Stereoscopic View-Dependent Visualization of Terrain Height Fields (Regular Paper)

Uğur Güdükbay and Türker Yılmaz

Bilkent University

Department of Computer Engineering

06533 Bilkent, Ankara, Turkey

Tel: +90 (312) 266 41 33, **Fax:** +90 (312) 266 41 26

e-mail: {gudukbay, yturker}@cs.bilkent.edu.tr

Corresponding Author: Uğur Güdükbay

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 loose 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 % faster than drawing the two images separately and a smooth stereoscopic visualization can be achieved at interactive frame rates using continuous multi-resolution representation of height fields.

Keywords: Stereoscopic visualization, terrain height fields, multiresolution rendering, quadtrees.

I. Introduction

Modern graphics workstations allow rendering of millions of polygons per second. Although the power of these systems increases greatly, it cannot catch up with the quality demand needed for graphics systems used for visualizing complex geometric environments since the data that needs to be processed increases quite fast as well. 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 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 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. Generally, triangles are used as modeling primitives for complex scenes. The triangulation must be adaptive in order to reduce the number of polygons to be processed and make efficient use of the limited memory sources. This means that high frequency elevation changes should be triangulated with more triangles than low frequency regions.

In stereoscopic visualization, the two views must be generated fast enough to achieve interactive frame rates. It is apparent that there will be some limitations in terms of the features that could be incorporated to increase the realism of the visualizations as compared to monoscopic visualizations. Since the amount of data that can be processed decreases drastically, complex visualizations, such as the visualization of urban scenery over the terrain, cannot be done easily. 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:

1. A traversal algorithm on the quadtree representation of the terrain data that is preventing the formation of cracks using dependency information between the vertices.
2. 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.

3. An algorithm to speed-up the generation of the stereo pairs for stereoscopic view-dependent visualizations, namely Simultaneous Generation of Triangles.
4. Several strategies to optimize the view frustum culling process that are user-specifiable and can be switched according to navigation characteristics while the program is running: coherency utilization between the frames of a visualization, deferred view frustum culling that culls at predefined intervals, view frustum culling based on the deviation of the viewer location that culls when the user moves a prespecified distance from its position, and culling with respect to the far plane whose distance is determined based on the altitude of the viewer.
5. A morphing technique that works in the same manner for both refining and coarsening operations while visualizing the terrain data.

The rest of this paper is organized as follows. In Section 2, we describe related work on both multi-resolution modeling of terrain data and stereoscopic visualization. Our quadtree based multi-resolution 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 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. Finally, conclusions are given in Section 6.

II. Related Work

A. 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 was used to represent height fields and to visualize terrain at real time. The simplification hierarchy is represented using a quadtree structure. During 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, the difference between the projections of a vertex when it is active and inactive is compared to a prespecified pixel threshold. If this value is smaller than the threshold then the vertex is removed.

In [2], a framework for monoscopic visualization of regular grid elevation data is proposed. The framework that addresses different problems of visualizing terrain data represented as a quadtree structure is as follows: In order to achieve a valid triangulation, the basic quadtree construction scheme has been turned into a restricted one by applying a dependency relation between the vertices. Every vertex is dependent on the two other vertices of the same or the next higher level in the quadtree hierarchy. This means that if a vertex is selected for triangulation then the dependents must also be selected. A breadth-first search is performed in the quadtree for progressive mesh construction. For triangle strip construction, the quadtree is traversed using Hamiltonian paths. 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 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. Texture binding is used for large texture mapping. Although the swapping cost is very high, this is necessary if large textures are to be used. To hide the appearance of cracks, each crack is closed by an additional triangle. Although this scheme produces a stepped view on cracked regions this appearance is decreased by the use of textures.

Other techniques, which decrease the number of polygons to be processed, hence optimize 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.

B. 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]. Most of the applications support stereoscopic display by completely generating the two images for the left and right eye views separately. Except large-scale simulator applications such as flight simulators, there are not many applications for low-end systems, especially personal computers, that allow the user to navigate freely over the data. In our work, we propose algorithms to reduce the overhead for stereoscopic visualization while the user is navigating over the data.

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 change 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 re-projection for fast direct 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.

III. Multiresolution Modeling

A. Data Structures

Here, we present the data structures used in our implementation. 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 stage the vertex is, a pre-calculated value showing the distance between active and inactive states of the vertex, and a morph lock flag to prevent the morph field from being decremented again by other neighboring vertices at the same frame (see Figure 1).

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 structure can be modified to reduce the amount of storage required by calculating the dependent vertices on the fly, which reduces the storage requirement at the expense of some processing overhead. However, dependent blocks for a vertex must be determined using a search process in the preprocessing phase since there is no straightforward way of determining the neighbors of an arbitrary quad block [15].

The `QuadTree` array occupies 26 bytes per node. Since each lowest level quad block (highest detail) contains four vertices, 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.

```
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[COL][ROW]; // 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

B. Terrain Representation

The quadtree structure is represented as a one-dimensional array (Figure 2). In this tree, each level represents a different level of detail on the terrain. We use the quadtree to store the indices, minimum and maximum elevations, and minimum and maximum activation distances for the vertices. Activation data for the vertices are not stored in the quadtree. In addition, we use an array-based representation of the quadtree to eliminate the need

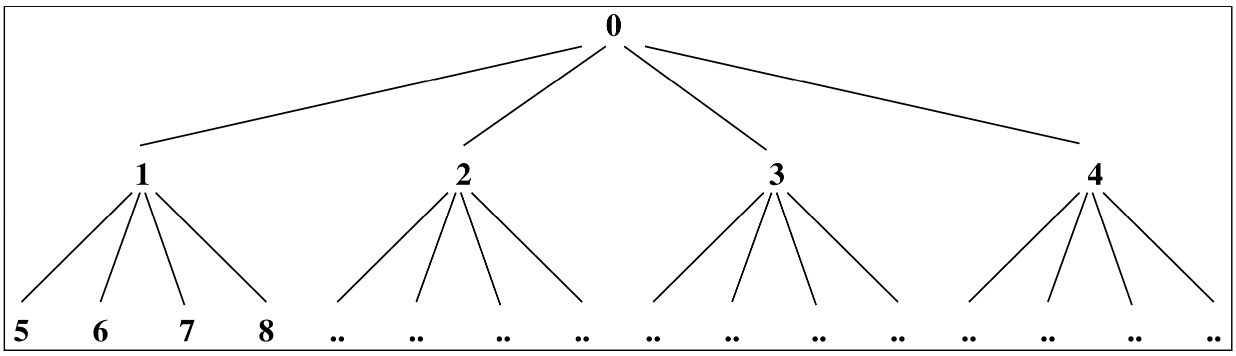

Fig. 2. Quadtree structure

84	83	80	79	68	67	64	63
81	82	77	78	65	66	61	62
72	71	76	75	56	55	60	59
69	70	73	74	53	54	57	58
36	35	32	31	52	51	48	47
33	34	29	30	49	50	45	46
24	23	28	27	40	39	44	43
21	22	25	26	37	38	41	42

Fig. 3. Numbering scheme for the quad blocks

for pointer manipulation. The numbering scheme for the quadtree structure when it is stored in a one-dimensional array is illustrated in Figure 3. The root is labeled as 0 and the rest is numbered recursively in counter-clockwise direction.

C. Approximation Criterion

As mentioned previously, screen-space error criterion for approximating the terrain is not sufficient in order to achieve a correct stereoscopic view. This is illustrated in Figure 4. Elevation differences are taken into account to evaluate a vertex for removal when screen-space error metric is used. The number of projected pixels for the vertex is calculated for this purpose. If this number is greater than the pre-specified pixel tolerance then the vertex is kept, otherwise it is removed. The problem here is that if the eye is above a quad block then its projection to the camera plane will be very small yielding to the elimination

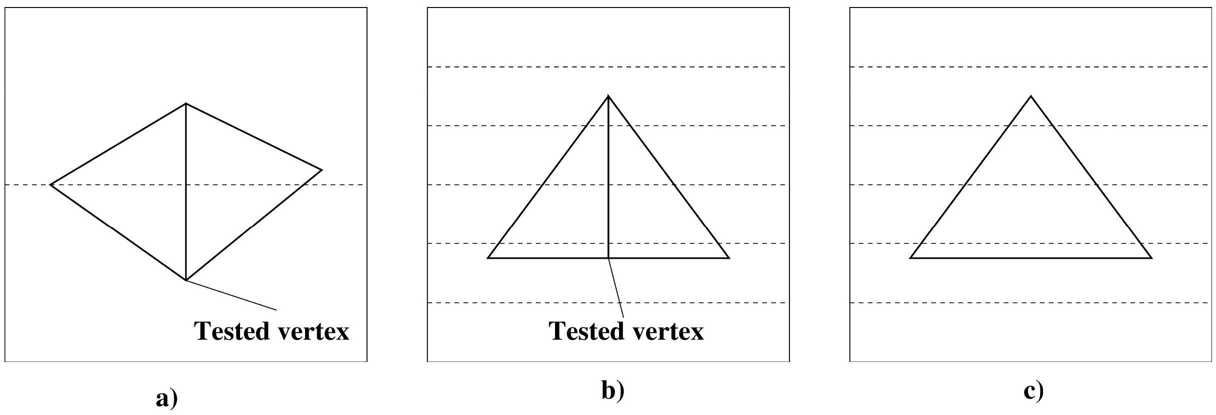

Fig. 4. Screen-space error metric. (a) Side view of a quad block. (b) Top view of the same block. (c) Edge removal by screen-space error based algorithm.

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, tested vertices will be removed. Therefore, although the screen-space error metric is suitable for monoscopic view, it degrades the stereo effect and results in incorrect stereoscopic vision.

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. We accept the eye to be in the center of a sphere. The candidate vertices tested for the elimination are located on the surface of the sphere. Our threshold value at a vertex location is computed by using the prespecified angular threshold value and the radius of the sphere. The greater the radius of the sphere (i.e., the distance from eye to the vertex) is, the larger the size of the threshold will be. We can derive the elevation threshold by taking the tangent of the angular threshold at the given distance. Figure 5 illustrates our angular error metric for evaluation of a vertex for removal.

The distance from the eye position to the vertex is

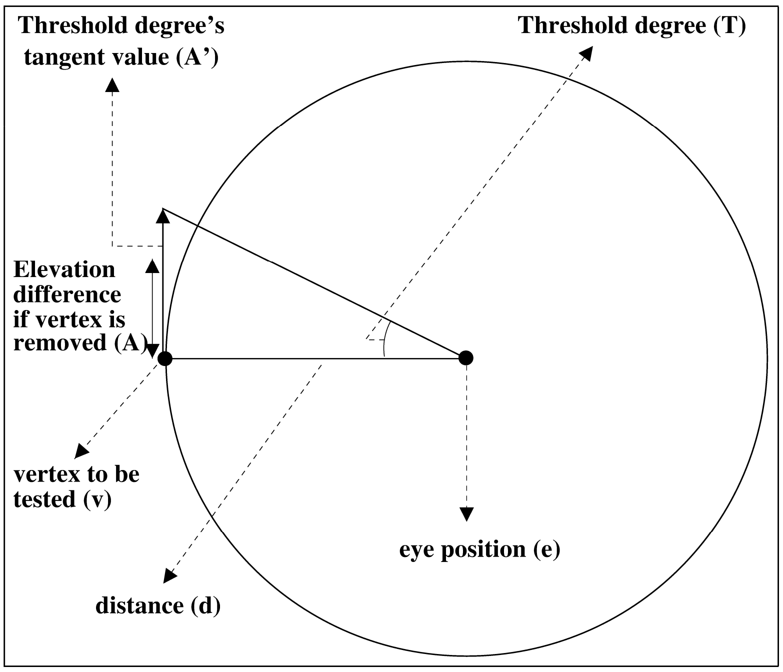
Threshold degree's tangent value (A')
Threshold degree (T)
Elevation difference if vertex is removed (A)
vertex to be tested (v)
eye position (e)
distance (d)

Fig. 5. Angular error threshold representation for vertex removal.

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

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

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

The tangent value of the angular threshold in degrees is given by $A' = tan(\tau)\ d$.

Hence, our rule for enabling or disabling a vertex is

```
if $\delta \le A'$ 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). So

$$\delta = A' \tag{3}$$
$$\delta = tan(\tau)\ d \tag{4}$$
$$d = \frac{\delta}{tan(\tau)} \tag{5}$$

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

```
if $v_{act} < d$ then
   disable vertex
else
   enable vertex
```

When the quadtree is built, v_{act} values for each vertex are computed. In addition, the maximum distance necessary for at least one vertex to be activated (min$_{act}$) and the minimum distance necessary for all vertices to be activated (max$_{act}$) are precomputed for each quad-cell. If the distance is greater than min$_{act}$, then the lowest resolution block is drawn. If it is less than max$_{act}$, then the full resolution block is drawn without checking internal vertices. Otherwise, vertices are considered individually.

D. View Frustum Culling

An efficient view frustum culler (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.

1. 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. This method is applicable if the VF is not culled according to the far plane.

2. 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. One problem with this approach is the navigation speed. If the user moves very fast involving rotation and backward motion then the screen may not refresh itself on time. This is suitable for slow motion walkthroughs of the terrain.
3. As another approach, VFC depending on the deviation of the viewer location is used. Deviation based culling is very suitable for walkthroughs in which the viewer navigates through the terrain very fast. Here, we run the VFC only if the user moves a prespecified distance from the previously culled position.
4. If the terrain is large then we have to test for the far plane too. In this case, an altitude-based scheme is used for far plane distance determination. If the altitude of the viewer is at lower levels in the terrain, the far plane is brought closer to the viewer in proportion to the altitude of the viewer because it is not possible to see farther distances. This approach establishes a balance between the frustum distance and the terrain resolution.

All of the optimizations mentioned above are user-specifiable and can be switched on/off while the program is running. Besides, it is possible to see the performance differences during fly-through. In deferred VFC and VFC depending on the deviation of the viewer location, the VFC is not done for every frame.

In view frustum culling algorithm (Figure 6), 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, our scene construction is not graph based, we do not use rotation coherency as in [5].

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

Fig. 6. View frustum culling algorithm

```
Algorithm CheckFrustum;
   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)
      }
   node=sibling(node)
   return node
```

Fig. 7. Frustum checking algorithm

In frustum checking part (Figure 7), we 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 then the children of the block are checked further and its index is returned to view frustum culling function. Otherwise, we conclude it is completely inside or outside the frustum and there is no need to check the children. If the block is completely inside the frustum then the flags of all children of the checked block are cleared and marked as inside.

E. Vertex Activation

Vertex activation takes place after view frustum culling. In this module (Figure 8), the quadtree is again traversed in preorder but only traverse the nodes that are in the view

```
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-vertex_x)**2 + (eye_y-vertex_y)**2 + (eye_z+vertex_z)**2
         if (((QuadTree[node].maxact*QuadTree[node].maxact)>=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].minact*QuadTree[node].minact)>=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. 8. Vertex Activation Algorithm

frustum. In this step, the distance from the viewer position to the center of the quad block is calculated and this value is compared with the vertex activation value of the block. Since the maximum of the activation values in the block is assigned to the activation value of the center vertex, block selection decision requires only a comparison of the distance from the quad-center to the eye-point and the activation value of the block.

If the distance is smaller than the maximum activation distance, it means that the viewer is close enough to the quad block and all vertices in the quad block should be 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 can safely be activated without further investigation. If the distance falls between the minimum and maximum activation distances then we check each border vertex individually, measuring the eye-point and vertex distances and comparing with their activation distances. If the activation distance is greater than the distance measured then it means that the viewer is close enough and the vertex should be activated. While activating a vertex, it is also necessary to activate its dependents.

F. Handling Cracks

Cracks are one of the artifacts on the geometry when the two neighboring quad blocks differ in level of detail. There are several approaches to crack handling. These include drawing another triangle patch in the cracked position [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 are imposed between vertices. 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. In such a case, the dependency relation works. If any of the border vertices is activated then the neighboring quad-center vertex at the same level is activated as well. Since this newly activated quadrant will include the common border vertex in its draw-list, a triangle including the common border vertex will be drawn for the neighboring block (Figure 9). For this purpose, the dependency relationships over the data are generated and used. As shown in Figure 10, center vertices are dependent on the four corner vertices, and if they are activated then the dependents are activated accordingly. Likewise, the border vertices are dependent on the center vertices of the two neighboring blocks at the same level. If a border vertex is activated then its dependent vertices are activated as well. The dependencies of the vertices are stored in the elevation structure.

During the vertex activation process, vertex dependents are locked by calling the dependency locking procedure (Figure 11) when a vertex is activated. This procedure activates

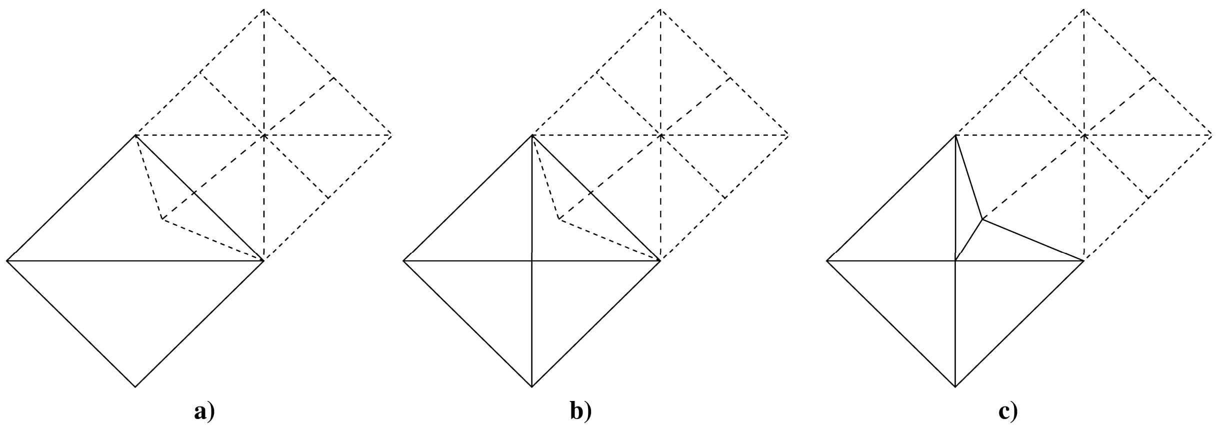

Fig. 9. Crack prevention: a) crack formation, b) activate the center vertex in the higher level block, and c) use it in the triangulation to eliminate the crack.

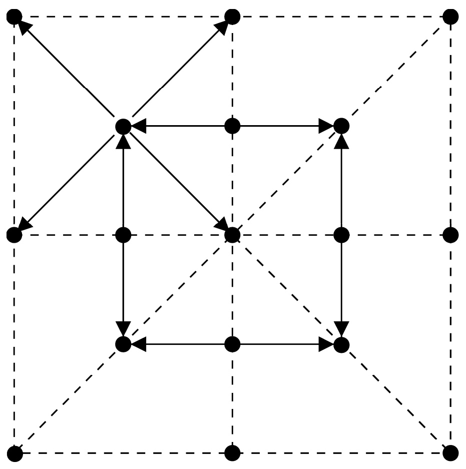

Fig. 10. Dependency relationships of center and border vertices.

```
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. 11. Locking the vertex dependents

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

Fig. 12. Notifying parents

the related vertex by turning its flag on. The dependent vertices are located sequentially in the dependent slots. A vertex can have at most four dependent vertices. If the vertex is a border vertex then the number of the dependent vertices is two. Therefore, the procedure checks if the dependent vertex is null or not; and if not, it informs its parents that the corresponding quad block is activated. It then calls itself recursively to further lock the dependents of the dependent vertex. It is important to stop the locking process if the dependent vertex has been enabled previously because this means that locking has already been done and there is no need to go further. Figure 12 gives the algorithm for notifying the parents of a node. In this algorithm, parents are notified in order not to generate a triangle towards the location of the activated child block. 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.

G. Valid Triangulation of the Mesh

In order to prevent cracks on the terrain, a valid triangulation should be maintained. Our distance-based vertex activation scheme accomplishes this by using the activation values assigned to each vertex at preprocessing phase. In this part, a *corner vertex* refers to one of the vertices on the four corners of a quad block; *border vertex* refers to the vertex between the corner vertices on the same edge of a quad block; *center vertex* refers to the vertex located at the center of the quad block; *sub-center vertex* refers to the center vertex of the sub-quad (see Figure 13). The activation values are assigned starting from the level just above the lowest level in the quadtree structure as explained below:

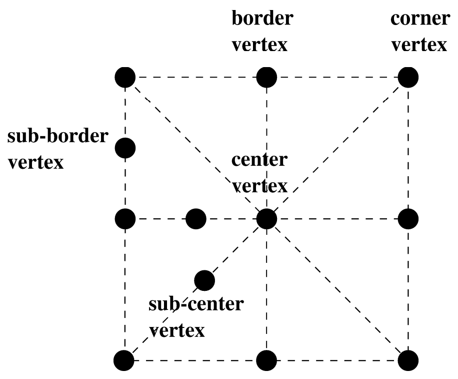

Fig. 13. Naming of the vertices in a quad block.

- Calculate the activation values for each border vertex of the quad blocks.
- Find the activation distances for the sub-center vertices by taking into account the diagonals based on the position of the sub-center in the quad block and assign the maximum of its four border activation distances and the calculated value as the sub-center vertex activation distance.

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 nodes as explained below.

- Calculate activation distance values for border vertices for each edge and assign the maximum of the sub-border vertex activation distances on the same edge and the calculated value as the border vertex activation distance.
- Find activation distances for sub-center vertices by taking into account the two corner vertices (based on its position in the larger quad block) and assign the maximum of nine values to the centers (maximum of four sub-subcenter, four sub-border and the calculated value).

This process is repeated going up until the root of the quadtree is reached. At the root, only the activation distance of the center vertex is calculated.

In the draw-list construction algorithm (Figure 15), each block is checked whether its center vertex is activated or not. If it is activated (i.e., the case when the eye is closer than the minimum activation distance of the quad block), the quadrants of the block are

```
Algorithm TriangulateBlock(node, eye);
   center_elevation=Morph(centerrow,centercol)
   // triangulate the bottom left quadrant
   if (EyeBlock[node].bottomleft) {    // bottom left quad can be triangulated
      if (EyeBlock[node].leftborder) { // vertex at left border is activated
         PutVertex(eye,centerrow,centercol,center_elevation)
         border_elevation=Morph(centerrow,minc)
         PutVertex(eye,centerrow,minc,border_elevation)  // left border
         border_elevation=Morph(minr,minc)
         PutVertex(eye,minr,minc,border_elevation)       // bottomleft corner
      }
      if (EyeBlock[node].bottomborder) { // vertex at bottom border is activated
         PutVertex(eye,centerrow,centercol,center_elevation)
         border_elevation=Morph(minr,minc)
         PutVertex(eye,minr,minc,border_elevation)       // bottomleft corner
         border_elevation=Morph(minr,centercol)
         PutVertex(eye,minr,centercol,border_elevation)  // bottom border
      }
      else
         if (EyeBlock[node].bottomright) { // bottom right quad can be triangulated
            PutVertex(eye,centerrow,centercol,center_elevation)
            border_elevation=Morph(minr,minc)
            PutVertex(eye,minr,minc,border_elevation)    // bottomleft corner
            border_elevation=Morph(minr,maxc)
            PutVertex(eye,minr,maxc,border_elevation)    // bottomright corner
         }
   }

   // triangulate the bottom right quadrant
   .....
   // triangulate the upper right  quadrant
   .....
   // triangulate the upper left quadrant
   .....
```

Fig. 14. Triangulation algorithm

checked for triangulation. 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. The triangulation algorithm, which is called from the draw-list construction algorithm, is given in Figure 14. The triangulation of a quad block resulting from an execution of the triangulation algorithm is illustrated in Figure 16.

```
Algorithm ConstructDrawList(eye);
node=0
while (nodes are not finished) {
   if (Terrain[centerrow][centercol].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
         EyeBlock[node].bottomborder=YES    // are inactive & border is active

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

Fig. 15. Construction of the draw-list

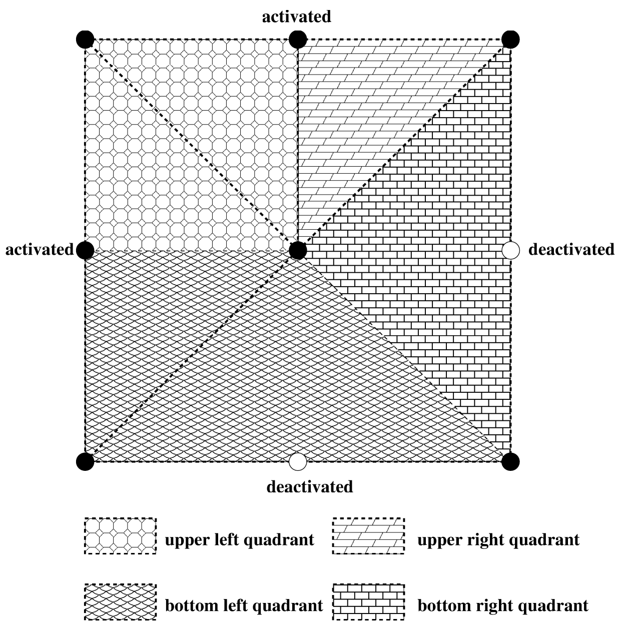

Fig. 16. A sample execution of the triangulation algorithm for a quad block.

H. 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. Blending, which is a less expensive solution for eliminating popping artifacts, cannot be used in stereo visualizations, especially in the interleaved drawing of the triangles for stereoscopic viewing.

The proposed morphing scheme works as follows. The distances between activated and deactivated states of the vertices are precalculated. A prespecified morph-segment value is used to decide at how many steps should the enabling or disabling vertex reach its new position. If this value is specified to be too large then the vertices being morphed do not reach their new positions when navigation is very fast. In order to get the morphing state of each vertex, a field is kept for each elevation on the terrain. At each frame, the morph value of a vertex is modified and the calculated distance is used as 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. The morphing algorithm is given in Figure 17.

This approach provides a uniform morphing scheme for both refinement and coarsening operations during the navigation. A positive morph value is set if the vertex is to be enabled and a negative morph value is set for a vertex to be disabled. While the viewer gets closer to the terrain, vertices are enabled and morphing is started. As soon as the viewer begins to get away from the terrain, morphing for the coarsening vertices starts.

The determination of the morph segment value is very important. This is due to the fact that the terrain does not come to its new state on time if the viewer moves very fast and the morphing lasts too long. On the other hand, if the morph segment value is too small then a popping-like appearance may result. Our experiences show that morphing should not last more than one second. The morph segment value can be determined adaptively

```
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. 17. The morphing algorithm

according to the frame rate when frame budgeting is implemented. Frame budgeting is not implemented in our work.

The morphing scheme imposes approximately ten to thirty 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 moves fast. Non-zero morph values are taken into account while drawing the next frame since taking only activated center vertices as the starting point of the triangulation is not correct for coarsening vertices. Therefore, these out-of-frustum vertices need to be cleared at each view frustum culling operation to make them ready for the next frame. The overhead comes from the traversal of the out-of-frustum nodes.

This morphing scheme is used when the culling is done for every frame. We also propose another morphing scheme that is used when the culling is not done on every frame. This approach is used for deferred VFC and the VFC based on the deviation of the viewer location. In this approach, the vertex activation and the view frustum culling processes will not run for every frame. This way, the frame rate is increased considerably and flickering in stereo visualization due to the insufficient frame rate is eliminated. This approach has another advantage in terms of the reduced number of morphing vertices. That is the morph locking flags will not be cleared when the VFC is not running. The distance between activated state and deactivated state of any vertex will be divided to morph segments and no morphing may take place without user navigation. As opposed to the continuous vertex morphing, there will be no vertex movement while the viewer is not moving.

IV. 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 (Figure 18 (a)). 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. Therefore, on-axis projection (Figure 18 (b)), 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 view frustum in half of the inter-ocular distance for each eye (Figure 18 (c)).

With the correction of the ghosting effect, the coverage of the stereoscopic area is the same as the stereoscopic area in off-axis projections. Besides, this ensures that each element in the view frustum is in stereo. Since the inter-ocular projection is very small with respect to the terrain, elimination of the ghosting effect does not cause significant processing overhead.

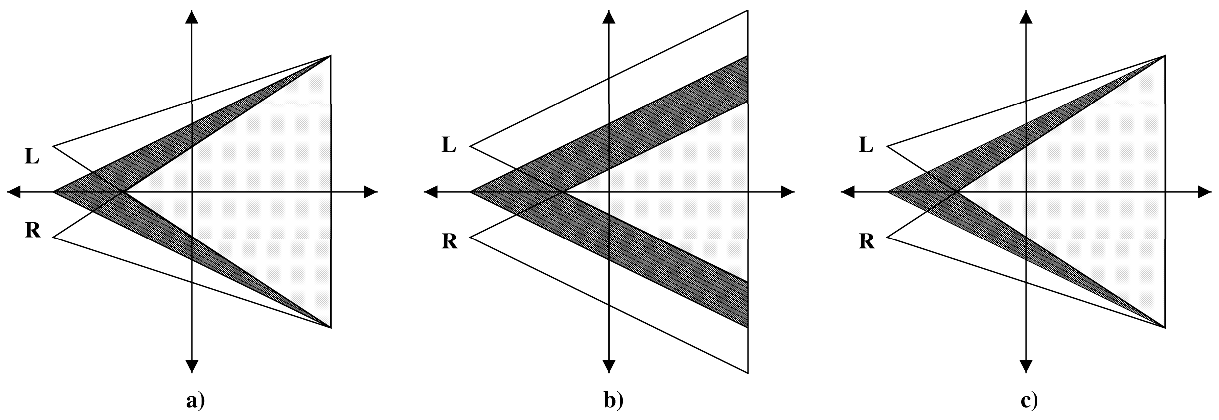

Fig. 18. Stereo projections. (a) Off-axis projection, (b) on-axis projection, (c) elimination of the ghosting effect.

A. Simultaneous Generation of Triangles

Terrain data is huge with respect to the inter-ocular 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 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. The critical point here is that necessary flags should be cleared during morphing and vertex activation processes without bringing too much overhead.

The second eye image is generated during the draw-list construction for the first eye. In the `EyeBlock` array, the flags indicating the activated vertices in the quad block are stored. For the second eye drawing interval, we do not repeat the view frustum culling and vertex activation processes. Furthermore, we do not clear any flags for morphing because the same state will apply to the left eye view as well. 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 Figure 15 as in Figure 19. Stereoscopic drawing algorithm using the SGT approach is given in Figure 21. The algorithm that decides when vertex activation and frustum culling operations should be done according to different culling schemes is given in Figure 20.

```
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. 19. Using the draw-list constructed for the right eye in generating triangles for the left eye image in the SGT algorithm

```
Algorithm Draw(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
          // the viewer is moving while dynamic morphing is off
          if (CullingNeeded) {
             ViewFrustumCulling
             ActivateVertices
          }
   ConstructDrawList(eye)
```

Fig. 20. The algorithm that calls view frustum culling and activation procedures depending on the culling scheme used

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

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

Fig. 21. Stereoscopic drawing using the SGT Algorithm

We could do several other optimizations that can be used in stereoscopic visualization. In general, many algorithms designed for speeding up second stereo pair use the mathematical characterization of the data when the eye point shifts horizontally. However, since the subject includes navigation over the data and there is not a mathematical characterization for the terrain but only the elevation data, it is not possible to shift the right eye data to the left only by its x-coordinates. We also tried to calculate the correct positions of the second eye vertices, but since the translations are implemented in hardware the approach of using the same draw-list resulted in faster draw times with respect to calculating projected vertex coordinates of the second eye. We could also use the frame buffer data drawn for right eye view for producing the left eye image by copying the right buffer to the left and shifting the data by the IOD. However, this would result in divergent parallax, which is very hard to visualize in stereo and may cause eye fatigue.

V. Performance Results

In the visualization experiments, approximately 4,000 polygons were rendered for each eye on the average at each frame. Number of polygons were almost the same in corresponding frames for the different types of visualizations. Our terrain is a part of Grand Canyon that has very sharp ridges in it, with 513x513 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. Screen shots from the monoscopic flythrough is shown in Figure 26. The number of polygons rendered during the flythrough is shown in Figure 23. The reason for sudden changes in the polygon count is that the viewer gets close to the edges and corners of the terrain. Figure 22 shows the performance comparisons 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 I. In this table, theoretical performance gain is calculated as the performance gain when the normal stereoscopic rendering speed is taken as half of monoscopic rendering speed. Practical performance gain is calculated by using the performance results obtained for normal stereoscopic visualization. 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 % over normal stereo implementation is achieved.

In Figure 24, 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 it is 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. Turning dynamic culling off becomes advantageous when the viewer does not move. In this case, the frame rate increases since the view frustum culling and vertex activation operations are not performed. 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.

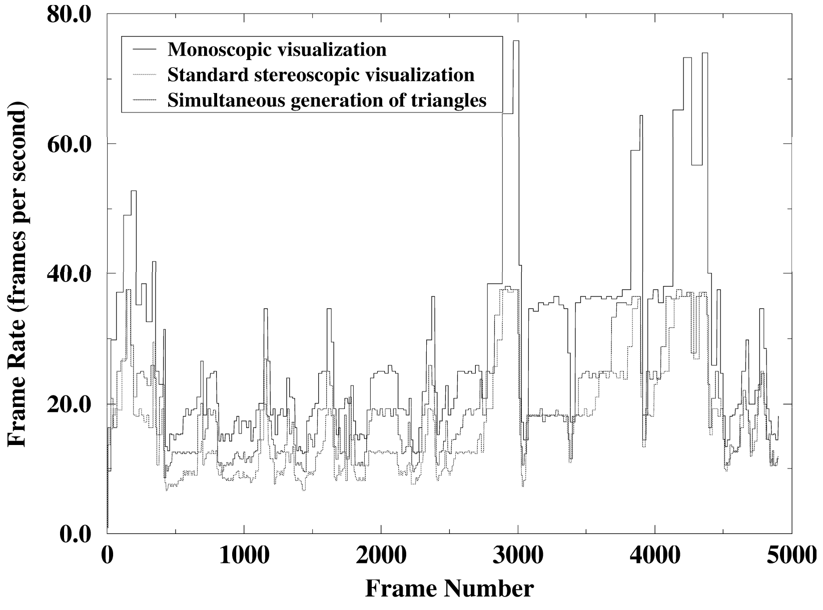

Fig. 22. Comparison of the frame rates of different types of visualizations: *monoscopic visualization* where only one image is generated for each frame; *standard stereoscopic visualization* where two images are generated for each frame; *simultaneous generation of triangles* for stereoscopic visualization where we utilize triangle list for one eye to generate the triangles for the other eye.

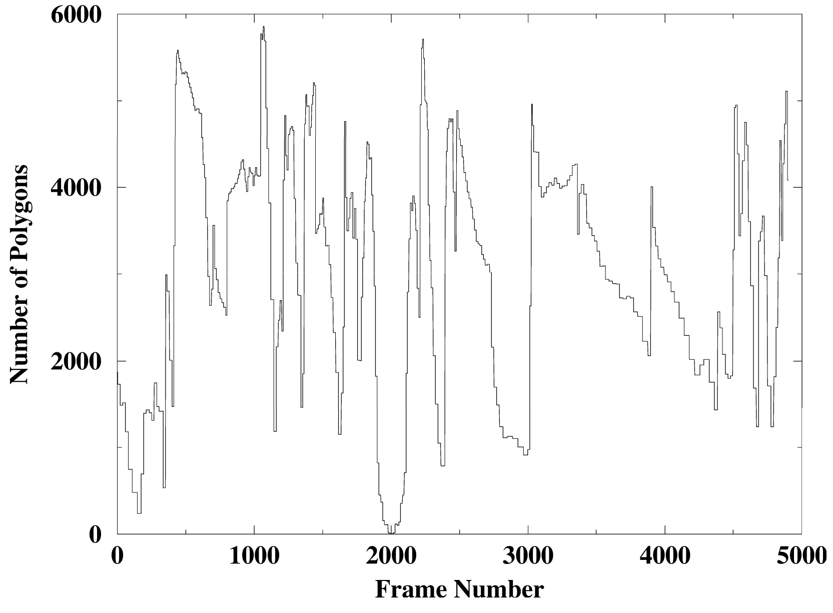

Fig. 23. Number of polygons for the experimental visualization

TABLE I
Performance Results

Visualization Method	Morph	Dynamic Culling	Deferred VFC	Deviation VFC	Average FPS	Theoretical Performance Gain	Practical Performance Gain
Monoscopic	OFF	OFF	OFF	OFF	30.66		
SGT	OFF	OFF	OFF	OFF	21.83	42.41	42.61
Normal Stereo	OFF	OFF	OFF	OFF	15.31		
Monoscopic	OFF	OFF	ON	OFF	44.88		
SGT	OFF	OFF	ON	OFF	26.83	19.56	12.15
Normal Stereo	OFF	OFF	ON	OFF	23.92		
Monoscopic	OFF	ON	OFF	OFF	30.85		
SGT	OFF	ON	OFF	OFF	21.90	41.99	43.27
Normal Stereo	OFF	ON	OFF	OFF	15.29		
Monoscopic	ON	OFF	OFF	OFF	22.56		
SGT	ON	OFF	OFF	OFF	16.42	45.58	27.61
Normal Stereo	ON	OFF	OFF	OFF	12.87		
Monoscopic	ON	OFF	ON	OFF	31.38		
SGT	ON	OFF	ON	OFF	19.13	21.91	27.54
Normal Stereo	ON	OFF	ON	OFF	15.00		
Monoscopic	ON	ON	OFF	OFF	22.57		
SGT	ON	ON	OFF	OFF	16.43	45.61	37.29
Normal Stereo	ON	ON	OFF	OFF	11.97		
Monoscopic	OFF	OFF	OFF	ON	43.25		
SGT	OFF	OFF	OFF	ON	27.48	27.07	8.54
Normal Stereo	OFF	OFF	OFF	ON	25.32		
Monoscopic	ON	OFF	OFF	ON	31.91		
SGT	ON	OFF	OFF	ON	20.70	29.73	15.25
Normal Stereo	ON	OFF	OFF	ON	17.96		

In Figure 25, 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.

VI. 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 loose 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

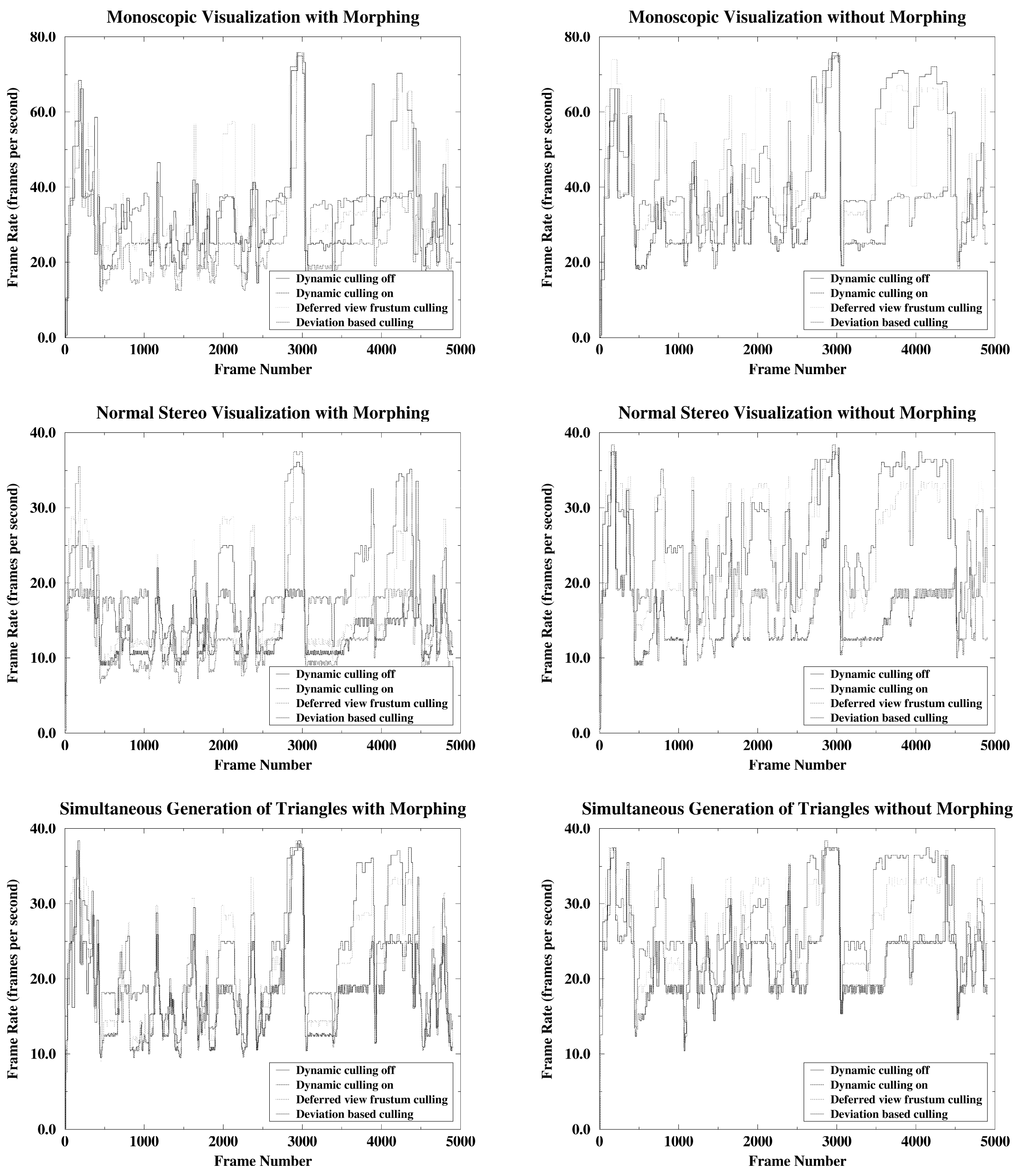

Fig. 24. Comparison of the frame rates for each type of visualization with different morphing/culling options: (a) *monoscopic visualization* with morphing; (b) *monoscopic visualization* without morphing; (c) *standard stereoscopic visualization* with morphing; (d) *standard stereoscopic visualization* without morphing; (e) *simultaneous generation of triangles* with morphing; (f) *simultaneous generation of triangles* without morphing.

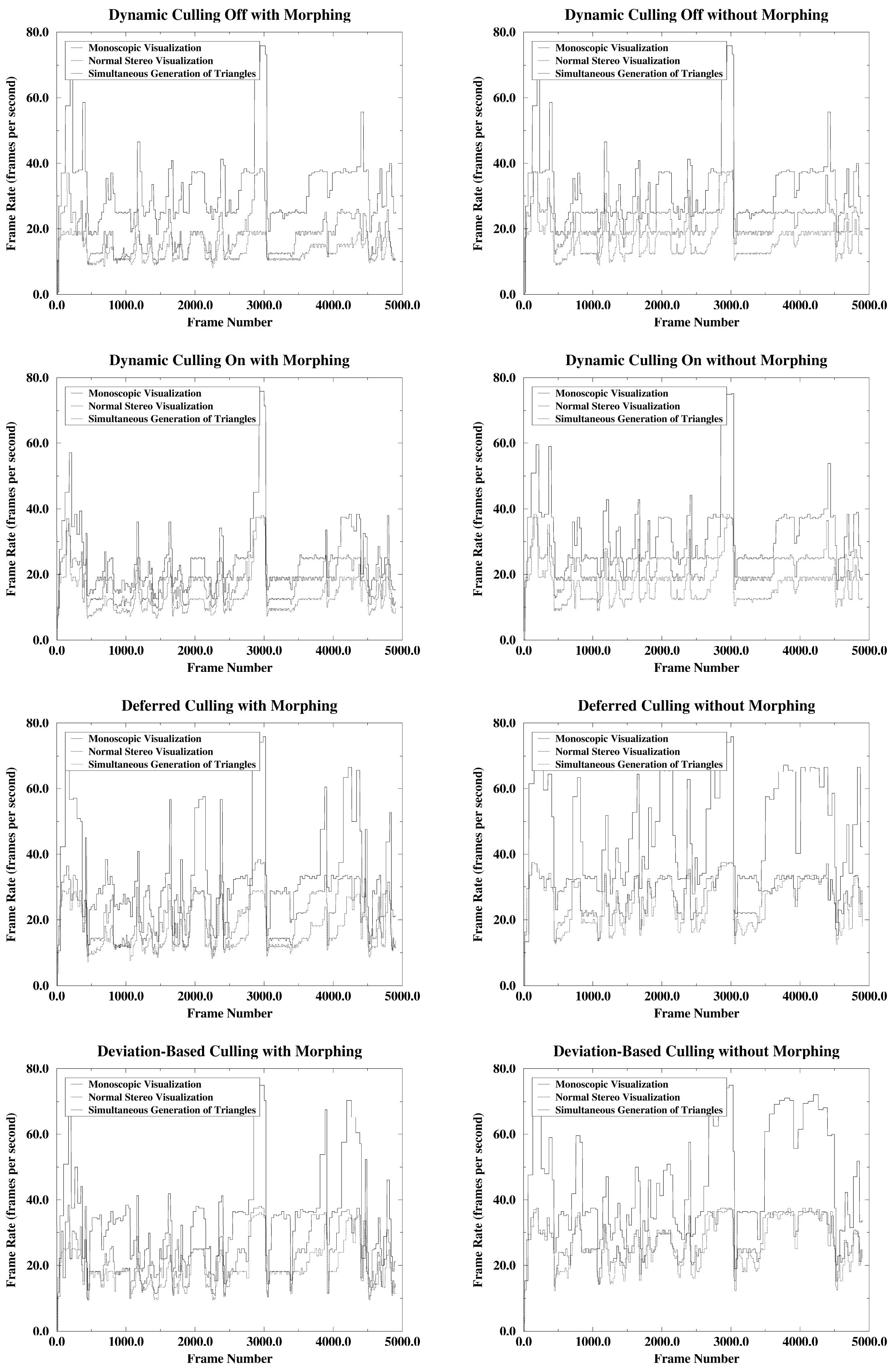

Fig. 25. Comparison of different culling schemes for each type of visualization: (a) *dynamic culling off* with morphing; (b) *dynamic culling off* without morphing; (c) *dynamic culling on* with morphing; (d) *dynamic culling on* without morphing; (e) *deferred culling* with morphing; (f) *deferred culling* without morphing; (g) *deviation-based culling* with morphing; (h) *deviation-based culling* without morphing.

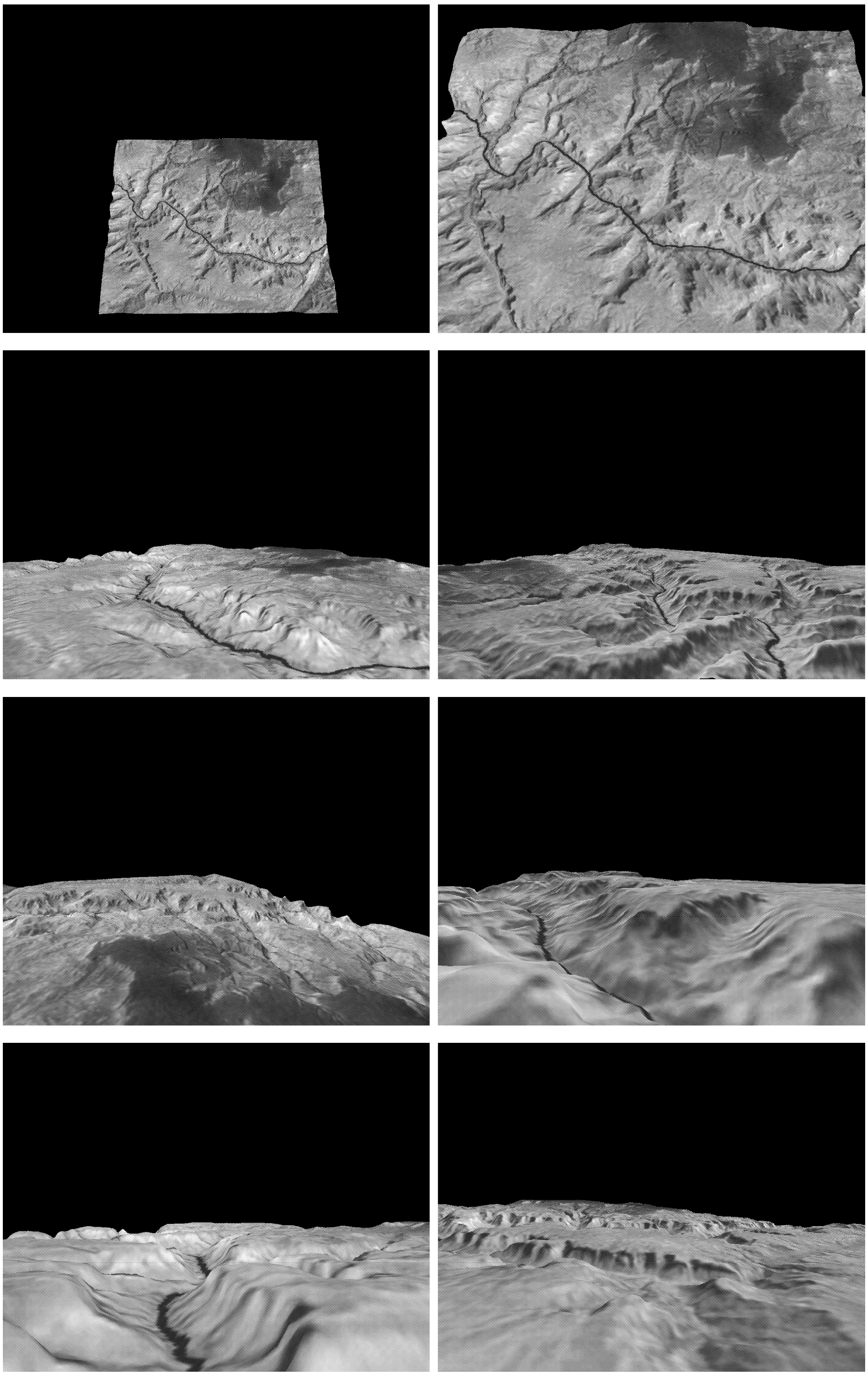

Fig. 26. Still frames from a walkthrough

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 % faster than drawing the two images separately and a smooth stereoscopic visualization can be achieved at interactive frame rates using continuous multi-resolution representation of height fields.

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," in *ACM Computer Graphics (Proc. of SIGGRAPH'96)*, 1996, pp. 109–118.

[2] R. Pajarola, "Large scale terrain visualization using the restricted quadtree triangulation," in *Proc. of IEEE Visualization'98*, 1998, pp. 19–26.

[3] H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *Proc. of IEEE'98*, 1998, pp. 35–42.

[4] P. Lindstrom, "Level-of-detail management for real-time rendering of phototextured terrain," Tech. Rep. GIT-GVU-95-06, Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, GA, USA, 1995.

[5] U. Assarsson and T. Möller, "Optimized view frustum culling algorithms for bounding boxes," *Journal of Graphics Tools*, vol. 5, no. 1, pp. 9–22, 2000.

[6] D. Bartz, M. Meisner, and T. Hüttner, "OpenGL-assisted occlusion culling for 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," in *Proc. of SPIE 1256, Stereoscopic Display and Applications I*, 1990, pp. 298–306.

[10] S.J. Adelson and L.F. Hodges, "Stereoscopic ray tracing," *the Visual Computer*, vol. 10, no. 3, pp. 127–144, 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," in *Proc. of Symposium on Volume Visualization*, 1994, pp. 3–9.

[13] H. Taosong and A. Kaufman, “Fast stereo volume rendering,” in *Proc. of IEEE Visualization’96*, 1996, pp. 49–56.

[14] R. Hubbold, D. Hancock, and C. Moore, “Stereoscopic volume rendering,” in *Proc. Visualization in Scientific Computing’98*, 1998, pp. 105–115.

[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,” in *Proc. of IEEE Visualization’99*, 1999, pp. 285–290.

Acknowledgement

This project is supported by Turkish Scientific and Technical Research Council (TÜBİTAK) with grant no. 198E018. Grand Canyon Data is obtained from The United States Geological Survey (USGS), with processing by Chad McCabe of the Microsoft Geography Product Unit. We would like to thank Uğur Doğrusöz and Murat Akbay for their valuable comments.