation,

$V(g)$ specify the morphing variance for a generation,

and if generation is maximised (see section 2.8.1) by number N, this will add three auxiliary maps of generation maximised to 1 giving $R(1)$, $M(1)$ and $V(1)$; this will also add three auxiliary maps of generation maximised to 2 giving $R(2)$, $M(2)$ and $V(2)$; and so on. This will then give $3 \times N$ auxiliary maps.

3.3.3 Limitations

There is a point where variability should stop. These auxiliary maps can use constants morphing factors or we can add them other morphing factors. Anyway, the use of only one generation of morphing factors can be enough for most applications.

Like said in section 2.8.2, using multiples maps is affordable even if the number of maps increase drastically.

[2]This function is just a list of morphing factors for every generation. Indeed, every point must only remember morphing factors for generation beneath itself

Chapter 4

Macro-structures

Sometimes, there are rules about landscape that should be applied but that get far away from the point of view scale. A good example is the river formation in nature; rules are simple (water "get down slopes") and bring structures no one will see in whole by his eyes with the feet on ground.

A naive approach should be to put sources and to simulate water run to find out where the rivers are; the point here is that we need a good precision and we want to avoid generating thousands of river falls to know wether we have feet in water or not. Anyway, we need a good "random-like shapes" for river realism.

4.1 Macro maps

Random continuous shapes is something terra maps can do, the problem can be reduced then to the problem of generating *regular maps* that just need to be transformed by a terra map *lens*.

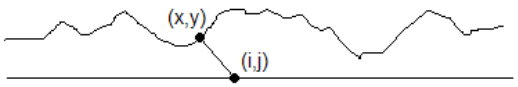
4.1.1 Theory

So, what is needed to be generated are a terra map lens (fig. 4.1.1), a regular map obeying rules of macro-structures (fig. 4.1.1); these will be enough to get fig. 4.1.1.

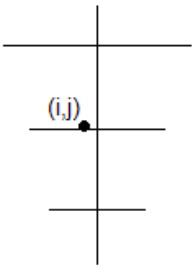
4.1.2 Application

Instead of using refraction formulas, we can just consider the *lens map* is a map specifying directly offsets $(\triangle x, \triangle y)$ so that $(x, y) \rightarrow (i, j) = (x + \triangle x, y + \triangle y)$.

It's what I did to get the example on fig. 4.2, using indeed two terra maps : one to get $\triangle x$ and another to get $\triangle y$.



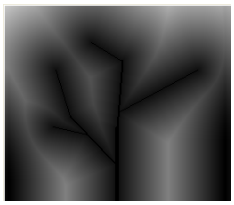
(a) Lens refraction



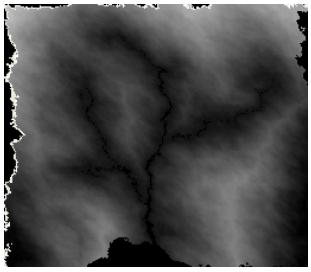
(b) Regular map

(c) Generated map

Figure 4.1: Terra map lens principles



(a) Regular map



(b) Generated map

Figure 4.2: Terra map lens 2D example

4.1.3 Limitations

The point here is that, to compute the height of a point of the generated map, we need to know the *map offset* so the values for that point on the *refraction maps*[1] and also, the value(s)[2] for the transformed point on the regular map. I used in the example a hand-made map but something sure is that it should be generated randomly and that is needed *only* a function $(x, y) \rightarrow values$ without the knowledge of each points of the map : this map should be huge, we're working on *macro*-structures !

[1] problem solved, even for multi-maps

[2] Several values may be needed. For our example : height, wether there is water, ...

4.1.4 Regular map generation

The points here are to create maps that obey some rules, without having to generate the entire map to know one point. A naive should be to generate a fern and to calculate, for each point of the background, the distance of this point to the nearest black one.

A way to apply this to our problem should be to generate that fern but not to draw it, only to remember each segments in a tree structure. After, when a point is queried, to calculate for each branch[3] the distance to the queried point. The minimum distance is retained and is the answer given by the "regular map".

I'm sure there are ways better to get it quicker and with more complexity in random ... work remains here.

Also, to generate a fern doesn't fulfill an earth-like river system. We should then either find a way to join regular maps one to one, either to generate a huge complex and complete regular map.

4.1.5 Macro map usage

Once a *macro-structured* map is generated, it can be used "as is" for landscape or, even better, be added some random from a common height terra map. In this case, the height terra map shouldn't interfere with macro-structure created by macro-map; macro map should then be added a value specifying sensibility of structure on a point. For example of rivers, "near water" points shouldn't be lowered beneath water level (care should then be token about the water that could fill another zone). In front of this, higher points can be modified to get random complex shapes without any danger of non-realism.

Macro-map should then specify an α-value[4] that'll specify wether point height should be taken more likely from macro map height or from added terra map one.

[3] That should be optimised by sorting branches by distance from point of view : if a point is near to a certain branch, it's neighbour is likely to be near the same one, fern is a macro-structure so that point of view scale move are really tiny compared to fern structure.

[4] α is chosen here in reference to image mixing where it is a color-like component specifying the "transparency" of a pixel.

Contents