


A hybrid representation for modeling, interactive editing, and real-time visualization of terrains with volumetric features

Çetin Koca & Uğur Güdükbay


To cite this article: Çetin Koca & Uğur Güdükbay (2014) A hybrid representation for modeling, interactive editing, and real-time visualization of terrains with volumetric features, International Journal of Geographical Information Science, 28:9, 1821-1847, DOI: 10.1080/13658816.2014.900560

To link to this article: <https://doi.org/10.1080/13658816.2014.900560>

 View supplementary material 

 Published online: 09 Apr 2014.

 Submit your article to this journal 

 Article views: 292

 View Crossmark data 

 Citing articles: 6 View citing articles 

A hybrid representation for modeling, interactive editing, and real-time visualization of terrains with volumetric features

Çetin Koca and Uğur Güdükbay*

Department of Computer Engineering, Bilkent University, Ankara, Turkey

(Received 16 December 2013; accepted 28 February 2014)

Terrain rendering is a crucial part of many real-time applications. The easiest way to process and visualize terrain data in real time is to constrain the terrain model in several ways. This decreases the amount of data to be processed and the amount of processing power needed, but at the cost of expressivity and the ability to create complex terrains. The most popular terrain representation is a regular 2D grid, where the vertices are displaced in a third dimension by a displacement map, called a heightmap. This is the simplest way to represent terrain, and although it allows fast processing, it cannot model terrains with volumetric features. Volumetric approaches sample the 3D space by subdividing it into a 3D grid and represent the terrain as occupied voxels. They can represent volumetric features but they require computationally intensive algorithms for rendering, and their memory requirements are high. We propose a novel representation that combines the voxel and heightmap approaches, and is expressive enough to allow creating terrains with caves, overhangs, cliffs, and arches, and efficient enough to allow terrain editing, deformations, and rendering in real time.

Keywords: terrain representation; terrain editing; terrain visualization; caves; overhangs; cliffs; voxel; heightmap

1. Introduction

The most popular terrain representations are based on heightmaps, which usually define the terrain surface as a regular grid on a 2D plane. Each vertex on the plane is displaced along the height axis according to the heightmap. There are variations to this approach, such as using a nonuniform grid of vertices; however, as heightmap representations sample the terrain from a top-down view, they cannot represent volumetric features such as caves, overhangs, arches, and even vertical cliffs.

Volumetric (voxel-based) representations are inherently able to model various volumetric features because they sample the 3D space, but they have their own set of problems. Ray tracing, for example, is still very slow at rendering large and detailed terrains in real time. Modern GPUs are designed to render polygons efficiently, and many real-time rendering applications that use volumetric representations extract a polygonal surface to use hardware-accelerated rendering. This, however, incurs extra overhead. Because of sampling of the 3D space, a voxel representation is several orders of magnitude larger than a heightmap representation. Volumetric representations thus usually need to be compressed for memory efficiency, which incurs another performance

*Corresponding author. Email: gudukbay@cs.bilkent.edu.tr

penalty due to decompression in real time. It is also difficult to use smoothly varying level-of-detail (LOD) techniques with voxel representations.

To deal with the issues mentioned above, many real-time applications model volumetric features as separate 3D meshes and place them on relevant parts of the terrain. Because these meshes do not seamlessly blend into the terrain, various tricks are used to conceal artifacts where the model and mesh meet, such as placing rocks at the entrance of a cave. Such hacks, however, constrain the size and detail level of volumetric features because they cannot use optimizations specific to terrain models.

Dedicated graphics hardware has recently become common and powerful, and the quality of real-time renderings has greatly increased. Although a terrain can now be rendered in a much higher level of detail, due to the limitations of the traditional approaches, model complexity has not shown the same amount of improvement. Many simple and efficient algorithms are designed to operate on heightmaps but are not directly usable with any other representation. A new representation must be designed to benefit from existing algorithms, or new algorithms must be developed. We propose a terrain representation that can uniformly model elevation data and volumetric features in real-time applications to create a surface terrain, which is also able to represent the artifacts such as caves, overhangs, and cliffs. These features are not add-ons but rather they are modeled as part of the terrain itself.

1.1. Overview of the proposed approach

Heightmap-based representations cannot handle volumetric terrain features. Voxel representations cannot be directly rendered using hardware acceleration; they require large amounts of memory, and they lack the necessary algorithms for high-quality real-time rendering. We propose a hybrid terrain representation: we use a relatively low-resolution voxel representation to model coarse volumetric features. Then, 2D surface patches are generated to construct the polygonal terrain surface for rendering. To further increase the resolution, heightmaps are used to displace these 2D surface patches in a third dimension. Our approach contributes the following to the field:

- A hybrid representation that can model terrains with volumetric features such as caves, overhangs, arches, and cliffs.
- A surface extraction method to extract a polygonal surface of a volumetric representation, where the terrain surface is constructed using surface patches.

2. Related work

Terrain representation is essential for many types of applications, but there is no silver bullet that addresses the needs and constraints of all applications. Furthermore, real-time terrain rendering is closely coupled with advances in GPU technology.

2.1. Heightmap-based representations

Heightmap-based representations sample the terrain surface from a top-down view. This sampling can be performed regularly or irregularly. Regular sampling is easy to work with; the geometry is constrained, well-defined, and memory efficient. Since vertex connectivity is implicit, it is sufficient to store a grid of height values.

2.1.1. Triangulated irregular networks

Triangulated irregular networks (TINs), on the other hand, sample heights irregularly to represent more-detailed areas using more samples (Peucker *et al.* 1978). Kumlér (1994) states that a regular grid representation requires less memory than a TIN when their detail levels are equal. Computing a TIN is usually more complex, but algorithms such as Delaunay triangulation can be used to generate TINs. Peucker *et al.* (1978) propose an algorithm that creates the optimal triangulation of a terrain surface for a given number of triangles. It is more difficult to manage TINs once they are generated, as the whole model needs to be regenerated every time the resolution changes. Texturing TINs is also more complex than regular grids (Fan *et al.* 2004) and generating TINs is CPU-intensive. Garland and Heckbert (1995) propose an optimized algorithm for generating a TIN from a heightmap. The resulting TIN, however, may result in artifacts, such as thin triangles.

Gross *et al.* (1995) propose an efficient real-time LOD computation approach that uses a quadtree to represent the terrain surface. Cohen-Or and Levanoni (1996) propose a continuous LOD approach for TINs where several TINs at different LODs are generated and blended at the vertex level in real time to avoid popping artifacts. Lindström *et al.* (1996) propose a continuous LOD approach for terrains that use regular grid sampling of a heightmap. They divide the terrain into blocks of different LODs and store the blocks in a quadtree. The LOD computation is performed at the block level by selecting the appropriate block, and at the vertex level by selecting the important vertices. Although the internal representation uses a regular grid, the resulting geometry used for rendering is a TIN.

Evans *et al.* (2001) propose the right-triangular irregular network (RTIN), a more-restricted TIN, that using a bintree, ensures that the generated triangles are right-angled. The bintree provides more detail where it is needed by iteratively splitting the triangles at a higher level. The vertex positions and connectivity information are stored implicitly, making it more memory efficient than TINs. Duchaineau *et al.* (1997) propose another algorithm that uses RTIN, called ROAM: real-time optimally adapting meshes. This is one of the most popular real-time terrain-rendering algorithms because it addresses some important problems in an efficient way, such as LOD, view frustum culling, and creating triangle stripes for efficient rendering. ROAM utilizes frame-to-frame coherence to reduce computational cost and is one of the few approaches that supports real-time terrain deformation. ROAM can control the generated triangle count thanks to its hierarchical representation, but as it requires geometry updates on every frame, it is not as popular for modern GPUs. Several algorithms similar to ROAM are proposed but they have rather simple geometry updates and work on batches of primitives (Pomeranz 2000, Levenberg 2002, Ulrich 2002).

2.1.2. LOD management

Hoppe proposes progressive meshes which can work on any type of mesh, and later updates the approach to refine the mesh in a view-dependent way, taking view frustum, surface orientation, and screen-space geometric error into account (Hoppe 1997). Hoppe (1998) still later adapts the approach to real-time terrain rendering and introduces geomorphs to provide temporal coherence. Hu *et al.* (2009) then update it to utilize GPU parallelism.

Chunk-based LOD algorithms are not precise because they assume that the required LOD for the entire chunk is the same. Röttger *et al.* (1998) propose a precise continuous LOD algorithm for heightmap-based terrains, which uses a quadtree and works at the vertex level. The surface is generated by recursively visiting the quadtree in a top-down manner. The distance to the observer and surface roughness are taken into account so that smooth surfaces are rendered with fewer vertices even if they are closer. The continuity of the surface, however, requires there to be at most one LOD difference on the borders. Smooth LOD is obtained by geometry morphing.

Ulrich (2002) proposes to render massive terrains by combining the quadtree representation with RTINs. Each internal quadtree node stores its own chunk of geometry and texture. When a node is to be rendered, the geometry is sent to the GPU. Ulrich also proposes a paging scheme, where the node data is loaded on demand. Cignoni *et al.* (2003b) propose an approach where a hierarchical representation of vertex batches is stored in a bintree. This allows rendering massive terrains because it does not require the entire data to be loaded in the main memory. There is a high computational cost for preprocessing large terrains. The authors later extend this method to render planet-sized terrains (Cignoni *et al.* 2003a), using pre-fetching to guess the chunks that will be needed.

In the early 2000s, GPUs became mainstream and more powerful than CPUs for graphics processing. Real-time terrain representations and algorithms adapted to this by moving computations to GPUs. Algorithms thus focused more on feeding the GPU, rather than trying to fine tune and sort out each vertex on the CPU. One such method uses geometrical mipmaps (de Boer 2000), where a regular grid is divided into equal-sized square vertex batches. The mipmap level of a particular batch is determined by the distance to the observer, and vertex morphing is used to prevent popping. The method does not continuously update the geometry but rather updates the connectivity as necessary. Hence, much less data is sent to the GPU.

Livny *et al.* (2009) describe a LOD scheme that uses GPU to construct the meshes of terrain patches and uses cached textures to assign elevations to the vertices. Since the adjacent patches agree on the resolutions of the common edges, no cracks occur, and the transitions between different levels of detail are seamless.

Losasso and Hoppe (2004) propose geometry clipmaps, targeting an efficient LOD scheme for modern GPUs. This approach is based on texture clipmaps (Tanner *et al.* 1998) but operates on the terrain geometry. It uses a regular grid representation for the terrain, dividing it into grids of different LODs, where the LOD of a grid is determined by its distance. The approach makes use of vertex buffers for rendering. The entire data is loaded in a compressed format to the main memory and when a grid is to be updated, the relevant part of the data is decompressed, and the vertex buffers are updated. Transition regions are defined to prevent visual artifacts caused by different LODs. Each vertex is also morphed geometrically between different LODs to prevent popping. The geometry is updated in the CPU and sent to the GPU for rendering. Asirvatham and Hoppe (2005) extend this approach such that almost all computation is done on the GPU.

2.1.3. Terrain generation and editing

Hnaidi *et al.* (2010) describe a method for terrain generation from a set of parameterized curves that represent the terrain features. Their approach uses a multigrid diffusion algorithm that can be implemented on GPU. However, this approach allows to represent only terrain surface features, such as cliffs, ridge lines, and riverbeds. Treib *et al.* (2012) describe an efficient approach for editing of large terrains and use GPU ray-casting for

rendering to avoid height-field triangulation. However, it does not provide functionality to create volumetric features, only allowing to edit the terrain surface.

McAnlis (2009) uses a regular grid heightmap that allows individual vertices to be displaced by a full vector-field displacement along three axes rather than only along one axis. This makes it possible to add simple overhangs and vertical faces; however, the terrain resolution must be increased significantly to make up for vertex displacements. Similarly, McRoberts (2011) uses geometry images to store the displacement of regular grid vertices along three axes. Gamito *et al.* (2001) describe a procedural technique that applies nonlinear deformations to an initial height field surface to generate overhangs. Zhou *et al.* (2007) describe an example-based system for terrain synthesis from digital elevation models. They utilize a user-sketched feature map that describes terrain features on the resulting terrain and use patches from the example data that matches the specified features to generate the terrain. Bruneton and Neyret (2008) present a method that uses a view-dependent quadtree refinement scheme to edit large terrains with features such as roads, rivers, lakes, and fields. These approaches are restricted to edit terrain surface, not allowing volumetric features.

There are some hybrid terrain representations combining the grid representations and TIN networks that allow adaptive tessellation of the grid cells in the neighborhood of the TIN structure so that the discontinuities are removed (Bóo *et al.* 2007, Amor and Bóo 2008, Bóo and Amor 2009, Paredes *et al.* 2012). These representations only model the terrain surface, not allowing volumetric features. Their usage of the term ‘hybrid’ refers to the use of the grid and TIN representations together, whereas we use the term ‘hybrid’ to refer to the combination of volumetric and surface representations.

2.1.4. Terrain rendering

Mantler and Jeschke (2006) perform ray-casting in the fragment shader to render terrains. The data is stored as a texture representing the elevations. Ray marching is used to determine the point at which a pixel ray intersects the model. Ray-casting is optimized by an empty-space skipping method. The performance depends on the number of pixels; however, terrain size is limited by the largest texture size the GPU supports, unless the CPU is used to continuously update the elevation texture. This method cannot render volumetric representations because only the elevation data is available.

Dick *et al.* (2009) define a ray-casting-based surface rendering technique to overcome the limitations of mesh-based renderers for rendering large terrains. They achieve real-time frame rates for very large terrains; however, their method is also restricted to rendering terrain surfaces and cannot be used to render volumetric terrain features.

Tessellated surfaces use subdivision techniques to obtain high-quality curved surfaces. Displacement mapping can be applied to them to render highly detailed models (Lee *et al.* 2004, Bunnell 2005). Niessner *et al.* (2012) propose a GPU-based rendering technique for Catmull–Clark subdivision, which uses displacement mapping. These techniques can only be applied on the surface of a terrain and cannot model volumetric features.

There are some studies for terrain rendering that allows editing and deformation on the terrain surface. Bhattacharjee *et al.* (2008) describe a GPU-based system to store, render, and manipulate the terrains using a regular-grid representation. De Carpentier and Bidarra (2009) propose *procedural brushes* that combine local and global approaches for editing terrain surfaces used for game applications. Atlan and Garland (2006) describe a system for online multiresolution edits to a large terrain represented as a wavelet quadtree

hierarchy. Brandstetter III *et al.* (2011) describe out-of-core rendering of deformable large terrains where the deformation is limited to terrain surface.

2.2. Volumetric representations

2.2.1. Surface extraction from volumetric data

Most volumetric representations are based on voxels, making fine tuning possible. Ray tracing volumetric terrains with high resolutions is not very practical for interactive applications. The internal voxel representation is usually converted to a form that can be used for hardware-accelerated rendering. Almost all approaches use marching cubes (Lorensen and Cline 1987) or a variant for this purpose.

Geiss' (2007) approach can use a procedural density function or a discrete representation stored as a 3D texture. If a density function is chosen, the values returned are stored in a 3D texture and used in a second pass to do the actual rendering. This method uses a geometry shader to generate a polygonal surface and can render terrains with volumetric features, but generating density functions to obtain a desired terrain model is difficult. Our representation allows terrain editing at two levels: editing the coarse volumetric representation to outline the volumetric features and editing the terrain surface to apply fine details at an increased resolution over the coarse model by modifying heightmaps.

Forstmann and Ohya (2005) propose a similar approach, based on the interactive view-dependent isosurface rendering proposed by Gregorski *et al.* (2002) and inspired by geometry clipmaps (Losasso and Hoppe 2004). It uses clipboxes in 3D instead of clipmaps in 2D and is very efficient, but the resolution of the rendered sample models is too low, and further details must be added to the extracted surface by applying noise. This method shares most of the downsides of Geiss' (2007) method: it is difficult to represent a detailed terrain model with isosurfaces and to fine tune it.

To remedy the problems of marching cubes such as high memory requirements and aliasing artifacts, alternative isosurface extraction and rendering algorithms are proposed. Kloetzli *et al.* (2008) describe a volume representation format, called *Bézier Tetrahedra (BT)*, and describe a raytracing-based isosurface rendering technique utilizing GPU. Marinc *et al.* (2011) use ray-casting on the GPU to render quadratic C^1 splines extracted from the volumetric data represented as uniform tetrahedral partitions. Steinberger and Grabner (2010) describe an interactive multiresolution rendering method for isosurfaces of a voxel grid based on spline wavelet hierarchy.

Dual contouring is a scheme that extracts isosurfaces from adaptive octrees using triangulation (Ju *et al.* 2002). In our approach, we are not merely interested in extracting polygons that correspond to the given voxels. The polygons are not the output of the proposed framework, but rather the output of the surface extraction step. We then cluster triangles for surface generation for the purposes of applying the LOD scheme and terrain editing and deformation. Thus using marching cubes and its variants, dual contouring, or any similar surface extraction algorithm does not completely fulfill the requirements of the surface generation step of our approach.

Rendering voxel-based large volumetric terrains in real time has not been popular until recently due to GPU and memory limitations and problems related to the algorithms used to extract polygonal surfaces of voxel representations. One such problem is the difficulty of LOD management in surface extraction. Marching cubes and other similar algorithms do not work well when the resolution of the sampling grid is not constant. LOD

approaches, however, require sampling grid resolutions to vary among different LODs, which causes artifacts, such as cracks, at the LOD boundaries. Lengyel (2010) proposes the Transvoxel algorithm, which eliminates visual artifacts of marching cubes and thus allows LOD management in real time. However, popping artifacts occur because there is no morphing between different LODs, and the method is CPU intensive because it frequently updates the geometry by recomputing parts of the terrain surface as the view-point changes; CPU-to-GPU geometry updates are frequent for the same reason.

2.2.2. Ray-casting-based volume rendering

Ray-casting-based volume rendering (Gobbetti *et al.* 2008) requires a very high resolution to obtain a detailed terrain rendering, and their suitability for interactive exploration of very large terrain models is questionable because of memory requirements and compression/decompression of the volumetric data.

Laine and Karras (2011) define a data structure for storing voxels and an algorithm for ray-casting highly detailed volumetric models using this structure. Their work is similar to our hybrid representation in that they augment the voxel data with contour information to increase geometric resolution, allowing smooth surfaces over the volume data. Their approach does not provide editing functionality to create volumetric features.

Hybrid sample-based surface rendering (Reichl *et al.* 2012) is proposed as an alternative to mesh-based rendering to improve the rendering performance. This technique uses rasterization and ray-casting in every frame simultaneously to determine eye-ray intersections. It selects the most optimal technique at run time for each part.

Gigavoxels and its extensions (Crassin *et al.* 2009, 2010) provide an elegant approach for efficient and detailed rendering of general volumetric models, not restricted to terrains. However, these approaches do not provide functionality of editing volumetric data to create volumetric features or deforming the existing model.

2.3. Hybrid (layered) terrain representations

Peytavié *et al.* (2009) propose a framework for modeling complex terrains with volumetric features. Similar to ours, their approach uses a hybrid terrain model combining volumetric and implicit surface representations. They use a layered volumetric representation where a terrain is defined as a 2D grid of material stacks; the layers are air, water, sand, bedrock, and rocks. They construct volumetric features such as overhangs, arches, and caves by inserting an air layer between two bedrock layers. They generate the surface of different material layers using implicit surfaces. The convolution surfaces they use for this purpose smooth the surfaces of different material layers. The difference of our approach from theirs is in the formation of the hybrid terrain representation; we use an adaptive octree structure for the coarse level volumetric representation whereas they use a layered 2D grid of material stacks. We generate the surface of the terrain using biquadratic Bézier surface patches whereas they use implicit surfaces. We also apply a LOD scheme for efficient visualization and rendering of complex large-scale terrains. Both approaches allow editing and realistic rendering of terrains with textures and shadows. They use a physically based simulation to automatically stabilize layers of sand and rocks, which removes the burden of finely editing details. The extension by Löffler *et al.* (2011) supports terrain rendering with different levels of granularity.

Loeffler *et al.* (2012) propose a method to extract bicubic patches from volume data.

3. The proposed approach

The proposed approach has the following properties:

- It can model anything a heightmap approach can; in addition, it can model volumetric features such as caves, overhangs, cliffs, and arches.
- It supports interactive frame rates with the exploitation of hardware-accelerated rasterized graphics. Hence, it is suitable for rendering using a modern GPU.
- It represents the terrain surface in terms of smaller logical parts (chunks), which makes the algorithms more efficient because they can work at a level higher than that of primitives such as vertices and triangles.
- It allows editing and deformation dynamically in real time where editing only causes local changes. The proposed algorithms to update data structures stored in the CPU and GPU can work in real time. Changes made to data stored in the CPU are reflected in the data in the GPU's video memory in real time; thus, the amount of data sent through the CPU–GPU data bus does not exceed the capabilities of a modern GPU.
- It can handle fairly large terrains, as long as they can still fit into the main memory. Paging schemes can be used to render data sets that do not fit.
- It is suitable for applying basic 3D rendering elements such as lighting, texturing, and shadowing.
- Rendering large terrains in real time without proper LOD support is not easy; thus, we define an LOD management scheme for the proposed terrain representation.
- Visual artifacts do not occur while extracting the terrain surface, rendering, or at the LOD boundaries where the terrain resolution is changed abruptly to accommodate for difference in surface proximity.

3.1. Terrain representation

The internal representation used to store terrain data should be efficiently convertible to a polygonal mesh. Modern GPUs are designed to accelerate rasterization-based polygon rendering. Voxel representations of large and detailed 3D models are not suitable for real-time applications because of intensive memory usage. To achieve a 1 m resolution at each axis in a 1 km³ working space, it is necessary to store one billion voxels. Even if a single byte of data is used to store each voxel, this representation would require about 1 GB of memory just for the terrain. Approaches based on heightmap and voxel representations alone do not suffice to achieve the defined goals. The proposed hybrid approach combines the expressive power of voxel-based approaches, and the simplicity and efficiency of heightmap-based approaches. In our approach, the terrain geometry is generated in two steps:

- (1) First, a relatively low-resolution voxel representation is used to coarsely define the terrain geometry. The surface geometry is extracted using a novel technique such that it consists of regular patches.
- (2) Next, each terrain patch is assigned a heightmap, which is used to displace the vertices of that patch. This increases the resolution of the terrain surface.

Our approach uses an octree to store the volumetric representation to develop a compact representation by exploiting large groups of occupied and empty voxels. The resolution of the initial coarse volumetric representation is a performance/quality trade-off issue. If we

choose a high resolution, the editing and deformation of the volumetric representation will be computationally intensive, but the volumetric features, such as caves, overhangs, and arches, could be represented well, whereas when we choose a low resolution, the editing and deformation will be easier, but the quality of the volumetric features will be low. It should also be noted that it is possible to adaptively increase the resolution of the volumetric representation at detailed parts of the terrain.

Patches are used to generate the surface geometry represented by voxels. Each patch is then subdivided into a number of triangles for rendering. Thus, the primitive type of the representation is patches, while the primitive type used at the rendering stage is triangles. Any type of terrain data obtained from different sources can be transformed to a form that fits into our representation. For example, height fields could be used to deform the surface of the terrain. This could be done by simplifying a 2.5D terrain model using polygonal simplification techniques to match the resolution of the surface of the volumetric representation (the surface grid) and using the height field data as the control points for the biquadric Bézier surfaces to generate the terrain surface.

3.2. Surface extraction

Surface extraction is the process of computing the coarse terrain surface from the 3D voxel model. This is not the ultimate form of the terrain surface but only an intermediate representation that will be used for terrain surface generation. Marching cubes and its derivatives are popular in extracting the surface of a volumetric representation. In our approach, however, we cannot use such an algorithm because its output is just a bunch of triangles, i.e., a triangle soup. Our approach generates the surface as regular patches on which heightmaps can be applied to achieve a higher resolution. The generated patches must have the following properties:

- Connected and continuous to prevent rendering artifacts; there must be no holes, overlapping or colliding patches.
- Rectangular so that vertices can be mapped to planar (u, v)-coordinates and values in the heightmap can be used to displace the patch vertices.
- Smooth on the interior; terrains usually lack sharp corners and edges. The combination of patches must also not introduce sharp edges, corners, or dramatic slope changes (continuity on the boundaries of patches); it is possible to add such details by displacing the vertices using heightmaps.
- Each edge exactly aligned with only one edge of only one other patch. This requirement simplifies the surface structure so that LOD algorithms can work more efficiently and produce better results.
- Vertices generated in a controlled manner such that the overlapping edges of two patches coincide, which makes it easier to seamlessly combine them.

We choose Bézier surfaces as the underlying patch representation (and use the term *Bézier surface* and *terrain patch* (or just *patch*) interchangeably).

3.2.1. Two-dimensional case

Our approach obviously needs surface extraction to work in 3D. A 2D version of the extraction process (curve extraction) is similar to the 3D case and easier to comprehend visually. We explain the 2D version first and build the 3D version on top of it.

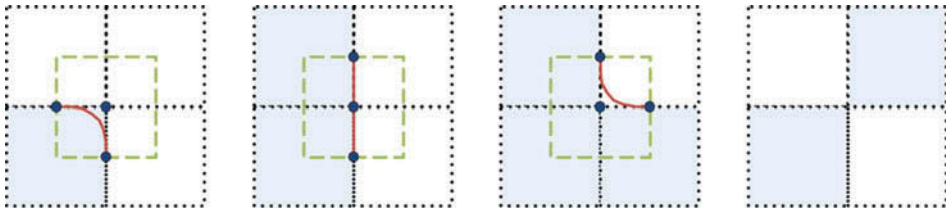


Figure 1. Several quadtree cell configurations in the 2D case. The green rectangles show the intersection zones.

The extraction algorithm works on each vertex at the intersection of four quadtree cells. For each such vertex, the algorithm generates a curve inside the intersection zone. This zone is rectangular, centered at the intersection vertex and is equal to the size of a cell. Since an intersection zone overlaps four cells, and each cell can be either empty or occupied, there are $2^4 = 16$ possible configurations. Several possible configurations are shown in Figure 1. Before the extraction process, the configurations must be normalized, where all occupied cells share an edge with another occupied cell in the same configuration. The configurations in Figure 1a, b, and c are normalized, while the one in Figure 1d is not.

In the normalization process, each cell is first put in a normalized group. Then the algorithm combines any two groups where the cells in the groups share an edge. This step is repeated until no more normalized groups can be combined (see Figure 2). Normalization is useful for decreasing the complexity of the extraction algorithm; a curve can be extracted for each normalized group, and these curves can be combined if there is more than one normalized group. Each intersection zone in a normalized cell group has three control points for curve generation (see Figure 1). The proposed approach uses quadratic Bézier curves because of their simplicity.

In Figures 1 and 2, the control points are depicted as solid circles, and the generated curves are drawn in red. The curves extracted for some configurations are simply flat lines. The algorithm does not generate a curve if all or none of the cells are/is occupied. The curve extraction algorithm performs the following steps for each intersection zone in the cell space:

- (1) If all cells in the zone are occupied or are all empty, skip the zone without generating any curve.
- (2) Split cells into normalized cell groups.
- (3) Determine the control points for the curve of each cell group, depending on occupied cells.
- (4) Generate curves for each normalized cell group.

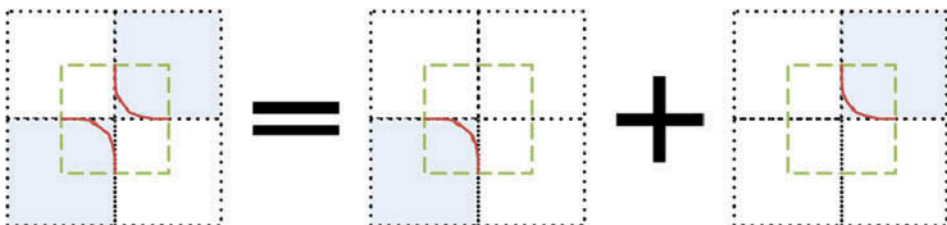


Figure 2. Configurations may be split during normalization.

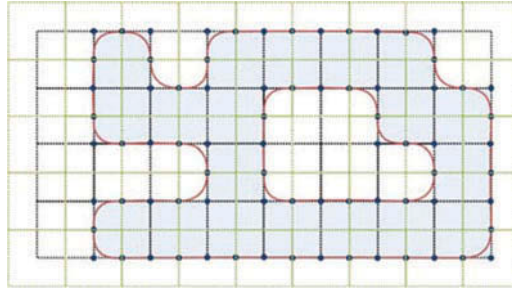


Figure 3. The curves extracted by the proposed algorithm.

Figure 3 illustrates 2D curve extraction. The blue rectangles are occupied cells, the white rectangles are empty cells, and the solid circles are control points of the curves of each intersection zone, which are shown as green rectangles. The control points of adjacent intersection zones are placed in such a way that one control point is shared between them, which ensures the continuity of the generated curve. The algorithm may generate many unconnected curves if the data represents such an area.

3.2.2. Three-dimensional case

The surface extraction algorithm also works on intersection vertices in the 3D case, where the intersection zone is a volume centered at the intersection vertex. The size of the intersection volume is equal to the size of a voxel and it overlaps eight voxels. The intersection volume is defined by whether each of these eight voxels is occupied; thus there are $2^8 = 256$ possible configurations.

In the 3D case, biquadratic Bézier surfaces are used to generate the surface for each intersection volume (see Figure 4a). The algorithm does not generate a surface if all voxels in an intersection volume are occupied or all are empty. For all other cases, there must be at least one Bézier surface for the volume. When there are more than four edges to be included in the surface, up to three Bézier surfaces may be required. For example, Figure 4b shows a configuration with five edges to be patched: (A, B, C), (C, D, E), (E, F, G), (G, H, J), and (J, K, A). Two Bézier surfaces patch two external edges and one internal edge, while one surface patches one external edge and two internal edges. External edges are shared between surfaces of different intersection volumes. Edges shared between surfaces in the same intersection volume are called internal edges, and are introduced

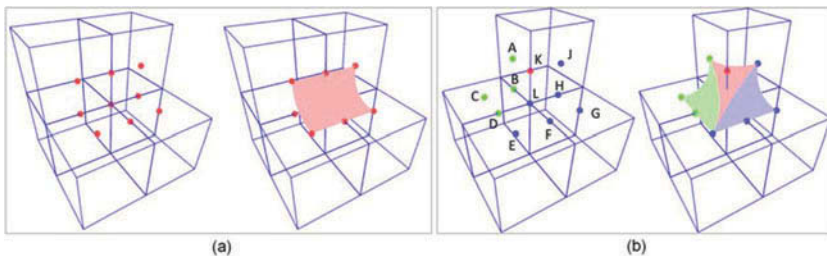


Figure 4. (a) A normalized intersection configuration with six occupied voxels and the generated surface. (b) An intersection configuration where three Bézier surfaces are required to generate a connected surface.

when there are multiple surfaces in an intersection volume. The control points of these three surfaces are:

- Surface 1: {A, B, C}, {A, L, D}, {A, L, E};
- Surface 2: {J, H, G}, {J, L, F}, {J, L, E}; and
- Surface 3: {A, L, E}, {K, L, E}, {J, L, E}.

Internal edges {A, L, E} and {J, L, E} are shared between surfaces, which ensures connectivity. For some configurations it is necessary to collapse one edge of the Bézier surface to obtain a surface with three edges. In [Figure 4b](#), all three surfaces are generated in this way. The three control points of Surface 1, for instance, are A, where one edge has been collapsed into a single point.

It is possible to compute a bit string of eight that represents the intersection volume configuration, with each bit representing whether the corresponding voxel is occupied or not. This bit string can then be used as a pointer to a lookup table to retrieve the precomputed control points for the surface of that volume configuration. There can be 9, 18, or 27 precomputed control points for each volume configuration, depending on the number of surfaces defined for that case. For the connectivity and continuity of the terrain surface, it is essential that the control points of the adjacent volumes coincide. Furthermore, if a configuration contains multiple Bézier surfaces, the control points of each edge must either be shared with another surface defined for the same configuration or shared with a surface defined for an adjacent volume configuration (see [Figure 4b](#)).

For 3D surface extraction, intersection configurations must be normalized before the surface extraction process is applied to the intersection volume, as in the 2D case. In the 3D case, however, each occupied voxel in a normalized intersection volume must share a face with another occupied voxel in the same normalized intersection volume rather than an edge. Voxel index fields can be used to determine what is shared among two voxels in the same intersection volume. They share

- a face if two of the index fields are the same;
- an edge if one of the index fields is the same;
- a vertex if no index fields are the same.

Although there are 256 possible configurations, most are related to a base configuration in one of the following two ways:

- symmetric about the xy -, xz -, or yz -planes or
- rotated by 90° , 180° , or 270° around x -, y -, or z -axis.

Surfaces for such configurations can be obtained by applying affine transformations to the Bézier surface control points of the corresponding base configuration. Of the 256 configurations, there are only 12 unique ones, excluding the cases where no surface is generated. These unique configurations, the control points for the surfaces, and the surfaces generated are shown in [Figure 5](#).

3.2.2.1. Handling voxels at different levels. The surface patch generation method presented here assumes that the sizes of all voxels in an intersection volume are equal; i.e., they are at the same level. In practice, this may not be true; the octree allows neighbor voxels to be at different levels. To handle these cases, the surface extraction algorithm

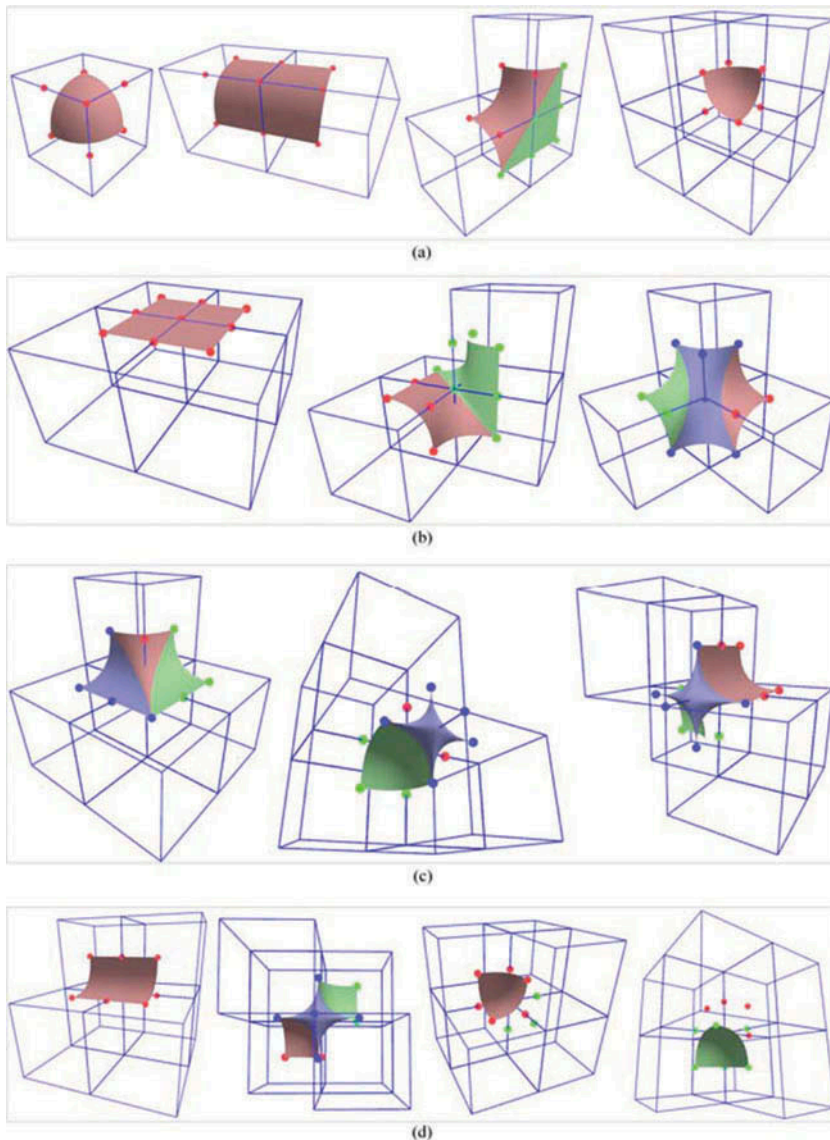


Figure 5. Unique intersection volume configurations. (a) One, two, three, and seven voxels are occupied, respectively. (b) Four voxels are occupied. (c) Five voxels are occupied. (d) Six voxels are occupied (the last two in (c) and (d) are the same configurations from different views).

computes the maximum level of voxels in the intersection volume and generates several new smaller volumes on the surface of the voxels that are on a lower level than the maximum level (see Figure 6). The surface patches for these smaller volumes are generated instead of the larger one, and only for the three faces of the larger voxels that overlap the actual volume. The inner volumes of these larger voxels are not processed because no surface is generated inside a voxel.

In actuality, larger voxels are not split into smaller voxels; the octree representation does not change in memory. Smaller intersection volumes are generated and processed on

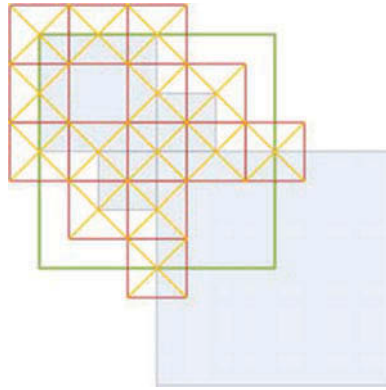


Figure 6. The division of the actual intersection volume (green rectangle) into smaller intersection volumes (red squares with yellow X's inside).

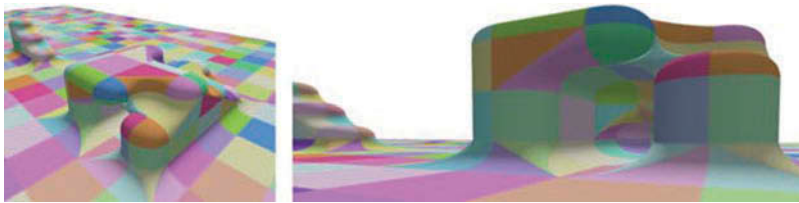


Figure 7. The result of surface extraction from two different views (patches are rendered in different colors).

the fly, and queries for smaller virtual sub-voxels are answered considering the larger voxel they belong to. If the larger voxel is occupied, then all its smaller virtual sub-voxels are also occupied, and vice versa. After we find out the occupied virtual sub-voxels by also handling voxel level transitions, the surface extraction algorithm generates surface patches for each virtual sub-voxel. The result of the surface extraction for a sample voxel configuration is shown in Figure 7.

3.3. Terrain surface generation

The 3D voxel model defines coarse terrain; surface patches must be generated from this representation to render the terrain using hardware acceleration. We do this with parametric equations and apply heightmaps to each patch to obtain the final surface.

3.3.1. Generating vertices

The first step of terrain surface generation is to generate a number of vertices for each Bézier surface. The minimum number of vertices to approximate each patch is nine, in which case each vertex coincides with a surface control point. The actual number of vertices per patch is determined by the desired LOD and is constrained by the memory available for storing these vertices.

At this stage of surface generation, only the original vertex positions are computed (rather than the displaced positions); the heightmaps will be applied at a later stage.

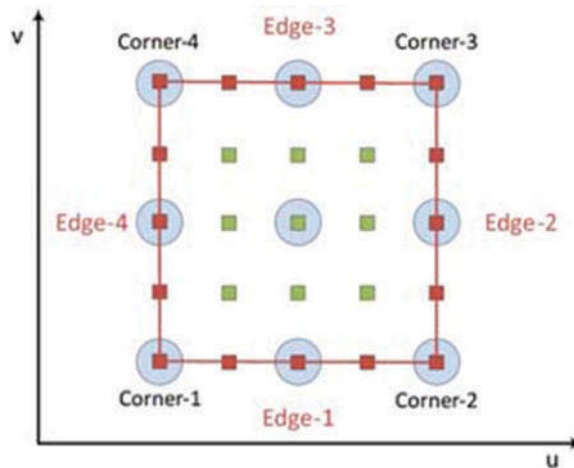


Figure 8. Vertices that approximate a patch where $N = 5$. Red boxes are border vertices, green boxes are internal vertices, and blue circles are control points.

Computing vertex positions is straightforward, using the parametric surface equation of the biquadratic Bézier surfaces. In the proposed surface generation method, the numbers of vertices at each surface edge are equal, meaning that surface parameters u and v are uniformly sampled. The total number of vertices on a surface patch is N^2 , where the number of vertices per edge is N . It is possible to use different frequencies for u and v . This usage, however, results in vertex alignment problems in neighbor surface patches. It must be ensured, therefore, that vertices on the edges of neighbor surface patches align correctly (this constraint is later relaxed for LOD purposes). Each vertex can be categorized as an *internal* or a *border* vertex (see Figure 8).

Border vertices are shared vertices, meaning that multiple vertices spatially coincide and are located at exactly the same coordinates in 3D. The vertices can be shared externally or internally, relative to the surface patch for which the vertex is generated. External and internal sharing occur on the edges and corners of patches, i.e., for border vertices. Internal vertices are never shared.

Internal sharing of a vertex means that multiple vertices that belong to the same patch are located at the same coordinates, which occurs when one edge is collapsed into a point so that there are three edges instead of four.

External sharing of a vertex means that multiple vertices that belong to different patches are located at the same coordinates, which occurs on the common edges and corners of neighboring patches. Vertices strictly on patch edges (i.e., not on corners) are shared by exactly two patches because each edge of a patch can coincide with an edge of only one other patch. Corner vertices, however, are shared among at least four patches. In some configurations, the number of patches sharing a corner vertex can be more than four. To determine the externally shared instances of a vertex that belongs to patch P , it suffices to examine the patch vertices of the voxel that includes patch P and its neighbor voxels. To improve the performance, first the bounding boxes of the patch pairs are compared to check for overlap.

3.3.1.1. Storing vertices. There are a total of N^2 vertices in a patch, where a single edge is approximated by N vertices. Of those N^2 vertices, $4 \times (N - 1)$ are shared (i.e., border) vertices. The attributes of shared vertices should not be stored separately because:

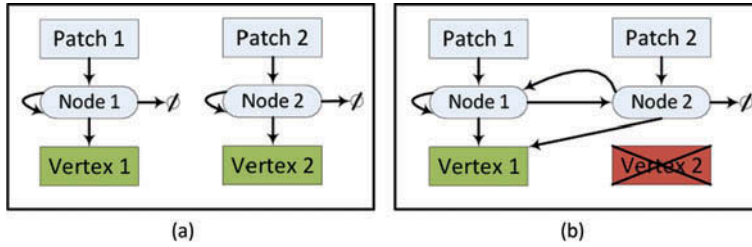


Figure 9. (a) Two distinct shared vertex lists, each having one node and pointing to different vertices. (b) The shared vertex lists are merged into a single shared vertex list, and the remaining vertices (Vertex 2 in this case) are deleted.

- this is redundant and causes the application to unnecessarily use more memory space,
- different heightmaps are applied to different patches, and thus displaced vertex positions may cause gaps in the terrain surface, and
- this causes normals to be computed incorrectly.

For efficiency and correct operation of the surface generation algorithm, shared vertices must be stored in a common data structure. The number of patches that share a border vertex is not constant; it depends on the voxel intersection volume configurations around the vertex. A linked list is a good choice for storing shared vertex data, where each node of the list represents an instance of the shared vertex, and each node is stored by a patch that shares the vertex.

Each shared vertex list node stores a pointer to the first and next node in the list, a pointer to a patch that shares the vertex, and a pointer to the actual vertex where the vertex attributes are stored. The number of shared vertex list nodes for a shared vertex is always equal to the number of patches that share the vertex. Initially, i.e., at the vertex generation stage, a separate node is generated for each border vertex. Once all vertices are generated for all patches, the algorithm searches for externally shared vertices by comparing the locations of border vertices of neighbor patches. If two vertices are in the same location, their shared vertex lists are merged (cf. Figure 9).

It is essential to maintain the mapping between patch vertices and the 2D parameter space. Vertices must be quickly accessible with a 2D vertex index in the format (i, j) , as this provides a mapping between the vertices and the (u, v) -coordinates. The range of both i and j is $[0, N - 1]$, where N is the number of vertices per edge.

To allow fast access to vertices through the 2D vertex indices, a 1D array of vertex pointers of size N^2 is created to store pointers to the vertex attributes. Different elements of the array can store pointers to the same vertices where the vertices are shared. The attributes of shared vertices are stored only once in the main memory.

3.3.2. Generating faces

The vertices generated for each patch are connected to form triangles that approximate the surface with no holes or cracks. Figure 10 shows different connection patterns. Although the pattern shown on the left is simpler, the pattern on the right yields better results with LOD. The minimum number of vertices per edge required by this connection pattern (N) is three. The connection pattern of size 3×3 can then be tiled along the u - and v -axes for $\frac{N-1}{2}$ times. This places a constraint on the number of vertices per edge in a patch, such that $N = 2 \times k + 1$, where

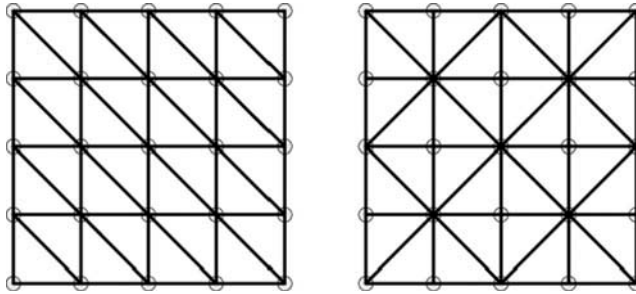


Figure 10. Two patterns for connecting vertices ($N = 5$).

k is a positive integer. The primitive pattern is then tiled k times along the u - and v -axes. The total number of triangles generated for a patch is $8 \times k^2$, which is equal to $2 \times (N - 1)^2$.

3.3.3. Vertex normals

The presented approach requires vertex normals because most operations are performed per vertex. Each vertex has two kinds of normals:

- (1) The *displacement normal* merely defines the direction through which the vertex is displaced according to the applied heightmap values. This normal depends only on the voxel definition of the terrain surface and does not change when the vertices are displaced by a heightmap.
- (2) The *vertex normal* is used for rendering, i.e., lighting and texturing. This normal changes when the displaced position of that vertex or one of the connected vertices is modified. The vertex normals must be recomputed in such cases.

3.3.4. Displacement of terrain surface vertices

This is the last step in terrain surface generation. It is performed using the original vertex positions and the displacement normals. The displaced positions can be computed by Equation (1), where \vec{P} is the original position, \vec{N} is the displacement normal, and \vec{D} is the displaced position. The 2D $h(u, v)$ function is the heightmap function, returning a scalar value for planar (u, v) -coordinates, where the range for u and v is $[0, 1]$.

$$\vec{D} = \vec{P} + \vec{N} \times h(u, v) \quad (1)$$

3.3.5. Terrain deformation

The terrain should be editable and deformable in real time. *How* to deform the terrain, e.g., by erosion or by other objects, is beyond the scope of this study, we do try to define a framework for deforming the terrain in real time. The limitation of our approach, however, is that terrain deformation can only be performed at the heightmap level in real time; it is not possible to smoothly modify the voxel model in real time because that would trigger surface extraction.

Editing heightmaps causes the displaced positions of affected vertices to change. Modifying a displaced vertex position invalidates the face normals of all triangles sharing

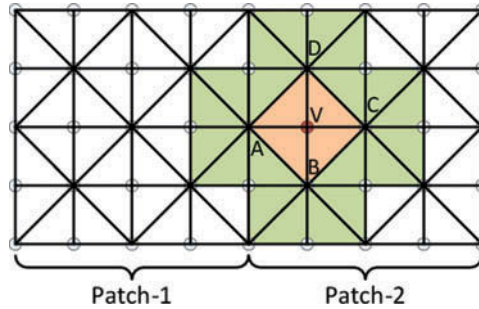


Figure 11. Modifying the displaced position of Vertex V causes the vertex normals of A, B, C, and D to be recomputed using the face normals of the colored triangles.

that vertex, which invalidates the normals of the vertices that constitute these triangles. Modifying the heightmap of one patch affects the vertices of neighbor patches. For example, in Figure 11, if the displaced position of Vertex V is modified, the face normals of the orange triangles are invalidated. This then invalidates the vertex normals of A, B, C, and D and recomputing them requires access to the orange and green triangles. Although the displaced position of a patch's internal vertex has been modified, face normals of affected triangles must be recomputed because face normals are not stored.

A straightforward solution to this problem could simply be to recompute the vertex normals of the entire terrain. However, even for only moderately large terrains, it may not be possible to do this in real time. In the proposed approach, the granularity of vertex normal recomputations is defined as patches, which means that whenever a patch's heightmap changes, the vertex normals of that patch are marked as invalid. These vertex normals are updated as follows:

- (1) Invalidate all vertices of the modified patch.
- (2) Invalidate all triangles of the modified patch and its neighbors if at least one vertex is invalidated.
- (3) Invalidate all vertices of invalidated triangles.
- (4) Recompute vertex normals of invalidated vertices.

4. Performance evaluation

The properties of the test environment are depicted in Table 1. Statistics of the sample terrain generated for performance evaluation are given in Table 2. (The number of nontrivial voxel intersection volumes are those for which one or more patches is generated.) Static surface culling eliminates patches will never be seen in practice. The statistics of the terrain surface generated with different parameters are given in Table 3. Figure 12 shows still frames of the visualization of a terrain with volumetric features.

Table 1. Test environment.

<i>Operating system</i>	Windows 7 Professional x64, SP1
<i>CPU</i>	Intel Core i7-2600K 3.4 GHz
<i>#CPU cores</i>	4 + 4 (Hyper-threading)
<i>Main memory</i>	8 GB DDR3
<i>GPU</i>	nVidia GeForce GTX260
<i>CPU-GPU data bus</i>	PCI Express x8 Gen2

Table 2. Coarse terrain model statistics of the test scene.

# Occupied voxels	24,239
# Nontrivial voxel intersection volumes	2544
# Patches (without static surface culling)	7481
# Patches (with static surface culling)	3560

Table 3. Terrain surface statistics of the test scene.

Max. LOD	# Patch vertices	# Patch triangles	# Total vertices	# Total triangles
1	3×3	8	32 K	28 K
2	5×5	32	89 K	114 K
3	9×9	128	288 K	456 K
4	17×17	512	1 M	1.8 M
5	33×33	2048	3.9 M	7.3 M
6	65×65	8192	15 M	29.2 M

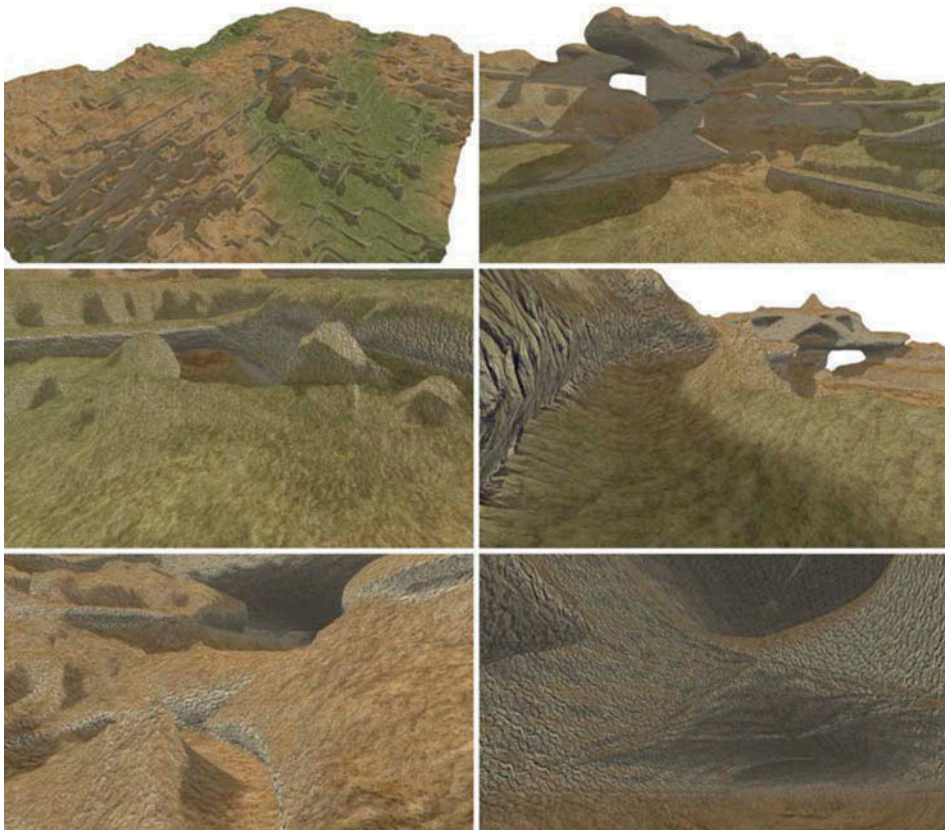


Figure 12. Still frames from the visualization of a terrain with volumetric features.

We use an artifact-free LOD management scheme to render large terrains in real time. The LOD scheme maintains temporal continuity while the view parameters are changing by using geometry morphing to support smooth transitions. The LOD scheme is based on the idea that the areas of all displayed triangles on the terrain surface should be more or less equal. As the observer moves farther away from a patch, the patch LOD becomes lower and the number of triangles on the same surface area decreases, and consequently, the area covered by a single triangle increases. The LOD indices of each patch can dynamically change at run time. A constant maximum LOD is determined when the terrain is created. All vertices at the highest LOD are then computed and stored with each patch. Once a maximum LOD is set, no patch can be rendered at a higher LOD than the maximum. When it is desired to render a patch at a low LOD, a subset of the vertices is selected accordingly.

4.1. Memory usage

Most of the main and video memory space is used to store vertex attribute data (see Tables 4 and 5), which requires 52 bytes each of the main memory and the video memory. It is necessary to store the vertex attributes on the main memory so that when the terrain is deformed they can be updated, but they can be discarded if real-time terrain deformations are not needed. The main memory usage for the test scene is shown in Table 6. Memory space required to store the vertex attributes and the triangle vertex indices is the most significant. Vertex attributes stored in the main memory and the video memory differ because some attributes stored in the former are not used in GPUs, and the video memory is usually considerably smaller than the main memory.

Table 4. Vertex attributes stored in the main memory.

Vertex attribute	Data type	Size
Original position	float [3]	12 bytes
Displacement normal	float [3]	12 bytes
Actual normal	float [3]	12 bytes
Displacement	float	4 bytes
Lower LOD neighbor	pointer [2]	8 bytes
LOD index	unsigned int	3 bits
Re-compute normals	unsigned int	1 bit
Vertex buffer index	unsigned int	28 bits

Table 5. Vertex attributes stored in the video memory.

Vertex attribute	Data type	Size (bytes)
Position	float [3]	12
Normal	float [3]	12
Geomorph transition threshold	float	4
Geomorph transition position	float [3]	12
Geomorph normal	float [3]	12

Table 6. Main memory usage of the test scene.

Max. LOD	Vertex attributes (MB)	Shared vertices (MB)	Index data	Heightmaps (MB)
1	0.3	0.67	512 KB	0.1
2	1.6	1.5	1.7 MB	0.4
3	9	3.2	7.3 MB	1.2
4	40	6.5	29 MB	4.1
5	175	13.3	117 MB	15.5

Table 7. Video memory usage of the test scene.

Max. LOD	Vertex attributes (MB)	Index data (avg.)	Textures (MB)
1	1	60 KB	6
2	3	125 KB	6
3	12	250 KB	6
4	46	1 MB	6
5	188	4 MB	6

The video memory usage of the test scene is shown in [Table 7](#); the video memory space required to store shadow maps is not included because that depends on the number of cascaded shadow maps used. We use four cascaded shadow maps of resolution 2048×2048 with 24-bit depth; hence the video memory required is 48 MB.

4.2. Performance

4.2.1. Terrain surface generation

The performance of the terrain surface generation is depicted in [Table 8](#). The columns represent the time required to extract the surface of the coarse voxel model, approximate the patches by creating vertices, find shared vertices between patches, compute displacement and vertex normals, and compute triangle vertex indices for each patch and different LODs. The surface extraction step does not depend on the maximum LOD because it is not affected by the number of vertices approximating a patch. These steps are performed only once, at the terrain creation or loading step.

Table 8. Performance of the terrain surface generation.

Max. LOD	Surface extraction (ms)	Surface approx. (ms)	Shared vertices (ms)	Normals (ms)	Indices (ms)
1	40	15	55	6	4
2	40	20	70	22	13
3	40	40	110	60	50
4	40	90	230	240	150
5	40	210	660	890	460

4.2.2. Terrain surface deformation

When the terrain is deformed in real time, the vertex displacements are computed, vertex normals are recomputed, and updated vertex attributes in the main memory are copied to vertex buffers. Table 9 lists the execution time of updating vertex displacements, recomputing vertex normals, and updating vertex buffers in the video memory.

4.2.3. Rendering

We render the scene geometry in $N + 1$ passes, where N is the number of cascaded shadow maps. The first N passes render each of the N shadow maps, and the final rendering pass renders the actual terrain surface using the shadow maps and applying various per-pixel effects, such as lighting and texturing. The performance-related statistics of the rendering passes for maps of resolution 2048×2048 are listed in Table 10. Table 11 shows the overall rendering performance. The values in Tables 10 and 11 are the averages for the flythrough over the test scene.

Table 9. Performance of real-time terrain surface deformation.

Max. LOD	Updating vertex displacements	Recomputing normals (ms)	Updating vertex buffers (ms)
1	0.6 μ s	6	16
2	2 μ s	25	22
3	6 μ s	0.1	74
4	24 μ s	0.3	0.28
5	0.1 ms	1.2	1.1

Table 10. Performance of shadow map rendering passes.

Max. LOD	# Shadow maps	Frustum culling (ms)	Index buffer updates (ms)	Rendering (ms)
	1	0.05	0.04	0.6
4	2	0.1	0.14	1.1
	4	0.2	0.3	2.3
5	1	0.05	0.3	2.1
	2	0.1	0.6	4.3
	4	0.2	1.1	8.5

Table 11. Overall performance of the proposed rendering pipeline (frame rates are given as frames per second).

Max. LOD	# Shadow maps	Frustum culling (ms)	Index buffer updates (ms)	Rendering (ms)	Frame rate
4	1	0.05	0.15	3.9	235
	2	0.1	0.25	4.4	205
	4	0.25	0.4	5.4	160
5	1	0.05	0.7	10	91
	2	0.1	1	12.1	74
	4	0.25	1.5	16	57

5. Conclusions and future work

We propose a hybrid terrain representation that combines voxel- and heightmap-based approaches. Our method models volumetric terrain features such as caves, overhangs, arches, and vertical cliffs using a coarse voxel-based approach. Terrain surface resolution is then increased further to add surface detail by applying heightmap displacement. To do this, we extract the terrain surface from the voxel model, where the extracted surface consists of patches similar to the blocks of a 2D regular grid heightmap.

The proposed approach provides a useful trade-off between the simplicity and efficiency of heightmap approaches and the expressiveness of volumetric approaches. Because the terrain surface consists of patches that can easily be mapped to 2D planar coordinates, we can, with minimal changes, use many of the existing algorithms designed for regular grid representations.

The proposed approach provides a suitable representation for detailed geographical information system (GIS) analysis, such as the coverage calculation for the communication and information systems, military GIS analysis, such as operational planning. The educators and scientists can explore the navigation and walkthrough functionality to explore volumetric features such as caves. The proposed framework can be extended to be used by natural resource managers to plan for pipelines, local governmental institutions may use it to make more detailed plans for their cities.

Our approach can be exploited in a number of ways:

- *Vector field displacement* could be used to apply displacement on patches along all directions rather than just the direction of the displacement normal, as described in McAnlis (2009).
- *Hardware-accelerated tessellation* is one of the newest methods supported by modern GPUs and it could be used to tessellate patches to increase terrain surface resolution.
- *Collision detection* algorithms could be developed specifically for the proposed representation. It is possible to create a voxel representation for the sole purpose of answering physics queries efficiently. The resolution of this representation would be determined by the desired level of precision, and because it would only be used for physics computations, it would not need to store visualization attributes.
- *Terrain synthesis* techniques could be developed to convert a high-resolution voxel representation to the proposed one. This could be done by first lowering the resolution of the given voxel representation, which could then be used as the coarse representation in the proposed approach. The actual high-resolution representation could then be approximated with the displacement of patch vertices.
- A *2.5D terrain model* (i.e., height field) could be converted into a volumetric representation by carving a volumetric block composed of a regular three-dimensional grid of voxels. The 2.5D height field data could be used to turn the voxels on or off. This requires the resolutions of the height field and the regular volumetric grid must be identical. To convert this regular volumetric representation to our coarse volumetric representation, the regular grid structure must be converted to an adaptive octree structure. We can use algorithms for converting a regular grid volumetric model to an octree. However, this approach can only model the terrain surface; volumetric features, such as caves, arches, cannot be modeled. Such

features could then be added by using the terrain editing and deformation facilities provided for the coarse volumetric representation. Please refer to the ‘Supplementary Material’ for the details of terrain editing and deformation functionalities.

- *LIDAR* mainly captures the height attribute for terrain data; hence, a height-field terrain can be easily produced from *LIDAR* data. This height-field data could be used as the control points for the biquadric Bzier surfaces to generate the terrain surface.

Acknowledgments

We thank ASELSAN Inc. for their support during this study. We are grateful to Dr. Trker Ylmaz for valuable comments and suggestions, Rana Nelson for proofreading and Ate Akaydn for narrating the videos.

Note: Terrain editing and walkthrough videos can be viewed from

- (1) Terrain Editing: <http://www.youtube.com/watch?v=zfg8Qzk8whE>
- (2) Terrain Walkthrough: http://www.youtube.com/watch?v=1F_yY9uy1-I

References

- Amor, M. and Bo, M., 2008. A new architecture for efficient hybrid representation of terrains. *Journal of Systems Architecture*, 54 (1–2), 145–160. doi:10.1016/j.sysarc.2007.04.005.
- Asirvatham, A. and Hoppe, H., 2005. Terrain rendering using GPU-based geometry clipmaps. In: M. Pharr and R. Fernando, eds. *GPU Gems 2*. Boston, MA: Addison-Wesley, 46–53.
- Atlas, S. and Garland, M., 2006. Interactive multiresolution editing and display of large terrains. *Computer Graphics Forum*, 25 (2), 211–223. doi:10.1111/j.1467-8659.2006.00936.x.
- Bhattacharjee, S., Patidar, S., and Narayanan, P., 2008. Real-time rendering and manipulation of large terrains. In: *Proceedings of the IEEE Sixth Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP '08)*, 16–19 December 2008 Bhubaneswar. Piscataway, NJ: IEEE, 551–559.
- Bo, M. and Amor, M., 2009. Dynamic hybrid terrain representation based on convexity limits identification. *International Journal of Geographical Information Science*, 23 (4), 417–439. doi:10.1080/13658810801932039.
- Bo, M., Amor, M., and Dllner, J., 2007. Unified hybrid terrain representation based on local convexifications. *GeoInformatica*, 11 (3), 331–357. doi:10.1007/s10707-006-0003-y.
- Brandstetter III, W.E., et al., 2011. Multi-resolution deformation in out-of-core terrain rendering. *International Journal of Computers and Their Applications (IJCA)*, 18 (4), 263–272.
- Bruneton, E. and Neyret, F., 2008. Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum*, 27 (2), 311–320. doi:10.1111/j.1467-8659.2008.01128.x.
- Bunnell, M., 2005. Adaptive tessellation of subdivision surfaces with displacement mapping. In: M. Pharr, ed. *GPU Gems 2*. Boston, MA: Addison-Wesley, 109–122.
- Cignoni, P., et al., 2003a. Planet-sized Batched Dynamic Adaptive Meshes (P-BDAM). In: *Proceedings of the 14th IEEE conference on visualization (VIS'03)*, 19–24 October 2003. Washington, DC: IEEE Computer Society, 147–154.
- Cignoni, P., et al., 2003b. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22 (3), 505–514. doi:10.1111/1467-8659.00698.
- Cohen-Or, D. and Levanoni, Y., 1996. Temporal continuity of levels of detail in Delaunay triangulated terrain. In: *Proceedings of the 7th IEEE conference on visualization (VIS '96)*, 27 October–1 November 1996. Washington, DC: IEEE Computer Society Press, 37–42.
- Crassin, C., et al., 2009. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: *Proceedings of the symposium on interactive 3D graphics and games (I3D'09)*, Boston, MA. New York: ACM, 15–22.

- Crassin, C., *et al.*, 2010. Efficient rendering of highly detailed volumetric scenes with GigaVoxels. *In: W. Engel, ed. GPU Pro*. Natick, MA: A. K. Peters, 643–676.
- de Boer, W.H., 2000. *Fast terrain rendering using geometrical Mipmapping* [online]. Available from: http://www.flipcode.com/archives/article_geomipmaps.pdf [Accessed 11 October 2013].
- de Carpentier, G.J.P. and Bidarra, R., 2009. Interactive GPU-based Procedural Height-field Brushes. *In: Proceedings of the 4th international conference on Foundations of Digital Games (FDG '09)*, Orlando, FL. New York: ACM, 55–62.
- Dick, C., Krüger, J., and Westermann, R., 2009. GPU ray-casting for scalable terrain rendering. *In: D. Ebert and J. Krüger, eds. Proceedings of Eurographics 2009 – Areas Papers*, November. Geneva, CH: Eurographics, 43–50.
- Duchaineau, M., *et al.*, 1997. ROAMing terrain: real-time optimally adapting meshes. *In: Proceedings of the 8th IEEE conference on visualization (VIS 97)*, October. Washington, DC: IEEE Computer Society Press, 81–88.
- Evans, W., Kirkpatrick, D., and Townsend, G., 2001. Right-triangulated irregular networks. *Algorithmica*, 30, 264–286. doi:10.1007/s00453-001-0006-x.
- Fan, M., Tang, M., and Dong, J., 2004. A review of real-time terrain rendering techniques. *In: Proceedings of the 8th international conference on computer supported cooperative work in design*, Vol. 1, May. Piscataway, NJ: IEEE, 685–691.
- Forstmann, S. and Ohya, J., 2005. Visualization of large ISO-surfaces based on nested clip-boxes. *In: ACM SIGGRAPH 2005 posters*, Los Angeles, CA. New York: ACM.
- Gamito, M.N., Iscte, E., and Musgrave, F.K., 2001. Procedural landscapes with overhangs. *In: Proceedings of 10th Portuguese computer graphics meeting*. Lisboa: ISCTE-IUL, 33–41.
- Garland, M. and Heckbert, P.S., 1995. *Fast polygonal approximation of terrains and height fields*. Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-95-181.
- Geiss, R., 2007. Generating complex procedural terrains using the GPU. *In: GPU Gems 3*. Boston, MA: Addison Wesley, Chap. 1, 7–37.
- Gobbetti, E., Marton, F., and Iglesias Guitián, J.A., 2008. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24 (7–9), 797–806. doi:10.1007/s00371-008-0261-9.
- Gregorski, B., *et al.*, 2002. Interactive view-dependent rendering of large isosurfaces. *In: Proceedings of the IEEE conference on visualization (VIS'02)*, Boston, MA. Washington, DC: IEEE Computer Society, 475–484.
- Gross, M.H., Gatti, R., and Staadt, O., 1995. Fast multiresolution surface meshing. *In: Proceedings of the 6th IEEE conference on visualization (VIS 95)*, 29 October–3 November. Washington, DC: IEEE Computer Society Press, 135–142, 446.
- Hnaidi, H., *et al.*, 2010. Feature based terrain generation using diffusion equation. *Computer Graphics Forum*, 29 (7), 2179–2186. doi:10.1111/j.1467-8659.2010.01806.x.
- Hoppe, H., 1997. View-dependent refinement of progressive meshes. *In: Proceedings of the ACM SIGGRAPH conference*. New York: ACM, 189–198.
- Hoppe, H., 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. *In: Proceedings of the IEEE conference on visualization (VIS'98)*, Research Triangle Park, NC. Washington, DC: IEEE Computer Society Press, 35–42.
- Hu, L., Sander, P.V., and Hoppe, H., 2009. Parallel view-dependent refinement of progressive meshes. *In: Proceedings of the symposium on interactive 3D graphics and games (I3D '09)*, Boston, MA. New York: ACM, 169–176.
- Ju, T., *et al.*, 2002. Dual contouring of hermite data. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH '02)*, 21 (3), 339–346.
- Kloetzli, J., Olano, M., and Rheingans, P., 2008. Interactive volume isosurface rendering using BT volumes. *In: Proceedings of the symposium on interactive 3D graphics and games (I3D'08)*, Redwood City, CA. New York: ACM, 45–52.
- Kumler, M.P., 1994. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica: The International Journal for Geographic Information and Geovisualization*, 31, 1–99. doi:10.3138/TM56-74K7-QH1T-8575.
- Laine, S. and Karras, T., 2011. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17 (8), 1048–1059. doi:10.1109/TVCG.2010.240.
- Lee, A., Moreton, H., and Hoppe, H., 2004. Displaced subdivision surfaces. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH'04)*, 26 (3), 85–94.

- Lengyel, E., 2010. *Voxel-based terrain for real-time virtual simulations*. Thesis (PhD). University of California, Davis.
- Levenberg, J., 2002. Fast view-dependent level-of-detail rendering using cached geometry. In: *Proceedings of the IEEE conference on visualization (VIS'02)*, November. Piscataway, NJ: IEEE, 259–265.
- Lindstrom, P., et al., 1996. Real-time, continuous level of detail rendering of height fields. In: *Proceedings of the ACM SIGGRAPH conference*. New York: ACM, 109–118.
- Livny, Y., Kogan, Z., and El-Sana, J., 2009. Seamless patches for GPU-based terrain rendering. *The Visual Computer*, 25 (3), 197–208. doi:10.1007/s00371-008-0214-3.
- Löffler, F., Müller, A., and Schumann, H., 2011. Real-time rendering of stack-based terrains. In: *Proceedings of 16th international workshop on Vision, Modeling, and Visualization (VMV)*, Berlin. Geneve, CH: Eurographics, 161–168.
- Löffler, F. and Schumann, H., 2012. Generating smooth high-quality isosurfaces for interactive modeling and visualization of complex terrains. In: *Proceedings of 17th international workshop on Vision, Modeling, and Visualization (VMV)*, Magdeburg. Geneve, CH: Eurographics, 79–86.
- Lorensen, W.E. and Cline, H.E., 1987. Marching cubes: A high resolution 3D surface construction algorithm. In: *Proceedings of the ACM SIGGRAPH conference*. New York: ACM, 163–169.
- Losasso, F. and Hoppe, H., 2004. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Transactions on Graphics (TOG) (Proceedings of ACM SIGGRAPH '04)*, 23 (3), 769–776. doi:10.1145/1015706.1015799.
- Mantler, S. and Jeschke, S., 2006. Interactive landscape visualization using GPU ray casting. In: *Proceedings of the 4th international conference on computer graphics and interactive techniques in Australasia and Southeast Asia (GRAPHITE '06)*, Kuala Lumpur. New York: ACM, 117–126.
- Marinc, A., et al., 2011. Interactive isosurfaces with quadratic C1 splines on truncated octahedral partitions. In: *Proceedings of SPIE-IS & T electronic imaging*, Vol. 7868, San Francisco, CA. Bellingham, WA: SPIE, Article No. 786808, 8p.
- McAnlis, C., 2009. Halo wars: the terrain of next gen. In: *Game developers conference*, March. San Francisco, CA: IGDA.
- McRoberts, D.A.K., 2011. *Real-time rendering of synthetic terrain*. Master's thesis. University of Johannesburg.
- Niessner, M., et al., 2012. Feature-adaptive GPU rendering of Catmull–Clark subdivision surfaces. *ACM Transactions on Graphics*, 31 (1, Article No. 6), 11p.
- Paredes, E., et al., 2012. Extended hybrid meshing algorithm for multiresolution terrain models. *International Journal of Geographical Information Science*, 26 (5), 771–793. doi:10.1080/13658816.2011.615317.
- Peucker, T.K., Fowler, R.J., and Little, J.J., 1978. The triangulated irregular network. In: *Proceedings of the ASP Digital Terrain Models (DTM) symposium*, October. Falls Church, VA: ASP.
- Peytavie, A., et al., 2009. Arches: a framework for modeling complex terrains. *Computer Graphics Forum (Proceedings of Eurographics'09)*, 28 (2), 457–467. doi:10.1111/j.1467-8659.2009.01385.x.
- Pomeranz, A.A., 2000. *ROAM Using Surface Triangle Clusters (RUSTiC)*. Master's thesis. University of California, Davis.
- Reichl, F., et al., 2012. Hybrid sample-based surface rendering. In: *Proceedings of 17th international workshop on Vision, Modeling and Visualization (VMV)*, Magdeburg. Geneve, CH: Eurographics, 47–54.
- Röttger, S., Heidrich, W., and Seidel, H.P., 1998. Real-time generation of continuous levels of detail for height fields. In: V. Skala, ed. *Proceedings of the sixth international conference in Central Europe on Computer Graphics and Visualization (WSCG '98)*. Plzen: Union Agency, 315–322.
- Steinberger, M. and Grabner, M., 2010. Wavelet-based multiresolution isosurface rendering. In: *Proceedings of the IEEE/EG international symposium on Volume Graphics (VG '10)*, Norrköping. Geneve, CH: Eurographics.
- Tanner, C.C., Migdal, C.J., and Jones, M.T., 1998. The clipmap: a virtual mipmap. In: *Proceedings of the ACM SIGGRAPH conference*. New York: ACM, 151–158.
- Treib, M., et al., 2012. Interactive editing of gigasample terrain fields. *Computer Graphics Forum*, 31 (2pt2), 383–392. doi:10.1111/j.1467-8659.2012.03017.x.

- Ulrich, T., 2002. Rendering massive terrains using chunked level of detail control. *In: Super-size It! Scaling up to Massive Virtual Worlds*, ACM SIGGRAPH '02 Course Notes, No. 35. San Antonio, TX: ACM.
- Zhou, H., *et al.*, 2007. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13 (4), 834–848. doi:[10.1109/TVCG.2007.1027](https://doi.org/10.1109/TVCG.2007.1027).