

Thoughts on Generating Terrains



May 24, 2009

Abstract

Fractal terrain generation is a wide-used technique to generate random virtual terrains. This paper provides an insight into an algorithm for generating fractal terrains and some thoughts on the terrains representation and implementation.

1 Introduction

Many todays games are set on islands, mountains, etc. So it is useful to know how to generate and represent such terrains.

Many algorithms have been developed to generate random terrains.

Part I

Terrain representation

2 Height matrices

We are going to represent a terrain as a (rectangular) $H \times W$ matrix T (heightfield), whose coordinates at (h, w) describe the height at the point (h, w) . As we will see later, it is useful to define H and W as follows:

$$H = W = 2^k + 1 \quad \text{for some } k \in \mathbb{N} \quad (2.1)$$

That is, we require the terrain to be as wide as long.

Figure 1 shows a terrain Matrix and the corresponding heightfield. Dark means low, bright means height. We see a very small terrain with a peak at $(2, 2)$ (we start counting indices at 0).

2.1 Level of detail (LOD)

For viewing a terrain, a discrete matrix is not the best model, since we have here only distinct points. But a terrain is a continous structure, so that we have to interpolate adjacent matrix entries if we want to render the terrain. It's obvious that the closer we come to the terrain, the more "cracked" it looks, we have to less matrix entries.

It's obvious, that a 5×5 -matrix doesnt suffice for rendering a large terrain – even if we interpolate, because too less information is stored in this small matrix. So we come to the problem of how to select the appropriate size of the matrix. Of course a bigger matrix results in a longer rendering process, since more matrix entries have to be processed. On the other hand, a smaller matrix results in a coarse terrain.

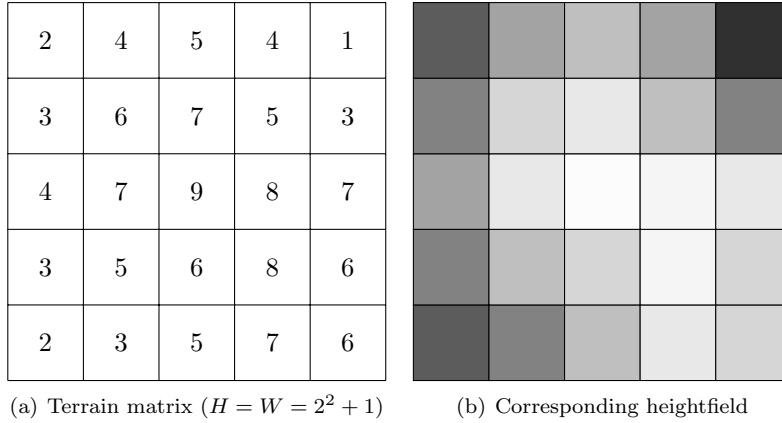


Figure 1: Sample terrain

So we do a little trick: We save the terrain in several level of details. In particular, we divide our terrain into four equal subterrains. The subterrains themselves are divided into subterrains themselves and so on (see figure 2). Each subterrain has as much entries as its parent. But the subterrain entries are “smaller-grained”. Figure 3 illustrates this issue.

Now we see, that it is convenient to say $H = W = 2^k + 1$, because then we can easily subdivide our terrains. Note that the subterrains edge points overlap. That is, in figure 3 the upper left and the lower left subterrain have the points $(0, 2), (0.5, 2), (1, 2), (1.5, 2), (2, 2)$ common.

So it is (theoretically) possible to create an infinitely accurate terrain representation.

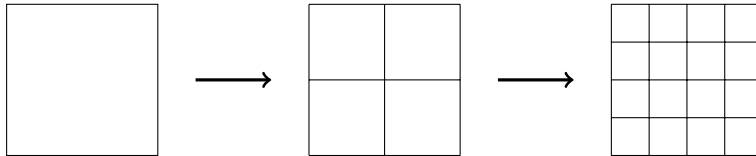


Figure 2: Terrain subdivision

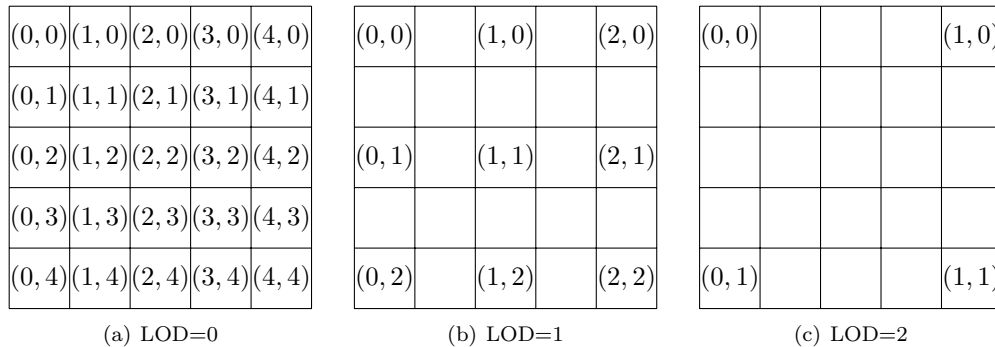


Figure 3: Upper left subterrains

So where is the advantage of this technique? Up to now, we have only gained more occupied memory. But we have also our terrain in several levels of detail. If we are far enough away from the terrain, it's sufficient to render the coarse terrain. The closer the terrains are at the view point, the

more detailed versions we take. This means, the closest subterrains (to the view point) are rendered in high-resolution, while the farer away subterrains are rendered in lower resolution to save time (see figure 4).

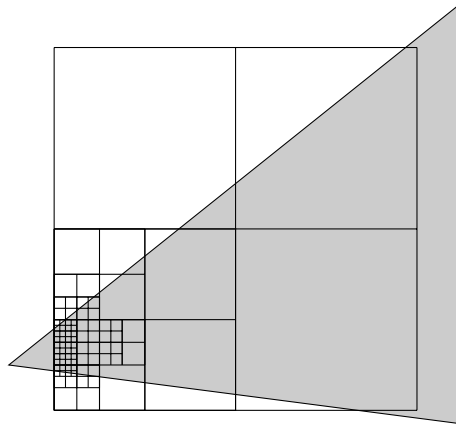


Figure 4: View point and levels of detail

We obtain a quadtree-like structure with this method.

Part II

Generating terrains

For more information on terrain generation algorithms, see [Mil86].

3 Midpoint-Displacement-Algorithm

The algorithm works as follows: First, we assign the corner heights, so we have some start values. These start values can either be fixed or random.

Then we calculate the midpoints of the yet generated points, interpolate and add a random value. Figure 6 illustrates this process on a single line.

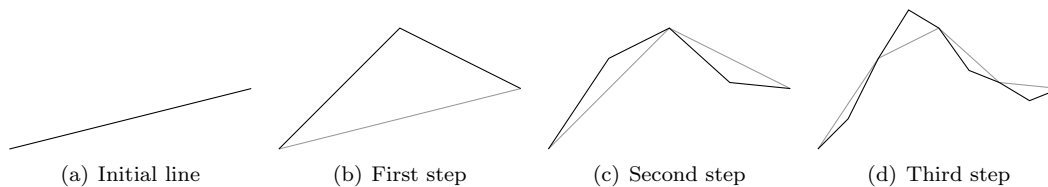


Figure 5: Midpoint displacement in 1D

So we calculate each new point by interpolating the neighbouring points and then adding a random number (wich can be negative). Logically, the range of this random number gives the “roughness” of the terrain (i.e. a range of $(-10, 10)$ gives a more uneven terrain than a range of $(-1, 1)$). This range should be variable and depend on the level of detail. The higher level of detail, the smaller the random number range. A possible way of reducing the range dynamically is to half it in every iteration, for example.

No we are going to extend this process to our heightfield. As we said before, we first set the corner values and then bit by bit the remaining values (see figure 6).

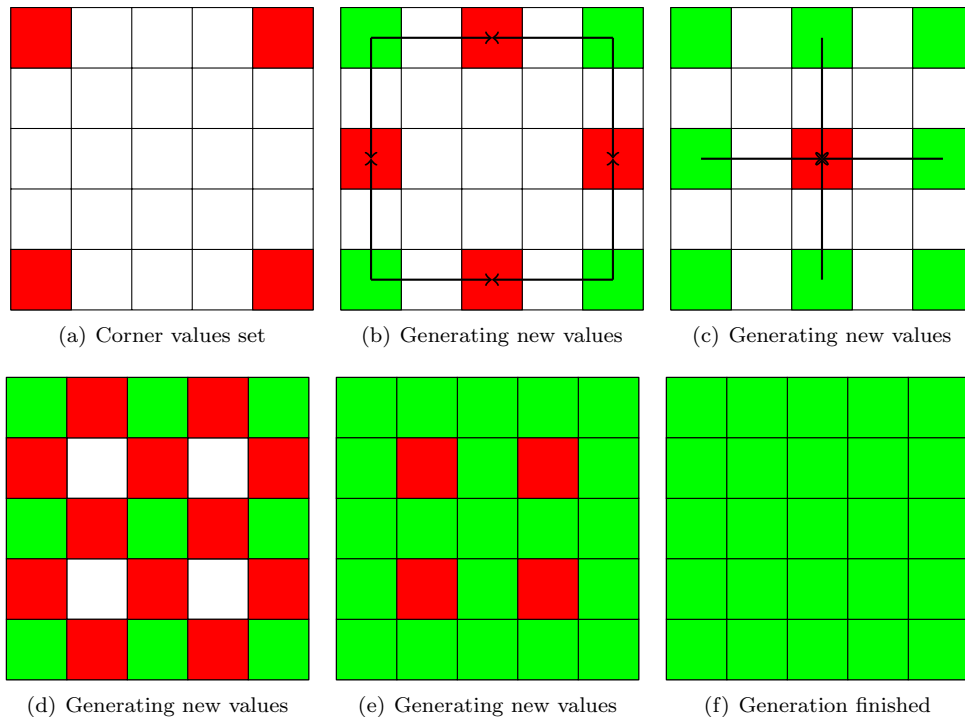


Figure 6: Midpoint displacement: **Known points**, **Points to be set**

A problem of this terrain generation method is that it produces a “checkerboard-like” appearance of the terrain.

4 Diamond-Square-Algorithm

The checkerboard effect mentioned above is unwanted, so we are looking for a better method. The Diamond-Square-Algorithm is a derived version of the midpoint displacement algorithm, that calculates the midpoint first and according to this midpoint the edge points.

We see, that the major difference between midpoint displacement and diamond-square is, that diamond-square generates diagonal elements first and according to them the other points. Diamond-square uses interpolation and addition of a random number analogue to midpoint displacement.

5 Level of detail consistency

We have seen two algorithms for terrain generation. If we want to generate a terrain with several subterrains, we do have to consider, that the subterrains must be consistent with the parent terrain. Our task will be to set the subterrain matrix according to the parent terrain matrix (see figure 3). There are two central approaches to keep consistency.

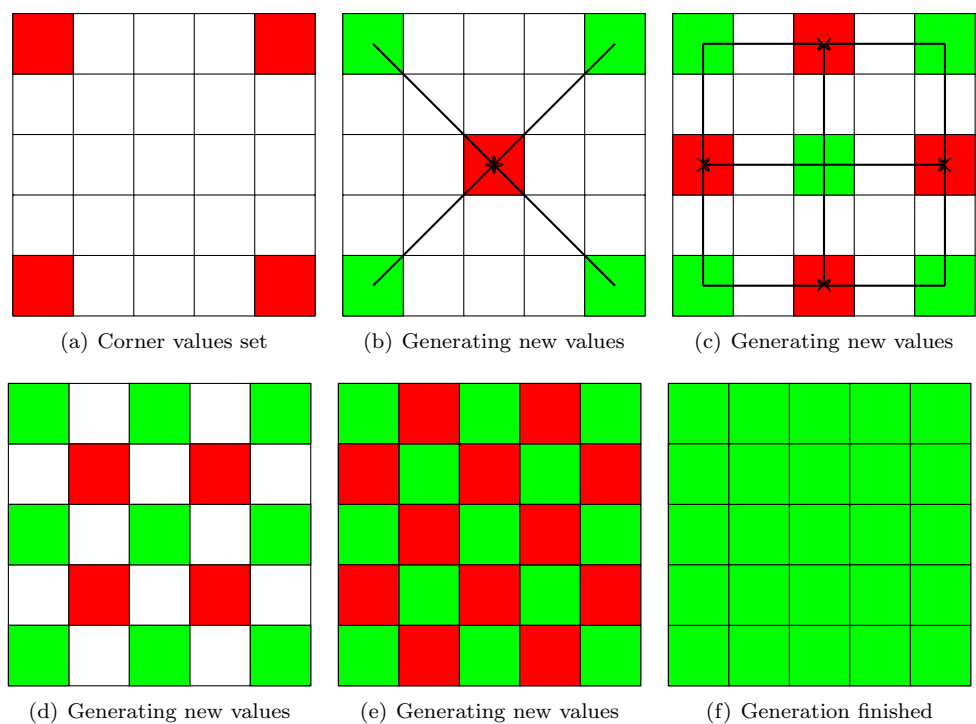


Figure 7: Diamond-Square: **Known points**, **Points to be set**

5.1 Top-down approach

An intuitive approach is a top-down approach. We construct a parent terrain and transfer the heightfield entries correctly to the children. Each child constructs only the points that are not given from the parent terrain.

This approach assures, that we don't allocate unnecessary memory, since it only needs to allocate as much space as each child occupies.

However, there is a problem with this method. If we want our matrix elements to carry only (x, y, z) -coordinates, everything works correctly. But if our heightfield elements should also contain further information, it can be problematic.

5.2 Bottom-up approach

Suppose we want to store the normal vectors in our heightfield entries, so that each entry has the direction of its normal vector. A normal vector of such an entry v is computed as follows:

- Calculate the normal vectors of *all* faces, that have v as a corner.
- Average the above calculated vectors and eventually normalize the result.

(See figure 8).

Now we come to the problem: We saw before, that adjacent subterrains share some specific heightfield entries. If we want to compute a normal vector for such a shared heightfield entry, we have to consider all bordering faces. That means, we have to consider faces from two different subterrains, which makes our generation quite complicated. Each information, that is based upon facts that concern heightfield entries from an adjacent subterrain, is complicated to compute.

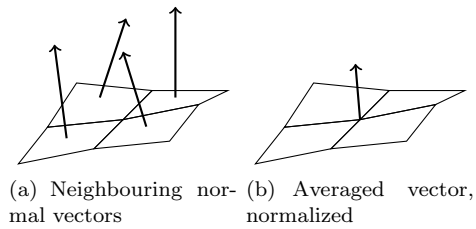


Figure 8: Normal vector

Because of that it is more feasible to generate the whole terrain at full detail first. After that we assign the subterrain matrices bit by bit. That means, we first have to generate a huge height matrix, which yields the entries for the subterrains.

It is clear, that, if we define $H = W = 2^k + 1$ and want to generate up to detaillevel d , we have to generate a terrain with dimensions $H' = W' = 2^{k+d} + 1$ (we begin counting detail levels at 0). So with this method we have a huge memory overhead – this is the price we pay for calculating normal vectors easily.

References

- [Mil86] Gavin S. P. Miller. The definition and rendering of terrain maps. *SIGGRAPH 1986 Conference Proceedings (Computer Graphics, Volume 20)*, 1986.