Back-to-Front Orderings of Triangles in Triangulated Terrains Over Regular Grids

J. Alonso, R. Joan-Arinyo
Grup d’Informàtica a l’Enginyeria
Universitat Politècnica de Catalunya
Av. Diagonal 647, 8º, 08028 Barcelona, Catalonia

Abstract

In this work we report on a complete and correct set of configurations to realistically rendering triangulated heightfields defined over regular grids by visiting triangles in a back-to-front order. The experiments conducted show that a simple CPU-based implementation performs as well as an up-to-date native graphics card z-buffer and allows real time interaction when the viewing position is fixed and when it moves along a 3D path.

Keywords: Digital Terrain Models, Elevation Terrain Models, Back-to-front Rendering, Visibility, Triangle Strips.

1 Introduction

Efficient visualization of Digital Terrain Models (DTM) is important in many applications such as computer graphics, resource management, earth and environmental sciences, civil and military engineering, surveying and photogrammetry, and interactive 3D games programming.

Real-time rendering of DTMs with ever increasing complexity imposes strict efficiency constraints on the visualization algorithms and interactively rendering large terrain DTMs by brute force is unpractical. Consequently, a number of algorithms have been proposed that render simplified representations of terrains, [7]. The most widely used approach relies on the multi-resolution concept. The approach increases the frame rate, ideally without any noticeable change in the terrain geometry, by reducing the number of polygons rendered according to an adaptive level-of-detail control that adjusts the terrain tessellation as a function of the view parameters, [15].
However, the level-of-detail introduces some new problems that violate the terrain coherence in both space and time. Spatial coherence is broken when a visible crack appears along an edge between two adjacent regions of different level of detail if the region of less detail cannot represent the height at a point where the region of higher detail can. Concerning temporal coherence, difficulties arise because temporally a coherent terrain does not rapidly change over time resulting in the so called *popping* effect. When the viewer is approaching a small detail from the distance, then that detail will suddenly emerge at a specific distance. While travelling around the terrain these changes of the terrain occur quite frequently. The changes might be very small, nevertheless the eye is still sensible enough to detect them. Fixing cracks requires an accurate strategy to define transition regions to guarantee visual continuity, [14]. Suppressing popping requires specific techniques like geomorphing, [18].

In this work we report on a minimal, complete and correct set of configurations which define a back-to-front ordering of triangles in a coherently triangulated DTM. The ordering includes a set of ten configurations for DTMs unformly triangulated and a set of configuration for fans of triangles seamless stitching triangles belonging to different level of detail in level of detail-based renderings. These orderings are a practical tool to build CPU-based algorithms to realistically render DTMs in real time when the viewing point is placed in a fixed point and when the viewing point describes a predefined 3D path.

As a proof of concept, we have implemented an algorithm based on our configurations used as look-up tables. The approach considers the terrain floor (XY plane) divided into six sectors defined by a framework centered at the projection of the current viewing position, plus the bisector of the first and third quadrants. Then DTM triangles are visited according to a back-to-front sequence specific to each sector associated with the projection of the DTM triangles on the floor plane. Thus no intersections need to be computed to figure out visibility. The algorithm is simple to implement and only requires graphics boards featuring basic capabilities: triangle strips, backface culling, clipping to the view frustum and the like.

The manuscript is organized as follows. Section 2 is devoted to review related previous work. In Section 3 we briefly recall general concepts that will be used later on. The set of configurations for uniform triangulations is defined in Section 4. Our implementation is described in Section 5. Results are given and discussed in Section 6. In Section 7 the basic set of configurations is generalized to deal with multi-resolution representations of terrains. We close with a brief summary in Section 8.
2 Previous work

A huge amount of literature has been devoted to the hidden line and hidden surface elimination problem. Here we are interested on those algorithms developed specifically to take advantage of the data structures underlying in grid surfaces, that is, surfaces defined as bivariate functions computed on a set of regular grid points.

The floating horizon algorithm dates back to 1968, see [13], and was improved in [6, 10, 20, 21] and in [22]. The algorithm was developed to remove hidden lines from three dimensional representations of surface functions of the form \( f(x, y, z) = 0 \). The underlying idea consists in transforming the three dimensional problem into a series of two dimensional problems by intersecting the surface with a set of parallel cutting planes at constant values of one of the variables, for instance \( z \). Then the visible surface is built up of a series of curves each on a cutting plane that are projected onto the plane \( z = 0 \) by increasing distances from the viewpoint. At each step, the set of visible segments is updated. If the corresponding \( y \) (or \( x \)) value is larger than the \( y \) value for any previous curve at that \( x \) (or \( y \)) value, then the curve is visible. Otherwise, it is hidden.

The floating column algorithm reported in [9] is a modified floating horizon that has been adapted for rendering shaded function surfaces. The surface is sampled at regular intervals by a plane perpendicular to the floor plane. As the sampling plane moves forward, the projection of each surface point is compared against the current minimum and maximum and displayed only if it is visible.

Work in [16] reports on an algorithm sensitive to both the terrain size and the size of the visible terrain parts. The algorithm first decomposes the terrain edges into monotone chains, then edges are sorted according to increasing distances from the viewpoint. Finally the algorithm detects intersections between individual terrain edges and chains.

A hidden-curve elimination algorithm to render grid surfaces is presented in [12]. The algorithm considers the surface as a set of prismatic tubes the walls edges of which are borderlines of visibility. A particular sequence for processing tubes walls is defined according to which wall is hidden and which is visible as seen from the viewer position.

In [17] a hierarchical visibility technique is applied to solve the terrain visibility problem. The terrain is stored as a hierarchy of rectangular regions at various resolutions. In a preprocessing step, a set of occlusion regions is computed for each terrain point. These occlusion regions are then combined to form occlusion regions for each rectangular region at each resolution. Since occlusion regions depend on the viewpoint, the approach does not apply in
terrain navigation where it is the user who controls the viewpoint position. An algorithm that figures out the terrain image to be rendered without transforming it into a polygonal mesh is reported in [2]. Basically the approach applies an incremental scan-conversion. The terrain image is generated by a parallel projection along the Z axis after aligning the axis of the image with the corresponding world axis what results in a restriction on the possible terrain views that can be rendered.

Back-to-front visibility ordering was first introduced in [8]. The ordering is based on the simple observation that, given a volume grid and a view plane, for each grid axis there is a traversal direction that visits geometric elements defined over the grid in order of decreasing distance to the viewing point. Authors only reported results for implementations considering orthographic projections.

The floating point-perimeter algorithm reported in [19] considers nine different regions defined in the viewing plane by four lines, $x = X_{\min}, x = X_{\max}, y = Y_{\min},$ and $y = Y_{\max}$. With each region, there is associated an enumeration that defines a previously computed sequence to process faces and edges to update the visibility status with respect to an active perimeter. The algorithm needs to process sets of edges individually to figure out crossing points with respect to the floating perimeter. The paper does not discuss whether the method supports geometric elements straddling over regions. The work in [3] gives a formal theoretical basis for the algorithm in [19].

Visibility ordering algorithms for rectilinear grids where thoroughly studied in [11]. After showing that the basic back-to-front approach as well as a number of variations fail for perspective projections, a correct perspective back-to-front visibility ordering is introduced.

A technique based on predefined configurations to painterly render terrains is reported in [4]. Configurations define a back-to-front ordering of quad cells. With each quad cell, the approach associates a brush stroke aligned along the maximum local slope which is then rendered. The set of configurations given suffers from some drawbacks. For example, configurations for some quadrants are redundant and no specific configurations are given to render quads overlapping more than one sector. In these conditions, the approach can lead to quads which are wrongly rendered and to holes in the surface thus turning the approach useless for realistically rendering terrains.

3 Preliminaries

For the sake of completeness, here we recall basic concepts on triangulated DTMAs as well as a well known short rational to justify that properly render-
Figure 1: Projections on the $XY$ plane of two different possible triangulations associated to a DTM cell.

Boiling a 3D triangulation over a regular grid can be solved by boiling it down to a 2D problem.

### 3.1 The Digital Terrain Model

A Digital Terrain Model (DTM) is a digital description and representation of a ground surface topography or terrain. Among the different ways of defining a DTM, we shall make use of a simple, regular, coherent triangulation defined as follows. Every pair of neighbor heights in a cell along a sampling axis define an edge. Each DTM cell is subdivided into two surface triangles which are used as the drawing primitives that will be sent to the graphics pipeline. There are two possible different ways of subdividing cells into two triangles. Figure 1 shows their projections onto the $XY$ plane. Vertices are labeled with grid coordinates. In the sequel, we consider the cells subdivided into triangles as shown in Figure 1a. There is nothing essential in the choice but, as we will see later on, it has an effect on the resulting set of cell configurations needed to properly render the terrain.

We organize triangles in a DTM as triangle strips as shown in Figure 2a. The vertex distinguished with the small filled circle will denote the first vertex in the triangle strip. As defined, the triangle strips are sequential because they turn alternating to the right and to the left visiting each vertex in the strip and describing sets of consistently oriented triangles. The triangle strip in Figure 2a includes a sequence of eight vertices and six triangles. Sets of triangle strips will be represented by stacks of triangle strips. The triangle strip distinguished with the small filled circle will denote the top of the stack. See Figure 2b.

### 3.2 Boiling the 3D Problem Down to a 2D Problem

Rendering triangulated terrains over a regular grid takes advantage of the fact that heightfields do not allow terrain overhangs and that a triangle
is a convex shape. In these conditions, the 3D hidden surface elimination problem can be solved as a 2D problem considering the projection onto the XY plane of the DTM surface triangulation, the viewing position and the line of sight.

Let $T_i$ and $T_j$ be two different triangles in the DTM surface triangulation. Clearly, triangles $T_i$ and $T_j$ share at most one common edge. See Figure 3. Let $O$ be the point of view and $l$ the line of sight as illustrated in Figure 3. Let $p_i$ and $p_j$ be the points where the line of sight $l$ intersects triangles $T_i$ and $T_j$, respectively.

Let $T_i'$, $T_j'$, $O'$, $l'$, $p_i'$ and $p_j'$ denote the parallel projections onto the XY plane of the corresponding geometric elements in the 3D space. Clearly $T_i'$ and $T_j'$ are convex. Since projections preserve incidence, $p_i'$ both is on $l'$ and belongs to $T_i'$ and $p_j'$ is on $l'$ and belongs to $T_j'$.

Assume that $p_i$ is closer to $O$ than $p_j$ and that the line of sight $l$ through $p_i$ and $p_j$ is not parallel to the Z axis. Then the relationship $p_i'$ is closer to $O'$ than $p_j'$ trivially holds. As considered, DTM triangulations do not allow terrain overhangs. Taken into account that DTM projected triangles are convex and do not overlap, triangles in a DTM triangulation over a regular grid can be sorted according to distances to the viewing point just by considering the projection of the 3D geometry onto the XY plane.

4 Back-to-Front Ordering

A back-to-front ordering of the triangles in the DTM is at the heart of the algorithms that explore triangulations over regular grids for fast processing to solve the visibility problem.

In order to define a complete and correct set of orderings, we split the projection of the DTM triangulation onto the XY plane into a number of regions that we call sectors. The number of sectors depends on whether the viewing point projects within the projected triangulation or not. Let us first consider the case where the viewing point projects within the projected triangulation.
Figure 3: a) DTM, point of view pictured as a camera and the projection plane. b) Projections of the DTM triangulation, the point of view, the line of sight and 3D X and Y axis onto the XY plane.

See Figure 4a. Let \( O = (x, y) \) denote the projected viewing position which is not necessarily a grid point. We define a set of local orthogonal axis, \( X \) and \( Y \), with origin at the projected viewing position \( O \) and aligned with the terrain sampling directions. Now let \( B \) be the bisector of the first and third quadrants defined by axis \( X \) and \( Y \). In this situation, the triple \( \{X, Y, B\} \) partitions the terrain into six sectors, see Figure 4a, labeled NE1, NE2, NW, SW1, SW2 and SE respectively.

Next we consider DTM tiles located within one sector. Then we will consider DTM tiles straddling over sectors.

### 4.1 Orderings Triangles in Tiles Within Sectors

With each sector we associate a unique and particular configuration that defines the path in which DTM cells must be visited to guarantee a back-to-front ordering of triangles. Consider first a set of terrain tiles placed within the NW sector with respect to the viewing position along with the projection of the viewing frustum as depicted in Figure 5a. Visiting the DTM cells following the red arrows from top to bottom and from left to right guarantees a back-to-front ordering for any line of sight \( l \) starting at the viewing position and running through the frustum. When the set of tiles to be displayed is within the SE sector all what we need to do is to follow the same path but in a bottom-up and right-left order, Figure 5b.

Now consider terrain tiles within the NE quadrant. For the DTM cell triangulation we have chosen, see Figure 1a, the relationship \textit{closer than} applied to DTM triangles within a DTM cell, as discussed in Section 3.2, depends
Figure 4: Sectors defined by a viewing position in a DTM. a) Viewing position is projected within the DTM. b) Viewing position is projected outside the DTM.

Figure 5: Back-to-front orderings for terrain tiles within sectors. a) NW sector. b) SE sector.
on the line of sight slope. First assume that the slope of the line of sight \( l \) through a DTM cell is smaller than 45° as shown in Figure 6a. Clearly triangle \( T' \) is closer to the point of view than \( T \). However, if the line of sight is \( l' \) with a slope larger than 45°, triangle \( T \) is closer to the point of view than triangle \( T' \).

A similar situation arises when considering triangles in rows or columns of a triangulated DTM. If the line of sight slope is smaller than 45° the right ordering of triangles is given by visiting first rows, Figure 6b. When the line of sight slope is larger that 45°, triangles are properly scanned visiting cells by columns, Figure 6c. A similar rationale applies to DTM tiles within the SW quadrant.

Similarly to what happens for the NW and SE sectors, configurations for sectors SW1 and SW2 are symmetric with respect to the point of view of configurations for sectors NE1 and NE2. The set of configurations to be applied to tiles in sectors within either NE or SW quadrants are listed in Figure 7.

When the viewing point projects outside the projected triangulation, some of the sectors discussed above do not appear on the projection plane. See for example Figure 4b. However, the configurations for visiting sectors described

![Figure 6: Closer than relation among triangles in DTM cells within the NE quadrant.](image)

![Figure 7: Orderings in triangles within the NE and SW quadrants. a) NE1 sector. b) NE2 sector. c) SW1 sector. d) SW2 sector.](image)
Figure 8: Sectors of a terrain tile overlapped by the field of view. a) One sector. b) Two sectors. c) Three sectors.

above apply.

4.2 Orderings Triangles in Tiles Straddling Over Sectors

In general, frustum angles are smaller than 90°. Thus the projection of the frustum onto the XY plane straddles at most over three different sectors. For a field of view on the X and Y axis of 60°, Figure 8 shows the projected frustum as a triangle in dashed lines when the viewing position projection falls within the terrain projection.

When triangles in a DTM tile straddle over two or more sectors, an approach to solve the triangles ordering would consist in two steps. First, one could compute the set of triangles in the tile within each terrain sector overlapping the field of vision. Then to each set of triangles in a sector, we could apply the associated configuration defined in Section 4.1. However taking the terrain tile as the unit to be sent to the rendering pipeline leads to a simpler approach. Thus we first classify tiles according to the configuration induced in the tile by the viewing position O, the local X and Y axis and the bisector B. Then we apply specific configurations to visit triangles depending on the sectors overlapped by the terrain tiles.

We distinguish two families of tile configurations according to whether the projected viewing position is outside the tile or inside the tile. Then, within each family we further consider different configurations depending on the geometry of the cell regions defined on the sectors by the axis X, Y and bisector B. Figure 9 shows the four possible types of regions created when a DTM tile straddles over sectors sharing either axis X or axis Y.

Triangles covering cells intersected by just the local X or Y axis, in general, have vertices that belong to different sectors. See Figure 9. Taking into account that triangles in these cells are trivially ordered and that the precision
The viewing point projection is outside the tile.

of the DTM representation is given by the triangle size, we coherently assign them to one of the two neighboring sectors and apply the configuration specific for each sector.

When the bisector $B$ intersects the terrain tile the situation is a little bit more complex. The sectors that must be considered are shown in Figure 10. The two raws at the top correspond to situations where the point of view is projected outside the tile at hand. The two bottom raws include the situations where the view of point is projected within the tile.

To illustrate the situation, consider a tile in the NE quadrant crossed by the bisector $B$ as depicted in Figure 11a and assume that triangles are labeled according to the sequence generated by back-to-front configuration associated to the NE1 sector. Assume that triangles labeled 2, 6, 12 and 20 are assigned to the NE1 sector while triangles labeled 22, 26 and 32 are assigned to the NE2 sector. Then according to what has been said in Section 4.1 and illustrated in Figure 6, pairs of triangles (22, 6), (26, 12) and, (32, 20) are incorrectly sorted. Notice that labeling triangles according to the back-to-front ordering associated with the NE2 sector just would yield the symmetric incorrect result.

A way to solve the problem would be to exclude from the ordering triangles in the cells intersected by the bisector and sort them on their own. But this would disallow the use of triangle strips. Therefore, we look for a different approach.

When the bisector $B$ intersects a tile, we group cells into rectangular subregions defined by the points where bisector $B$ and local $X$ and $Y$ axis intersect the terrain tile boundaries. Figure 11b illustrates the situation for the NE quadrant when the viewing point is projected outside the terrain tile.

Subregions labeled R1, R2 and R3 are each fully within a different sector. Therefore triangles in each of these subregions are ordered applying respectively the already defined back-to-front orderings the one associated with SE sector for subregion R1 and the ordering of NE1 for subregions R2 and R3.

Subregion R4 is always squared and we subdivide it into two subsectors. DTM cells below the bisector $B$ define the NE12 subsector and DTM cells
Figure 10: Sectors involved in the ordering when tiles straddle over bisector $B$. Top two rows: the point of view is projected outside the tile. Bottom two rows: the point of view is projected within the tile.

Figure 11: Tiles intersected by bisector $B$ in the first quadrant. a) Incorrectly sorted triangles in cells intersected by the bisector. b) Regions defined by the $X$ axis and the bisector $B$. 

12
above the bisector define the NE22 subsector. When the bisector $B$ intersects the tile boundary on the left and top edges as in Figure 11b, the back-to-front ordering starts by labeling triangles in the column of cells with the largest column grid index. Next triangles in the terrain row with the largest row grid index and not yet ordered are considered. Successive strip triangles are ordered by alternatively labeling columns and rows of terrain cells. The back-to-front ordering for NE12 is given in Figure 12a where triangles in the strip labeled $k + 1$ are closer to the point of view than those in the strip labeled $k$.

When the bisector $B$ intersects tile boundary edges at the bottom edge and the edge on the right side, columns and rows interchange their roles with respect to those played in the ordering for NE12. See ordering NE22 in Figure 12b.

A similar rationale allows to define the orderings for subsectors SW12 and SW22 originated by the bisector $B$ in the quadrant SW. See Figures 12b and c. Notice that these ordering are symmetric with respect to the point of view of those given for NE12 and NE22.

Therefore, our approach includes a total of ten different back-to-front orderings of triangles in a triangulated DTM. Six orderings correspond to tiles that fall within a single terrain sector. Four orderings are associated with tiles straddling over more than one terrain sector.
5 Implementation

As a proof of concept, we have implemented a CPU-based algorithm to realistically render triangulated DTM s by sending to the graphics card triangle strips according to a back-to-front ordering strategy using the configurations described in Section 4.

Pseudo-code for the algorithm is listed in Algorithm 1. We assume that the algorithm is fed with the set of terrain tiles to be rendered which have been properly selected in the DTM model, the point of view \( O \), the line of sight \( L \) and the octree depth \( D \). The output is the rendered terrain.

Figure 13 illustrates conceptually how our algorithm works. Figure 13a shows an orthographic projection onto the \( XY \) plane of the DTM terrain tiles, the \( XY \) axis and the bisector \( B \). Tiles crossed by the axis and bisector are depicted in brown. Tiles within each sector have a different color. The point of view projection falls within the tile common to the axis and bisector projections. Selecting one tile, as depicted in Figure 13b, results in the set of sectors and triangles shown in Figure 13c. Figure 13d shows the triangles rendered by our algorithm. Colors of triangles rendered are coincident with those of the corresponding sectors induced in the \( XY \) plane.

The algorithm has two main parts: data initialization and rendering. In the data initialization part, the algorithm first organizes the projection onto the \( XY \) plane of the set of terrain tiles to be rendered in a uniform depth quadtree. Terrain tiles included in each quadtree node are recursively labeled following a standard back-to-front quadtree ordering as illustrated in Figure 14b.

Then the terrain tile corresponding to each quadtree leaf is loaded into the GPU as a different vertex buffer object. Next, the six visiting rules for tiles falling within one terrain sector are built for the specific quadtree depth as a sequence of integer indices which identify the path for traversing the set of triangles in the tile. Finally each configuration is loaded into the GPU as an index array.

The rendering part starts by culling tiles placed outside the viewing frustum. Then for each remaining tile the specific ordering rule is identified according to the terrain sector it belongs to. If the tile is included within a unique terrain sector, it is rendered. When the tile straddles over more than one terrain sector the situation is a little bit more complex. Notice that in this case the tile subdivision depends on the specific configuration determined by the axis and bisector on the tile boundary. See Figure 9. Therefore the corresponding ordering configuration for each straddling type tile must be loaded into the GPU. Finally the tile is rendered.
Figure 13: a) Projection parallel to the Z axis of the DTM onto the XY plane. b) Selecting a tile. c) Triangles within the selected tile and overlapped regions. d) DTM triangulation rendered as viewed from the given point of view.

Figure 14: Quadtree. a) Quadrants labeling. b) A depth two quadtree.
Algorithm 1 DTM-painter

Input: DTM, a block of terrain tiles
      O, viewing point
      L, line of sight
      D, the octree depth

Output: A render of the DTM

Q = quadtree(DTM, D)
for each leaf node (terrain tile) T_i in Q do
    VBO = buildVBO(T_i)
    loadVBOinGPU(i, VBO)
end for
for each sector S in \{NE1, NE2, NW, SW1, SW2, SE\} do
    I = buildVisitingRuleIndex(D, S)
    loadIndexInGPU(i, I)
end for
CT = cullTiles(Q, O, L)
for each tile T_i in CT do
    R = identifyVisitingRule(T_i, O, L)
    if R in \{NE1, NE2, NW, SW1, SW2, SE\} then
        renderVBOinGPU(i, R)
    else
        if R not loadedInGPU() then
            I = buildExtraVisitingRuleIndex(T_i, O, L, R)
            loadIndexInGPU(i, I)
        end if
        renderVBOinGPU(i, R)
    end if
end for
6 Results and Discussion

To assess the performance of the algorithm implemented and therefore of the back-to-front ordering technique, we have implemented two extra algorithms. One extra algorithm was a DTM rendering algorithm using the standard z-buffer provided by the graphics card. Since the path for visiting triangles in a GPU vertex buffer object does not matter, we filled the index array with the ordering associated to the NE sector. The other extra algorithm just renders vertex buffer objects in the GPU following our approach but using always an *a priori* fixed ordering, say the NE ordering. Clearly, this algorithm does not solve the hidden-surface problem but yields the highest rendering frame ratio the available graphics card can attain and is used as a reference.

The experiments have been conducted on a laptop Pentium Intel Core i7 at 2.20 GHz, with 8GB RAM, featuring an AMD Radeon HD6750M graphics board with 1GB running Visual Studio 2010 under Windows 7. The graphics API used was OpenGL and the GLUT library was used for events and window management.

The benchmark consisted in the terrain shown in Figure 15 represented as a digital elevation model of height fields sampled on a regular grid parallel to the X and Y axis.

We considered two different series of experiments. In one series, the point of view was static, in the other series the point of view moved along a predefined 3D path. For each case, we tested three different terrain resolutions with respectively $512 \times 512$, $1024 \times 1024$ and $2048 \times 2048$ uniformly distributed grid points. For the lowest and middle resolutions, eight different quadtree subdivision depths were considered. Due to the limited available storage...
Figures 16, and 20 plot the number of frames per second rendered for each grid resolution and quadtree depth by respectively the wrong algorithm, (■), the graphics card z-buffer algorithm, (○), and our approach, (●). Tables included in Appendix A list measured values.

6.1 Fixed Point of View

For the static point of view and 512 × 512 and 1024 × 1024 terrain precisions, plots of frame ratios follow the same pattern. For small quadtree depths, curves show a plateau where the number of frames per second rendered is almost constant. Then the frame ratio drops off sharply.

As expected, the wrong algorithm always performed better than both the graphics card z-buffer and our algorithm. In general, our algorithm performs as well as the graphics card z-buffer. However, for the 2048 × 2048 height points grid and zero and one quadtree depths, our algorithm performs worst than z-buffer. Searching for a rational for this behavior, we observed that small quadtree depths result in large tiles that, when crossed by the bisector, must be rendered according to visiting rules which require breaking the set of height points indices into a number of disjoint sequences. See Figure 12. But as the tile size decreases, the number of tiles which straddle over sectors increases and, consequently, the number of triangles rendered according to a simple basic configuration which can be described as large sequences of sorted triangles decreases. See Figure 17.

To check whether these observations have an effect on the algorithm performance, we conducted an additional experiment. We fed the wrong algorithm with the terrain as a zero depth quadtree. Consequently, only one vertex buffer object was loaded in the graphics card. Culling was avoided by
Figure 17: The number of tiles straddling over the bisector depends on the quadtree depth. (a) Quadtree depth 2. (b) Quadtree depth 4. (c) Quadtree depth 8.

placing the point of view far enough.

We simulated different lengths of sorted index sequences considering five levels of randomness, 0%, 25%, 50%, 75% and 100%. 0% means that contiguous cells in the index array stored consecutive indices and 100% means that all the indices were loaded in the index array cells at random.

The test was applied to a terrain considering three different grids. Figure 18a plots the frame ratio yielded by the wrong algorithm versus the randomness level for a 512 \times 512 grid (●), 1024 \times 1024 grid (■), and 2048 \times 2048 grid (▲). Specific measured values are given in Appendix B. Clearly randomness in indices stored in the index array has an effect on the graphics card performance: The performance worsen as the indices randomness increases. Plots in Figure 18b show for each terrain grid the ratios normalized with respect to the 0% randomness level. We can observe that the drop off of the graphics card performance is up to a 50% when the randomness level in the index array is 100%. As the quadtree depth increases, tiles crossed by the bisector and the corresponding graphics card index arrays became smaller and the influence of the indices randomness vanishes.

To explain the frame ratio sharp drop off when the quadtree depth is in the range 5 to 7, we have conducted another experiment. We used the terrain including a grid with 512 \times 512 height values. Then we fed our wrong algorithm with the data organized according to an increasing number of tiles. Each terrain tile was stored in the graphics card as a vertex buffer object. Since we do not care about the correctness of the rendered terrain, we loaded into the graphics card a unique index array used to visit all the vertex buffers. The frame ratios output by the algorithm versus the number of data buffers in the graphics card are shown in Figure 19. Clearly, the sharp drop off appears when the number of buffers was 1024, which corresponds to the number of leaf nodes in a quadtree depth of 5, that is, $2^{5 \times 5}$. Notice that this agrees with the results shown in Figure 16a. Therefore, the graphics card performance is the limiting factor in the frame ratio rendered by the
6.2 Moving Point of View

When the viewing point moved along a 3D path, the number of frames per second rendered for the terrain precisions considered show patterns consistent with those yielded for the static point of view. The drop off also appears for quadtree depths of about six and the rational given for the static point of view also applies.

For small quadtree depths, our approach always performs worse that the graphics card z-buffer. The rational given for the static point of view case also applies here. Besides, when the point of view moves the sectors where terrain triangles project change and look-up ordering tables must be re-sent to the graphics card resulting in a lose of vertex buffer objects efficiency in data transfer. However, performance of our approach steadily increases with

Figure 18: Frame ratio versus randomness in the graphics card index array. a) Frame ratio. b) Frame ratio variation normalized with respect to frame ratio for 0% randomness.

Figure 19: Frame ratio versus number of vertex buffer objects stored in the graphics card.
the quadtree depth. When the quadtree depth reaches the value four, the frame ratio reaches the plateau where values are equal or higher than those yielded by the graphics card z-buffer.

7 DTM with Level of Detail

Interactive rendering of large DTMs with high resolution is a challenging problem. Consequently, a number of algorithms have been developed that render simplified representations of terrains, [7].

Multi-resolution terrain models provide efficient mechanisms to represent and manipulate DTM by optimizing the tradeoff between complexity and accuracy of representation. In level of detail multi-resolution DTM simplification schemes, terrain regions close to the viewing position are approximated more accurately than regions that are far away. The resulting image does not show any noticeable visual difference with respect to the one rendered with the full terrain information.

We adopt the multi-resolution level of detail approach reported in [5] also applied, for example, in [1]. The level of detail of two neighboring terrain tiles can differ at most in one level. Geometry gaps at common edges of tiles with different level of detail are avoided by changing the connectivity of heightpoints in the higher detail tile and building a stitching fan of triangles. Figure 21 shows the four possible cases that arise in neighboring tiles of 5 x 5 heightpoints. Note that stitching fans have been defined to agree with the topology of the underlying triangulation. Heightpoints marked with a small filled circle in Figure 21 are still part of the triangulation but they are not rendered.

To deal with different levels of detail in our algorithm, we need to define the ordering to visit the triangles in each stitching fan. Since there are four possible tile neighborhoods and each of them can be found within each
terrain sector, we have to consider 24 cases. Figure 22 shows the ordering to visit triangles in the stitching fan according to decreasing distances to the viewing position along the line of sight when the fan is located within viewing sector NE2. Labels in triangles define the visiting ordering. Similar orderings have been defined for stitching fans located in NW, SW1 and SW2 terrain sectors.

When stitching fans are located in sectors NE1 and SW1, there are two neighborhoods that require two different orderings depending on how the line of sight intersects the stitching fan. Figure 23 depicts the orderings for these cases. The slope of the line of sight \( l \) for the NE1 sector is in the range \([0, 45^\circ]\) while for the SE1 sector the slope is in \([180^\circ, 225^\circ]\). The row on the top includes the orderings for the NE1 terrain sector while the bottom shows the orderings for the SW1 terrain sector.

Figure 22: Visiting sequence for triangles in a stitching fan located within viewing sector NE2.
Figure 23: Visiting sequences depends on how the line of sight intersect the stitching fan. Top) NE1 terrain sector. Bottom) SW1 terrain sector.

8 Summary

Fast rendering of digital terrain models is a challenging problem in a number of applications like animation and video games programming. A way to fast rendering triangulated terrains is based on rendering triangles by applying either back-to-front or front-to-back orderings according to predefined configurations.

In this paper we provide a complete and correct set of configurations for realistically rendering triangulated terrains. The set includes six configurations for terrain tiles the projection of which fall within one of the six sectors defined on the projection plane by the point of view and the local set of axis plus four additional rules for terrain tiles straddling over terrain sectors. Besides, we provide configurations for fans of triangles that seamless stitch different levels of triangulations when a level of detail approach is applied to rendering the terrain.

We tested our algorithm on a CPU-based implementation. A comparison of our approach performance with the performance of a graphics card native z-buffer showed that when the point of view was static, our approach performs as well as the graphics card native z-buffer. When the point of view moved along a 3D path our approach showed a strong dependence on the quadtree depth used to organize the set of tiles to be rendered. Frame rate plots show a plateau for quadtree depths ranging from 3 to 6 where frame rates yielded by our implementation are equal or higher than those yielded by the z-buffer. In any case, experimental results prove that our approach is easy to implement, is robust and supports real time interaction.

The need of reloading into the graphics card the ordering look-up tables when the point of view describes a path entails a significative overhead data
transfer in our approach. In this context, allowing graphics cards to automatically generate index patterns for ordering look-up tables would be a great improvement over the currently available fixed index arrays. Assuming that the unit sent to the graphics card to be rendered is a squared tile of terrain, all what would be needed to define an index pattern for each ordering in our approach is: The number of rows and columns in the terrain tile; an ordering look-up table code to identify the sector where the terrain projects; an offset for the first vertex in the terrain to be rendered; and the number of rows and columns spanned by the set of triangles within the tile grid to be rendered.

In what follows we use an OpenGL like notation. Assume that the terrain size to be loaded into the graphics card is \texttt{dataSize}, which is conceptually organized as an array with \texttt{dataSize\_rows} rows and \texttt{dataSize\_cols} columns. Assume that the flag \texttt{GL\_PATTERN\_ELEMENT\_ARRAY\_BUFFER} denotes the binding for an automatic patterned index array generation in the graphics card. An example of data-index array binding could be

\begin{verbatim}
// generate a new VBO and get the associated ID
glGenBuffers(1, &vbo_id)
// bind VBO in order to use it
glBindBuffer(GL\_ARRAY\_BUFFER, vbo_id);
// upload data to VBO
glBufferData(GL\_ARRAY\_BUFFER, dataSize, vertices, GL\_STATIC\_DRAW);
// generate patterned vertex index array
glBufferData(GL\_PATTERN\_ELEMENT\_ARRAY\_BUFFER, pattern,
   x\_offset, y\_offset, n\_rows, n\_columns,
   dataSize\_rows, dataSize\_cols, GL\_STATIC\_DRAW);
\end{verbatim}

Here \texttt{pattern} would take values in the set \texttt{GL\_NE1, GL\_NE2, GL\_NW, GL\_SW1, GL\_SW2, GL\_SE, GL\_NE, GL\_NW1, GL\_NW2, GL\_SW, GL\_SE1, GL\_SE2}. The index array patterns described here are just an example and one can think of patterns useful in other applications. Graphics cards featuring such a tool would allow to avoid the overhead incurred by the need of transferring indices from the CPU.

\section*{References}


A Frame Rate Measured Values

Tables 1, 2 and 3 show the number of frames per second for each grid resolution and quadtree depth rendered by the wrong algorithm, the native graphics card z-buffer and by our correct implementation. In all cases the table top lists values for the static point of view and the table bottom lists values when the point of view moved along a 3D path.
Table 1: Frames per second for nine different levels of terrain subdivision. Terrain with 512×512 points.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Wrong</td>
<td>763</td>
<td>762</td>
<td>760</td>
<td>758</td>
<td>748</td>
<td>686</td>
<td>214</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Z-buffer</td>
<td>722</td>
<td>723</td>
<td>721</td>
<td>718</td>
<td>709</td>
<td>608</td>
<td>207</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Correct</td>
<td>744</td>
<td>744</td>
<td>744</td>
<td>739</td>
<td>728</td>
<td>610</td>
<td>177</td>
<td>51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moving</td>
<td>Wrong</td>
<td>805</td>
<td>796</td>
<td>777</td>
<td>762</td>
<td>758</td>
<td>680</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Z-buffer</td>
<td>738</td>
<td>720</td>
<td>716</td>
<td>712</td>
<td>714</td>
<td>642</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>Correct</td>
<td>517</td>
<td>742</td>
<td>753</td>
<td>749</td>
<td>750</td>
<td>636</td>
<td>179</td>
</tr>
</tbody>
</table>

Table 3: Frames per second rendered for eight different levels of terrain subdivision. Terrain with 2048×2048 points.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Wrong</td>
<td>63</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td>63</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Z-buffer</td>
<td>62</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>62</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>Correct</td>
<td>55</td>
<td>60</td>
<td>65</td>
<td>65</td>
<td>63</td>
<td>62</td>
<td>62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moving</td>
<td>Wrong</td>
<td>63</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td>63</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Z-buffer</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>63</td>
<td>62</td>
<td>61</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Correct</td>
<td>41</td>
<td>52</td>
<td>58</td>
<td>59</td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
</tbody>
</table>

B Index Array Randomness Measured Values

Table 4 lists the frame ratio rendered by the wrong algorithm considered in this work versus the randomness of the index array loaded in the graphics card. We considered randomness values from 0% (index arrays storing increasing indices in contiguous cells) up to 100% (index arrays storing indices at random in contiguous cells). Measures where taken using one vertex buffer object with the z-buffer enabled and the point of view placed far enough to avoid culling.
Table 2: Frames per second rendered for nine different levels of terrain subdivision. Terrain with 1024x1024 points.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wrong</td>
<td>249</td>
<td>250</td>
<td>250</td>
<td>242</td>
<td>238</td>
<td>239</td>
<td>207</td>
<td>49</td>
<td>16</td>
</tr>
<tr>
<td>Z-buffer</td>
<td>237</td>
<td>239</td>
<td>239</td>
<td>236</td>
<td>233</td>
<td>234</td>
<td>198</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>Correct</td>
<td>245</td>
<td>248</td>
<td>248</td>
<td>241</td>
<td>238</td>
<td>239</td>
<td>165</td>
<td>43</td>
<td>13</td>
</tr>
<tr>
<td>Moving</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wrong</td>
<td>244</td>
<td>245</td>
<td>245</td>
<td>238</td>
<td>234</td>
<td>233</td>
<td>207</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>Z-buffer</td>
<td>233</td>
<td>235</td>
<td>235</td>
<td>232</td>
<td>229</td>
<td>228</td>
<td>205</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>Correct</td>
<td>209</td>
<td>222</td>
<td>229</td>
<td>233</td>
<td>232</td>
<td>232</td>
<td>169</td>
<td>43</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4: Frames per second rendered by the graphics card versus randomness in the index array.

<table>
<thead>
<tr>
<th></th>
<th>0%</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>755</td>
<td>640</td>
<td>555</td>
<td>495</td>
<td>445</td>
</tr>
<tr>
<td>1024x1024</td>
<td>245</td>
<td>199</td>
<td>165</td>
<td>142</td>
<td>121</td>
</tr>
<tr>
<td>2048x2048</td>
<td>62</td>
<td>51</td>
<td>41</td>
<td>35</td>
<td>31</td>
</tr>
</tbody>
</table>