A New Data Structure for Terrain Models

Petr Lobaz
lobaz4@students.zcu.cz

Department of Informatics and Computer Science[1]
The University of West Bohemia
Plzeň, Czech Republic

Abstract

In computer applications, work with terrains is very often needed. Requirements on data structure of the terrain model depend on the application. Here we present data structure destined mainly for image synthesis software. Presented data structure is able to store complex terrain features such as overhangs and caves. Notes about implementation are also included.

KEYWORDS: terrain synthesis, image synthesis, data structure, terrain models

1 Introduction

In many applications it is required to work with terrain models. We can divide these applications into several types. Main three types are geographical information systems (GIS), programs for simulating processes on the earth surface and below it and programs for image synthesis. Everyone of these applications has special requirements how a terrain model (its data structure) should look like. This depends on what terrain features we want to represent, what accuracy we need, how we obtain data and what are we going to do with the model. Next, we must consider memory complexity of the model and how difficult is to implement tools for its maintenance. In this article we will take an interest in data structure aimed at image synthesis software.

Image synthesis software does not need an exact model of the terrain. It means that some distances (heights) can be slightly different from exact ones. Nevertheless, topological properties of terrain must be stored correctly. Focus of image synthesis software does not lie in work with terrain, so the data structure should be simple and fast. It is often needed to represent complex terrain features like arches, overhangs etc. However, currently there is no data structure that can store such complex terrains. In the article we will discuss currently used data structures, give brief overview of methods for terrain generation and rendering and last we will derive data structure destined for storing complex terrains.

[1]This work was supported by The Ministry of Education of the Czech Republic: projects VS 97155, ME 259 and project GA AV A2030801.

2 Currently used data structures

We can divide models of terrain into two types, real and virtual. In GIS, there is of course interesting first case only, that is real terrain. Data can be obtained either by taking photographs from airplane or satellites, or by direct measuring in terrain. In both cases, we obtain data about terrain in finite number of representative points (usually from thousands to hundreds of thousands points), next we use a hypothesis that these data (e.g. height) are varying continuously, so we use some kind of interpolation (e.g. bilinear, bicubic). Note that points that are derived from representative ones via interpolation are just an approximation of real terrain.

If we use direct terrain measuring, representative points are distributed "randomly", that is they do not create any regular geometric pattern. In fact, these points should be distributed more dense in places where bigger accuracy is needed (e.g. steep slopes, river banks). Natural data structure for such a data is triangulated irregular network (TIN), where vertices of triangles are representative (measured) points, edges give information how to interpolate between these points. This data structure can be modified to hierarchical version to achieve a) faster point location in terrain b) storing of several representations of the same terrain in various resolutions (level of detail), mainly for speeding up rendering. TIN is able to describe any terrain feature (but almost nowhere perpendicular slopes or overhangs are considered), point location can be done quite fast (especially with hierarchical modification). Opposite to this, modifications are not simple, with every change we must check (and possibly modify) the triangulation. If we do this with simple (and fast) methods, we can obtain thin triangles that are very unsuitable for interpolations.

If we measure from bird's eye view we get representative points in regular pattern, for example square grid, no matter how locally complicated terrain is. With this view, we can of course measure only one piece of information (e.g. height) above one point of ideal earth surface, model is then in the form of function of earth coordinates. If we have representative points in square grid we can store data efficiently in two dimensional array. Most often stored data are heights and so this data structure is called height field. Pros and cons are near inverse against TIN. It is not possible to describe every terrain feature,[2] there is no adaptivity if a terrain is locally more complicated (sampling is the same in the whole height field). On the other hand, modification of such a model is trivial (in sense of work with data; modification itself, e.g. erosion, can be a complex task), point location is very fast. Thanks to listed purposes this data structure is used if it is necessary to modify it often, if information from it is extremely often needed (collision detection), if we do not require high accuracy (terrains of area of thousands km^2) or if a focus point of application lies somewhere else (image synthesis software).

To choose what data structure to use for virtual terrain model depends on (besides others) how do we want to generate it and what manipulations do we want to do. Most of algorithms work with common height field; there are also algo-

[2]Sometimes height field is modified so that it can describe perpendicular slopes, but more complicated features are still not possible

rithms that work with TIN or with regular lattice different from square (triangles, hexagons); see [3].

The last criterium how we can choose data structure is algorithm for rendering (hidden surface elimination). Algorithms such as ray casting prefer structures based on regular lattices, hierarchical if it is possible (for speeding up ray—object intersection test). Combination of painter's algorithm with regular lattice is very fast and often used (it is not necessary to sort polygons, arrangement is obvious from the lattice). In algorithms such as scan line or z-buffer it does not too depend what data structure is used, but if we want to avoid aliasing problems we choose structure that leads to bigger polygons. It is similar in case of continuous (vector, non-raster) algorithms that cannot use information about arrangement, we choose structure with less number of polygons (decimated TIN).

3 Motivation for new data structure creation

In image synthesis software, terrain is an often used object. Examples of applications are architecture, advertising, computer art, computer games, virtual environments etc. Here we do not need accurate terrain model, we are interested in appearance only. Due to interactivity of software or big number of rendered images it is necessary to achieve maximal possible speed, so almost always data structures with regular (almost always square) lattice are used, even if they have bigger memory requirements. This is possible because in listed applications we do not visualize large areas (mainly at most several km^2).

Here we get first problem that appears if we use regular sampling of terrain. Virtual camera is usually perspective, polygons that lie in the front are bigger than those in the back. If we choose linear (bilinear) interpolation of heights, we get very ugly "teeth" in the front. When we refine sampling we get two problems: size of data quadratically increases (the biggest commonly used height fields contain about 1000×1000 heights) while in the back, size of polygons decreases to subpixel size, causing aliasing problems. Time needed for rendering also quadratically increases (except of ray casting based algorithms).

We can solve this problem in several ways. First of them is not to use bilinear interpolation. Instead we can use approximation with e.g. Coons B-spline patches (let us recall that in our application it is not needed for sampled points to lie on the surface, we do not need exact model). Using this approach e.g. for height field of size 256×256 we get about 65,000 bicubic patches—it is sure this will slow down rendering. Next, if we use bad textures, terrain in the front will look unnaturally smooth.

Another choice is to use true displacement—this method works similar to bump mapping but geometry of the object is really changed. Its use can be for example such that in the front bilinear patch will be changed to "bicubic" (we can add some random height to increase number of details) and this change will smoothly disappear in the direction to the background; from some distance there will be no changes to geometry. Advance of this approach is adding procedurally generated

details of arbitrary small size, this cannot be done simply in other methods; see [1]. Unfortunately only small number of rendering engines support this ability (e.g. Photorealistic RenderMan) because its correct implementation is not trivial.

Next approach is based on using several models of parts of the same terrain. It is for example possible to use rough sampling for the whole terrain, in the neighbourhood of camera we cut it off and replace it with finer representation. This cutting can be done by using methods of *constructive solid geometry* (CSG) or with special textures (rough terrain will be invisible in the neighbourhood of camera causing the finer model to be visible). Structure of nesting these models is stored outside the height fields. Nesting can be of course recursive. If we do not work systematically there can appear problems with overlaping models at the same level of nesting. One opportunity how to avoid these problems is to constrain what part of terrain will be replaced with finer one. It can be done for example with quadtree data structure or with its extension, structured grid (area is not divided into 4 parts but into $m \times n$ parts). Problems with this data structure are obvious—much more complicated work with it, parts of terrain must be connected continuously.

If an application needs to work with simple terrain only it can use one of the modifications noted above. Constraint that follows from that there is defined just one height above every point will not appear in this case. However, in advertising off-road vehicle will look better in the complicated, hard accessible terrain rather than on the meadow. Here we can do nothing with height field representation—it cannot store highly inclined slopes (it depends on density of sampling), overhangs are not possible. Again, there are ways how to overcome this drawback.

If we want to represent walls of at most perpendicular slope we can slightly modify height field. Information about height will be extended with two bits—one for (for example) north, second for east direction. If one of these bits will be set to 1 it means that heights between current point and point in appropriate direction are not continuous, there is a jump. If a bigger perpendicular wall is created in this way it is necessary to use good texture for it to hide that this part of surface is smooth (it holds in general in height fields—the more steep surface the worse it looks). This modification needs of course special renderer that supports this extension, or to break terrain into triangles before rendering—doing this we loose possibility to render it (as a special object) faster.

Second possibility, often used, is to use height field as an ordinary object and to work with it. For example a simple cave can be created using two height fields, "convex" and "concave" one. Space between them is quite a good approximation of cave, we can add some objects into it to hide artifacts. If we define height field in some way as an object enclosing some volume (for example by cutting a cube with this height field) we can use CSG. Then we can create overhang or another complicated feature using CSG difference operation. As it was said, this technique is often used. Its drawback is that CSG has not any relation to a terrain and so algorithms that modify terrain (e.g. erosion) cannot be used in this case.

Next problem that must be solved is definition of attributes of terrain in its different parts. Namely these are textures (or materials), relation between terrain and another objects (both solids like vegetation or stones and particles like sand or water). Materials can have optical properties only, they can also influence algorithms for terrain modification (rivers will have steeper banks in softer rocks). In currently usually used algorithms there is considered only one material in the whole area—this is quite unreal even in the small scale.

Let us sum up previous information: for general model of terrain a data structure is needed that can describe every shape (like general TIN), memory requirements as well as complexity of work should be small for simple terrains (the best would be height field like complexity). It should be able to store terrain in different resolutions in different parts to achieve level of detail effect. Next it is necessary to have relations to volume properties of terrain and to objects that are connected with it. Existing algorithms for modifications, generation and rendering of terrain should be easily convertible to work with new data structure. Mainly this should hold for rendering algorithms so that it would be possible to use some existing (good) renderer. Currently there is no structure that fulfills all of these parameters.

Before we derive such a structure we must give brief overview of basic methods for generating and rendering terrains.

4 Methods for terrain generation

There exist two approaches for generating terrain surface: physical and empirical one. First approach is used mainly for generating whole planets, second one for mid-scale terrains (several km^2). Here we will take a look only on the second approach, empiric. Thanks the presumption of generating small terrain, we can neglect curvature of the ideal planetary surface and approximate it with a plane. Also other parameters that are connected with curvature (e.g. direction of gravitation force) are almost constant in this small area. In the following text we will use coordinate z as height of terrain above sea level, coordinates x and y will denote coordinates of ideal surface (approximated by plane).

Majority of empiric algorithms is based on observations from fractal geometry, mainly that surfaces with some so called fractal dimension look very similar like terrain. For its generation we can therefore use some method for generating surface with given fractal properties.

The simplest (and most used) algorithm is one of midpoint displacement methods (see [3], [2]). The principle of the simplest one (and the worst) is that height field with square shape with heights prescribed in the corners only is divided into four square parts. We must therefore generate five new (random) heights. After they have been generated we apply this process recursively on parts that have just been created. These methods work with height field as well as with TIN. Their advance are simplicity, speed and quite good results, drawback is that computa-

tion is dynamic (it is impossible to simply compute height above one point). They work well for generating rough, jointed terrains.

The second often used method is approximation with trigonometric polynomials (Fourier synthesis), see [3]. Using some rules we generate coefficients for sin and cos functions, by summing Fourier series we get height of one point of terrain. If we want to know all heights in the height field we can use fast Fourier transform (FFT). Advance of this method is removal of some artifacts present in midpoint displacement methods (e.g. *creasing*, ie. it is visible how the terrain was generated), drawback is much bigger time requirement. Property of such generated terrain is its periodicity, it can be both an advance or a drawback. It is better to use this method for generating smooth, old terrains.

Last mentioned method is functional synthesis, extension of previous method, see [2], [1]. If we want to hide that functions sin and cos are perfectly smooth, we have to sum big number of such functions. When we use another functions, more visually complex, number of summands need not be so large. Very suitable for this purpose are noise functions, for example Perlin noise, see [1]. On the other hand, we loose the advance of using FFT.

5 Methods of terrain rendering

Each used algorithm of visualizing terrain in raster space is a modification of known algorithms—ray casting and painter's algorithm. Thanks to nature of applications with new data structure, we will not have to consider vector algorithms at all. We will also discuss algorithms for rendering on the screen only (into memory, resp.), no special devices like plotters are assumed.

Terrain can be rendered with original, non-modified algorithm as well but by breaking terrain into primitives we loose information that can be used in rendering. In following paragraphs we will talk about algorithms that work with height fields only. Visualization of TIN works in similar way but thanks to irregularity, algorithms are much more complex.

Basic method for rendering terrain is painter's algorithm, forward or backward (known also as floating horizon method). If we know that heights are arranged in square grid we can easily decide what patch created with bilinear interpolation of four heights is nearer to the eye (instead of bilinear patch approximation with two triangles is often used). Then if we render patches in an order that can be easily constructed in dependance on camera type and view, surface visibility is automatically solved. Improvement of this method is to draw first patches into temporary buffer (every one with different colour) and to do texturing in second pass only on the patches that are really visible.

Backward algorithm works in similar way, first it renders primitives in the front and then in the back. It can draw only such pixels that have background colour. Algorithm is slightly more complex than previous but there is no need for two passes to reduce texturing.

Second type of algorithms is based on ray casting method. Basic method is grid tracing, see [2]. Here a ray is cast and perpendicular projection on the plane with height field is done. Then raster algorithm goes through all cells of height field (cell is formed by four adjacent heights, is is "a small square") that incide with this projection of ray. While this is done, it detects if a ray is still above terrain. Test of ray—primitive intersection is done if and only if a ray goes from the state "above terrain" to "below terrain" and vice versa.

Speedup of this algorithm is based on temporary hierarchy of bounding boxes. Box that bounds the whole object (height field) includes four bounding boxes inside that (like in quadtree) bound appropriate parts of height field. Name of this method is derived from the data structure idea—quad tracing; see [2]. If a height field is large this algorithm can lead to big speedup.

6 Development of new data structure

Comparing requirements for data structure with typical algorithms noted before, mainly for rendering, we come to conclusion that it would be better to use height field. Hierarchical approach and level of detail is achieved in quadtree-like structures, development thus could go this way.

If we want to represent rock wall only (including overhangs) we could generate height field that will not be in basic position. Instead we rotate this height field so that it will no longer represent function $z = f(x, y)$ (for explanation of coordinates see above) but function $x = f(y, z)$. Now we get a situation that we are able to represent slopes near vertical one, problem with horizontal slopes appeared. In real terrain orientation of surface does not vary chaotically, it is always possible to bound its part that can be represented with one of functions $f(x, y)$, $f(x, z)$ and $f(y, z)$, so we can use for its representation appropriately oriented height field.

Collecting ideas from previous paragraphs we can formulate outlines of new data structure. We want different parts of terrain to be represented with rotated height field as well as hierarchy. It follows immediately that we can extend quadtree structure to octree, or structured grid in 2D into structured grid in 3D. Its elements will be either nothing (cell is void, its contents is air only), material (whole cell is filled with one material), rotated height field (height field defined above one of six faces of cell) or structured grid.

It is obvious that this data structure must be able to represent any ordinary height field regardless how fine the resolution of grid is. Except of continuity in x and y directions we must therefore solve the case that surface can go through several cells one above other. We can solve it in several ways.

The first one creates new type of cell contents—cell can contain besides height field also a reference to height field defined in another cell (this will be called *forced height field*). This means that in fact we ignore structure of the grid; things that can be represented with one height field are really represented in this way. It follows that problem of continuity in z direction disappears, modifications of this height field is easier than if the terrain is divided into several parts one above other.

Nothing is for free, let us look at problems that arose. If one cell defines height field in $+z$ orientation (it means "common" orientation, ie. height says how high is space filled with material in $+z$ direction; this notation will be used throughout the rest of the paper), another cell above defines height field with $-z$ orientation, then there is not obvious that these surfaces do not intersect. This would mean that one cell (in which intersection would appear) must have references to several height fields. This is nonsense both in physical sense and data structure, next it would not be possible to use simplified painter's algorithm for rendering. So if we want to avoid intersections we must check height fields somewhere in program. This checking would be quite difficult, remember that data structure is hierarchical—height field from the cell at some level in hierarchy can interfere in cells that are in hierarchy more above or more below. If we constrain this we get again the problem that was in the beginning, ie. continuity in z direction.

But there exists more serious problem. If we modify terrain not locally, ie. modification goes through several height fields (for example creation of river valley) we must know what height fields are connected to current one in x and y directions. If the reference to forced height field is represented with pointer to a cell that defines it, there is no information available where this cell lies in hierarchy, so we do not have information what height fields are connected. We can solve it either that in the cell with height field we store references to height fields this one is connected to (thanks to hierarchy there can be arbitrary number of such references) or to represent reference to cell like a path to that cell (thanks to hierarchy this path can be arbitrary long). In both cases we get a trouble with items of undefined length, this is quite a hard problem. We see that forced height fields bring more problems than they solve, so we must reject this approach.

The second way of solving continuity problem is as follows. Height fields will not neighbour with cell faces, there will be a small margin. For rendering we must provide height fields as well as strips that fill these margins. Using this approach continuity problem would be solved in all directions, borders between cells would not be noticeable, there would be no software checking of continuity. The only thing that must be implemented is creation of these strips. Here we unfortunately get a problem that cannot be algorithmically solved. Even if we consider height fields in directions $+z$ and $-z$ only, there are cases that there is not clear how height fields should be connected. Example of this situation is at Figure 1. One of the presumptions was that there is no need for accurate description, but in sense of distances, not topology. For solving this problem we would (as in previous case) save topological information in cells. It can be seen that number of problems did not reduce. We must therefore reject this method and continue searching another alternative.

The third way of solving continuity problem must consider that the whole height field lies entirely in one cell and neighbours with its faces exactly. continuity of two height fields one above another can be solved with *aligning*—if a height overlaps borders of cell we clamp it into the appropriate interval. We can easily modify this idea so that two possible values in height field will denote "height undefined". The difference between them is that one height denotes "here should be height below the cell" and "here should be height above the cell". In this way

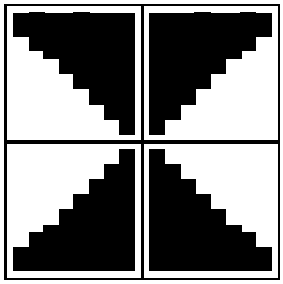

Figure 1: Here it cannot be algorithmically decided if there is to be a horizontal hole between height fields

we get in every cell height field that can have both on above and below a plateau. This does not matter on the side where this plateau will be covered with another height field; on the other side there must not be this plateau. It can be done so that renderer will obtain information both about height field and how to clip it (clipping planes)—most of renderers support this ability. Now height fields that lie one above another are exactly connected, it was done by modifying some distances. However, it was said several times that slight modification in distances does not matter too much. This approach saves topology. The problem here is a practical one: due to clamping of heights into appropriate interval some visible creases can appear on the borders of cells. These creases will respect grid structure, this is obviously undesirable.

With slight modification we can suppress this problem too. Modification comes from combination of ideas of clamping and forced height field. Height field in one cell could contain heights above and below the cell that would give information for better continuity. We can solve the situation so that we will not clamp heights that overlap cell borders. Nevertheless we give information about clipping planes to the renderer, so we have an illusion of forced height field but with well defined data structure. Of course, continuity of height fields must be checked in software, data on the cell borders will be redundant; this causes problems in every case. However there was not found better way how to achieve continuity.

In data structure there are another continuities than those described above. continuity between height fields of the same orientation "on the sides" (e.g. in x and y directions for height fields oriented $+z$) is trivial—it is sufficient to unify two appropriate heights. continuity between height fields of the same orientation "above and below" (e.g. in z direction for height fields oriented $+z$ and $-z$) was described above. Problem is continuity of height fields of different orientation. Here it must be taken into consideration that heights are represented with finite precision, border of height field must be representable in both orientations and so on. Complete solution of these cases is still not entirely done. But it is necessary to know that in the worst case we can solve this by clamping even if the result is not perfect; it follows that problem has solution in this data structure.

7 Implementation

Hierarchical data structure is always based on dynamic data types and so it is not good to implement this data structure in a language without support of pointers or dynamical arrays or something like this. Due to the thing that items of structured grid can be of various character, language with strong type checking would at least complicate the implementation. So it seems that C language should be suitable. Abilities that gives C++ are hard to use in this case, mainly due to effectivity. Thanks to these reasons it was chosen ANSI C as the best language for the implementation of this data structure.

Let us summarize information that can be stored in one cell:

- nothing;
- material:
  - memory needed for material definition is not uniform;
  - at least information about optical properties, hardness, permeability and distribution of cracks must be stored;
- height field:
  - size is various;
  - at least resolution, orientation, height field data, material below height field, another information like about water, objects on it etc. must be stored;
- structured grid:
  - size is various;
  - resolution and the cells must be stored.

It is obvious that we cannot store data of this variability in an array, so we have to work with pointers. As a good compromise this approach was chosen:

- array of cells contains heads of *lists*;
- if the *head* is equal to `NULL`, the cell is void;
- there is an (global) array of materials; if the head points somewhere into this array, the cell is entirely filled with appropriate material;
- in other cases it is true head of list:
  - *items* of list are of different type; all of them are based on `struct` type;
  - each item contains a pointer to the next item and type of current item;

- if the type is `HF`, it is a height field and `struct` contains extra information: resolution of height field, material below height field, orientation, information in what sides of cell continuity must be solved (only for speedup, it is not necessary), pointer to an array of heights;
- if the type is `GRID`, it is structured grid and `struct` contains extra information: resolution of the grid and pointer to an array of cells;
- another information like objects, textures etc. are described in other items of the list; at this time only specification is done

Now we must notice a property that was still only quietly assumed, it was not anywhere explicitly said. Solving continuity "on the sides" with height fields of the same orientation we assumed that resolution of adjacent sides are the same or the samples of the one are a subset of the second. One way how this can be always fulfilled is that we will require the resolution of structured grid to be of type $2^i \times 2^j \times 2^k$, resolution of height field of type $(2^m + 1) \times (2^n + 1)$ (due to splitting we need center element). Doing this we can store only numbers i, j, k (m, n, resp.). Then it is more than sufficient to store one dimension in one byte.

An interesting part is implementation of type for one height of height field. It follows from practice that 8 bits is too small resolution, 16 bits gives good results. We need to represent numbers that lie both inside and outside of the cell. It is not good choice to define global coordinates of height because with increasing depth in hierarchy the accuracy of details does not increase, too. Next, to find if a height lies outside of the cell needs an information where this cell lies in the space and how big it is. Due to these reasons local coordinates inside the cell were chosen. As a good choice it seems to reserve one bit for information if the height lies inside or outside the cell, other 15 bits will denote heights between 0 (including) and 1 (excluding) in fixed point representation (in fact type like `int` is used). One half of the range lies outside; dividing symetricaly this range we get that we can represent numbers approximately between -0.5 and 1.5 (in cell local coordinates). It remains to decide how to represent intervals above and below the cell.

Sometimes it is necessary to have linear ordered heights. Interval of visible heights then should lie between `0x4000` to `0x8000` (hexadecimal notation; now we assume that height 1 is visible), interval below the cell `0x000` to `0x3fff`, above the cell `0x8001` to `0xffff`. If we notice that summing in two-byte unsigned arithmetic is a commutative (Abel) group we can do conversion from linear representation to special one by adding number `0x8000` (by subtracting `0x4000`, resp.). In this way we get a special code with moved zero, where is

`0x0000`	the smallest visible height (0)
`0x4000`	center of interval of visibility (0.5)
`0x8000`	the highest visible height (1)
`0x8001`	the smallest height above the cell (approx. 1.000030518)
`0xbfff`	the highest height above the cell (approx. 1.499969482)
`0xc000`	the smallest height below the cell (-0.5)
`0xffff`	the highest height below the cell (approx. -0.000030518)

In this code visibility is easily checked; in linear ordering it is better to do comparsions.

8 Conclusions

It follows from above that data structure presented is able to store all information we need. It is for example possible to include more types of representation into the list in one cell, e.g. several height fields with different resolutions, both structured grid and its approximation with height field, etc. Everything depends on the implementation of tools for work with this structure. Next advance of the structure is its openess for including other information—it is needed only to add new type of list item; if a particular tools implementation cannot work with this type of item, it can just ignore it (so it is not necessary to change tools implementation by adding new feature).

References

[1] David S. Ebert, editor. *Texturing and Modeling: A Procedural Approach.* AP Professional, 1994.

[2] Kenton F. Musgrave. *Methods for Realistic Landscape Imaging (PhD. thesis).* Yale, 1993.

[3] Heinz-Otto Peitgen and Dietmar Saupe, editors. *The Science of Fractal Images.* Springer Verlag, 1988.