Stereoscopic View-Dependent Visualization of Terrain Height Fields

Ugur Güdükbay, *Member*, *IEEE Computer Society*, and Türker Yilmaz

Abstract—Visualization of large geometric environments has always been an important problem of computer graphics. In this paper, we present a framework for the stereoscopic view-dependent visualization of large scale terrain models. We use a quadtree based multiresolution representation for the terrain data. This structure is queried to obtain the view-dependent approximations of the terrain model at different levels of detail. In order not to lose depth information, which is crucial for the stereoscopic visualization, we make use of a different simplification criterion, namely, distance-based angular error threshold. We also present an algorithm for the construction of stereo pairs in order to speed up the view-dependent stereoscopic visualization. The approach we use is the simultaneous generation of the triangles for two stereo images using a single draw-list so that the view frustum culling and vertex activation is done only once for each frame. The cracking problem is solved using the dependency information stored for each vertex. We eliminate the popping artifacts that can occur while switching between different resolutions of the data using morphing. We implemented the proposed algorithms on personal computers and graphics workstations. Performance experiments show that the second eye image can be produced approximately 45 percent faster than drawing the two images separately and a smooth stereoscopic visualization can be achieved at interactive frame rates using continuous multiresolution representation of height fields.

Index Terms—Stereoscopic visualization, terrain height fields, multiresolution rendering, quadtrees.

1 Introduction

In general, geometry processing is the main bottleneck of all graphics applications. Even high-end graphics workstations have the ability to draw only a very small fraction of the triangles needed to draw large complex scenes at interactive frame rates. Furthermore, virtual reality applications need twice the processing power as needed for their monoscopic correspondents. Therefore, the surface has to be approximated up to a certain threshold.

The most common way to approximate a surface is to use algorithms based on screen-space error threshold that provide suitable heuristics for the approximation. However, one of the most important disadvantages of using screen-space error threshold as a simplification criterion is the loss of depth information, which is crucial in stereo visualizations. To solve this problem, we propose a distance-based angular error threshold criterion that preserves depth information of the terrain data during the simplification process.

In stereoscopic visualization, the two views must be generated fast enough to achieve interactive frame rates. Our goal in this work is to decrease the time needed for generating the second eye image so that complex stereoscopic visualizations can be possible. For this purpose, an algorithm is proposed to speed up the generation of stereo pairs for stereoscopic view-dependent visualizations. The algorithm, called Simultaneous Generation of Triangles (SGT), generates the triangles for the left and right eye images simultaneously, using a single draw-list, thereby avoiding the need for performing the view frustum culling and the vertex activation operations twice.

The contributions of the paper can be summarized as follows:

- A traversal algorithm on the quadtree representation of the terrain data that is preventing the formation of cracks using dependency information between the vertices.
- A distance-based angular error metric for view-dependent refinement of the terrain data that preserves the depth information of the terrain data during simplification process, which is necessary for correct stereoscopic view.
- An algorithm to speed-up the generation of the stereo pairs for stereoscopic view-dependent visualizations, namely, Simultaneous Generation of Triangles.
- Several strategies to optimize the view frustum culling process.
- A morphing technique that works in the same manner for both refining and coarsening phases while visualizing the terrain data.

The rest of this paper is organized as follows: In Section 2, we describe related work on both multiresolution modeling of terrain data and stereoscopic visualization. Our quadtree-based multiresolution modeling approach and distance based angular error threshold as the approximation criterion are explained in Section 3. The algorithm that is proposed to speed up the generation of the second eye image for stereoscopic visualization is explained in Section 4. Section 5 discusses the performance of the proposed algorithms in terms of processing speed and quality of the visualizations. Conclusions are given in

• *The authors are with Bilkent University, Department of Computer Engineering, 06533 Bilkent, Ankara, Turkey.*
E-mail: {gudukbay, yturker}@cs.bilkent.edu.tr.

Manuscript received 23 Jan. 2001; revised 16 Aug. 2001; accepted 19 Oct. 2001.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 113523.

Section 6. In the Appendix, we present the algorithms in C-like pseudocode.

2 Related Work

2.1 View-Dependent Visualization of Terrain Height Fields

In [1], a dynamic approach is presented for level of detail (LOD) construction of terrain data. In this work, grid elevation data is used to represent height fields and to visualize terrain at real time. The simplification hierarchy is represented using a quadtree structure. During the simplification process, block-based tests are done first to select discrete levels of detail for blocks of the quadtree. After this coarse level of simplification, a fine-grained simplification is performed in which individual vertices are considered for removal. To check a vertex for removal, a screen-space error metric is used.

In [2], a framework for monoscopic visualization of regular grid elevation data is proposed. In order to prevent cracks on the terrain, a dependency relation is generated. Every vertex is dependent on the two other vertices of the same or the next higher level in the quadtree hierarchy. A breadth-first search is performed in the quadtree for progressive mesh construction. Blending is used to prevent popping effects. A windowing mechanism is used for large terrains by applying spatial database access in order not to load the whole terrain data into the memory. The Euclidean distance between the vertices is used as a simplification criterion.

In [3], regular grid data is first approximated with minimum error and the triangulation is converted into a triangulated irregular network (TIN) model. Later, the blocks are simplified step-by-step for each LOD and simplification steps are recorded to construct hierarchical representation of the terrain. While switching between different resolutions, morphing is used to eliminate popping artifacts. The pixel threshold that is used to control the simplification process is adjusted according to the frame rate defined.

Grid elevations and quad cells are also used in [4]. The lowest acceptable rendering speed is chosen and the appropriate LOD for that rendering speed is selected. Elevation differences are taken into account for simplification and a distance-based polygon resolution technique is used for simplification. In order to hide the appearance of cracks, each crack is closed by an additional triangle.

Other techniques, which decrease the number of polygons to be processed, hence optimizing CPU usage, include view frustum culling, back face removal, and occlusion culling. In [5], some methods are proposed to speed up view frustum culling by using bounding boxes. They use movement coherency during visualization based on the properties of axis aligned and oriented bounding boxes.

Some other work use special capabilities of the underlying graphics system. In [6], selection buffer mechanism of OpenGL is used for view frustum culling. This mechanism is very effective in determining which quad blocks are in the view frustum and eliminates the need to make intersection tests. However, the bounding boxes must be drawn to the selection buffer as filled polygons and backface culling should not be performed. Otherwise, it is possible that the viewer is completely inside of a box and the selection buffer may not create a hit although the block is in the viewing frustum. Besides, culling tests bring additional overhead if it is needed to distinguish between the blocks that are completely inside and the blocks intersecting with the view frustum since a hit produced cannot differentiate between these cases. For occlusion culling, they use OpenGL's stencil buffer mechanism.

2.2 Stereoscopic Rendering and Visualization

Stereoscopic visualization systems are used in many applications, such as simulators and scientific visualization. These systems can be used with suitable hardware designed for this purpose. One of the most commonly used hardware is the time multiplexed display system that is supported by liquid crystal shutter (LCS) glasses and virtual reality (VR) glasses. In this work, we chose to use LCS glasses since they are less expensive and many users can simultaneously see the results of a visualization application in stereo. Detailed information about these systems can be found in [7] and [8].

For stereoscopic viewing, the application must support a kind of display technique to make each eye see the image generated for it. In visualization with LCS glasses, when the left eye view is drawn onto the screen, the right eye of the glasses dims to occlude the left eye image from the right eye. The same procedure is applied when the right eye image is drawn onto the screen. Average refresh rate of a real-time visualization application should be around 25 frames per second (fps) for monoscopic view. However, since two images should be generated for each frame in stereoscopic visualization, the application should be able to generate 50 or more images per second to achieve the same frame rate as the monoscopic correspondent. This means that, when you convert a monoscopic application to stereo without any improvement, the frame rate decreases by half.

The algorithms developed for speeding up stereo rendering generally make use of the mathematical characterization of an image that changes when the eye-point shifts horizontally and a recognition of the characteristics that are invariant with respect to the eyepoint, like the scanlines to which an object project, as stated in [7]. In [9], the authors present a visible surface ray-tracing algorithm that infers a right-eye view from a fully ray-traced left-eye view and this algorithm is further improved in [10]. In [11], a non-ray-tracing algorithm is described to speed up the second eye image generation for polygon filling, hidden surface elimination, and clipping. In [12], methods that take advantage of the coherence between the two halves of a stereo pair for ray traced volume rendering are presented. In [13], the authors present an algorithm using segment composition and linearly interpolated reprojection for fast stereo volume rendering. Hubbold et al. [14] propose extensions of a direct volume renderer for use with an autostereoscopic display in radiotherapy planning. Since the terrain data does not have any mathematical characterization, mentioned algorithms cannot be adapted easily to stereoscopic terrain visualization.

```
struct elevation
{
   short int  elevation;        // elevation in meters
   int        activation;       // vertex activation distance
   int        dependent[4][3];  // array of dependents: [][0]=col,[][1]=row,
                                // [][3]=belonging quad block number
   float      morphdistance;    // distance for each vertex to be morphed
   char       morph;            // level of morphing
   unsigned   activestate:1;    // keeps activation locks on vertex
   unsigned   morphlock:1;      // flag to prevent another quad
                                // decrement the morph value
};
struct elevation Terrain[ROW][COL]; // terrain data

struct quad
{
   short int  elevmin, elevmax; // minimum and maximum elevations
   int        minact, maxact;   // minimum and maximum vertex enabling distance
   int        rcenter, ccenter; // indices of the center vertex
   char       activated;        // the cell is activated or not
   char       culled;           // the cell is previously culled or not
   char       childactivated[4];
};
struct quad   QuadTree[NODECOUNT];  // quadtree array

struct tag
{
   char     center;
   unsigned leftborder     :1;
   unsigned bottomleft     :1;
   unsigned bottomborder   :1;
   unsigned bottomright    :1;
   unsigned rightborder    :1;
   unsigned upperright     :1;
   unsigned upborder       :1;
   unsigned upperleft      :1;
};
struct tag EyeBlock[NODECOUNT];
```

Fig. 1. Data structures.

3 Multiresolution Modeling

3.1 Data Structures

Here, we present the data structures used in our implementation. In order to visualize complex scenes, such as terrain height fields, at interactive frame rates, efficient data structures need to be used. Quadtree representation perfectly fits into grid elevation data. To allow morphing and crack prevention, the `elevation` structure has to be equipped with suitable fields. The `elevation` data structure stores elevation data, the distance at which the vertex will be activated, the state of the vertex (active or inactive), indices of its dependent vertices, morph field indicating at which morphing stage the vertex is, a precalculated value showing the distance between active and inactive states of the vertex, and a morph lock flag to prevent the vertex from being morphed again by other neighboring blocks at the same frame (see Fig. 1).

The quadtree structure was constructed as in [15]. In the `quad` structure, minimum and maximum elevations and minimum and maximum activation distances for a quad block are stored. Flags indicating whether or not the quad block is activated, previously culled, and its children are activated are also stored in this structure.

For a terrain with n^2 vertices, the `Terrain` structure holds $60n^2$ bytes. The `QuadTree` array occupies 26 bytes per node. The quadtree contains $N = ((n-1)/2)^2$ nodes at the most detailed level. Given $L = log_4 N$ levels and $T_N = \sum_{level=1}^{L} 4^{level-1}$ nodes, the quadtree structure occupies $26T_N$ bytes.

The `tag` data structure stores flags to indicate the activated vertices for the quad blocks and uses up two bytes for each node. This information is used while drawing the second eye image.

3.2 Approximation Criterion

As mentioned previously, the screen-space error criterion for approximating the terrain is not sufficient in order to achieve a correct stereoscopic view. In the screen-space error metric, elevation differences are taken into account to evaluate a vertex for removal. The visible pixel difference on the projection plane when the vertex is active and inactive is calculated for this purpose. If this number is greater than the prespecified pixel tolerance, then the vertex

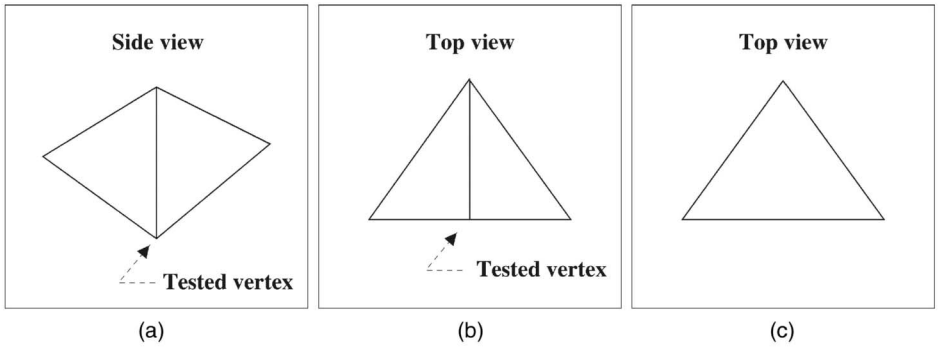

Fig. 2. Screen-space error metric. (a) Side view of a quad block. (b) Top view of the same block. (c) Edge removal by screen-space error-based algorithm when the block is viewed from the top.

is kept; otherwise, it is removed. This is illustrated in Fig. 2. The problem here is that if the viewer is looking at the terrain from above, then the projection of the vertices to the camera plane when they are active and inactive will be very small, yielding to the elimination of the candidate vertex. This problem can be illustrated by an example. Assume that we are looking at a tower from above and we use screen-space error tolerance. Since the projection of the elevation difference will be very small with respect to the position of the eye, the tested vertices will be removed, although they are important to the viewer (i.e., they will make the viewer see the height of the tower when viewed in stereo). Therefore, although the screen-space error metric is suitable for the monoscopic view [1], it degrades the stereo effect and may result in incorrect stereoscopic vision.

The elevation and distance of objects from the viewer are two important criteria that make us feel the depth and differentiate between objects. Therefore, the threshold value must be specified adaptively so that it takes into account both of these parameters to reflect the correct depth information. For this purpose, we specify our distance-based angular error threshold for simplification as follows: First, we define an *angular error threshold* that will be used to simplify the terrain. This value will be used to calculate *elevation thresholds* at each vertex location, which is adaptive to the distance of the viewer from the vertex. In order to do that, we accept the eye to be in the center of a sphere. The candidate vertices tested for the elimination are supposed to be located on the surface of the sphere. The elevation threshold value at a vertex location is computed by using the prespecified angular threshold value and the radius of the sphere (i.e., the distance from eye to the vertex). The greater the radius of the sphere is, the larger the size of the elevation threshold will be. We can derive the elevation threshold by calculating the tangent of the angular threshold at the given distance. Fig. 3 illustrates our angular error metric for the evaluation of a vertex for removal.

The distance from the eye position to the vertex is:

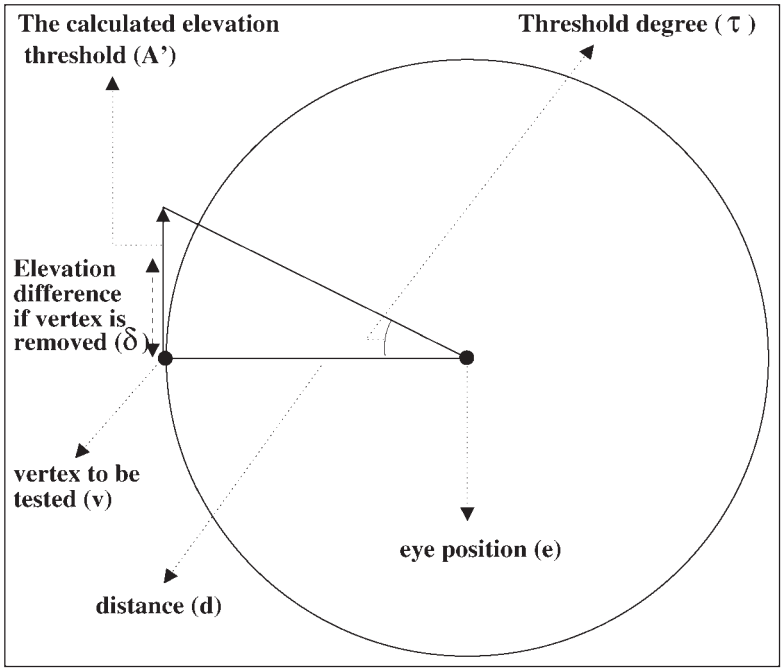

Fig. 3. Angular error threshold representation for vertex removal.

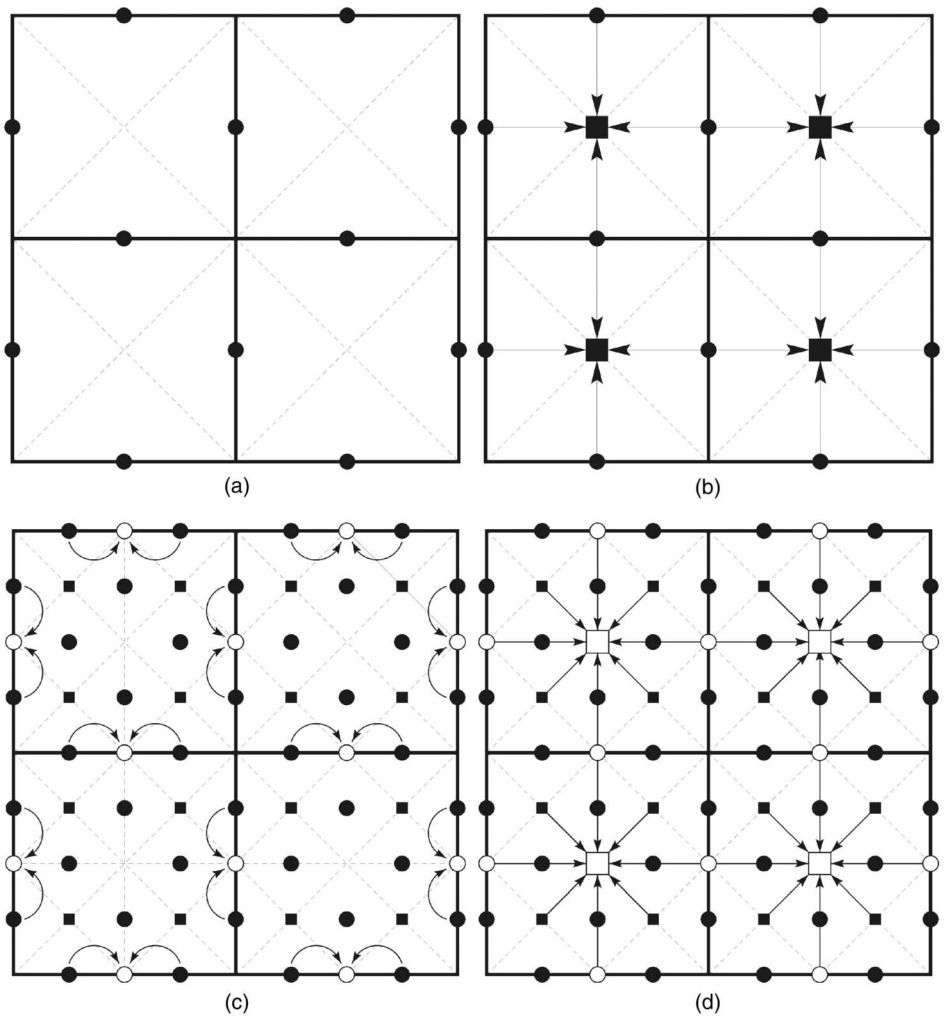

Fig. 4. Activation distance assignment. (a) Calculation of the activation values for each border vertex of the quad blocks. (b) Finding the activation distances for the center vertices by taking into account the diagonals based on the position of the quad block and assignment of the maximum of its four border activation distances and the calculated value as the center vertex activation distance. (c) Calculation of the activation distances for border vertices for each edge and assignment of the maximum of the border vertex activation distances on the same edge and the calculated value as the border vertex activation distance. (d) Finding the activation distances for center vertices by taking into account the two corner vertices (based on its position in the larger quad block) and assignment of the maximum of nine values to the centers.

$$d = \sqrt{(e_x - v_x)^2 + (e_y - v_y)^2 + (e_z - v_z)^2}. \quad (1)$$

The distance between the original and removed positions of the vertex is:

$$\delta = \left| v_z - \left(\frac{leftcorner_z + rightcorner_z}{2} \right) \right|, \quad (2)$$

where z implies the height of the vertex.

The elevation threshold that is calculated at the vertex location is given by $A' = tan(\tau)\ d$.

Hence, our rule for enabling or disabling a vertex is

```
if δ < A' then
  disable vertex
else
  enable vertex
```

Our aim is to find a distance at which the threshold value does not exceed the elevation difference (δ). Therefore,

$$\delta = A' \quad (3)$$

$$\delta = tan(\tau)\ v_{act} \quad (4)$$

$$v_{act} = \frac{\delta}{tan(\tau)}. \quad (5)$$

The vertex activation distance $v_{act} = \delta / tan(\tau)$ is a precomputable value. So, the rule for enabling or disabling a vertex can be restated as

```
if v_act < d then
  disable vertex
else
  enable vertex
```

This reduces vertex simplification to a comparison between the precomputed v_{act} value and the measured distance (d) between the viewer and the vertex location.

In order to prevent cracks on the terrain and provide a suitable heuristic for morphing, a valid triangulation should

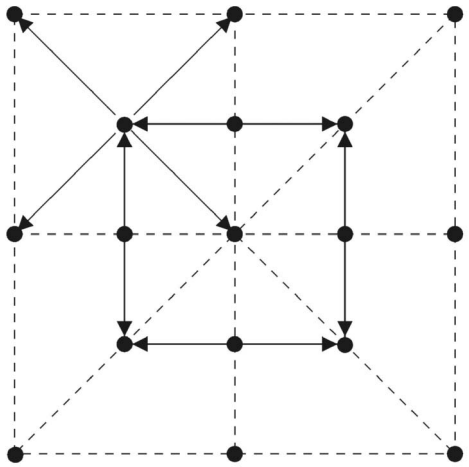

Fig. 5. Dependency relationships of center and border vertices.

be maintained. Our distance-based vertex activation scheme accomplishes this by using the activation values (v_{act}) assigned to each vertex in the preprocessing phase by using (5).

The activation values are assigned starting from the level just above the lowest level in the quadtree structure (Fig. 4a and Fig. 4b). After finding the activation distances for this level, we go up one level in the quadtree and the activation distances for the higher level nodes are calculated similarly. However, there are minor differences for the calculations at higher level blocks (Fig. 4c and Fig. 4d). This process is repeated going up until the root of the quadtree is reached. In addition, the distance necessary for at least one vertex to be activated ($\max_{act}$) and the distance necessary for all vertices to be activated ($\min_{act}$), which are used to speed up the simplification process, are precomputed for each quad-cell.

3.3 View Frustum Culling

Efficient view frustum culling (VFC) is crucial for interactive frame rates. While the quadtree is traversed, the nodes are checked against the viewing frustum and flags for the nodes in the quad block are cleared and set accordingly. To speed up frustum culling, frustum tests are done using bounding spheres enclosing the quad blocks.

In VFC, several optimizations can be performed, as listed below.

- One of the most important optimizations is to utilize the coherence between two frames when the user navigates through the terrain. If the user moves forward, then there is no need to cull the whole terrain again since the terrain is already culled in the previous frame. So, previously culled blocks can be used for the current frame [5].
- Another method is *deferred* VFC. By deferred VFC, we mean that VFC is not done for every frame, but at predefined intervals. In this way, the overhead brought by the VFC step can be decreased.
- As another approach, VFC depending on the deviation of the viewer location is used. Here, we run the VFC only if the user moves a prespecified distance from the previously culled position.

The control algorithm that decides when vertex activation and frustum culling operations should be done according to different culling schemes is as follows (Fig. 14 in the Appendix):

- If *deferred culling* is being used, then the time from the last VFC operation is calculated. If the period is reached, then the VFC and vertex activation algorithms are invoked.
- If *deviation-based culling* is being used, then we calculate the distance between the camera positions of the current frame and the last frame where the VFC operation is performed. If it is greater than the deviation threshold, then the VFC and vertex activation algorithms are invoked.
- If no VFC optimization is being used and the user is navigating, then the VFC and vertex activation algorithms are invoked at each frame.
- The draw-list construction algorithm is activated.

In the view frustum culling algorithm (Fig. 15 in the Appendix), the quadtree is traversed in preorder. If the viewer moves forward, the algorithm checks only the previously culled blocks. In this way, we make use of frame coherency. If the movement is not a forward movement then all quad blocks are to be checked. Since,

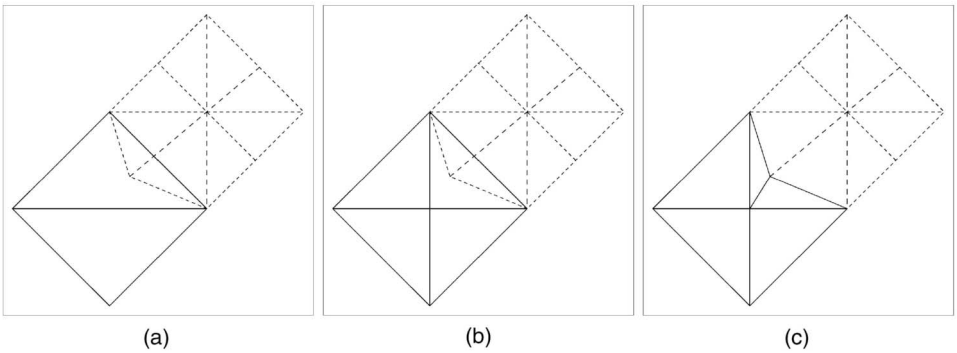

Fig. 6. Crack prevention: (a) crack formation, (b) activate the center vertex in the higher level block, and (c) use it in the triangulation to eliminate the crack.

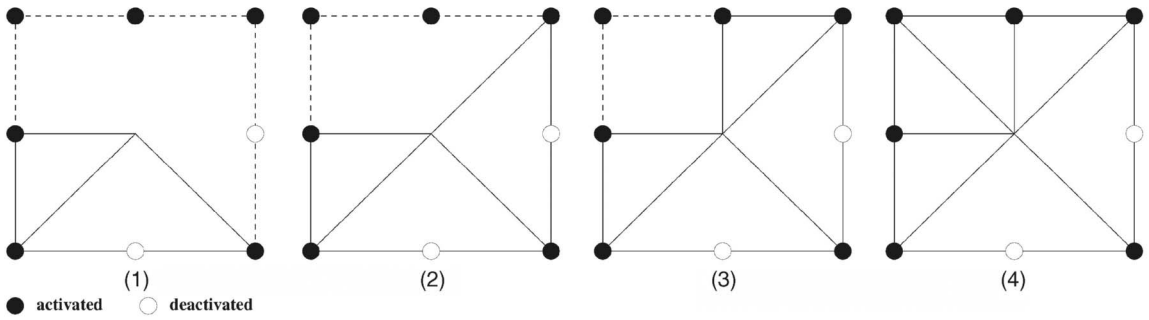

Fig. 7. A sample execution of the triangulation algorithm for a quad block.

our scene construction is not graph-based, we do not use rotation coherency as in [5].

The frustum checking algorithm (Fig. 16 in the Appendix) that is a part of the VFC algorithm can be summarized as follows:

- Test whether the block is completely inside, intersecting, or completely outside of the view frustum.
- If the block is intersecting with any of the planes of the frustum, then it is marked as *intersecting* and the VFC algorithm is called for the children of the block.
- If the block is completely inside the frustum, then all children of the checked block are marked as *inside* and the VFC algorithm is called for the sibling of the checked block.
- If the block is completely outside the frustum, then no marking takes place and the VFC algorithm is called for the sibling of the checked block.

3.4 Vertex Activation

Vertex activation takes place after view frustum culling. In this algorithm (Fig. 17 in the Appendix), the quadtree is again traversed in preorder, but we only traverse the nodes that are in the view frustum. In this step:

- The distance from the viewer position to the center of the quad block is calculated.
- If the distance is less than the minimum activation distance value of the quad block, then the viewer is close enough to the quad block and all vertices are activated. Since the maximized activation values are assigned to higher level quad blocks, it is not necessary to check the children of the quad block and they can be activated without further investigation.
- If the distance falls between the minimum and maximum activation distances, then each border vertex is checked individually, measuring the eye-point to vertex distances and comparing with their activation distances. If the distance measured is less than the activation distance, then the viewer is close enough and the vertex should be activated.

It should be noted that the dependents of a vertex are also activated during the activation of a vertex for crack prevention. Therefore, in the worst case, the number of vertex activation operations is $5T_N$ for a terrain with T_N nodes in the quadtree.

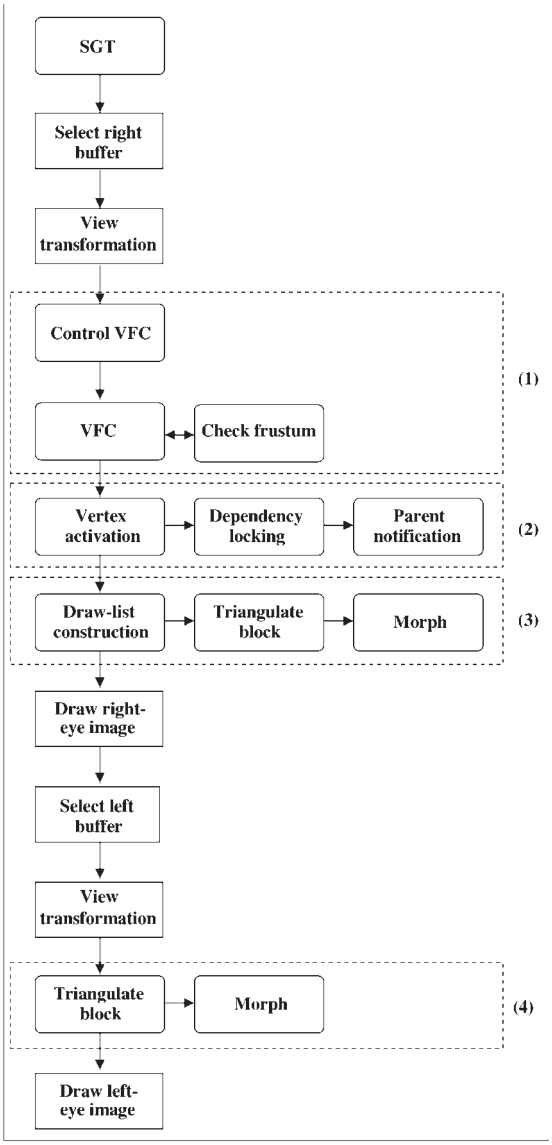

Fig. 8. The process flow diagram of the proposed framework for stereoscopic visualization. This diagram shows the invocation order of the presented algorithms. The data worked on by the processes in the dashed blocks are 1) the whole quad-tree, 2) the view-frustum culled data, 3) the activated blocks, and 4) the draw-list transferred.

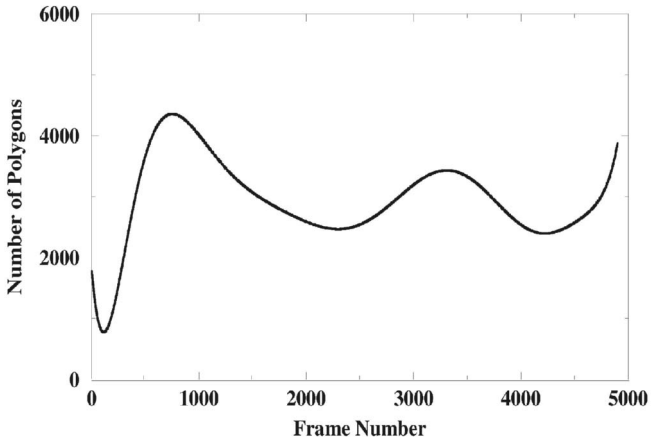

Fig. 9. Number of polygons rendered for the experimental visualization.

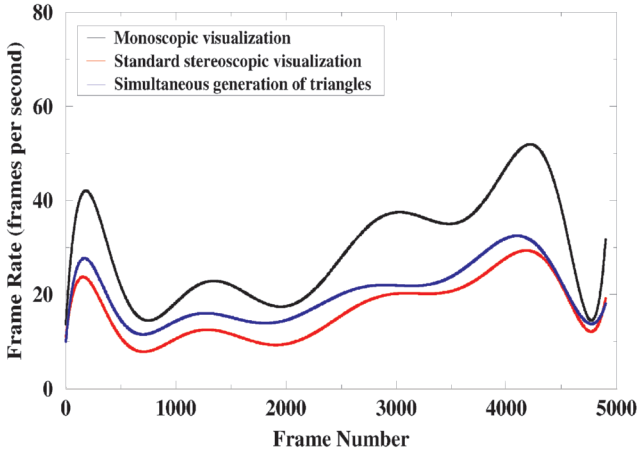

Fig. 10. Comparison of the frame rates of different types of visualizations: *monoscopic visualization* where only one image is generated for each frame; *standard stereoscopic visualization* where two images are generated for each frame; *simultaneous generation of triangles* for stereoscopic visualization where we utilize a triangle list for one eye to generate the triangles for the other eye.

3.5 Crack Handling and Triangulation

Cracks are one of the artifacts on the geometry when the two neighboring quad blocks differ in level of detail. If a border vertex is activated in a block, then a triangle including that vertex is drawn. If a neighboring block is not on the same level of detail, then no triangles including the common border vertex will be drawn for the neighboring block. This causes the formation of a crack. There are several approaches to crack handling. These include hiding the cracked position by drawing another triangle patch [4], triangulation of the gapped position [16], or not allowing crack formation by using the dependency relations [2].

In order to prevent cracks without causing discontinuities, dependency relations similar to the one in [2] are imposed between vertices. As shown in Fig. 5, center vertices are dependent on the four corner vertices. If they are activated, then the dependents are activated as well. Likewise, the border vertices are dependent on the center vertices of the two neighboring blocks at the same level. If a

TABLE 1
Performance Results

Visualization Method	Morph	Dynamic Culling	Deferred VFC	Deviation VFC	Average FPS	Performance Gain
Monoscopic	OFF	OFF	OFF	OFF	30.66	
SGT	OFF	OFF	OFF	OFF	21.83	42.61
Normal Stereo	OFF	OFF	OFF	OFF	15.31	
Monoscopic	OFF	OFF	ON	OFF	44.88	
SGT	OFF	OFF	ON	OFF	26.83	12.15
Normal Stereo	OFF	OFF	ON	OFF	23.92	
Monoscopic	OFF	ON	OFF	OFF	30.85	
SGT	OFF	ON	OFF	OFF	21.90	43.27
Normal Stereo	OFF	ON	OFF	OFF	15.29	
Monoscopic	ON	OFF	OFF	OFF	22.56	
SGT	ON	OFF	OFF	OFF	16.42	27.61
Normal Stereo	ON	OFF	OFF	OFF	12.87	
Monoscopic	ON	OFF	ON	OFF	31.38	
SGT	ON	OFF	ON	OFF	19.13	27.54
Normal Stereo	ON	OFF	ON	OFF	15.00	
Monoscopic	ON	ON	OFF	OFF	22.57	
SGT	ON	ON	OFF	OFF	16.43	37.29
Normal Stereo	ON	ON	OFF	OFF	11.97	
Monoscopic	OFF	OFF	OFF	ON	43.25	
SGT	OFF	OFF	OFF	ON	27.48	8.54
Normal Stereo	OFF	OFF	OFF	ON	25.32	
Monoscopic	ON	OFF	OFF	ON	31.91	
SGT	ON	OFF	OFF	ON	20.70	15.25
Normal Stereo	ON	OFF	OFF	ON	17.96	

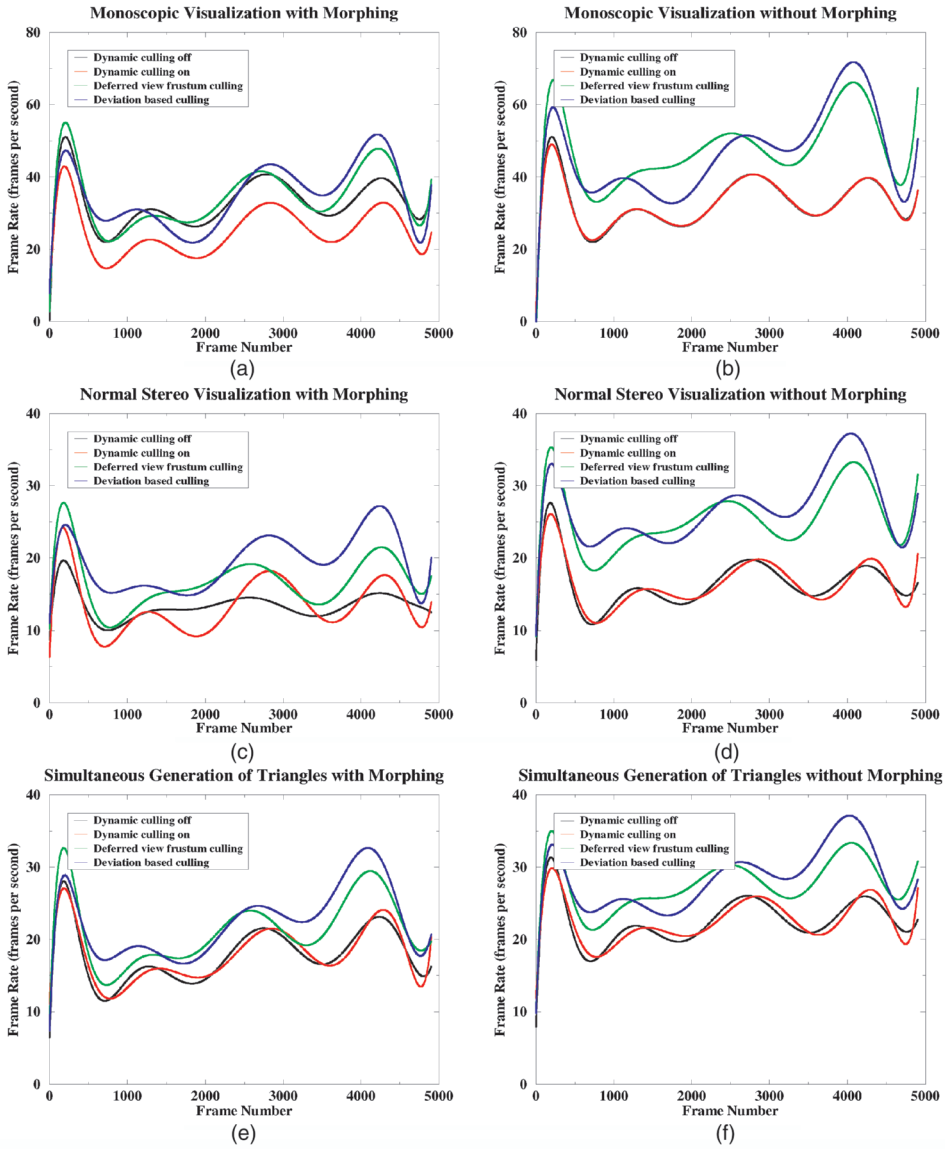

Fig. 11. Comparison of the frame rates for each type of visualization with different morphing/culling options: (a) *monoscopic visualization* with morphing; (b) *monoscopic visualization* without morphing; (c) *standard stereoscopic visualization* with morphing; (d) *standard stereoscopic visualization* without morphing; (e) *simultaneous generation of triangles* with morphing; (f) *simultaneous generation of triangles* without morphing.

border vertex is activated, then its dependent vertices at the same level are activated, too (Fig. 6).

During the vertex activation process, vertex dependents are locked by calling the dependency locking algorithm (Fig. 18 in the Appendix) when a vertex is activated. This procedure activates the related vertex as follows:

- Turn its flag on.
- Inform its parents that the corresponding quad block is activated.
- If the dependent vertex was enabled previously, then stop locking because locking has already been done and there is no need to go further.
- Call the procedure recursively to further lock the dependents of the dependent vertex.

In order to triangulate a quadrant, no children should be active in that quadrant. Otherwise, overlapping triangle patches may exist in that area. This is guaranteed for a block by checking the fields showing the activation status of its children, which are modified by a notification algorithm. Fig. 19 in the Appendix gives the algorithm for notifying the parents of a node. In this algorithm, the quadrant number of the subquad block that is notifying its parent is calculated and the related field of the parent is marked. The notification process is stopped if the location of the child block in the quad was marked before, which means the higher level quad blocks have already been notified.

Before the triangulation of each quad block, a drawlist for that block is constructed. The draw-list construction algorithm (Fig. 20 in the Appendix) can be summarized as follows:

- Check whether the center vertex of the quad block is activated or not.
- If the center is activated, then check the bottom-left quadrant of the block for triangulation.

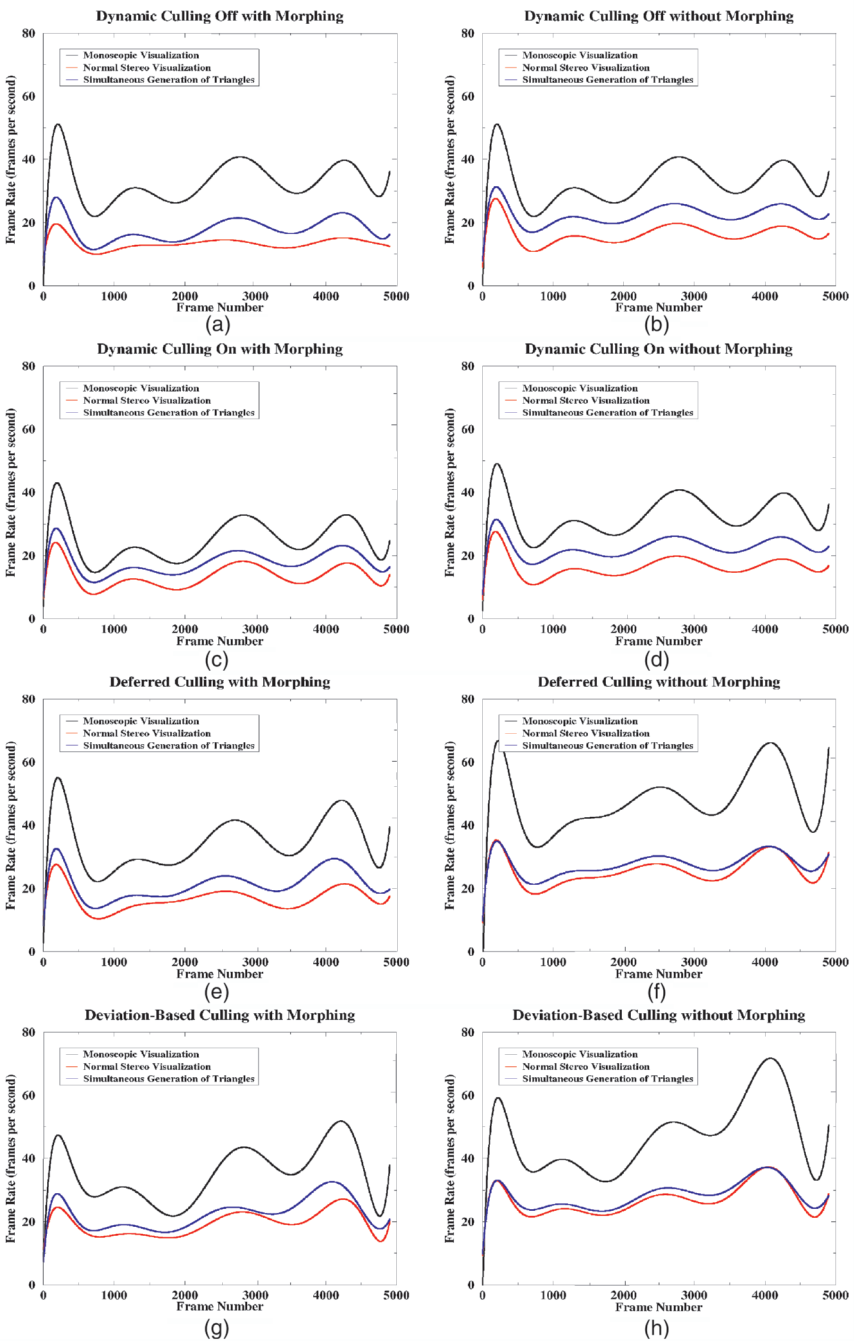

Fig. 12. Comparison of different culling schemes for each type of visualization: (a) *dynamic culling off* with morphing; (b) *dynamic culling off* without morphing; (c) *dynamic culling on* with morphing; (d) *dynamic culling on* without morphing; (e) *deferred culling* with morphing; (f) *deferred culling* without morphing; (g) *deviation-based culling* with morphing; (h) *deviation-based culling* without morphing.

- If no subquad block is activated in the bottom-left quadrant and the upper-left quadrant is not available, then put the left border vertex into the draw-list.
- If no subquad block is activated in the bottom-left, then put the bottom-left corner vertex into the draw-list.
- If the bottom-right quadrant is not available, then put the bottom border vertex into the draw-list.
- If the bottom border vertex is activated and the bottom-left and the bottom-right quadrants are available, then put the bottom border vertex into the draw-list.
- Process all the other quadrants as above.
- Triangulate the quad-block.
- Check the children of the block and repeat the process.

- If the center is not activated then check the sibling of the quad-block.

The triangulation algorithm essentially checks the vertices of a quad block starting from the bottom-left subquad and forms triangles with the activated vertices by

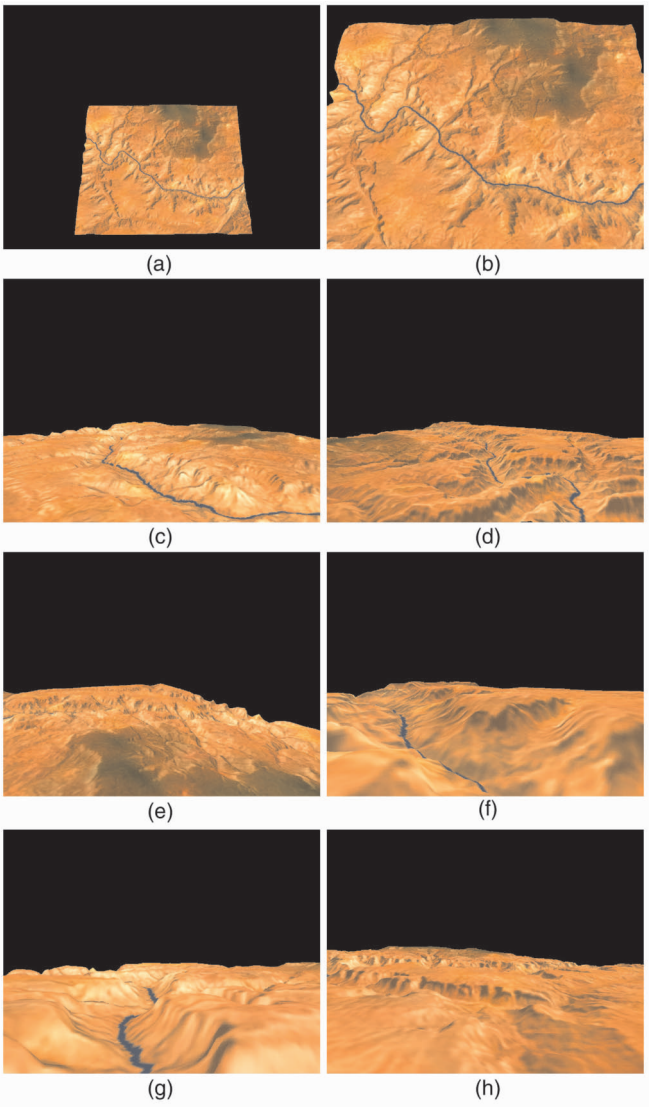

Fig. 13. Still frames from a monoscopic walkthrough.

traversing the quad block in counterclockwise direction. The triangulation of a quad block is illustrated in Fig. 7.

3.6 Morphing

One of the important issues while visualizing complex geometric environments using a multiresolution representation is that there should be no popping artifacts while switching between different levels of detail. The best way to achieve this is with a smooth morphing of the geometry between successive frames.

The proposed morphing scheme works as follows: The distances between activated and deactivated states of the vertices, namely δ values, are precalculated using (2). A prespecified morph-segment number is used to decide at how many steps should the enabling or disabling vertex reach its new position. In the morphing algorithm (Fig. 21 in the Appendix):

- If the vertex is a coarsening vertex, then δ value is added or subtracted from the elevation, depending on whether it is below or above its deactivated state.

```
Algorithm ControlVFC(eye)
  if (DEFERREDCULLING == ON) {
     time = gettime()
     if ((time - frustumtime) > DEFERRINGTIME ||
         rotating) {
        ViewFrustumCulling
        ActivateVertices
        rotating = no
        frustumtime = time
     }
  }
  else
     if (DEVIATIONCULLING == ON) {
        d = sqrt((current_x - old_x)**2 +
                 (current_y - old_y)**2 +
                 (current_z - old_z)**2)
        if (d > DEVIATIONDISTANCE || rotating) {
           old_x = current_x
           old_y = current_y
           old_z = current_z
           ViewFrustumCulling
           ActivateVertices
           rotating = no
        }
     }
     else
        // dynamic morphing is on, or viewer is
        // moving while dynamic morphing is off
        if (CullingNeeded) {
           ViewFrustumCulling
           ActivateVertices
        }
  ConstructDrawList(eye)
```

Fig. 14. The algorithm that calls view frustum culling and activation procedures depending on the culling scheme used.

- The δ value is divided by the morph-segment value.
- At each frame, the morphing state of the vertices is modified and the divided distance is used to calculate the new elevation for that point.
- If the morph-segment value is modified more than once by the neighboring quad blocks while drawing a frame, then gaps may occur between the neighboring quad blocks. In order to prevent the formation of such gaps, a flag is used to lock the vertex morphing at each frame.

During morphing, a positive morph value is used for a refining vertex and a negative morph value is used for a

```
Algorithm ViewFrustumCulling;
  while (nodes are not finished) {
    if (viewer moves forward)
       check previously culled blocks for culling
    else {
       check all nodes for culling
       clear the previous frame flags
    }
    node=CheckFrustum(node)
  }
```

Fig. 15. The view frustum culling algorithm.

```
Algorithm CheckFrustum(node);
  QuadTree[node].activated=test(node)
  if (QuadTree[node].activated==intersecting) {
     increment hits
     mark the node as intersecting
     return child(node)
  }
  else
     if (QuadTree[node].activated==inside) {
        increment hits
        mark the node as inside
        mark all children as inside
        return sibling(node)
     }
  // the block is completely outside
  return sibling(node)
```

Fig. 16. The frustum checking algorithm.

coarsening vertex. While the viewer gets closer to the terrain, vertices are enabled and morphing is started. Morphing for the coarsening vertices starts immediately, as soon as the viewer begins to get away from the terrain, which provides a uniform morphing scheme for both the refinement and coarsening phases during the navigation.

4 Stereoscopic Visualization

At first, we need to explain the stereoscopic projection system we used. In general, stereo projections are divided into two: *on-axis* and *off-axis* [7]. Off-axis projections require the implementation of asymmetric parallel view frustum projections. By using off-axis projections, a more accurate stereo view can be achieved in terms of reduced ghosting effect in the peripheries. However, it has a disadvantage in terms of execution speed because control of the center of projection is not implemented in hardware for most low-end systems [7]. Therefore, on-axis projection, which modifies the data with translations and rotations, has an important advantage over off-axis projections in terms of speed. The disadvantages of on-axis projections, namely, ghosting effect and loss of data in side-views, are eliminated with our simple correction: We operate on the data that is in the view frustum, plus the data on the left and right of the

```
Algorithm ActivateVertices;
node=0    // start activation from the root
if (hits!=NIL)
   while (all view frustum culled nodes are not finished) {
      if (QuadTree[node].culled==YES) {
         d=(eye_x-center_x)**2 + (eye_y-center_y)**2 + (eye_z+center_z)**2
         if (((QuadTree[node].minact*QuadTree[node].minact)>=d)) {
            // If the viewer is closer than all vertices' activation distance
            // lock all vertices down the quadtree without checking them individually
            for all quadblocks including this one do {
               LockDependents(centervertex)
               LockDependents(bottombordervertex)
               LockDependents(rightbordervertex)
               LockDependents(upbordervertex)
               LockDependents(leftbordervertex)
            }
            node=sibling(node)
         }
         else // the distance is in uncertainty section
            if ((QuadTree[node].maxact*QuadTree[node].maxact)>=d) {
               LockDependents(centerrow,centercol)
               for all border vertices do {
                   d=(eye_x-vertex_x)**2 + (eye_y-vertex_y)**2 + (eye_z+vertex_z)**2
                   if (Terrain[borderrow][bordercol].activation>=d)
                      LockDependents(borderrow,bordercol)
               }
               node=child(node)
            }
            else  // the block will not be activated
               node=sibling(node)
      }
      else
         node=sibling(node)
   }
```

Fig. 17. The vertex activation algorithm.

```
Algorithm LockDependents(row, col, node);
  Terrain[row][col].activestate=YES
  for (i=0; i<4; i++) {
      if (Terrain[row][col].dependent[i][0]!=NIL) {
         if (Terrain[Terrain[row][col].dependent[i][0]]
                [Terrain[row][col].dependent[i][1]].activestate==NO) {
            NotifyParents(node)
            LockDependents(Terrain[row][col].dependent[i][0],
                Terrain[row][col].dependent[i][1], Terrain[row][col].dependent[i][2])
         }
         else
            break // exit without checking the rest since no more dependents exist
      }
  }
```

Fig. 18. Locking the vertex dependents.

view frustum in half of the projection of the interocular distance for each eye.

With the correction of the ghosting effect, the coverage of the stereoscopic area is the same as the stereoscopic area in off-axis projections. Since the interocular projection is very small with respect to the terrain, elimination of the ghosting effect does not cause significant processing overhead.

4.1 Simultaneous Generation of Triangles

Terrain data is huge with respect to the interocular distance (IOD). In order to prevent the ghosting effect that can occur in on-axis projections, we add the necessary data to the view frustum by enlarging it by half the projection distance of IOD in both sides. Besides, we do not make occlusion culling since it does not increase the performance significantly for the terrain data [3]. Therefore, left and right eye views may operate on the same view frustum culled data safely, which is the most important condition for the SGT algorithm.

The second eye image is generated during the draw-list construction for the first eye. When drawing the second eye, we do not repeat the view frustum culling and vertex activation processes. We only traverse the draw-list constructed for the right eye and do not make any modifications on the data created previously, while the left eye view is drawn. This is achieved by modifying the algorithm for construction of the draw-list given in Fig. 20 as in Fig. 22. The modification adds a control block to the algorithm that checks the drawn eye and, if it is the second eye, then only uses the draw-list constructed while the first eye was being drawn. Stereoscopic drawing algorithm using the SGT approach is given in Fig. 23 in the Appendix. In this algorithm:

```
Algorithm NotifyParents(node);
 // calculate the quadrant index
 childno=node-child(parent(node))
 while (node) {
   node=parent(node)
   if(QuadTree[node].childactivated[childno]==NO)
     QuadTree[node].childactivated[childno]=YES
   else
     // exit since the rest of the parents know
     // that the quadrant is already activated
     break
   childno=node-child(parent(node))
 }
```

Fig. 19. Notifying parents.

```
Algorithm ConstructDrawList(eye);
node=0
while (nodes are not finished) {
  if (Terrain[crow][ccol].activestate==YES) {
     EyeBlock[node].center=YES
     // process bottom left quadrant
     if (canIdrawbottomleft(node)==YES) {
        // no subquads are active
        if (canIdrawupperleft(node)==NO)
           // above subquad is active
           EyeBlock[node].leftborder=YES
        EyeBlock[node].bottomleft=YES
        if (canIdrawbottomright(node)==NO)
           // next subquad is active
           EyeBlock[node].bottomborder=YES
     }
     if (canIdrawbottomborder(node)==YES)
        // neighboring quadrant centers
        // are inactive and border is active
        EyeBlock[node].bottomborder=YES

     // process bottom right quadrant
     .....
     // process upper right quadrant
     .....
     // process upper left quadrant
     .....
     TriangulateBlock(node,eye)
     node=child(node)
  }
  else
     node=sibling(node)
}
```

Fig. 20. Construction of the draw-list.

```
Algorithm Morph(cr, cc)
  if (MORPHING==ON) {
     emorph=Terrain[cr][cc].morph
     // if vertex is disabled then corrected elevation is taken
     if (Terrain[cr][cc].activestate==DISABLED)
        morphed_el=Terrain[cr][cc].elevation - Terrain[cr][cc].morphdistance
     else
        morphed_el=Terrain[cr][cc].elevation
     if (emorph) {
        // if elevation is above its deactivated state morphdistance > 0
        // if elevation is below its deactivated state morphdistance < 0
        morphdist=(emorph*(Terrain[cr][cc].morphdistance/MORPHSEGMENTS))
        if (emorph<0) { // the vertex is coarsening
           morphed_el=morphed_el-Terrain[cr][cc].morphdistance-morphdist
           if (!Terrain[cr][cc].morphlock) { // vertex not locked by another quad
              (Terrain[cr][cc].morph)++
              Terrain[cr][cc].morphlock=YES
           }
        }
        else {          // the vertex is refining
           morphed_el=morphed_el-morphdist
           if (!Terrain[cr][cc].morphlock) { // vertex not locked by another quad
              (Terrain[cr][cc].morph)--
              Terrain[cr][cc].morphlock=YES
           }
        }
     }
  return morphed_el
  }
```

Fig. 21. The morphing algorithm.

- The right eye buffer for stereo is selected.
- The view transformation for the right eye is calculated.
- The VFC and the vertex activation operations are performed.
- The right eye view is drawn while transferring the draw-list to the left eye.
- After drawing the right eye view, the left eye buffer is selected.
- The view transformation for the left eye is calculated.
- The left eye view is drawn.

The process flow diagram of the proposed framework for stereoscopic terrain visualization is given in Fig. 8.

5 Performance Results

In the visualization experiments, approximately 4,000 polygons were rendered for each eye on the average at each frame. The terrain used is a part of the Grand Canyon that has very sharp ridges in it, with 513×513 vertices. The results were obtained on a personal computer with Intel Pentium III-550 Mhz CPU and 64 MB of main memory with 32 MB of graphics memory.

We prepared a flythrough of the terrain with approximately 5,000 frames. Still frames from the monoscopic flythrough are shown in Fig. 13. The number of polygons rendered during the flythrough is shown in Fig. 9. All related figures are smoothed using a regression function for easy interpretation. Fig. 10 shows the average frame rates of different types of visualization techniques by using different morphing, culling, and rendering techniques at different parts of the flythrough. It gives a general overview about the performance of the visualization techniques. In this test, the average frame rate of the proposed SGT approach is 17.65 fps, whereas the frame rate of the monoscopic visualization is 25.05 fps. Performance comparison of the visualization methods with different types of culling, morphing, and rendering techniques are given in Table 1. These results show that the best performance in stereo is achieved when deviation-based culling is used without morphing with the proposed SGT approach. In this case, the average rendering speed for SGT is 27.48 fps, where its monoscopic correspondent is 43.25 fps. The largest performance gain is achieved when SGT approach is used with dynamic culling without morphing. In this case, a performance gain of 43.27 percent over normal stereo implementation is achieved. The morphing scheme imposes approximately 10 to 30 percent overhead on the frame rate due to clearance of the morph flags for the vertices going out of the view frustum when the user is navigating.

In Fig. 11, performance comparison showing the frame rates of our culling techniques with each visualization method is given. In deviation-based culling tests, the deviation threshold was taken as 500 meters. In deferred culling, the deferring time was taken as 0.1 second. As is

```
Algorithm ConstructDrawList(eye);
node=0
while (nodes are not finished) {
   if ((eye==left_eye)&&(LISTTRANSFER==ON)) {
      if (EyeBlock[node].center) {
         TriangulateBlock(node,left_eye)
         ClearEyeBlock
         node=child(node)
      }
      else
         node=sibling(node)
   }
   else
      .....
```

Fig. 22. Using the draw-list constructed for the right eye in generating triangles for the left eye image on the SGT algorithm.

seen in the figure, deviation-based culling and deferred culling perform better than dynamic culling. The performances are almost the same when dynamic culling is on or off since the viewer is continuously moving in the experiments. Turning dynamic culling off becomes advantageous when the viewer does not move. Under normal conditions, the viewer generally stops moving at undetermined instances. Since the screen will be rendered without being culled, the stereo effect will be much better.

In Fig. 12, the performances of the visualization methods for each of the proposed culling schemes are given. It is apparent that the proposed stereo visualization method performs much better than the normal stereoscopic visualization.

6 Conclusion

This paper presents a framework for the stereoscopic view-dependent visualization of large scale terrain models. A quadtree-based multiresolution representation is used for the terrain data. This structure is queried to obtain the view-dependent approximations of the terrain model at different levels of detail. In order not to lose depth information, which is crucial for the stereoscopic visualization, we make use of a different simplification criterion, namely, distance-based angular error threshold. An algorithm is proposed for the construction of stereo pairs in order to speed up the view-dependent stereoscopic visualization. The proposed algorithm simultaneously generates the triangles for two stereo images using a single draw-list so that the view frustum culling and vertex activation is done only once for each frame. The cracking problem is solved using the dependency information stored for each vertex. The popping artifacts that can occur while switching between different resolutions of the data are eliminated using morphing. The proposed algorithms are implemented on personal computers and graphics workstations. Performance experiments show that the second eye image can be produced approximately 45 percent faster than drawing the two images separately and a smooth stereoscopic visualization can be achieved at interactive frame rates using continuous multiresolution representation of height fields.

```
Algorithm SGTStereo {
  // Draw right-eye view.
  SelectRightBuffer
  ClearBuffer
  Calculate view transformation for right eye
  ControlVFC(right_eye)

  // Draw left-eye view.
  SelectLeftBuffer
  ClearBuffer
  Calculate view transformation for left eye
  ConstructDrawList(left_eye)
}
```

Fig. 23. Stereoscopic drawing using the SGT Algorithm.

Appendix

Algorithms in Pseudocode

In Fig. 14, Fig. 15, Fig. 16, Fig. 17, Fig. 18, Fig. 19, Fig. 20, Fig. 21, Fig. 22, and Fig. 23, we present the algorithms in pseudocode in order to make the techniques reimplementable. The algorithms are given in C-like pseudocode, where details are omitted for the sake of simplicity and clarity.

Acknowledgments

This project is partially supported by the Turkish Scientific and Technical Research Council (TÜBITAK) with grant no. 198E018. Grand Canyon Data obtained from the United States Geological Survey (USGS), with processing by Chad McCabe of the Microsoft Geography Product Unit. The authors would like to thank Ugur Dogrusöz and Murat Akbay for reading the manuscript. They are grateful to the anonymous reviewers for their valuable comments.

References

[1] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner, "Real-Time Continuous Level of Detail Rendering of Height Fields," *ACM Computer Graphics (Proc. SIGGRAPH '96)*, pp. 109-118, 1996.

[2] R. Pajarola, "Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation," *Proc. IEEE Visualization '98*, pp. 19-26, 1998.

[3] H. Hoppe, "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering," *Proc. IEEE '98*, pp. 35-42, 1998.

[4] P. Lindstrom, D. Koller, L.F. Hodges, W. Ribarsky, N. Faust, and G. Turner, "Level-of-Detail Management for Real-Time Rendering of Phototextured Terrain," Technical Report GIT-GVU-95-06, Graphics, Visualization, and Usability Center, College of Computing, Georgia Inst. of Technology, 1995.

[5] U. Assarsson and T. Möller, "Optimized View Frustum Culling Algorithms for Bounding Boxes," *J. Graphics Tools*, vol. 5, no. 1, pp. 9-22, 2000.

[6] D. Bartz, M. Meisner, and T. Hüttner, "OpenGL-Assisted Occlusion Culling for Large Polygonal Models," *Computers & Graphics*, vol. 23, no. 5, pp. 667-679, 1999.

[7] L.F. Hodges, "Tutorial: Time-Multiplexed Stereoscopic Computer Graphics," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, pp. 20-30, 1992.

[8] L.F. Hodges and D.F. McAllister, "Stereo and Alternating-Pair Techniques for Display of Computer-Generated Images," *IEEE Computer Graphics and Applications*, vol. 5, no. 9, pp. 38-45, 1985.

[9] J.D. Ezell and L.F. Hodges, "Some Preliminary Results on Using Spatial Locality to Speed-Up Raytracing of Stereoscopic Images," *Proc. SPIE 1256, Stereoscopic Display and Applications I,* pp. 298-306, 1990.
[10] S.J. Adelson and L.F. Hodges, "Stereoscopic Ray Tracing," *The Visual Computer,* vol. 10, no. 3, pp. 127-44, 1993.
[11] S.J. Adelson, J.B. Bentley, I.S. Chong, L.F. Hodges, and J. Winograd, "Simultaneous Generation of Stereographic Views," *Computer Graphics Forum,* vol. 10, pp. 3-10, 1991.
[12] S.J. Adelson and C.D. Hansen, "Fast Stereoscopic Images with Ray Traced Volume Rendering," *Proc. Symp. Volume Visualization,* pp. 3-9, 1994.
[13] H. Taosong and A. Kaufman, "Fast Stereo Volume Rendering," *Proc. IEEE Visualization '96,* pp. 49-56, 1996.
[14] R. Hubbold, D. Hancock, and C. Moore, "Stereoscopic Volume Rendering," *Proc. Visualization in Scientific Computing '98,* pp. 105-115, 1998.
[15] H. Samet, "The Quadtree and Related Data Structures," *ACM Computing Surveys,* vol. 16, no. 2, pp. 187-260, 1984.
[16] R. Nielson, D. Holliday, and T. Roxborough, "Cracking the Cracking Problem with Coons Patches," *Proc. IEEE Visualization '99,* pp. 285-290, 1999.

Ugur Güdükbay received the BSc degree in computer engineering from Middle East Technical University, Ankara, Turkey, in 1987. He received the MSc and PhD degrees, both in computer engineering and information science, from Bilkent University, Ankara, Turkey, in 1989 and 1994, respectively. Then, he conducted research as a postdoctoral fellow at the University of Pennsylvania, Human Modeling and Simulation Laboratory. Currently, he is an assistant professor at Bilkent University, Department of Computer Engineering. His research interests include physically based modeling, human modeling and animation, multiresolution modeling and rendering, and stereoscopic visualization. He is a member of the IEEE Computer Society and ACM SIGGRAPH.

Türker Yilmaz received the BSc degree in finance management from the Turkish Military Academy, Ankara, Turkey, in 1991. After finishing a one-year automated data processing program at Middle East Technical University, Ankara, Turkey, in 1998, he received the MSc degree in computer engineering from Bilkent University, Ankara, Turkey, in 2001. He is a captain in the Turkish Army under the signals corps. Currently, he is both an instructor at the Turkish Military Academy and a PhD student at Bilkent University, Department of Computer Engineering. His research interests include stereoscopic visualization, virtual reality and simulation programming, and multiresolution modeling and rendering.