FAST STEREOSCOPIC VIEW-DEPENDENT VISUALIZATION OF TERRAIN HEIGHT FIELDS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

by

Türker Yılmaz

June, 2001

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Güdükbay (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof.Dr. Bülent Özgüç

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Uğur Doğrusöz

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

FAST STEREOSCOPIC VIEW-DEPENDENT VISUALIZATION OF TERRAIN HEIGHT FIELDS

Türker Yılmaz
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. Uğur Güdükbay
September, 2001

Visualization of large geometric environments has always been an important problem of computer graphics. In this thesis, 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.

ÖZET

ARAZİ YÜKSEKLİK VERİLERİNİN STEREOSKOPİK BAKIŞA GÖRE HIZLI GÖRÜNTÜLENMESİ

Türker Yılmaz
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Uğur Güdükbay
Haziran, 2001

Geniş geometrik ortamların etkileşimli olarak görüntülenmesi bilgisayar grafikleri ile uğraşanlar için devamlı olarak bir problem olmuştur. Bu çalışmada geniş arazi verilerinin stereoskopik ve çok çözünürlüklü olarak görüntülenmesi için bir yapı sunulmuştur. Arazi verilerinin temsili için kullanılan dörtlü ağaç yapısı, arazinin bakışa göre değişken olarak çok çözünürlüklü modelinin elde edilmesi için sorgulanmıştır. Derinlik bilgisini kaybetmeyen mesafe temelli açısal sadeleştirme kriteri ile birlikte, stereoskopik olarak görüntülemeyi hızlandıran "Üçgenlerin Aynı Anda Üretilmesi" adı verilen yeni bir yöntem geliştirilmiştir. Bu yöntemde, birinci göz görüntüsü için çizilecek üçgenler üretildikten sonra, ikinci göz görüntüsünü elde etmek için çeşitli dönüşümlere tabi tutulmakta, böylece işlemci zamanını alan üçgenlerin bakış piramidinin dışında kalanlarının ayıklanması ve köşelerin aktif hale getirilmesi işlemleri iki göz görüntüsü için bir defa yapılmaktadır. Arazi üzerindeki kırılma problemi bağımlılık ilişkileri kullanılarak çözülmüştür. Farklı çözünürlükler arasındaki atlama etkileri başkalaşım yöntemi kullanılarak giderilmiştir. Önerilen algoritmalar bir görüntüleme sistemi haline getirilmiş ve kişisel bilgisayarlar ile grafik iş istasyonlarında uygulanmıştır. Performans ölçümleri sonucunda, arazinin düzgün ve çok çözünürlüklü stereoskopik görüntüleri, her iki gözü ayrı ayrı üretmeye göre % 45 daha hızlı olarak elde edilmiştir.

Anahtar Sözcükler: Stereoskopik görüntüleme, arazi yükseklik verisi, çok çözünürlüklü modelleme, dörtlü ağaçlar.

Türk Silahlı Kuvvetleri'ne

ve

Aileme.

ACKNOWLEDGMENTS

At the end of this study, I am very grateful to my supervisor, Assist. Prof. Dr. Uğur Güdükbay, for his invaluable support, guidance and motivation.

I also would like to thank my thesis committee members Prof. Dr. Bülent Özgüç and Assist. Prof. Dr. Uğur Doğrusöz for their valuable comments to improve this thesis. I would like to thank Prof. Dr. Bülent Özgüç for his invaluable guidance and imagination improving approach during the lectures. I am very grateful to Assist. Prof. Dr. Uğur Doğrusöz for his comments on the development of the thesis.

I would like to mention some people who helped me during this study in different ways. I would like to thank Captain Murat Akbay for his invaluable comments on earlier drafts of this thesis. I would also like to thank Second Colonel Ziya İpekkan for making Modeling and Simulation Laboratory available for me to make some performance tests on graphics workstations. I would like to thank to Major Hakan Maraş for his invaluable support on digital terrain elevation data formats. I would also like to thank to SGI İstanbul, for providing me the liquid crystal shutter glasses.

I really would like to thank my brother, Lawyer Tibet Yılmaz for helping me to have a well equipped personal computer to develop the code on.

Finally, I cannot forget my love and my wife, Canan Yılmaz. I would like to thank her for invaluable moral support and love.

This thesis was partially supported by TÜBİTAK (Turkish Scientific and Technical Research Council) under Grant 198E018. Grand Canyon Data is obtained from The United States Geological Survey (USGS), with processing by Chad McCabe of the Microsoft Geography Product Unit.

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

LOD	: Level of Detail
TIN	: Triangular Irregular Network
LCS	: Liquid Crystal Shutter
VR	: Virtual Reality
fps	: Frames per Second
quadtree	: A tree structure holding the data in a quadruple way
N	: Number of nodes at the bottom level of the quadtree
n	: Number of vertices in a line of the terrain
L	: Number of levels in the quadtree
T_N	: Number of nodes in the quadtree
d	: Distance between the vertex and the viewer
δ	: Elevation difference between the border vertex and the corners
τ	: Angular threshold
A'	: Tangent of the angular threshold
v_{act}	: Vertex activation distance
$(\min_{act})$	: The distance, that at least one vertex should be activated
$(\max_{act})$	: The distance, that all vertices should be activated in the block
VFC	: View Frustum Culling
$deferred$ VFC	: VFC done at predefined intervals
corner vertex	: One of the vertices on the four corners of a quad-block
border vertex	: One of the vertices between the corner vertices of a quad-block
center vertex	: Vertex located at the center of the quad-block
sub-center	: Center of a child of the corresponding quad-block
sub-border	: A border vertex of the child of the corresponding quad-block
sub-sub-center	: A center vertex at the two levels below of the quad-block
$activevertex$	: Vertex that is selected to be a part of a triangulation
IOD	: Inter-Ocular Distance
parallax	: Distance between the stereo pairs on the screen
SGT	: Simultaneous Generation of Triangles
Normal Stereo	: Stereoscopic visualization done by a naive approach

Chapter 1

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 counterparts. Therefore, the surface has to be approximated up to a certain threshold.

1.1 Simplification Criteria

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. Besides, the correctness degrades at the peripheries although the human eye compensates for this. However, this degradation should be prevented and the simplification criteria should be

defined adaptively to real life situations, especially in flight simulator like applications because pilots use peripheral views when landing and taking off, while they focus their attention towards the center of the view. To solve these problems, we propose a distance-based angular error threshold criterion that preserves depth information of the terrain data during the simplification process.

1.2 Multiresolution Representation

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. While doing this process, the artifacts that can occur on the terrain should be minimized as much as possible.

1.3 Stereoscopic Visualization

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.

1.4 Contributions

The contributions of the thesis 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 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.
- A morphing technique that works in the same manner for both refining and coarsening operations while visualizing the terrain data.

1.5 Outline of the Thesis

In the next chapter, 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 Chapter 3. A short information about stereoscopic projections and algorithms that we propose to speed-up the generation of second eye image for stereoscopic visualization are explained in Chapter 4. Chapter 5 discusses the performance of the proposed algorithms in terms of processing speed and quality of the visualizations. Conclusions are given in Chapter 6.

In Appendix A, we describe the user interface of the developed system. In Appendix B, we give brief overview about stereoscopic visualization formats. In Appendix C, a short introduction to OpenGL is given. In Appendix D, description about the user interface library we used is given. Finally, in Appendix E, the unsorted graphical results of our proposed algorithms are presented.

Chapter 2

Related Work

In this chapter we give an overview of the previous work that has been produced in this area.

2.1 View-dependent Visualization of Terrain Height Fields

In [12], 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 [17], 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 [9], 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 [11]. 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 [4], 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 [5], 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 back face 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]. Most of the applications support stereoscopic display by completely generating the two images for the left and right eye views separately. Parallel processing is very suitable for this type of stereoscopic visualization. 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 eye-point, like the scan-lines to which an object project as stated in [7]. In [6], 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 [3]. In [1], a non-ray-tracing algorithm is described to speed up the second eye image generation for polygon filling, hidden surface elimination and clipping. In [2], methods that take advantage of the coherence between the two halves of a stereo pair for ray traced volume rendering are presented. In [22], the authors present an algorithm using segment composition and linearly-interpolated re-projection for fast direct volume rendering. Hubbold et al. [10] propose extensions of a direct volume renderer for use with an autostereoscopic display in radiotherapy planning. Since the terrain data do not have any mathematical characterization, mentioned algorithms cannot be adapted easily to stereoscopic terrain visualization.

Chapter 3

Multiresolution Modeling

In this chapter we describe the methods we applied to construct multiresolution models of the terrain. At first we give the data structures we used in the implementation. Next we describe the terrain representation. In the later sections we describe how we achieved the properties of a proper terrain visualization system by dealing with simplification functions, crack prevention and morphing to prevent popping artifacts.

3.1 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 precalculated 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 3.1).

In the `quad` structure, minimum and maximum elevations and minimum and maximum activation distances for a quad block are stored. Flags indicating

```
struct elevation {
  short int elevation;       // elevation in meters
  int       activation;      // vertex activation distance
  int       dependent[4][3]; // array of dependents
  float     morphdistance;   // distance for morphed vertex
  char      morph;           // level of morphing
  unsigned  activestate:1;   // keeps activation locks
  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;  // min and max enabling distance
  int       rcenter, ccenter;// indices of the center vertex
  char      activated;       // if the cell is activated
  char      culled;          // if the cell is previously culled
  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];
```

Figure 3.1: Data structures

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 [19].

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.

3.2 Terrain Representation

The quadtree structure is represented as a one-dimensional array (Figure 3.2). 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 for pointer manipulation. The numbering scheme for the quadtree structure when it is stored in a one-dimensional array is illustrated in Figure 3.3. The root is labeled as 0 and the rest is numbered recursively in counter-clockwise direction.

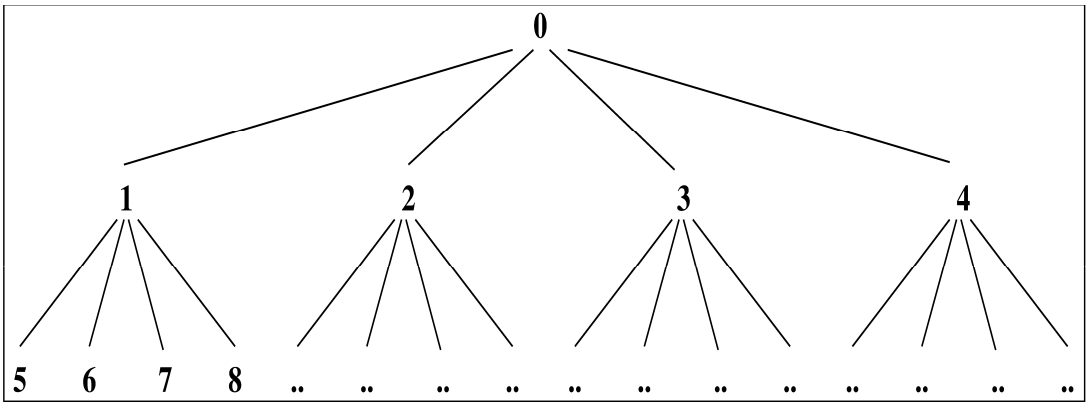


Figure 3.2: Quadtree structure

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

Figure 3.3: Numbering scheme for the quad blocks

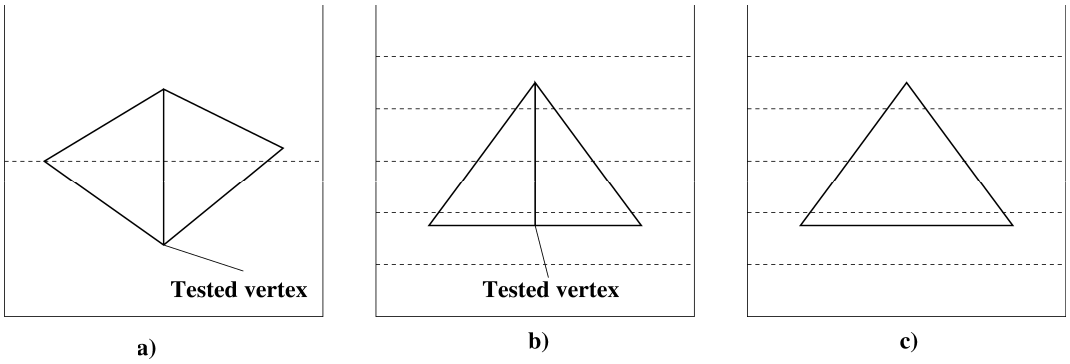


Figure 3.4: Screen-space error metric. (a) Side view of a quad block. (b) Top view of the same block. (c) Edge removal by screen-space error based algorithm.

3.3 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 3.4. 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 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.

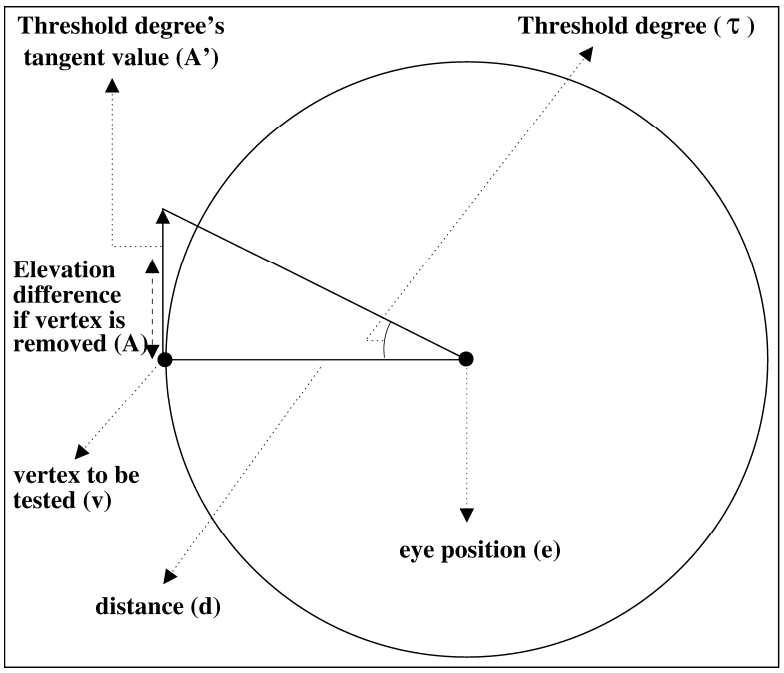


Figure 3.5: Angular error threshold representation for vertex removal.

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 3.5 illustrates our angular error metric for evaluation of a vertex for removal.

The distance from the eye position to the vertex is

$$d = \sqrt{(e_x - v_x)^2 + (e_y - v_y)^2 + (e_z - v_z)^2}. \tag{3.1}$$

The distance between the original and removed positions of the vertex (Figure 3.6) is

$$\delta \;=\; \left| v_z - \left(\frac{leftcorner_z + rightcorner_z}{2} \right) \right|. \tag{3.2}$$

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

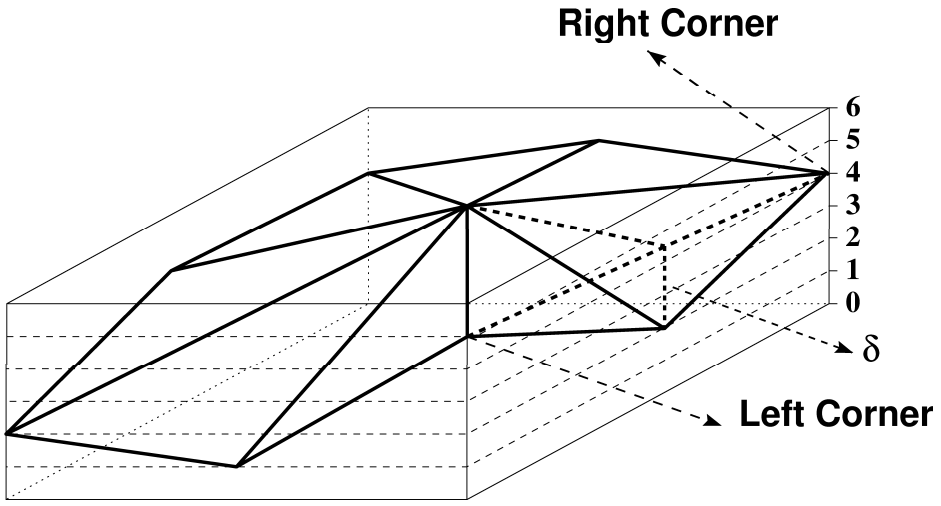


Figure 3.6: The distance δ between original and removed positions of the tested vertex.

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). So

$$\delta = A' \tag{3.3}$$

$$\delta = tan(\tau)\ d \tag{3.4}$$

$$d = \frac{\delta}{tan(\tau)} \tag{3.5}$$

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.

3.4 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.

- 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.
- 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.

- 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.

- 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. The algorithm that controls the VFC operation is given in Figure 4.8, in Chapter 4.

In view frustum culling algorithm (Figure 3.7), 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 [4].

In frustum checking part (Figure 3.8), 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

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

Figure 3.7: 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
```

Figure 3.8: Frustum checking algorithm

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.

3.5 Vertex Activation

Vertex activation takes place after view frustum culling. In this module (Figure 3.9), the quadtree is again traversed in preorder but only do we 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 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.

3.6 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.

```
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)
}
```

Figure 3.9: Vertex Activation Algorithm

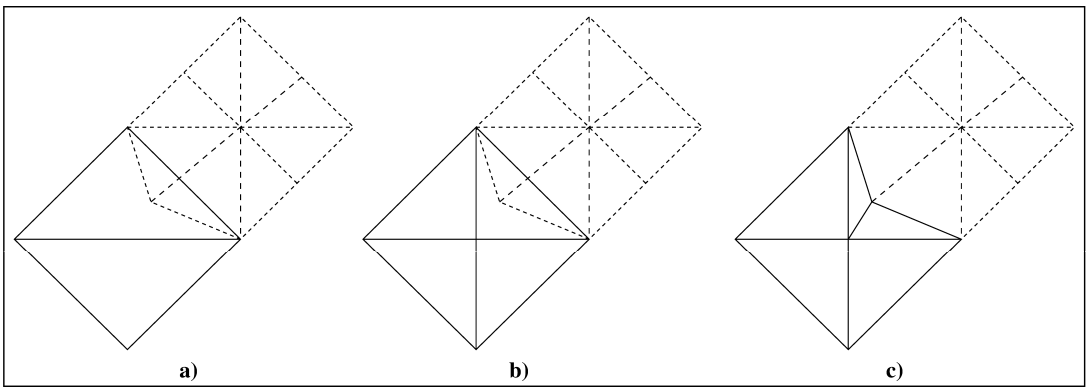


Figure 3.10: Crack prevention: a) crack formation, b) activate the center vertex in the higher level block, and c) use it in the triangulation to eliminate the crack.

These include drawing another triangle patch in the cracked position [11], triangulation of the gapped position [16], or not allowing crack formation by using the dependency relations [17].

In order to prevent cracks without causing discontinuation, 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 3.10). For this purpose, the dependency relationships over the data are generated and used. As shown in Figure 3.11, 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 3.13) when a vertex is activated.

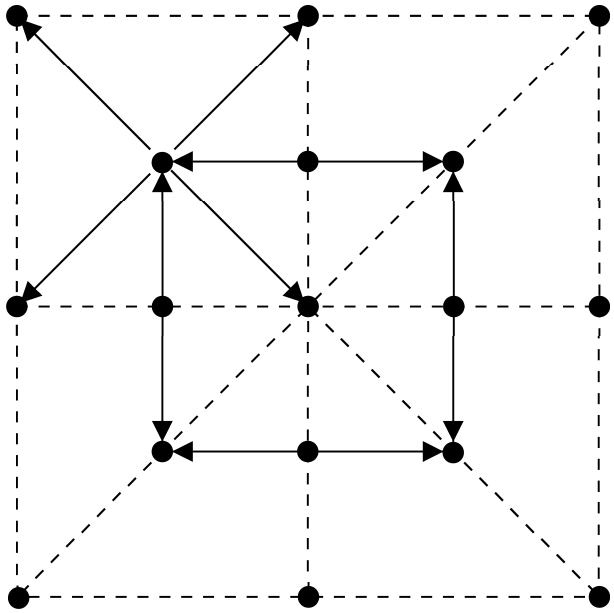


Figure 3.11: Dependency relationships of center and border vertices.

```
Algorithm NotifyParents(node);
   childno=node-child(parent(node)) // find the quadrant #
   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))
   }
```

Figure 3.12: Notifying parents

```
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
       }
   }
```

Figure 3.13: Locking the vertex dependents

This procedure activates 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 3.12 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.

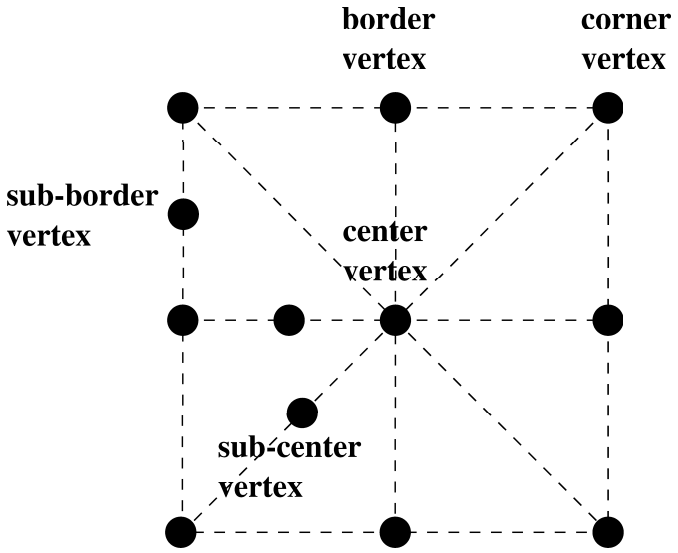


Figure 3.14: Naming of the vertices in a quad block.

3.7 Valid Triangulation of the Mesh

3.7.1 Triangle Construction

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 3.14). The activation values are assigned starting from the level just above the lowest level in the quadtree structure as explained below:

- Calculate the activation values for each border vertex of the quad blocks (Figure 3.15).
- 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 (Figure 3.16) and assign the maximum of its four border activation

distances and the calculated value as the sub-center vertex activation distance (Figure 3.17).

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.

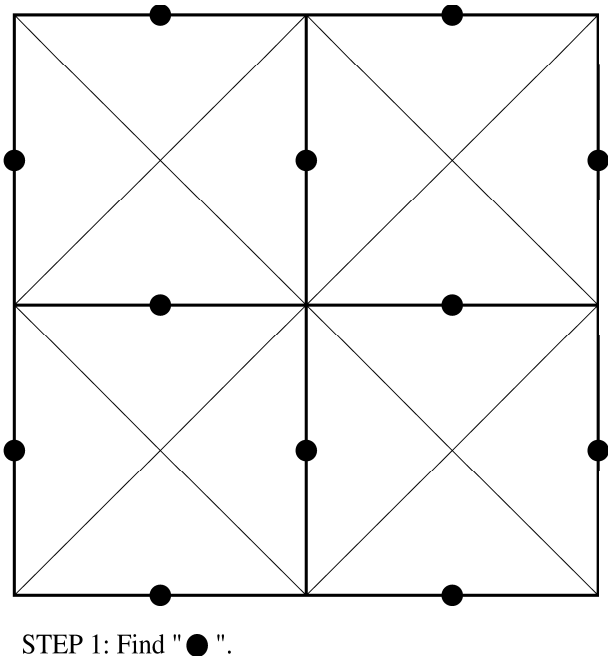


Figure 3.15: The first step of the vertex activation value assignment.

- Calculate activation distance values for border vertices for each edge (Figure 3.18) 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 (Figure 3.19).
- Find activation distances for sub-center vertices by taking into account the two corner vertices (based on its position in the larger quad block) (Figure 3.20) and assign the maximum of nine values to the centers (maximum of four sub-subcenter, four sub-border and the calculated value)(Figure 3.21).

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.

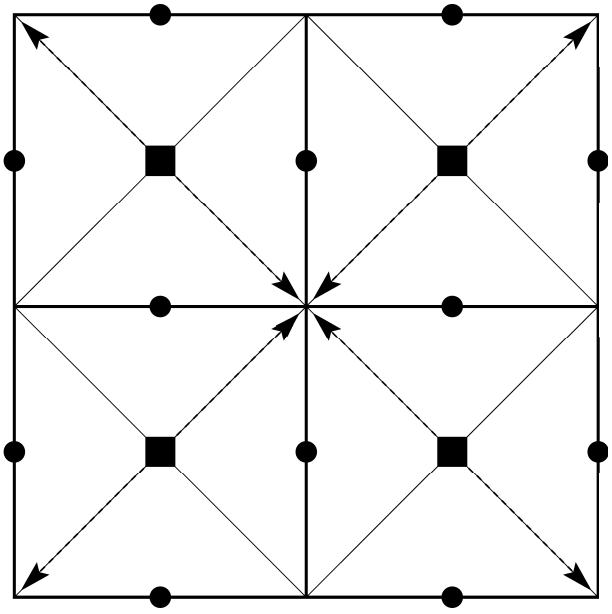


Figure 3.16: The first phase of the second step of vertex activation values assignment: Calculate activation distances for subcenter vertices.

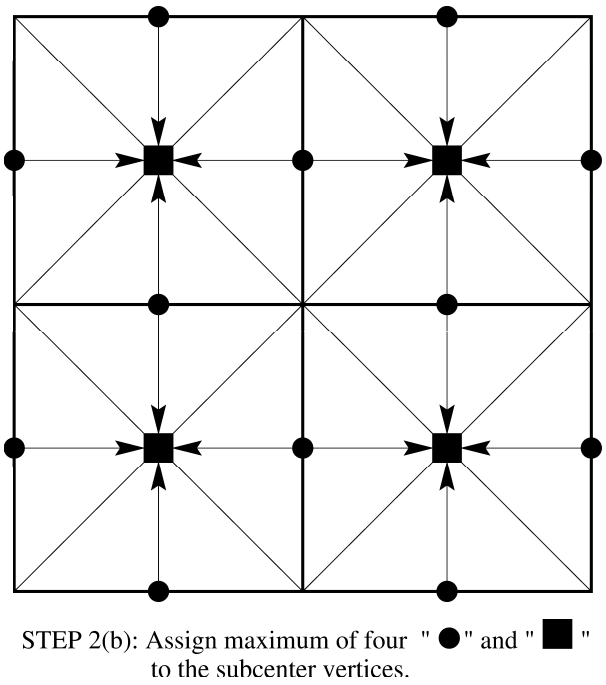
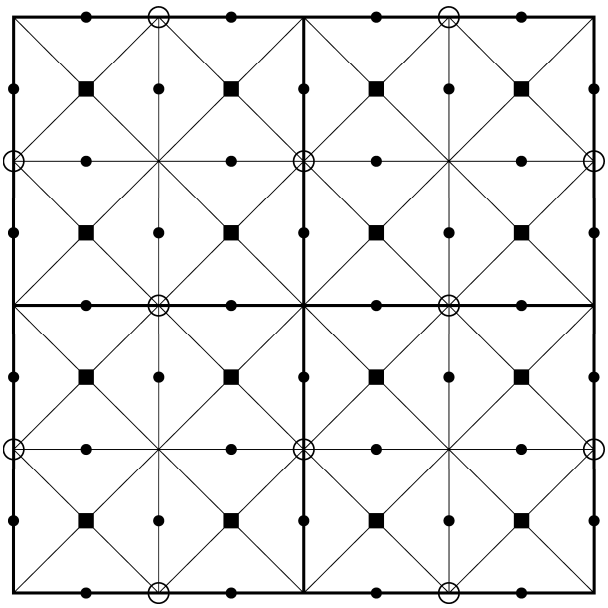
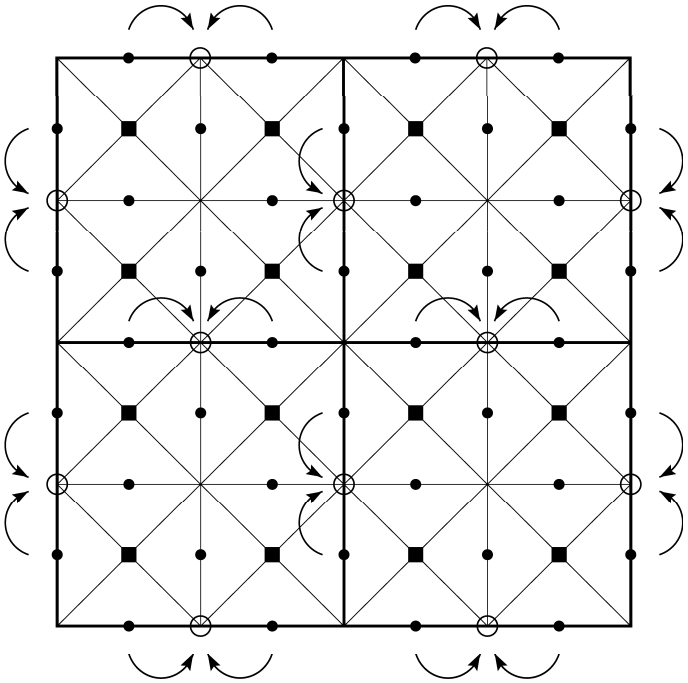


Figure 3.17: The second phase of the second step of vertex activation values assignment: Maximum operations on the center vertices.



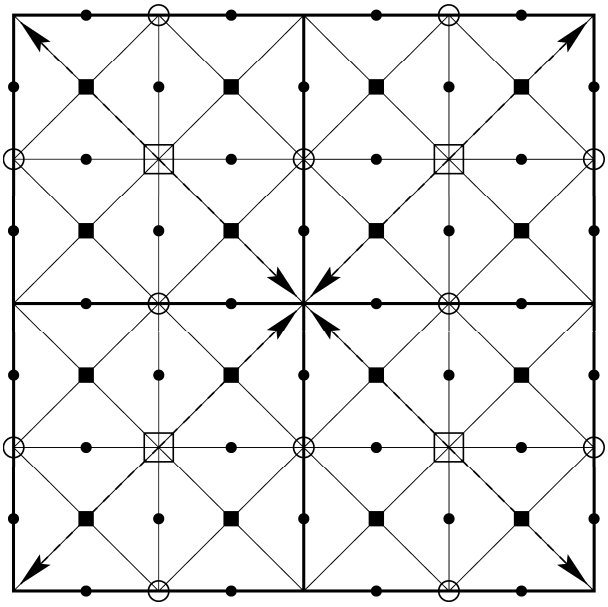
STEP 3(a): Find " ○ ".

Figure 3.18: The first phase of the third step of vertex activation values assignment: Calculate activation distances for border vertices.



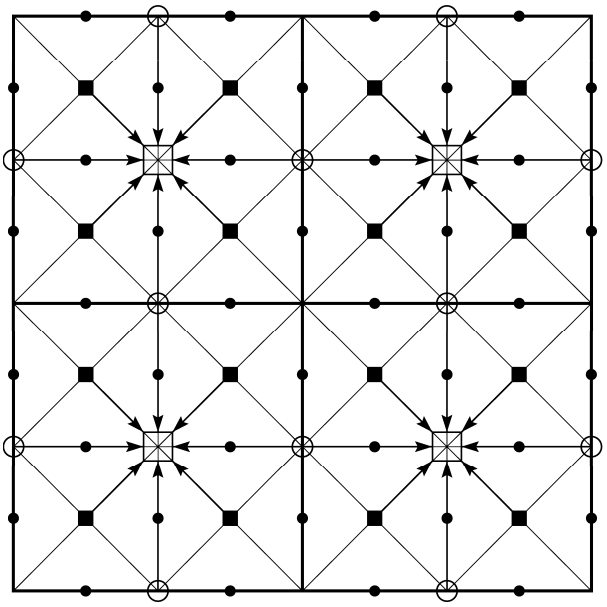
STEP 3(b): Assign max of the border values to " ○".

Figure 3.19: The second phase of the third step of vertex activation values assignment: Maximum operations on the border vertices.



STEP 4(a): Find " □ ".

Figure 3.20: The first phase of the last step of vertex activation values assignment: Calculate activation distances for subcenter vertices.



STEP 4(b): Assign maximum of nine values to center vertices.

Figure 3.21: The second phase of the last step of vertex activation values assignment: Maximum operations on the sub-center vertices.

```
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 act.
         if (canIdrawupperleft(node)==NO)   // above subquad act.
            EyeBlock[node].leftborder=YES
         EyeBlock[node].bottomleft=YES
         if (canIdrawbottomright(node)==NO) // next subquad act.
            EyeBlock[node].bottomborder=YES
      }
      if (canIdrawbottomborder(node)==YES)  // neighbor quadrant
         EyeBlock[node].bottomborder=YES    // centers are inact.
                                            // & border is act.
      // process bottom right quadrant
      .....
      // process upper right quadrant
      .....
      // process upper left quadrant
      .....
      TriangulateBlock(node,eye)
      node=child(node)
   }
   else
      node=sibling(node)
}
```

Figure 3.22: Construction of the draw-list

```
Algorithm TriangulateBlock(node, eye);
   center_elevation=Morph(centerrow,centercol)
   // triangulate the bottom left quadrant
   if (EyeBlock[node].bottomleft) {    // it can be triangulated
      if (EyeBlock[node].leftborder) { // left border activated
         PutVertex(eye,centerrow,centercol,center_elevation)
         border_elevation=Morph(centerrow,minc)
         PutVertex(eye,centerrow,minc,border_elevation)//l.border
         border_elevation=Morph(minr,minc)
         PutVertex(eye,minr,minc,border_elevation)     //b.l.cor.
      }
      if (EyeBlock[node].bottomborder) { // bottom border act.
         PutVertex(eye,centerrow,centercol,center_elevation)
         border_elevation=Morph(minr,minc)
         PutVertex(eye,minr,minc,border_elevation)     //b.l.cor.
         border_elevation=Morph(minr,centercol)
         PutVertex(eye,minr,centercol,border_elevation)//b.bor.
      }
      else
         if (EyeBlock[node].bottomright) { // can be triangulated
            PutVertex(eye,centerrow,centercol,center_elevation)
            border_elevation=Morph(minr,minc)
            PutVertex(eye,minr,minc,border_elevation)  //b.l.
            border_elevation=Morph(minr,maxc)
            PutVertex(eye,minr,maxc,border_elevation)  //b.r.
         }
   }

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

Figure 3.23: Triangulation algorithm

3.7.2 Triangle Drawing

In the draw-list construction algorithm (Figure 3.22), 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 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 3.23. A sample triangulation of a quad block resulting from an execution of the triangulation algorithm is illustrated in Figure 3.24.

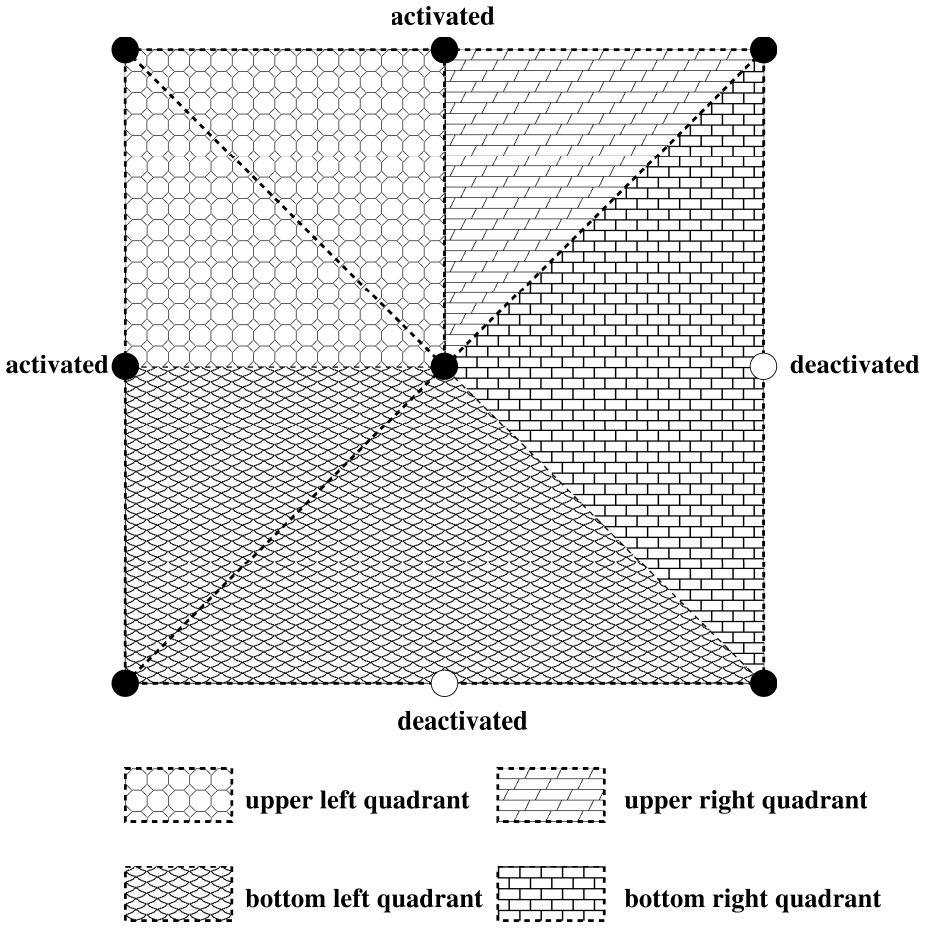


Figure 3.24: A sample execution of the triangulation algorithm for a quad block.

3.8 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.

The proposed morphing scheme works as follows. The distances between active and deactive 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 3.25.

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

```
Algorithm Morph(cr, cc)
  if (MORPHING==ON) {
     emorph=Terrain[cr][cc].morph
     // if vertex is disabled, 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) { // is not locked
              (Terrain[cr][cc].morph)++
              Terrain[cr][cc].morphlock=YES
           }
        }
        else {          // the vertex is refining
           morphed_el=morphed_el-morphdist
           if (!Terrain[cr][cc].morphlock) { // is not locked
              (Terrain[cr][cc].morph)--
              Terrain[cr][cc].morphlock=YES
           }
        }
     }
  return morphed_el
  }
```

Figure 3.25: The morphing algorithm

second. The morph segment value can be determined adaptively according to the frame rate when frame budgeting is implemented. Frame budgeting is not implemented in our work because our main purpose is to see how fast we can generate the second image needed in stereoscopic visualization.

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.

Chapter 4

Stereoscopic Visualization

In this chapter we give information about stereoscopic visualization. Next we describe the projection system that we chose for the implementation, and explain our approach to the generation of the stereo pairs.

4.1 About Stereoscopy

4.1.1 Stereoscopic Image Perception

Up to 19^{th} century, mankind was not aware that there was a separable binocular depth sense until relatively recently. Through the ages, people like Euclid and Leonardo understood that we see different images of the world with each eye. It was Wheatstone [25] who explained to the world, that there is a depth sense that is named as *stereopsis*, produced by retinal disparity. Wheatstone explained that the mind fuses the two planar retinal images into one with stereopsis (*solid seeing*).

A stereoscopic display is an optical system whose final component is the human brain. It functions by presenting the mind with the same kind of left and right views that the person sees in the real world.

4.1.2 Retinal Disparity

In order to explain the presence of the retinal disparity one can try this experiment: hold your finger in front of your face. When you look at your finger and try to see the finger in detail, your eyes start to converge on your finger. That is, the optical axes of both eyes cross on the finger. There are sets of muscles which move the eyes to accomplish this by placing the images of the finger on each fovea, or central portion, of each retina. If you continue to converge your eyes on your finger, paying attention to the background, you will notice that the background appears to be double. Now try to focus on the background and you will see that when your see the background in detail, your finger, because of the retinal disparity, will now appear to be double. If we could take the images that are on your left and right retina and somehow superimpose them as if they were a slide, you would see two *almost* overlapping images - left and right perspective viewpoints - which have what physiologists call disparity. Disparity is the distance, in horizontal direction, between the corresponding left and right image points of the superimposed retinal images. The corresponding points of the retinal images of an object on which the eyes are converged, will have zero disparity.

Retinal disparity is caused by the fact that each of our eyes sees the world from a different point of view. On the average the eyes are two and a half inches or 64 millimeters apart for adults [21]. The disparity is fused by the brain into a single image of the visual world. The minds ability to combine two different, although similar, images into one image is called fusion, and the resultant sense of depth is called stereopsis.

4.1.3 Parallax

A stereoscopic display is able to display parallax values. This makes stereoscopic display different from a monoscopic display. Disparity in the eyes produces parallax, and this provides the stereoscopic cue.

Electro-stereoscopic displays provide parallax information to the eye by

using a method related to that employed in the stereoscope. In a stereoscopic display, the left and right images are alternated rapidly on the monitor screen. When the viewer looks at the screen through shuttering eye-wear, each shutter is synchronized to occlude the unwanted image and transmit the wanted image. Thus each eye sees only its appropriate perspective view. The left eye sees only the left view, and the right eye only the right view. If the images (the term *fields* is often used for video and computer graphics) are refreshed fast enough (often at twice the rate of the monoscopic display), the result is a flickerless stereoscopic image. This kind of a display is called a field-sequential stereoscopic display.

When you observe an electro-stereoscopic image without eye-wear, it looks like there are two images overlayed and superimposed. The refresh rate is so high that you cannot see any flicker, and it looks like the images are double-exposed. The distance between left and right corresponding image points (sometimes also called "homologous" or "conjugate" points) is parallax, and may be measured in inches or millimeters.

Parallax and disparity are similar entities. Parallax is measured at the display screen, and disparity is measured at the retinal. When wearing eye-wear, parallax becomes retinal disparity. Retinal disparity produces parallax, and parallax in turn produces stereopsis. Parallax may also be given in terms of angular measure, which relates it to disparity by taking into account the viewers distance from the display screen.

Since parallax is the entity which produces the stereoscopic depth sensation we give a classification of the kinds of parallax one may encounter when viewing a stereoscopic view.

4.1.4 Types of Parallax

Four basic types of parallax are shown in Figure 4.1 [21]. In the first case, *zero parallax* (Figure 4.1(a)), the homologous image points of the two images exactly correspond or lie on top of each other. When the eyes of the observer, spaced apart at distance IOD (the interpupillary or interocular distance, on

average two and a half inches), are looking at the display screen and observing images with zero parallax, the eyes are converged at the plane of the screen. In other words, the optical axes of the eyes cross at the plane of the screen. When image points have zero parallax, they are said to have zero parallax setting.

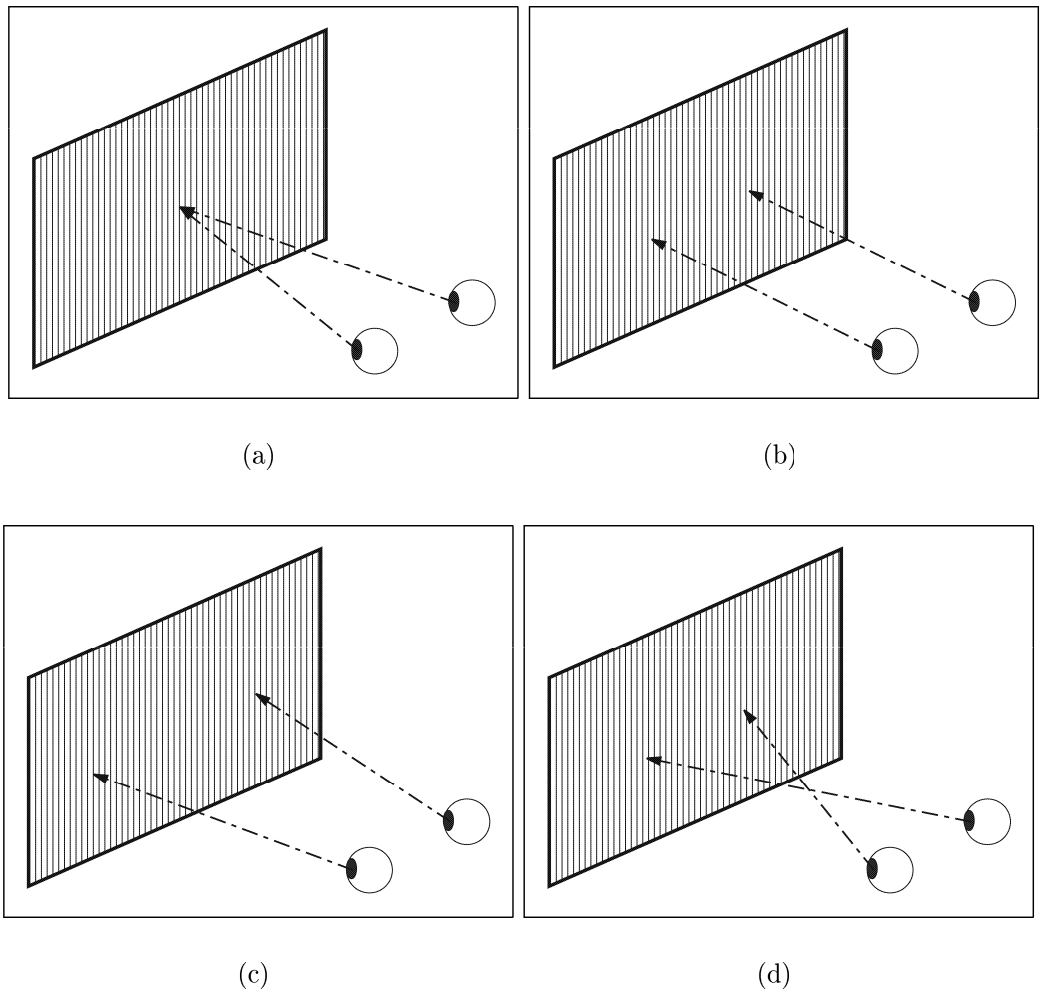


Figure 4.1: Types of parallax: (a) *zero parallax*; (b) *positive parallax*; (c) *divergent parallax (a kind of positive parallax)*; (d) *negative parallax.*

In one type of *positive parallax* (Figure 4.1(b)), the axes of the left and right eyes are parallel. This happens in the visual world when looking at objects that are at a great distance from the observer. For a stereoscopic display, when the distance between the eyes (*IOD*) equals the parallax, the axes of the eyes will be parallel, just as they are when looking at a distant object in the visual world. Experiences show that having parallax values equal to IOD, or nearly IOD, for

a small screen display will produce discomfort. Any uncrossed or positive value of parallax between IOD and zero will produce images appearing to be within the space of the cathode ray tube (CRT), or behind the screen. We say that such objects are within CRT space.

Another kind of positive parallax is shown in Figure 4.1(c), *divergent parallax*, in which images are separated by some distance greater than IOD. In this case, the axes of the eyes are diverging. This divergence does not occur when looking at objects in the visual world, and the unusual muscular effort needed to fuse such images may cause discomfort. There is no valid reason for divergence in computer-generated stereoscopic images. Objects with *negative parallax* (Figure 4.1(d)), appear to be closer than the plane of the screen, or between the observer and the screen. We say that objects with negative parallax are within viewer space.

4.1.5 Focusing and Convergence Relationship

The left and right image fields must be identical in every way except for the values of horizontal parallax. The color, geometry, and brightness of the left and right image fields need to be the same or to within a very tight tolerance, or the result will be *eye fatigue* for the viewer. If a system is producing image fields that are not suitable in these respects, there are problems with the hardware or software. It will never be able to produce good-quality stereoscopic images under such conditions. Left and right image fields congruent in all aspects except horizontal parallax are required to avoid discomfort [13].

The eyes converge on the objects in the real world. But in stereoscopic visualization, it is assumed that the eyes converge on the screen not on any specific object, and this convergence does not show up any change. This differentiation of real world and stereoscopic visualization causes some people depart from their natural feeling and those people may experience an unpleasant sensation when looking at stereoscopic images, especially images with large values of parallax. Experiences show that its better to use the lowest values of parallax possible for a good depth effect in order to help to reduce viewer discomfort. But the parallax value specification and visual discomfort should

be adjusted so that while providing a good depth effect visual discomfort is minimized.

The goal when creating stereoscopic images is to provide the deepest effect with the lowest values of parallax. This is accomplished in part by reducing the IOD. As a rule, parallax values should not exceed 1.6° [23]. Also the distance of the viewer from the screen should be kept in mind when composing a stereoscopic image.

4.1.6 Crosstalk (Ghosting)

Crosstalk in a stereoscopic display results in each eye seeing an image of the unwanted perspective view. In a perfect stereoscopic system, each eye sees only its assigned image. In particular, there are two reasons for crosstalk in an electronic stereoscopic display: departures from the ideal shutter in the eye-wear, and CRT phosphorus afterglow [14]. A third reason of ghosting is non-matching perspective projection for both eyes. This may occur when a point is projected for an eye and is not projected for the other.

In an ideal field-sequential stereoscopic display, the image of each field, made up of glowing phosphor, would vanish before the next field was written, but thats not what happens. After the right image is written, it will persist while the left image is being written. Thus, an unwanted fading right image will persist into the left image (and vice versa). The term ghosting is used to describe perceived crosstalk. Stereoscopists have also used the term *leakage* to describe this phenomenon. The perception of ghosting varies with the brightness of the image, color, and – most importantly – parallax and image contrast. Images with large values of parallax will have more ghosting than images with low parallax. High-contrast images, like black lines on a white background, will show the most ghosting. Given the present state of the art of monitors and their display tubes, the green colored phosphor has the longest afterglow and produces the most ghosting effect.

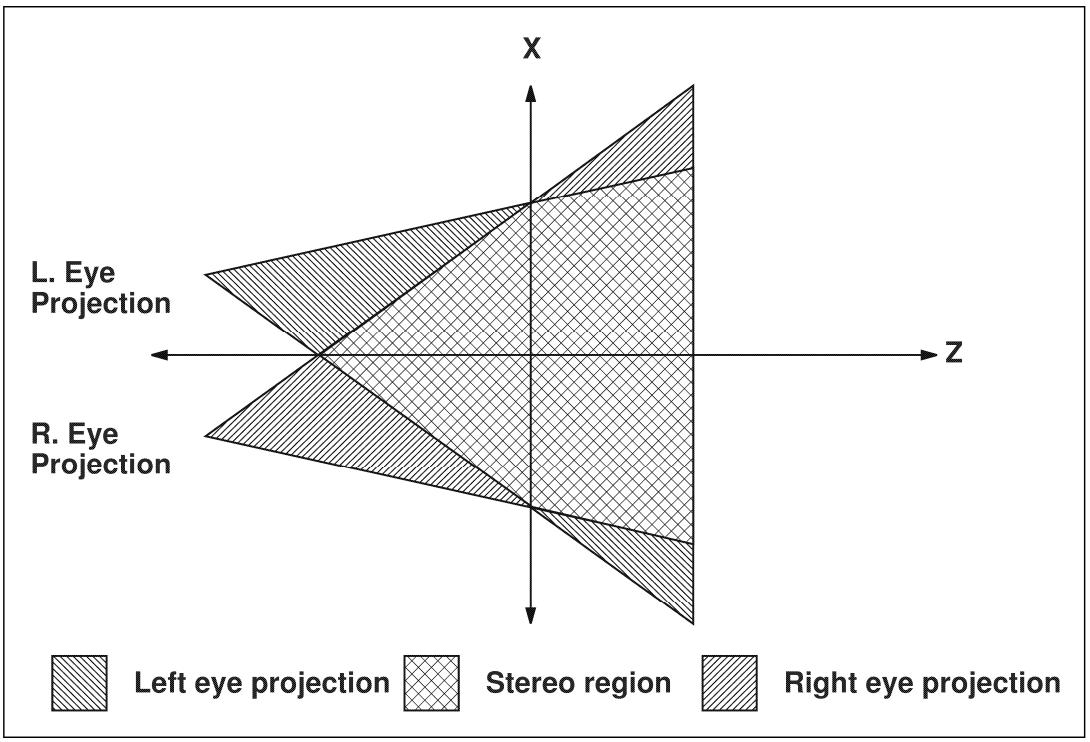


Figure 4.2: Off-axis projection.

4.2 Stereoscopic Projection System Used

Now, we need to explain the stereoscopic projection system we used. In general, stereo projections are divided into two: *on-axis* and *off-axis* [7]. A sample illustration of Off-axis projection is shown in Figure 4.2. As it can be seen from the figure this kind of stereo projection requires 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 cause by non-matching projections. 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, which modifies the data with translations and rotations, has an important advantage over off-axis projections in terms of speed. However, on-axis projections causes loss of data in both sides of the view frustum. The data loss is caused by translations of the data for right and left eye projections and is illustrated in Figure 4.3.

The disadvantage of on-axis projections, namely ghosting effect at the peripheries, is 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. The refinement process of stereoscopic view and comparison with the off-axis projection is shown in (Figure 4.4).

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 distance projection is very small with respect to the terrain, elimination of the ghosting effect does not cause significant processing overhead.

There is another way to prevent ghosting effect on the stereo view. This can be achieved by first culling and projecting the data with respect to right eye view and translating it with IOD to the left for the left eye view. This scheme may produce faster results in terms of processing speed because of a single translation for the left eye instead of two translations. However, it makes parallax control impossible and makes it suitable only for static stereoscopic image generation and viewing applications where parallax control is not needed. Since our platform is dynamic and makes navigation over the terrain possible, using the previous approach for on-axis projection does not cause any additional overhead for the application because we have to make two translations for the data in anyway. The approach is shown in Figure 4.5.

4.3 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 [9]. 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 3.22 as in Figure 4.6. Stereoscopic drawing algorithm using the SGT approach is given in Figure 4.7. The algorithm that decides when vertex activation and frustum culling operations should be done according to different culling schemes is given in Figure 4.8.

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.

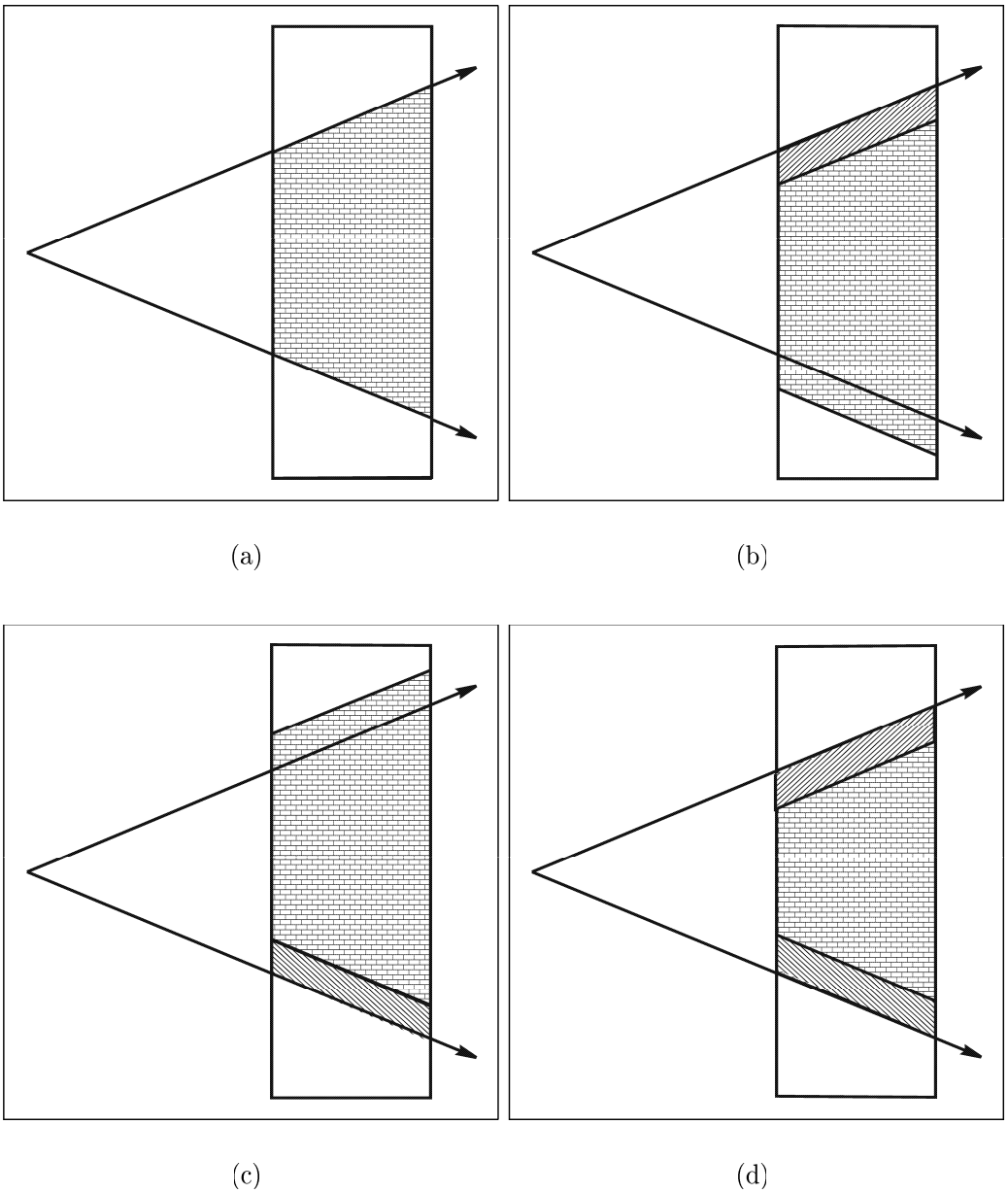


Figure 4.3: Loss of data in on-axis projection: (a) view frustum data; (b) translated data for right eye view; (c) translated data for left eye view; (d) resultant stereoscopic view.

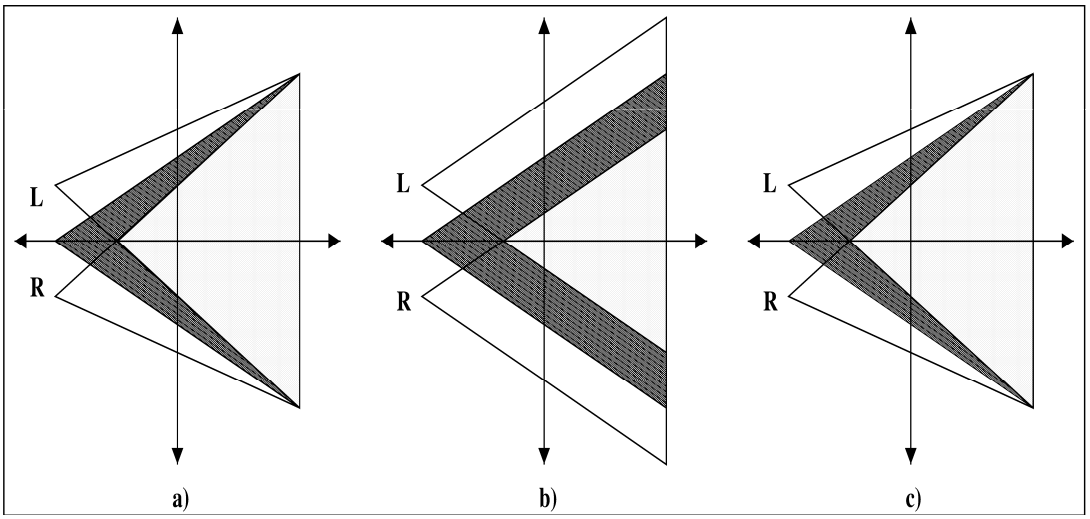


Figure 4.4: Elimination of disadvantages of on-axis projection: (a) Off-axis projection, (b) on-axis projection, (c) elimination of the ghosting effect.

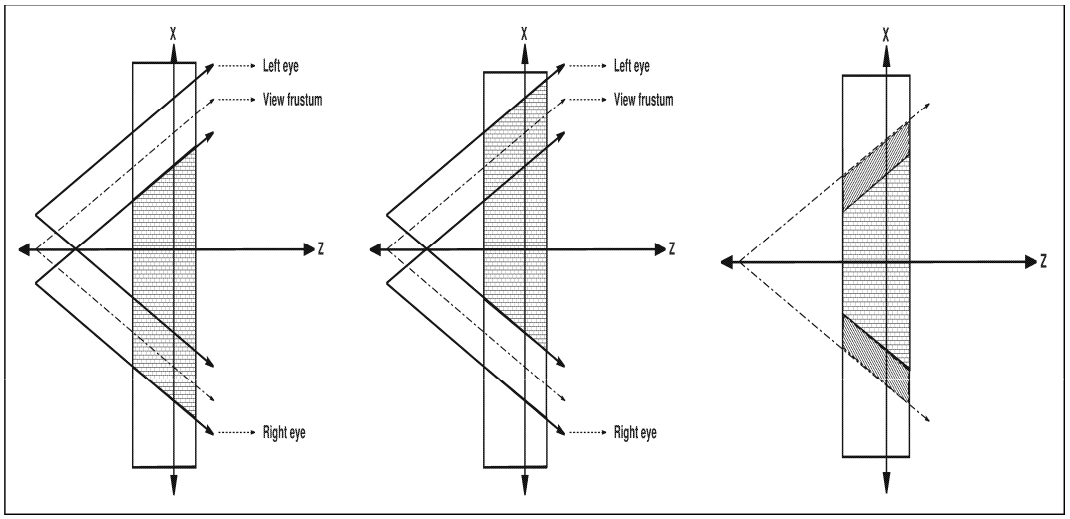


Figure 4.5: Producing the same result for on-axis projection with one translation: (a) right eye projected data; (b) translated data for left eye view; (c) resultant stereoscopic projection which is the same with previous on-axis projection.

```
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
      .....
```

Figure 4.6: Using the draw-list constructed for the right eye in generating triangles for the left eye image in the SGT algorithm

```
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)
}
```

Figure 4.7: Stereoscopic drawing using 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)
```

Figure 4.8: The algorithm that calls view frustum culling and activation procedures depending on the culling scheme used

Chapter 5

Empirical Study

This chapter is devoted to the performance study of the proposed algorithms.

5.1 Definition of the Test Platform

The proposed algorithms were implemented on personal computers and graphics workstations using *C language* with OpenGL[1] [15] (see Appendix C). We name this system as *Stereoscopic View-Dependent Terrain Visualization System.* 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, in other words 526,338 triangles. The elevations are stored in a gray-scale image, in which each value represents a discrete elevation representation of 10 meters. The elevation values were doubled, in order to make the test platform more challenging. Besides, the test flights included texture and lighting calculations. The threshold value is taken as 1 degree which correspond to approximately 17.8 meters at 1 kilometer distance. The results were obtained on a personal computer with Intel Pentium[2] III – 550 Mhz CPU and 64 MB of main memory with 32 MB of graphics memory. The

[1] OpenGL is a registered trademark of Silicon Graphics, Inc.
[2] Pentium is a registered trademark of Intel Corporation.

user interface was created using *GLUI* library (see Appendix D).

5.2 Performance Results

We prepared a flythrough of the terrain with approximately 5,000 frames. Screen shots from the monoscopic flythrough is shown in Figure 5.1 and Figure 5.2. Figure 5.3 contains screen shots from the stereoscopic flight in which different visualization techniques are used. The number of polygons rendered during the flythrough is shown in Figure 5.5.

All figures given in this chapter are modified in order to have a better information about the results of the empirical study. The values are put into a regression function and smoothed. The figures representing real values are given in the Appendix E. Figure 5.4 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 5.1. 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 Figures 5.6, 5.7 and 5.8 performance comparison showing the frame rates of our culling techniques with each visualization method are 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. It is apparent that, if we increase these thresholds, the performance gains will increase drastically with the expense of the need to decrease the navigation speed. In our experiments, we have determined that the deviation threshold can be increased to 1000 meters and the deferring time can be increased to 0.4 second, without loss of quality and the need for decreasing the navigation speed. As it is seen in the figure, deviation based culling and deferred culling methods 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.

In Figures 5.9, 5.10, 5.11 and 5.12 the performances of the visualization methods for each of the proposed culling schemes are given. It is apparent that the proposed stereo visualization method – simultaneous generation of triangles – performs much better than the normal stereoscopic visualization. In deferred and deviation-based culling without morphing, the results of our SGT approach seems to be close to normal stereo visualization. The reason is that deferred and deviation based culling improves performance for both SGT and normal stereo visualizations. It should also be noted that we kept threshold values for deviation-based culling and deferred culling methods very small, and our experiences showed that we can almost quadruple these values. Since the operations will be done for only one eye in SGT approach, the performance differences will be much greater.

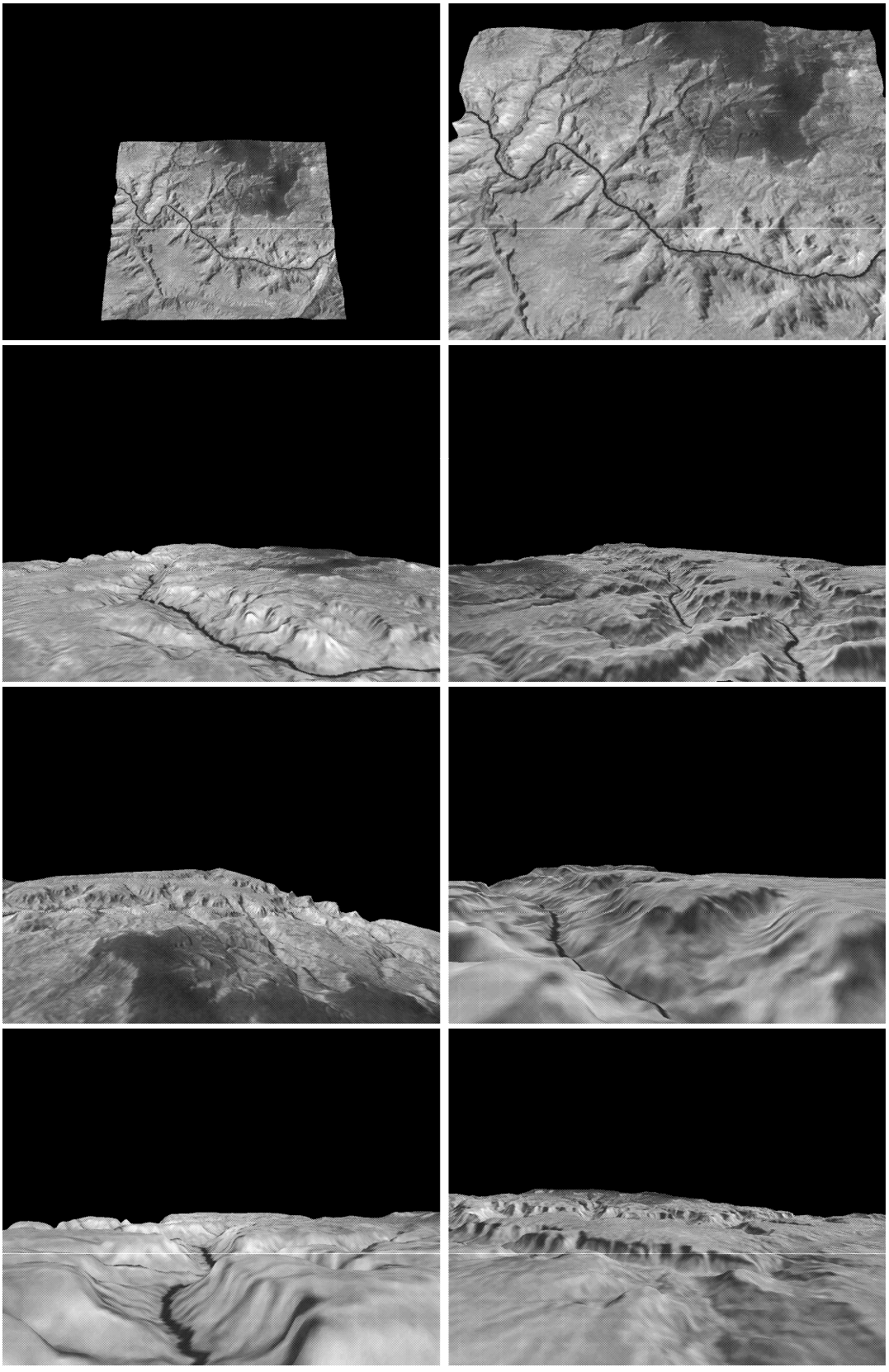


Figure 5.1: Still frames from a monoscopic walkthrough

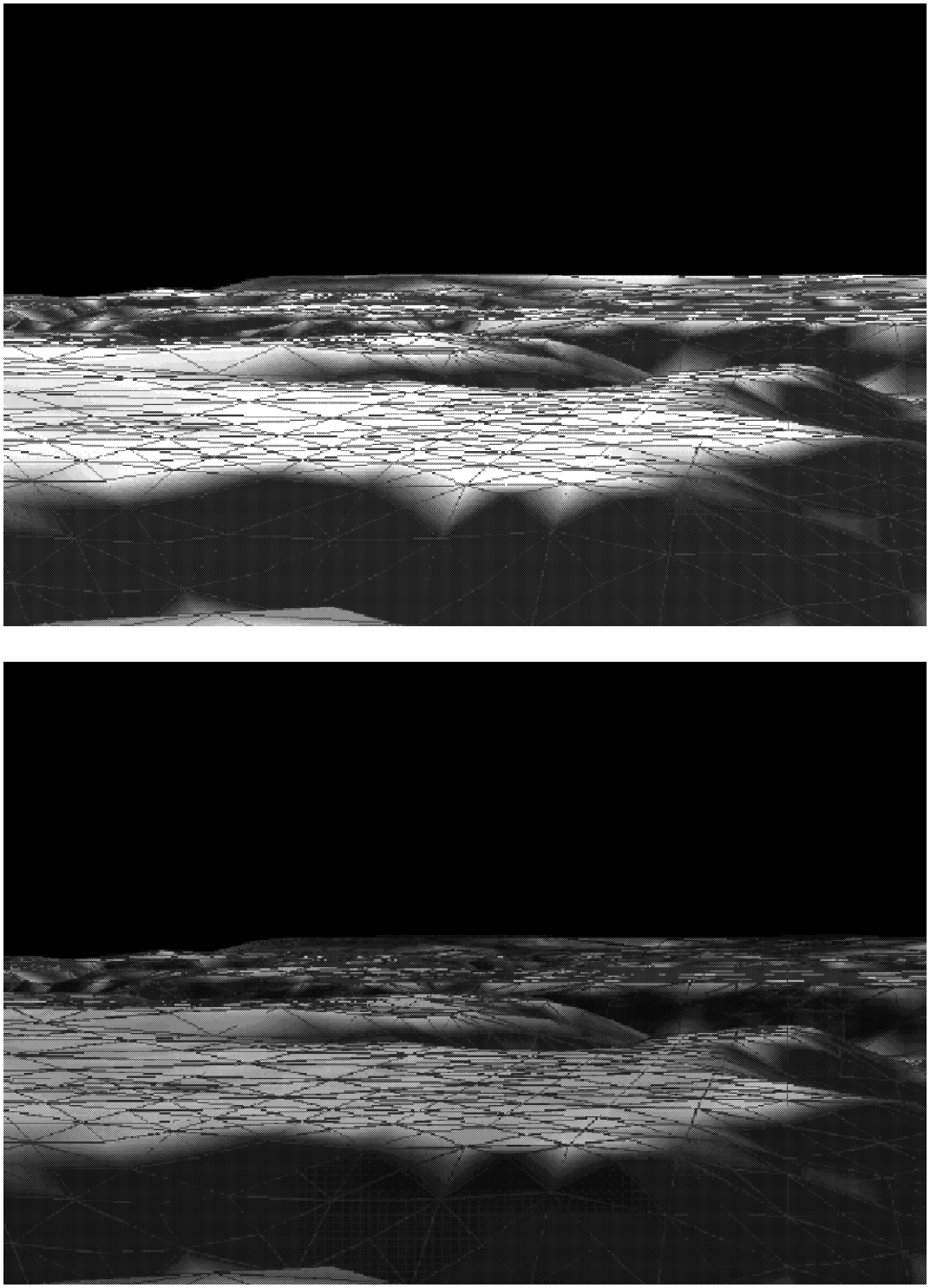


Figure 5.2: Another set of still frames from a monoscopic walkthrough, showing terrain in both filled and wire-frame modes with shading and texture.

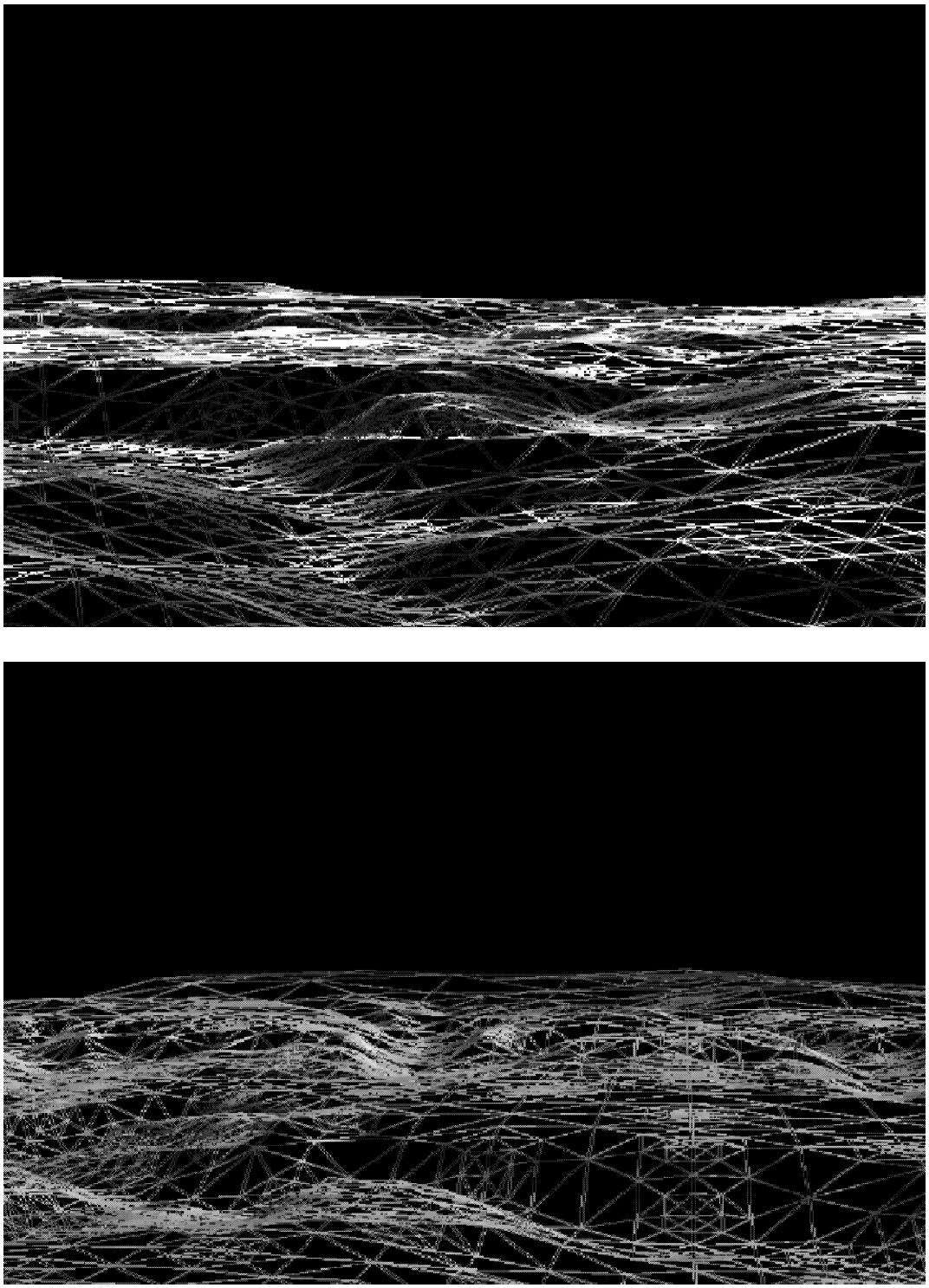


Figure 5.3: Still frames from a stereoscopic walkthrough, showing terrain in wire-frame.

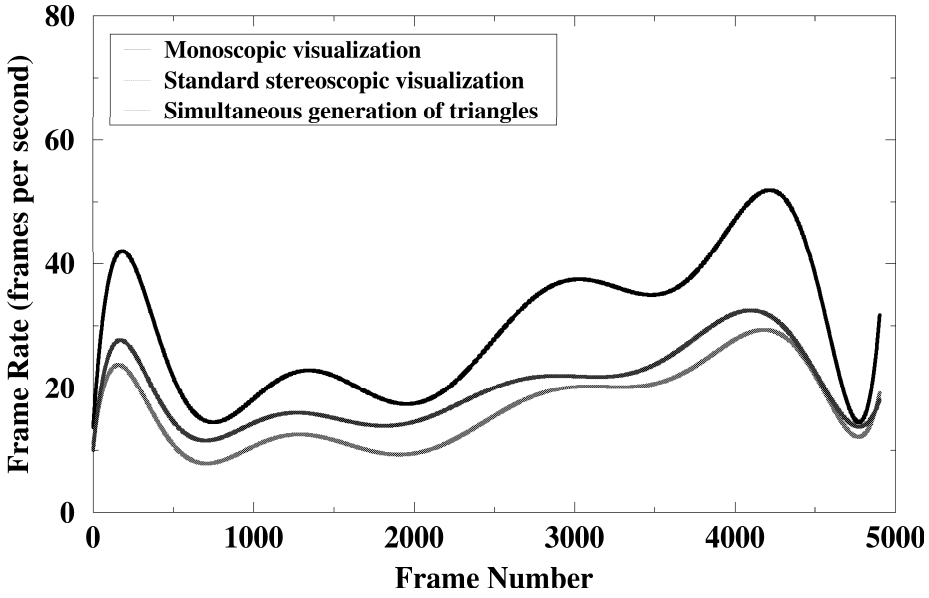


Figure 5.4: Comparison of the frame rates of different types of visualizations (SMOOTHED): *monoscopic visualization* where only one image is generated for each frame; *standard stereoscopic visualization* where two images are generated for each frame; *simultaneous generation of triangles* for stereoscopic visualization where we utilize triangle list for one eye to generate the triangles for the other eye.

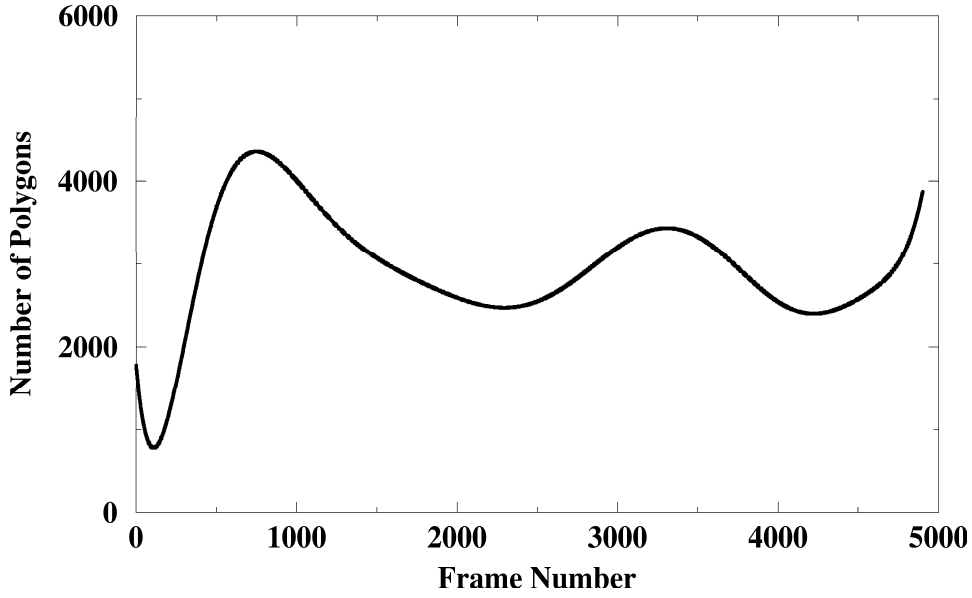
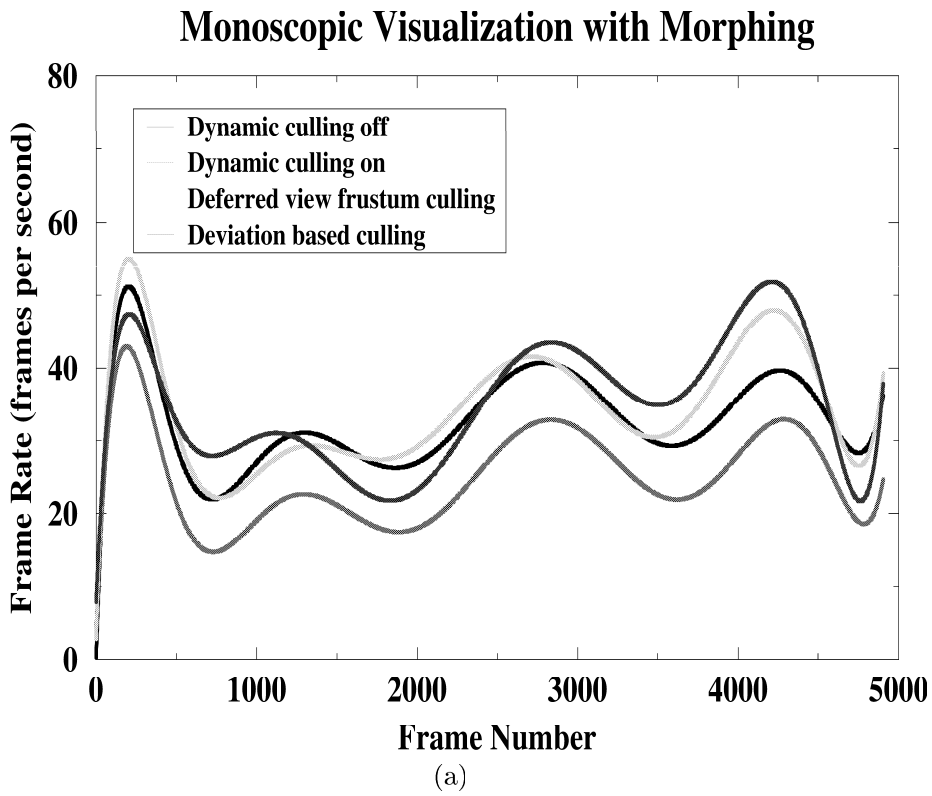
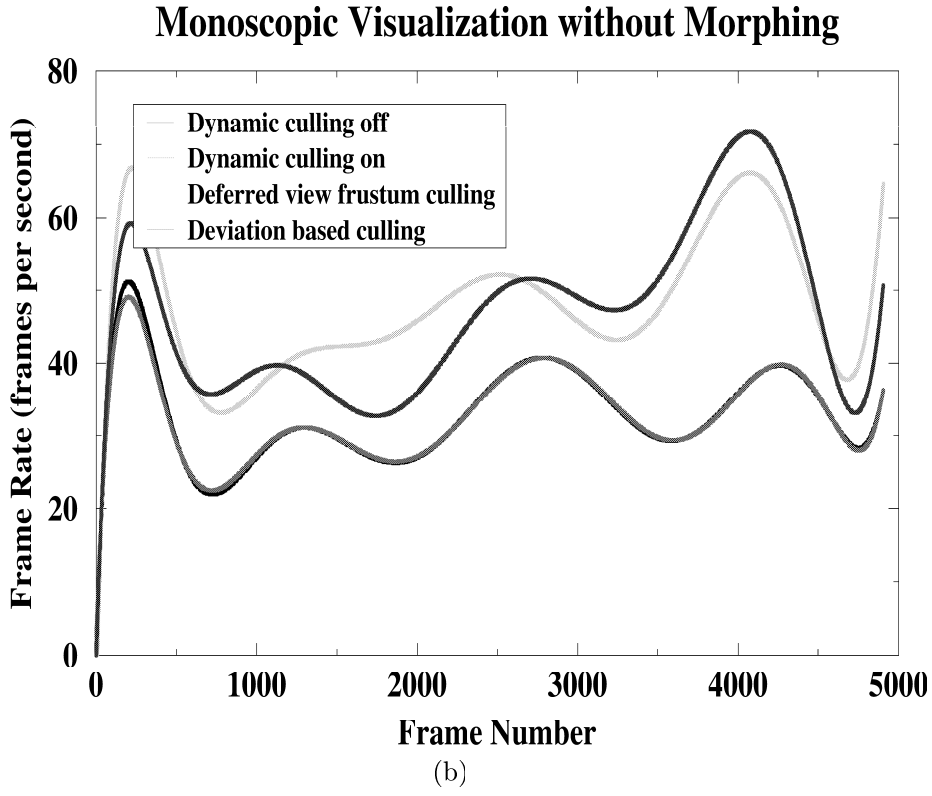


Figure 5.5: Number of polygons for the experimental visualization (SMOOTHED)

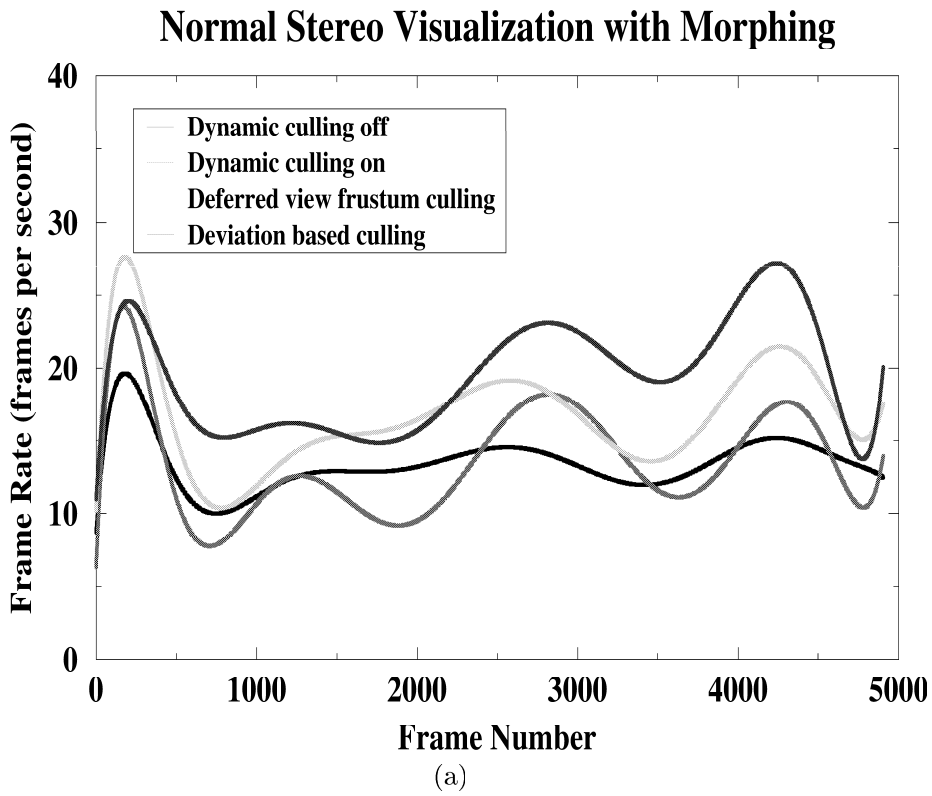


(a)

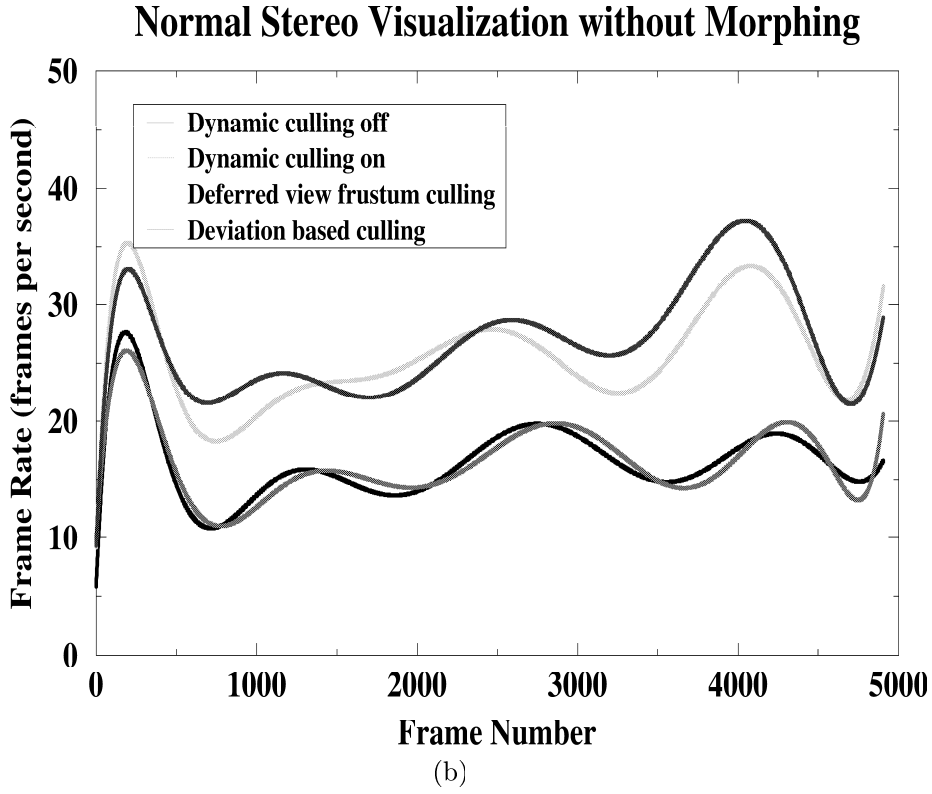


(b)

Figure 5.6: Comparison of the frame rates for visualization types with different morphing/culling options (SMOOTHED): (a) *monoscopic visualization* with morphing; (b) *monoscopic visualization* without morphing.

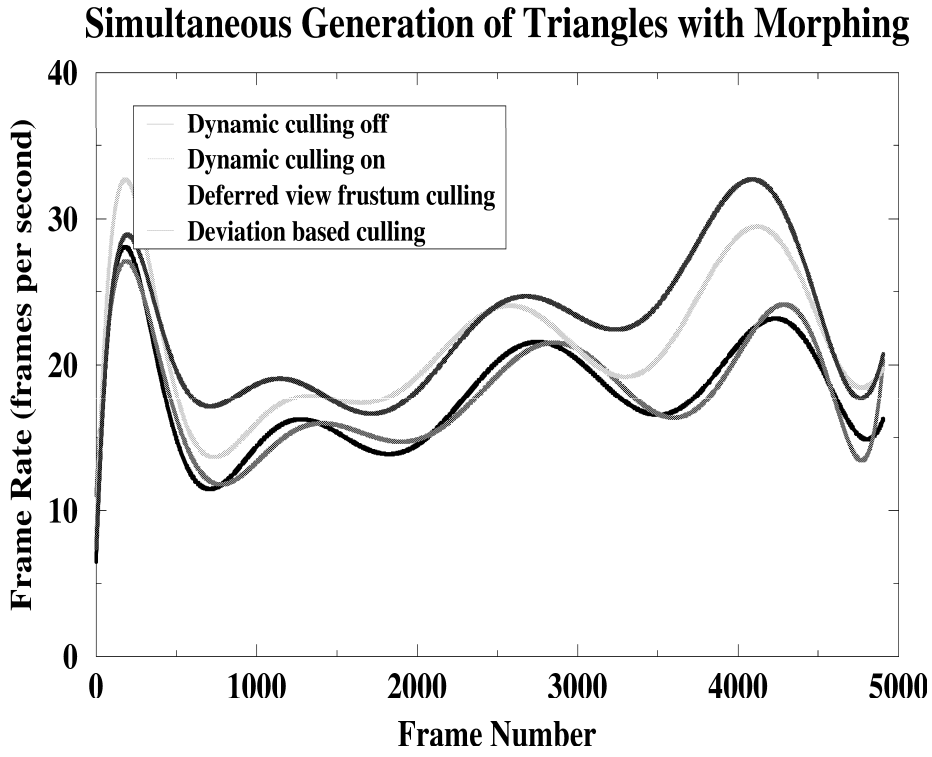


(a)

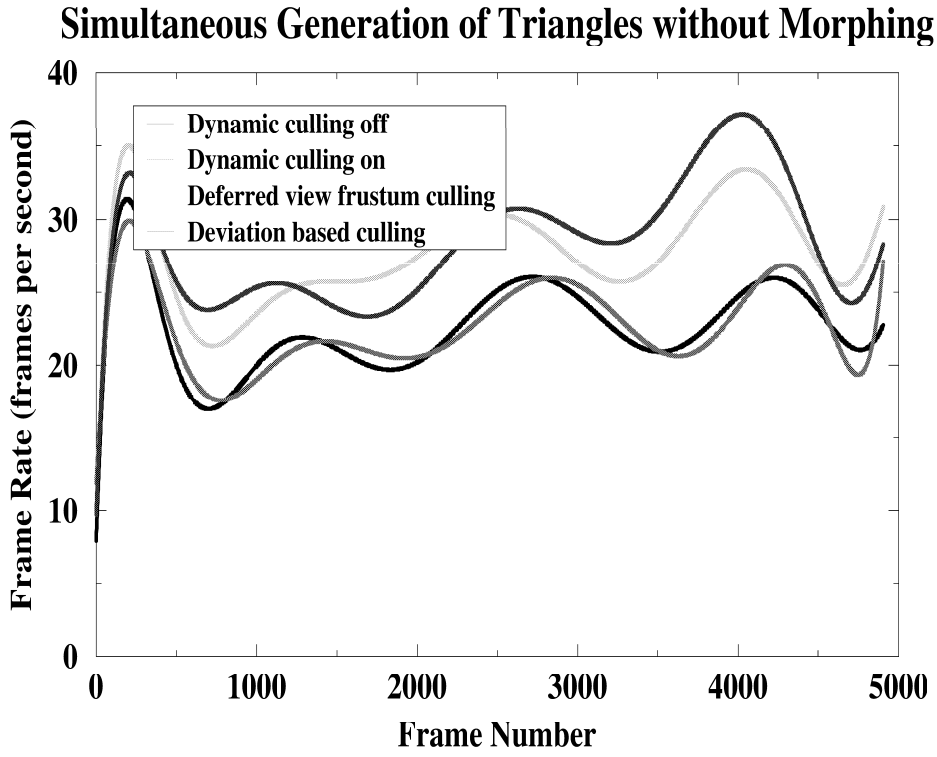


(b)

Figure 5.7: Comparison of the frame rates for visualization types with different morphing/culling options (SMOOTHED): (a) *standard stereoscopic visualization* with morphing; (b) *standard stereoscopic visualization* without morphing.

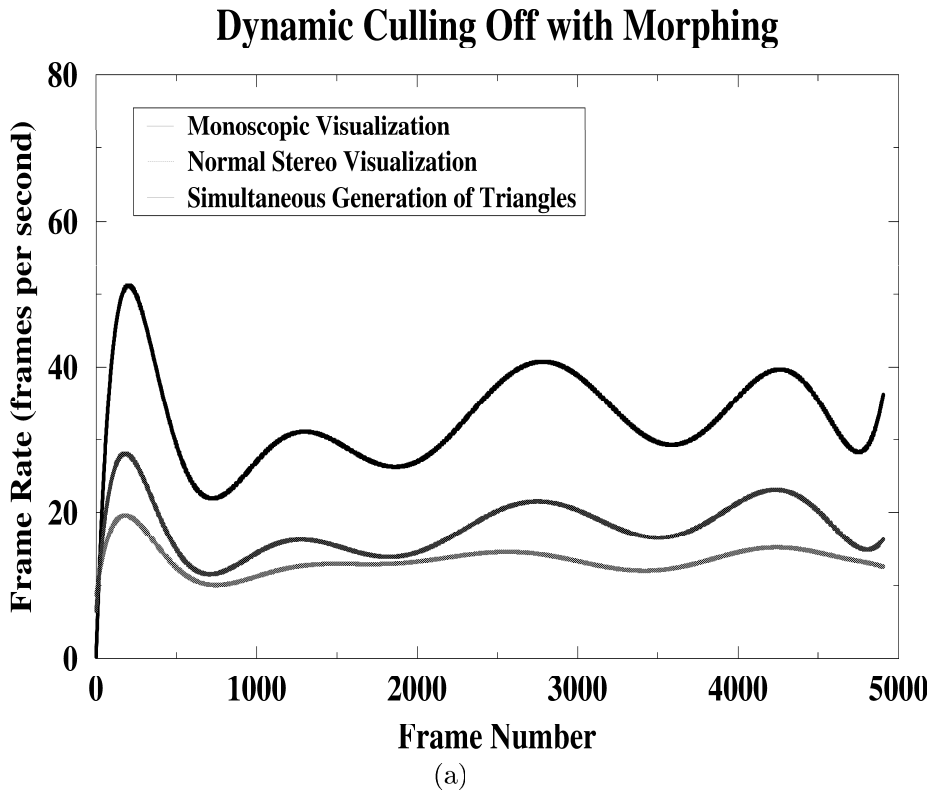


(a)

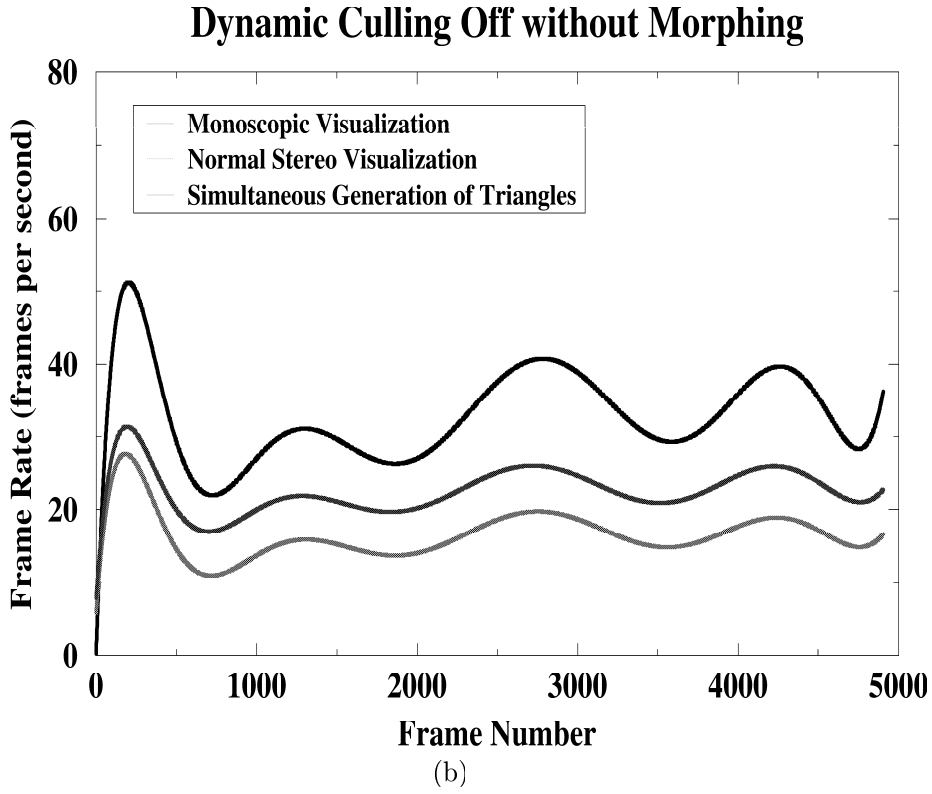


(b)

Figure 5.8: Comparison of the frame rates for visualization types with different morphing/culling options (SMOOTHED): (a) *simultaneous generation of triangles* with morphing; (b) *simultaneous generation of triangles* without morphing.

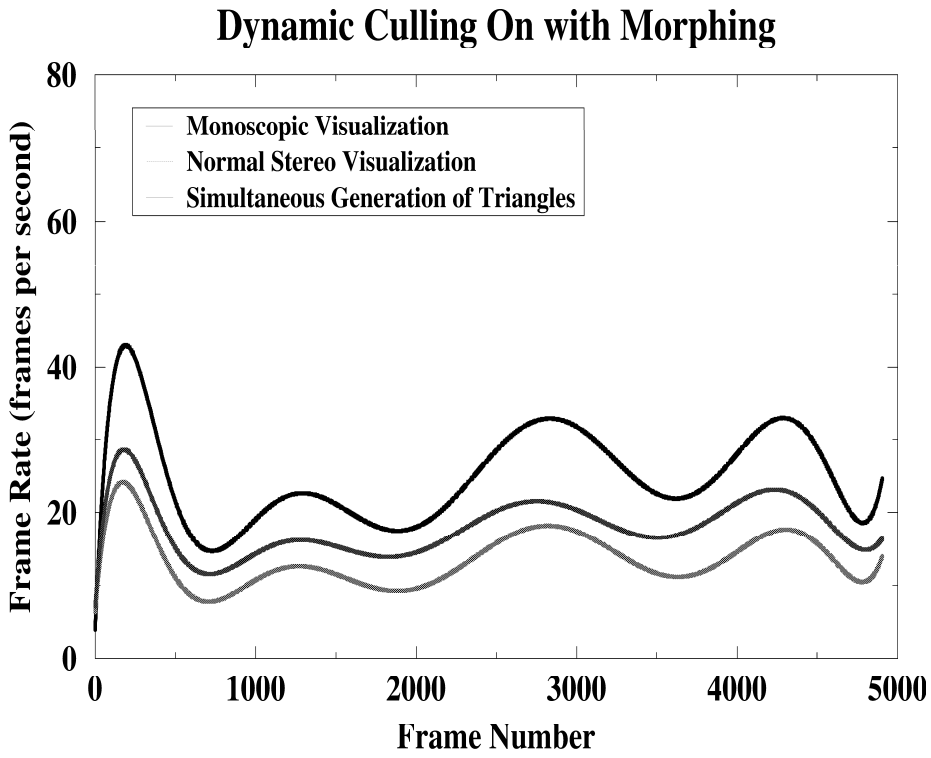


(a)

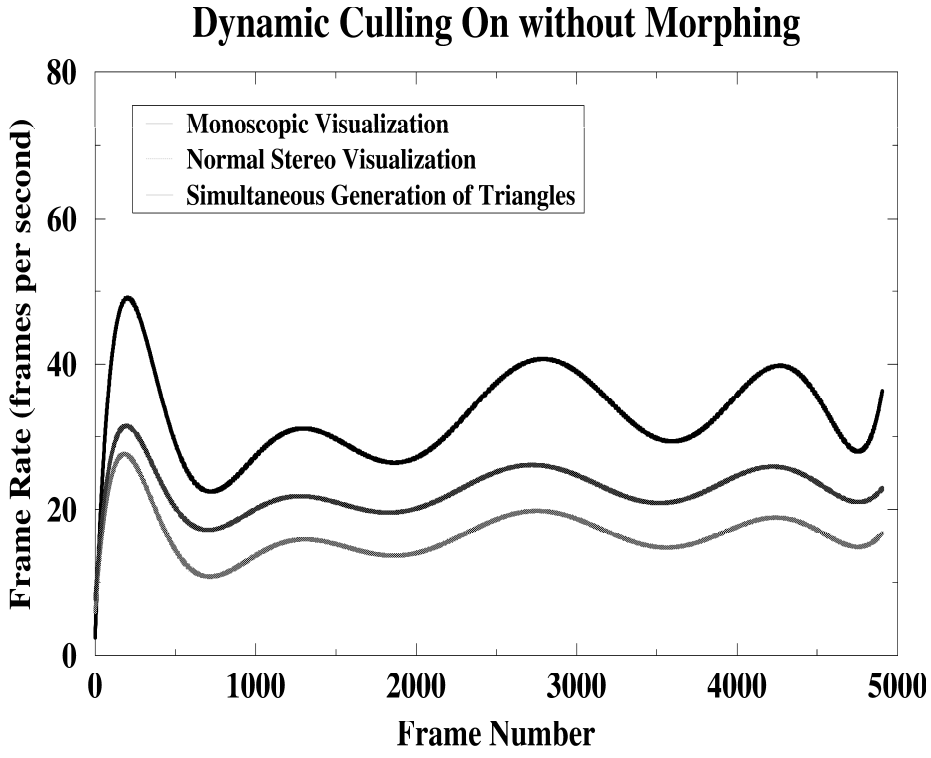


(b)

Figure 5.9: Comparison of different culling schemes for visualization types (SMOOTHED): (a) *dynamic culling off* with morphing; (b) *dynamic culling off* without morphing.

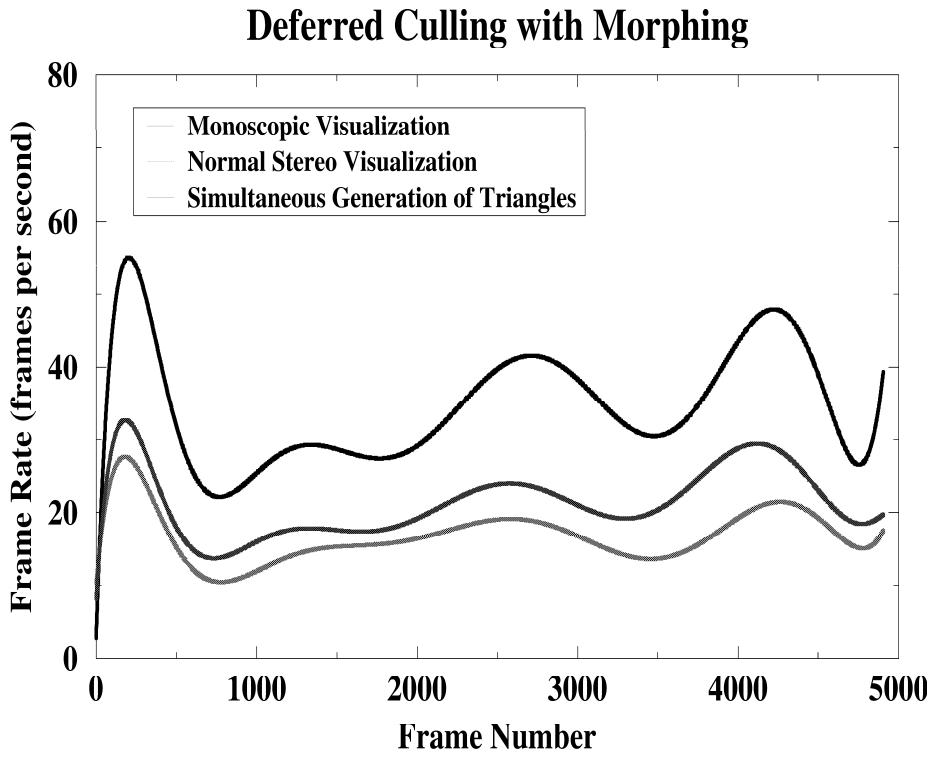


(a)

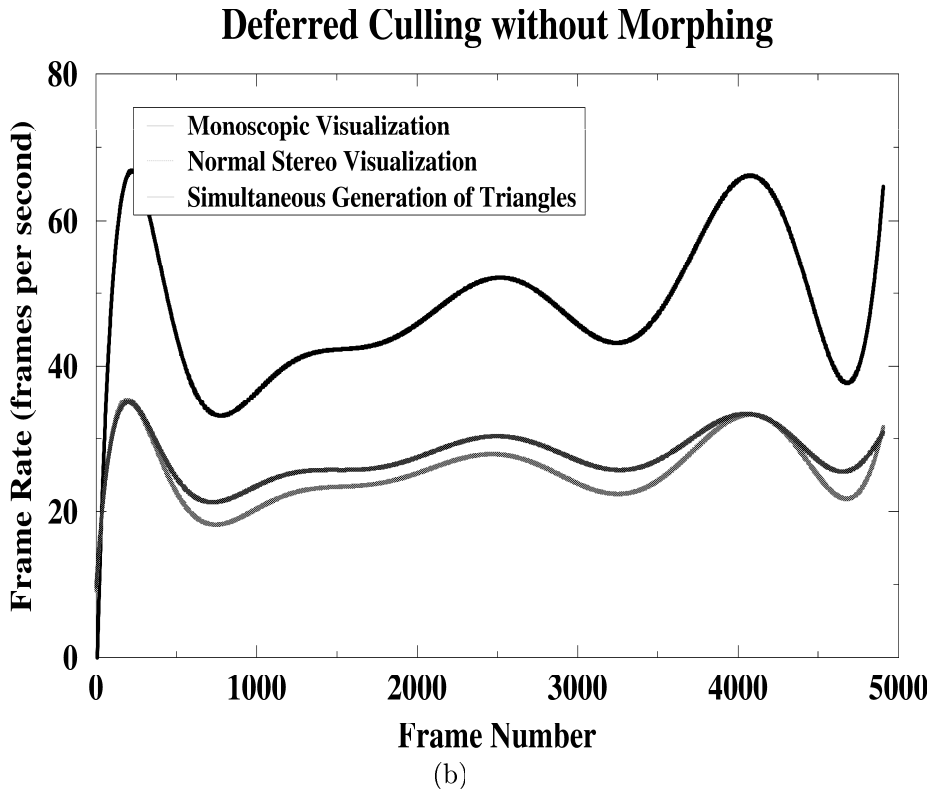


(b)

Figure 5.10: Comparison of different culling schemes for visualization types (SMOOTHED): (a) *dynamic culling on* with morphing; (b) *dynamic culling on* without morphing.

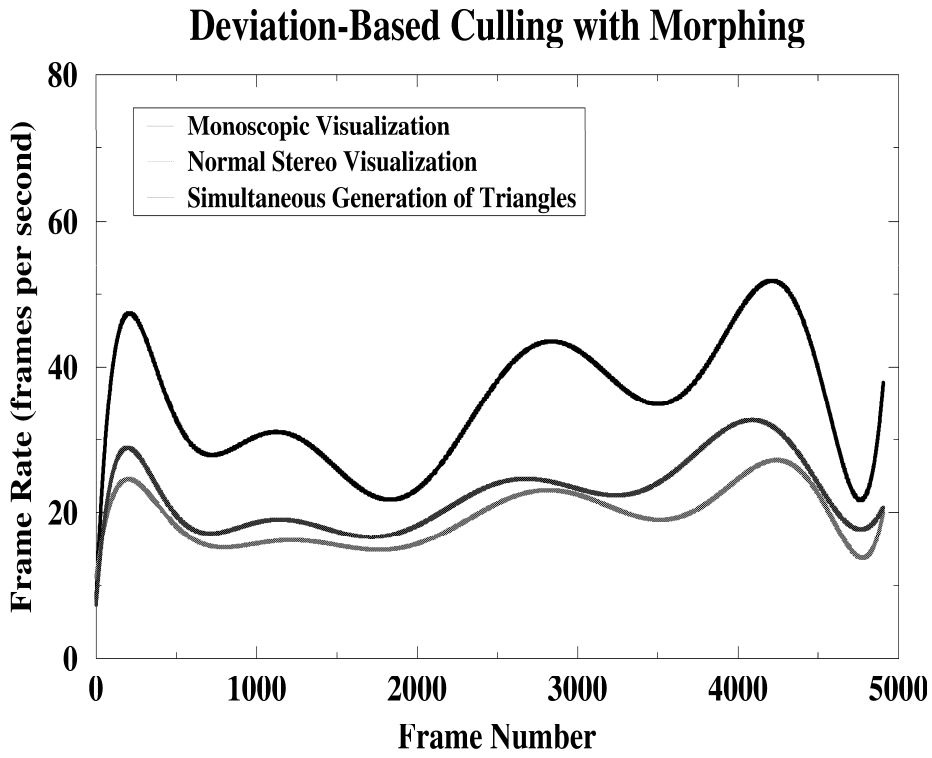


(a)

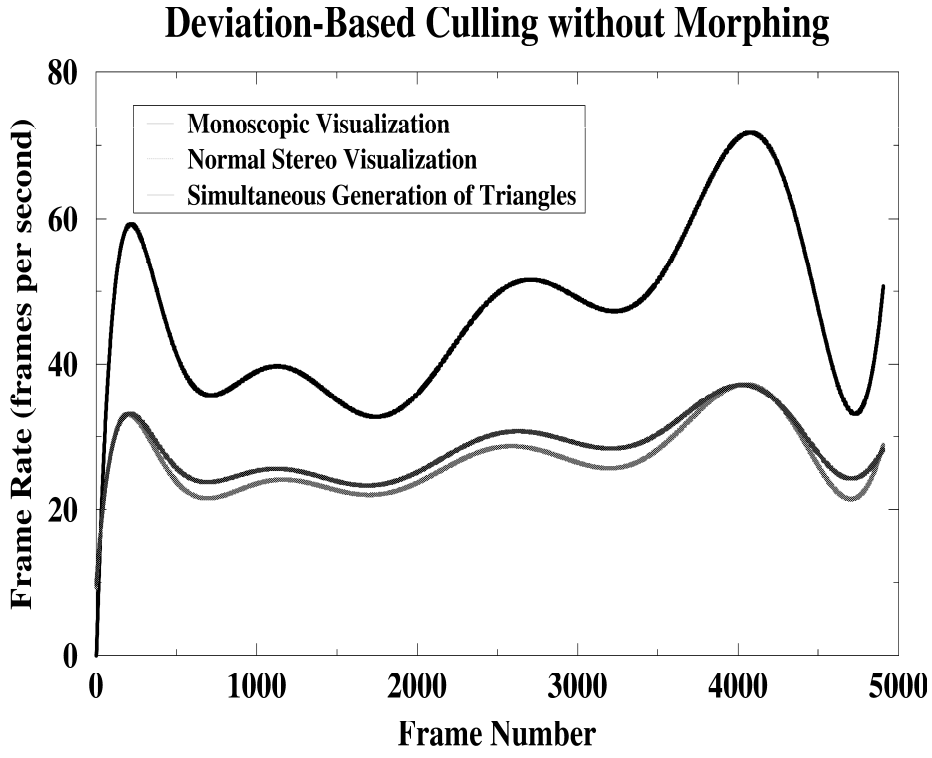


(b)

Figure 5.11: Comparison of different culling schemes for visualization types (SMOOTHED): (a) *deferred culling* with morphing; (b) *deferred culling* without morphing.



(a)



(b)

Figure 5.12: Comparison of different culling schemes for visualization types (SMOOTHED): (a) *deviation-based culling* with morphing; (b) *deviation-based culling* without morphing.

Table 5.1: Performance Results: The best performance gain is 43.27 % and the best average rendering speed is 27.48 fps.

Visual Method	Morph	Dynamic Culling	Def. VFC	Dev. VFC	Avg. FPS	Theoretical Perf. Gain	Practical Perf. Gain
Monoscopic	OFF	OFF	OFF	OFF	30.66		
SGT	OFF	OFF	OFF	OFF	21.83	42.41	42.61
Normal Ste.	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 Ste.	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 Ste.	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 Ste.	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 Ste.	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 Ste.	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 Ste.	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 Ste.	ON	OFF	OFF	ON	17.96		

Chapter 6

Conclusion and Future Work

6.1 Conclusions

This thesis 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. This simplification criterion is very useful in terms of performance needs. Because the calculations needed for the simplification criteria are very simple and if storage requirements are very critical, then it may easily be implemented at run time without significant overhead.

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 algorithm's performance is very satisfactory as shown in related figures and tables (Figures 5.4, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12 and Table 5.1).

The cracking problem is solved using the dependency information stored for each vertex. This prevention scheme is crucial in order to have a perfect view of the terrain. Because as the threshold value increases the appearance of the terrain with cracks becomes annoying.

The popping artifacts that may occur while switching between different resolutions of the data are eliminated using morphing. Morphing scheme imposes approximately 10–30% overhead on the performance. Especially for great threshold degrees, the walkthrough becomes full of popping artifacts that makes the visualization disturbing without morphing.

The proposed algorithms were implemented on personal computers and graphics workstations. Since the application is domain independent, we did not need to make significant changes in the code except the functions needed for stereoscopic buffering mechanisms. Performance experiments showed that the two views for a stereoscopic walkthrough 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 multiresolution representation of height fields.

6.2 Future Work

In this section we are going to propose some ideas that could be applied to improve and develop the stereoscopic view dependent visualization system.

Since, our aim was to see how fast we could generate the second eye view in a stereoscopic environment, we did not focus on modifying threshold value. Threshold value modification during the navigation helps us to adjust view parameters like morph-segment number and terrain level of detail dynamically, so that the intended frame rate during the navigation could be achieved. By the help of frame budgeting, the application would be more system independent and larger terrain visualizations in the system would be available.

For larger terrains, since the terrain data will not fit into memory, a paging mechanism is needed. Paging in terrain visualization systems should not be left to the operating system by only supporting enough virtual memory. Paging could be supported by implementing a robust paging algorithm and using spatial terrain database access. Spatial terrain database access would also help us to bind textures to terrain blocks making large texture usage available.

Memory optimization methods like paging would also make us to add other features to terrain, such as vegetation, hydrographic data and even urban visualization in large terrains.

With the implementation of frame budgeting and spatial database access, a more general purpose application for stereoscopic terrain visualization could be achieved at interactive frame rates using simultaneous generation of triangles approach with continuous multi-resolution representation of terrain height fields.

Appendix A

The User Interface

In this appendix we give information about the user interface implemented for the Stereoscopic View-Dependent Terrain Visualization System.

A.1 Overview

In the GUI of the system, it is intended to supply the most flexibility to the user while visualizing the terrain. The properties of the algorithms and visualization methods are tried to be emphasized.

The graphical user interface of the system is given in Figure A.2. The user interface is developed to give the user the ability to navigate over the terrain freely. Navigation over the data by using latitude and longitude locations in meters is possible. Each vertex coordinate is in 100m. scale. Also the user is able to use cursor keys to navigate over the data, and this is the recommended style of the navigation.

The user interface is a combination of two parts on the screen. One of them is the viewing area for the terrain data. The other is the commands section.

A.2 Viewing Area

To use cursor key navigation, and function keys it is needed to click the viewing area with mouse. Likewise in order to use commands section, commands area has to be clicked.

Viewing area contains information about some parameters. Those parameters are;

- Morphing status on/off (*F1*),
- Dynamic culling status on/off (*F2*),
- Deferred culling status on/off (*F3*),
- Deviation based culling status on/off (*F4*),
- Frames per second,
- Frame drawing time,
- Polygons drawn for that frame,
- Flight direction,
- Altitude,
- Wire-frame mode on/off (*F5*),
- Texture mode on/off (*F6*),
- Threshold change (*F7-F8*),
- Lights on/off (*F9*).
- Turn left ($\leftarrow$),
- Turn right ($\rightarrow$),
- Go forward ($\uparrow$),
- Go backward ($\downarrow$),

- Increase altitude (+),
- Decrease altitude (–).

Most of those options are self explanatory. The option *threshold change* is used to increase or decrease the threshold value used to simplify the terrain. As this value changes the distances measured from the user location to the vertices are multiplied with it and taken into account by the vertex activation algorithm.

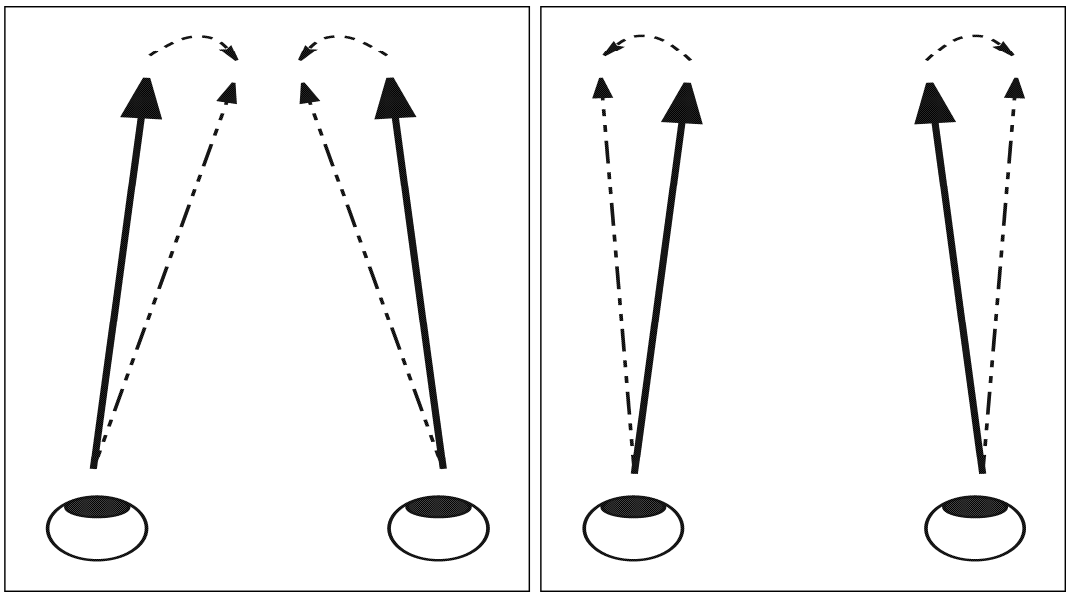
For example, if the value is decreased, the distances measure will be smaller and therefore the user will be treated to be closer than normal reflecting a decreased threshold value, which causes a refinement process. Likewise, if the value is increased, the distances will be greater than real, and vertices will not be activated, simulating a coarsening procedure. With this option it is able to change the threshold value and see a more refined or coarsened view of the terrain. This is very useful if the performance of the system is not sufficient and it is needed to decrease the terrain complexity.

A.3 Commands Area

The commands area occurs from collapsible/expandable menu blocks. The user may turn a block on by pressing its name button on the commands section. This makes commands menu area flexible supporting the viewing area to be enlarged as needed. Besides, it is also possible to collapse the whole commands area by pressing *commands* button.

A.3.1 Culling Techniques Block

This collapsible block occurs from three groups of radio-buttons representing the state of the culling algorithm used at that moment. Those are *dynamic culling*, *deferred view frustum culling* and *deviation based culling*. Each of these groups can be turned on or off. The selected algorithm is immediately taken into action.



(a) (b)

Figure A.1: Results of emboss change: (a) negative emboss; (b) positive emboss.

A.3.2 Visual Properties Block

In this block visual properties of the viewing area can be adjusted. It works in the same manner as the culling techniques block. Here it is possible to change from stereo to mono visualization. But since stereoscopic buffers are set up during the program start-up phase, monoscopic visualization is simulated by not drawing the second eye view and drawing only the right eye view.

The terrain can be viewed in wire-frame mode, textured, or in lighting enabled. Here it is possible to switch morphing on or off.

A.3.3 Navigation Block

In the navigation block, there are two parts:

- Looking direction part (a *rotating ball* and a *reset button*) and,
- Location control part.

Rotating ball makes it available for the user to move the terrain in any direction. This is very useful to visualize the terrain in any angle when the user comes to a point. *Reset Look* button is used to initialize the ball to its original position.

Location control part is used to bring the user to a specific location and direction. There are four spinner-buttons that enable the user to achieve this purpose. These are;

- *Direction*: Used to change the direction of the viewer. As the value increases the user turns to the right.
- *Altitude* : Used to increase or decrease the altitude of the user. The value shows the altitude of the viewer from the *average elevation* of the terrain.
- *Latitude* : Used to change the latitude of the user.
- *Longitude*: Used to change the longitude of the user.

A.3.4 Stereo Control Block

Stereo control block defines two main parameters of a stereoscopic visualization. These are;

- IOD and,
- Emboss.

IOD is inter-ocular distance between the eyes of the user. Initially it is set to the average eye separation value of adults that is 64mm. A user may change this value while looking through LCS glasses and can adjust it according to his or her eye separation. As this value increases, the depth effect also increases but this may result in eye fatigue if set to very high values.

Emboss changes convergence point of both eye images in the space by applying them a rotation. If this value is less than zero, then it means that the right eye view is rotated to left and left eye view is rotated to right causing the convergence point come nearer and making the stereo view appear behind the CRT display. It also enlarges the scene and increases the depth effect that can be achieved.

Positive emboss values causes a divergent like parallax to be achieved and makes the image pop out of the screen. Emboss values should be used to adjust depth effect with respect to the user distance to the screen. If the user is close to the CRT display then this value should be a positive value. If the distance of the user to the CRT is not very close with respect to the former situation, then this value should be a negative one, not to disturb the depth effect that can be achieved from the visualization. The effect of changing emboss value is shown in Figure A.1.

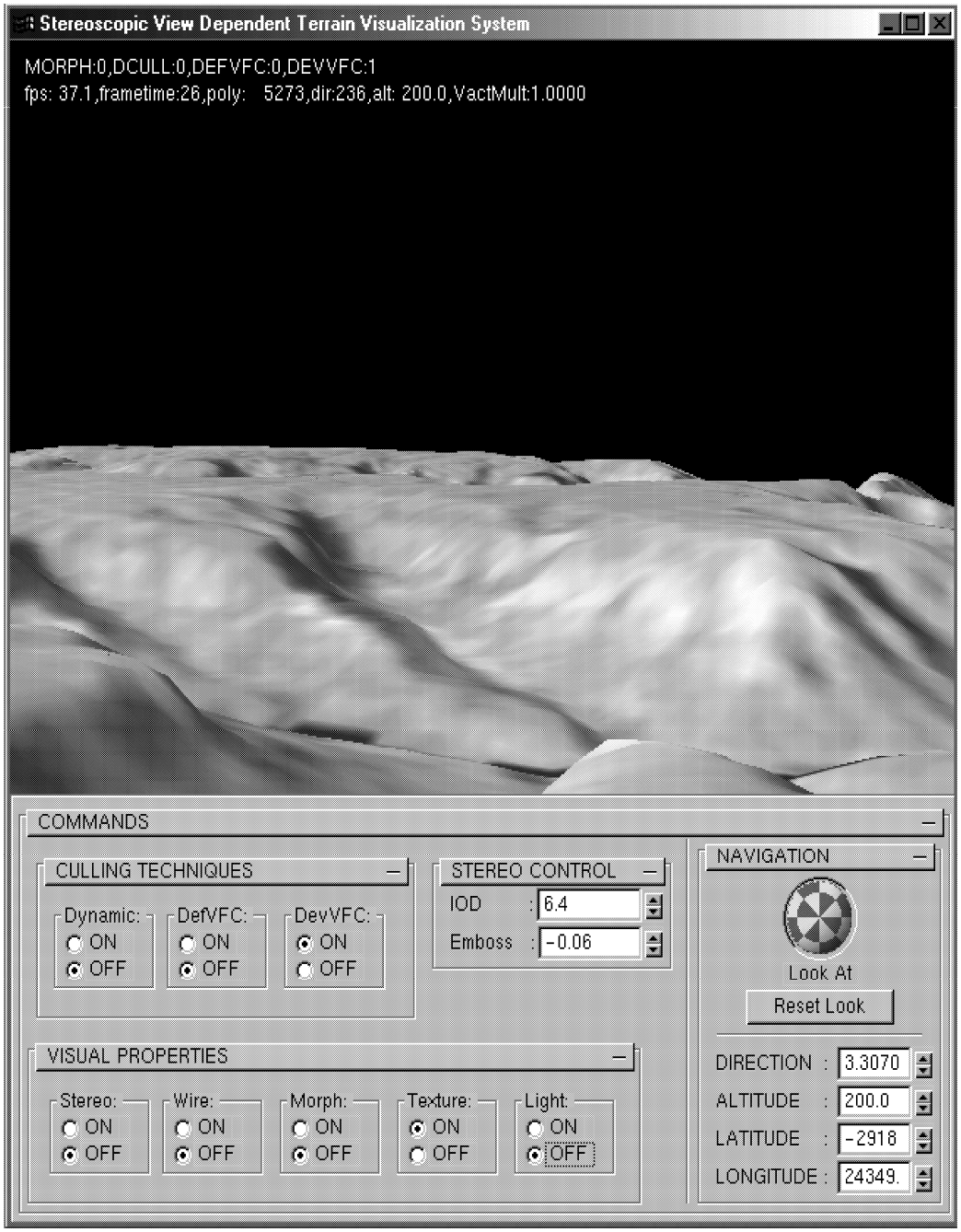


Figure A.2: Graphical user interface of the system

Appendix B

Types of Stereoscopic Displays

There are many methods developed for stereoscopic views [24].

- Anaglyph Method.
- Squished Side-by-side Method.
- Polar Method.
- Page Flipping Method.
- Line Alternate Method.

B.1 Anaglyph Method

The anaglyph method uses color to encode the right and left image pairs. This method requires that the user wear a special pair of glasses with color filters over each eye. These glasses have a red filter over one eye and a blue or green filter over the other eye. Most anaglyph glasses put the red filter over the left eye.

There are three ways to encode an anaglyph image:

- *Color* : Color anaglyphs try to preserve as much of the original image color as possible (Figure B.1(a)).
- *Gray* : Gray anaglyphs use a black and white version of the original image. Although the color information is not preserved (as is with the color anaglyph) the gray anaglyph is typically easier to view (Figure B.1(b)).
- *Pure* : The pure anaglyph method converts the original image into a pure red/blue or red/green image (depending on the type of glasses you have). The pure method gives the best 3D effect but sacrifices the color data and image intensity (Figure B.1(c)).

B.2 Squished Side-by-Side Method

Some more expensive viewing systems utilize lens arrays to help guide the right and left images into the correct eye of the viewer. These systems typically do not require the use of glasses and are classified as autostereoscopic displays. Current autostereo displays use a squished side-by-side format (Figure B.2).

The draw back is that your head must be in a specific location in order to view the image. If your head is not in the correct location, a 3D image will not be seen.

B.3 Polar Method

The polar method requires two projectors covered with polarizing filters and a special grey screen which reflects light in a more direct way then a standard white screen (Figure B.3). The polar filter glasses that are used to view polar stereo images are very inexpensive and do not cause color loss or distortion like red/blue glasses.

(a) Color anaglyph

(b) Gray anaglyph

(c) Pure anaglyph

Figure B.1: Types of anaglyph method [Courtesy of Vrex Inc., http://www.vrex.com, 1997]. If you look at these pictures with RED–BLUE anaglyph glasses, you can see them in stereo.

Figure B.2: Squished Side-by-Side method [Courtesy of Vrex Inc., http://www.vrex.com, 1997].

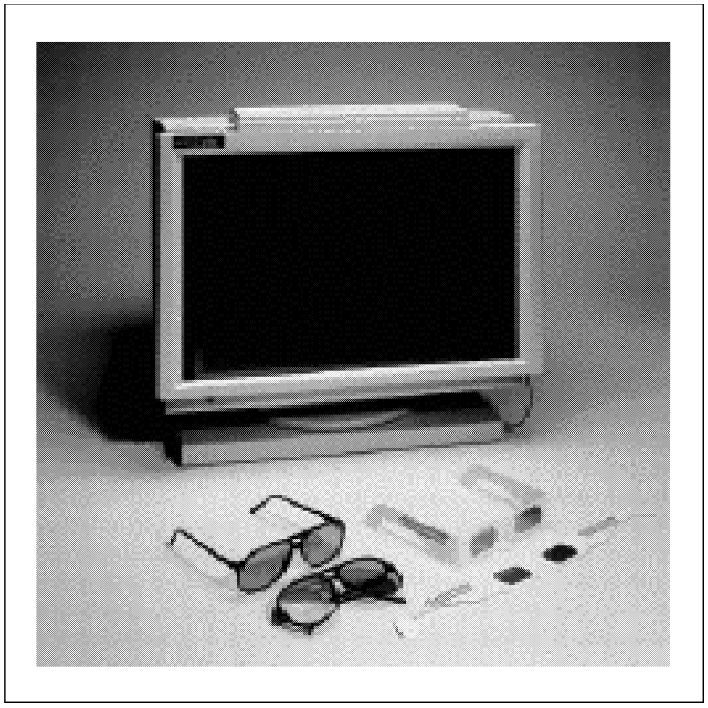


Figure B.3: Polar display equipment [Courtesy of Stereographics Co., http://www.stereographics.com, 1997].

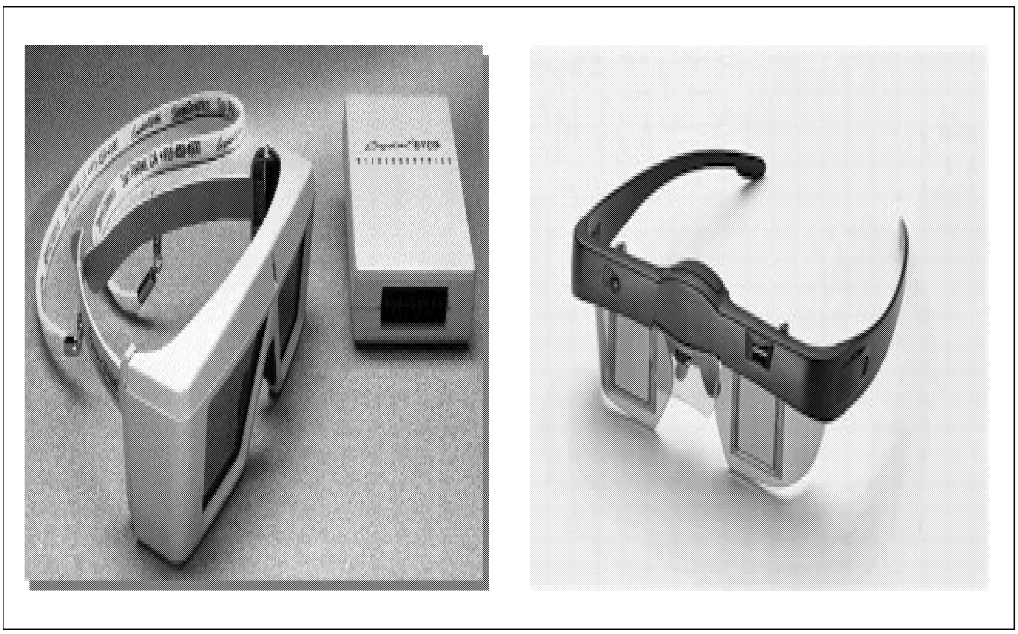


Figure B.4: Examples of shutter glasses [Courtesy of Stereographics Co., http://www.stereographics.com, 1997].

B.4 Shutter Glasses

Shutter glasses come in many forms (Figure B.4):

- Some have wires which connect to your video card.
- Some connect to your serial port or parallel printer port.
- Some are wire-less and use special transmitters which send out infra-red pulses to the glasses.

There are many view detection methods for shutter glasses. Some of them are explained in the following sections.

B.4.1 Page Flipped Methods

Page flipped stereoscopic images use a special feature of some video hardware to rapidly switch the monitor between the right and left images. A special pair of glasses must be used to view these images. The glasses have high-speed electronic shutters (typically made with Liquid Crystal material) which open and close in synchronization with the images on the monitor.

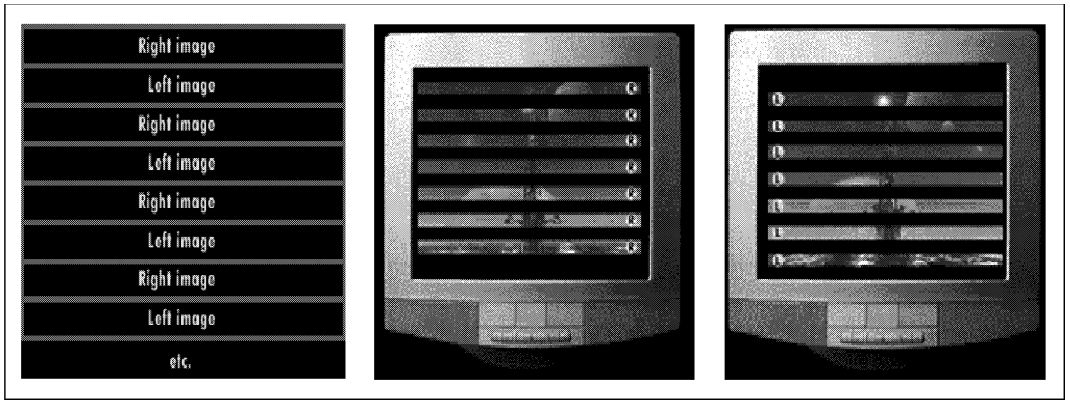


Figure B.5: Line alternate display method [Courtesy of Vrex Inc., http://www.vrex.com, 1997].

Page flipping methods allow you to see full color 3D stereoscopic image in high resolutions. The draw back is that typical video systems will exhibit some flicker. Special purpose video boards which support high-speed page flipping are available.

B.4.2 Line Alternate Methods

Line alternate stereoscopic images reformat the right and left images so that they are interleaved on a line-by-line basis. Each line alternates between the right and left image (Figure B.5).

B.4.3 Disadvantage of Page Flipped Methods : Flicker

Vertical scan rates of the monitors is very important when viewing a stereo image. The reason that if the rate is too slow, the viewer will be distracted by perceived flicker, the sense that the image is not solid, but appears and disappears as if the scene were lighted by candles blowing in a breeze. This effect is not only annoying, but it actually makes the viewer tired, dizzy, and can sometimes induce headaches. Even when refresh rates are fast enough that the viewer does not consciously notice the flicker, the unconscious or precognitive perception of flicker can produce the same stressing effects.

Anything less than 100 fields per second (50 pairs per second) is going to be uncomfortable for about 80% of the population. At 100 frames per second and above, frame sequential displays become acceptable to almost all of the general population with only a small fraction perceiving some flicker unconsciously.

At about 120 fields per second, a wonderful effect occurs; the flicker normally associated with frame sequential systems goes away for almost the entire population, both consciously and unconsciously. It is a phenomenon similar to the one discovered early in this century for the cinema; almost nobody sees flicker in (monoscopic) movies which are shown at 24 frames per second or more.

Appendix C

OpenGL

OpenGL (for *Open Graphics Library*) is a software interface to graphics hardware [20]. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects. Most of OpenGL requires that the graphics hardware contain a frame-buffer. Many OpenGL calls pertain to drawing objects such as points, lines, polygons, and bitmaps, but the way that some of this drawing occurs (such as when antialiasing or texturing is enabled) relies on the existence of a frame-buffer. Further, some of OpenGL is specifically concerned with frame-buffer manipulation.

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the frame-buffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn. A typical program that uses OpenGL begins with calls to open a window into the frame-buffer into which the program will draw.Then, calls are made to allocate a *GL* context and associate it with the window. Once a *GL* context is allocated, the programmer is free to issue OpenGL commands.

Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two– or three-dimensional model space to the two-dimensional screen. There are also calls to effect direct control of the frame-buffer, such as reading and writing pixels.

To the implementor, OpenGL is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable frame-buffer, then OpenGL must be implemented almost entirely on the host CPU. More typically, the graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering two- dimensional lines and polygons to sophisticated floating-point processors capable of transforming and computing on geometric data. The OpenGL implementor's task is to provide the CPU software interface while dividing the work for each OpenGL command between the CPU and the graphics hardware. This division must be tailored to the available graphics hardware to obtain optimum performance in carrying out OpenGL calls. OpenGL maintains a considerable amount of state information. This state controls how objects are drawn into the frame-buffer. Some of this state is directly available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this specification is to make OpenGL state information explicit, to elucidate how it changes, and to indicate what its effects are.

Appendix D

GLUI User Interface Library

D.1 Overview

GLUI is a GLUT-based C++ user interface library which provides controls such as buttons, checkboxes, radio buttons, spinners, and listboxes to OpenGL applications. We used *GLUI* to develop the user interface of the Stereoscopic View-Dependent Terrain Visualization System. *GLUI* library was developed by Paul Rademacher [18]. It is window-system independent, relying on GLUT to handle all system-dependent issues, such as window and mouse management. Features of the *GLUI* User Interface Library include:

- Complete integration with GLUT toolkit.
- Simple creation of a new user interface window with a single line of code.
- Support for multiple user interface windows.
- Standard user interface controls such as *buttons*, *checkboxes* for boolean variables, *Radio Buttons* for mutually-exclusive options, *editable* text boxes for inputting text, integers, and floating-point values, *spinners* for interactively manipulating integer and floating-point values *arcball* controllers for inputting rotation values.

- Controls can generate callbacks when their values change.
- Variables can be linked to controls and automatically updated when the value of the control changes.
- Controls can be automatically synchronized to reflect changes in live variables.
- Controls can trigger GLUT *redisplay* events when their values change.
- Layout and sizing of controls is automatic.

D.2 Background

The OpenGL Utility Toolkit (GLUT) is a popular user interface library for OpenGL applications. It provides a simple interface for handling windows, a mouse, keyboard, and other input devices. It has facilities for nested pop-up menus, and includes utility functions for bitmap and stroke fonts, as well as for drawing primitive graphics objects like spheres and teapots. Its greatest attraction is its window system independence, which (coupled with OpenGL's own window system independence) provides a very attractive environment for developing cross-platform graphics applications. Many applications can be built using only the standard GLUT input methods - the keyboard, mouse, and pop-up menus. However, as the number of features and options increases, these methods tend to be greatly overworked. It is not uncommon to find glut applications where almost every key on the keyboard is assigned to some function, and where the pop-up menus are large and cumbersome.

The *GLUI* User Interface Library addresses this problem by providing standard user interface elements such as buttons and checkboxes. The *GLUI* library is written entirely over GLUT, and contains no system-dependent code. A *GLUI* program will therefore behave the same on SGIs, Windows machines, Macs, or any other system to which GLUT has been ported. Furthermore, *GLUI* has been designed for programming simplicity, allowing user interface elements to be added with one line of code each. A sample *GLUI* window is shown in Figure D.1.

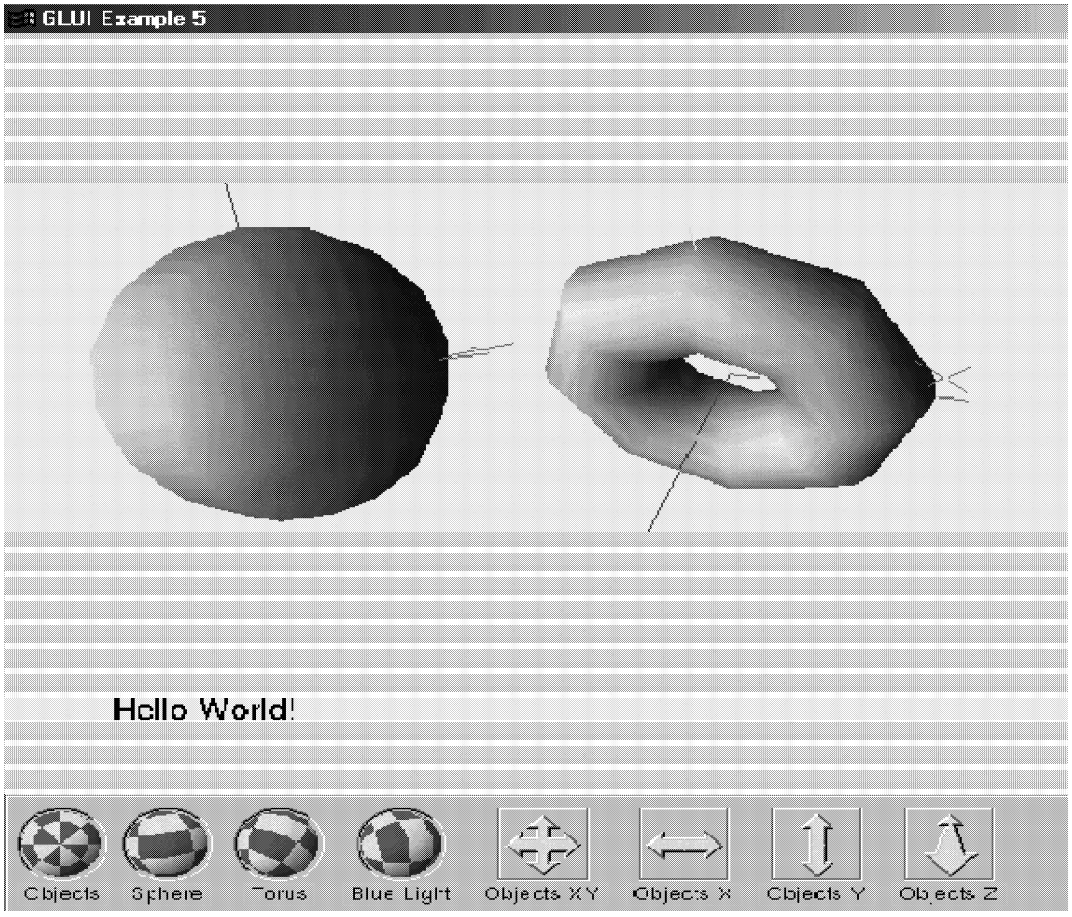


Figure D.1: A sample *GLUI* window [Snapshot of the 5^{th} example program distributed with *GLUI* Library V2.0].

D.3 Properties

GLUI is intended to be a simple yet powerful user interface library. This section describes in more detail its main features, including a flexible API, easy and full integration with GLUT, live variables, and callbacks.

D.3.1 Programming Interface

GLUI has been designed for maximum programming simplicity. New *GLUI* windows and new controls within them can be created with a single line of code each. *GLUI* automatically sizes controls and places them within their windows. The programmer does not need to explicitly give X, Y, width, and height parameters for each control -an otherwise cumbersome task.

GLUI provides default values for many parameters in the API. This way, one does not need to place NULL or dummy values in the argument list when some feature are not needed.

D.3.2 Full Integration with GLUT

GLUI is built on top of - and meant to fully interact with - the GLUT toolkit. Existing GLUT applications therefore need very little change in order to use the user interface library. Once integrated, the presence of a user interface will be mostly transparent to the GLUT application.

D.3.3 Live Variables

GLUI can associate live variables with most types of controls. These are regular C variables that are automatically updated whenever the user interacts with a *GLUI* control. For example, a checkbox may have an associated integer variable, to be automatically toggled between one and zero whenever the user checks or unchecks the control. An editable text control may maintain an entire character array as a live variable, such that anything the user types into the text box is automatically copied into the application's character array. This eliminates the need for the programmer to explicitly query each control's state to determine their current value or contents. In addition, a *GLUI* window can send a GLUT redisplay message to another window (i.e., a main graphics window) whenever a value in the interface is changed. This will cause that other window to redraw, automatically using the new values of any live variables. For example, a *GLUI* window can have a spinner to manipulate the radius of an on-screen object. When the user changes the spinner's value, a live variable (say, float radius) is automatically updated, and the main graphics window is sent a redisplay message. The graphics window then redraws itself, using the current (that is, the updated) value of radius - unaware that it was changed since the last frame. Live variables help make the *GLUI* interface transparent to the rest of the application. Live variables are automatically updated by *GLUI* whenever the user interacts with a control.

One can synchronize live variables. This procedure will check the current value of all live variables in a *GLUI* window, and compare them with the controls' current values. If a pair does not match (that is, the user changed a live variable without telling *GLUI*), then the control is automatically updated to reflect the variable. Thus, one can make a series of changes to variables in memory, and then use the single function call *sync_live()* to synchronize the user interface:

```
radius = radius * .05;          // Make changes to a group of
aperture = aperture + .1;       // variables that are linked
num_segments++;                 // Update user interface
toglui->sync_live();            // to reflect these changes
```

If a pointer to a live variable is passed to a control creation function (e.g., add_checkbox()), then the current value of that variable will be used as the initial value for the control. Thus, remember to always properly initialize live variables (including strings), before passing them to a control creation function.

D.3.4 Callbacks

GLUI can also generate callbacks whenever the value of a control changes. Upon creation of a new control, one specifies a function to use as a callback, as well as an integer ID to pass to that function when the control's value changes. A single function can handle callbacks for multiple controls by using a switch statement to interpret the incoming ID value within the callback.

D.3.5 API

The *GLUI* library consists of 3 main classes:

- GLUI_Master_Object,
- GLUI,
- GLUI_Control.

There is a single global GLUI_Master_Object object, named GLUI_Master. All *GLUI* window creation must be done through this object. This lets the *GLUI* library track all the windows with a single global object.

The GLUI_Master is also used to set the GLUT Idle function, and to retrieve the current version of *GLUI*.

Appendix E

Performance Graphics

In this appendix we give the figures related with the performances of the proposed algorithms. The results in these figures are *actual*, meaning that the recorded results are given without a regression function being applied. The figures in the Chapter 5 are smoothed in order to give a clear idea about the results of the algorithms. The figures presented in this appendix are given with actual values in order to be precise about the results of the empirical study.

Figure E.1 shows the performance comparisons of different types of visualization techniques by using different morphing, culling and rendering techniques at different parts of the flythrough and the number of polygons rendered for each eye is given in Figure E.2. The reason for sudden changes in the polygon count is that the viewer gets close to the edges and corners of the terrain. In Figures E.3, E.4 and E.5 performance comparison showing the frame rates of our culling techniques with each visualization method are given with the recorded values. In Figures E.6, E.7, E.8 and E.9 the performances of the visualization methods for each of the proposed culling schemes are given.

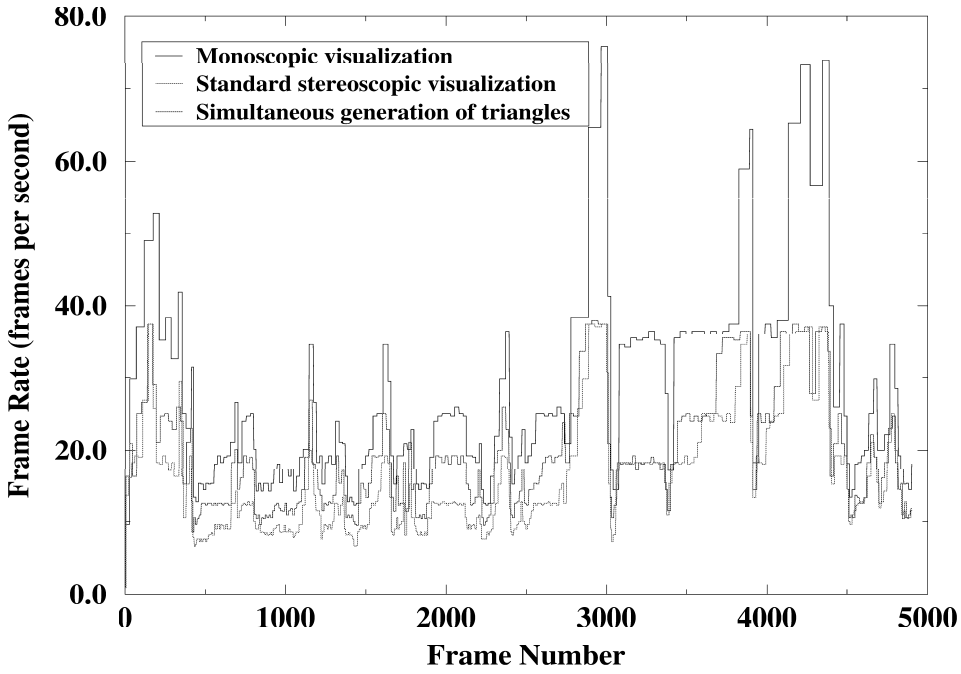


Figure E.1: Comparison of the frame rates of different types of visualizations (ACTUAL).

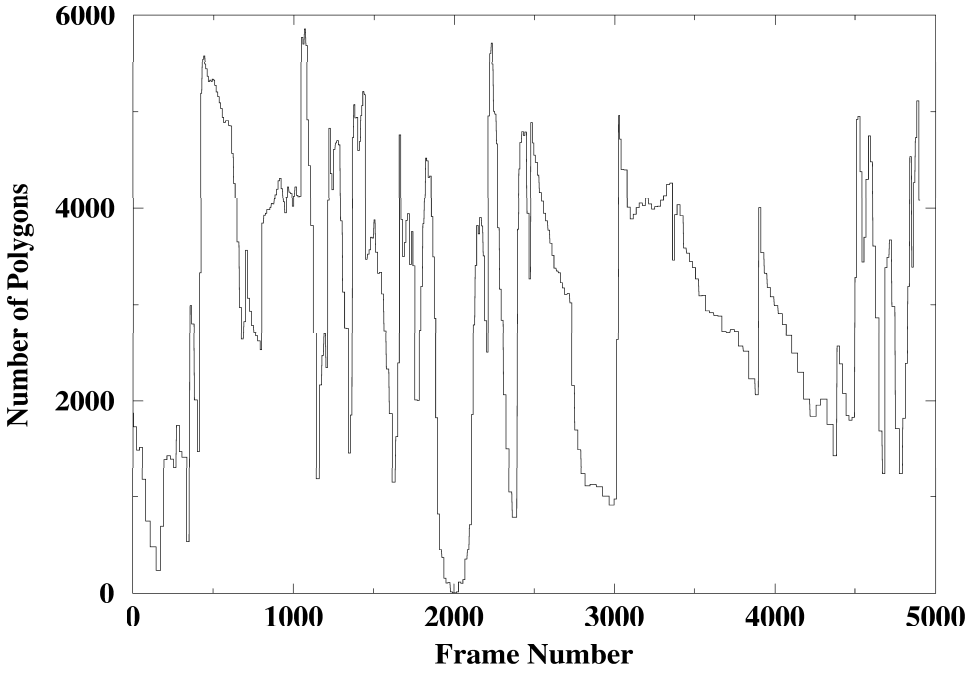
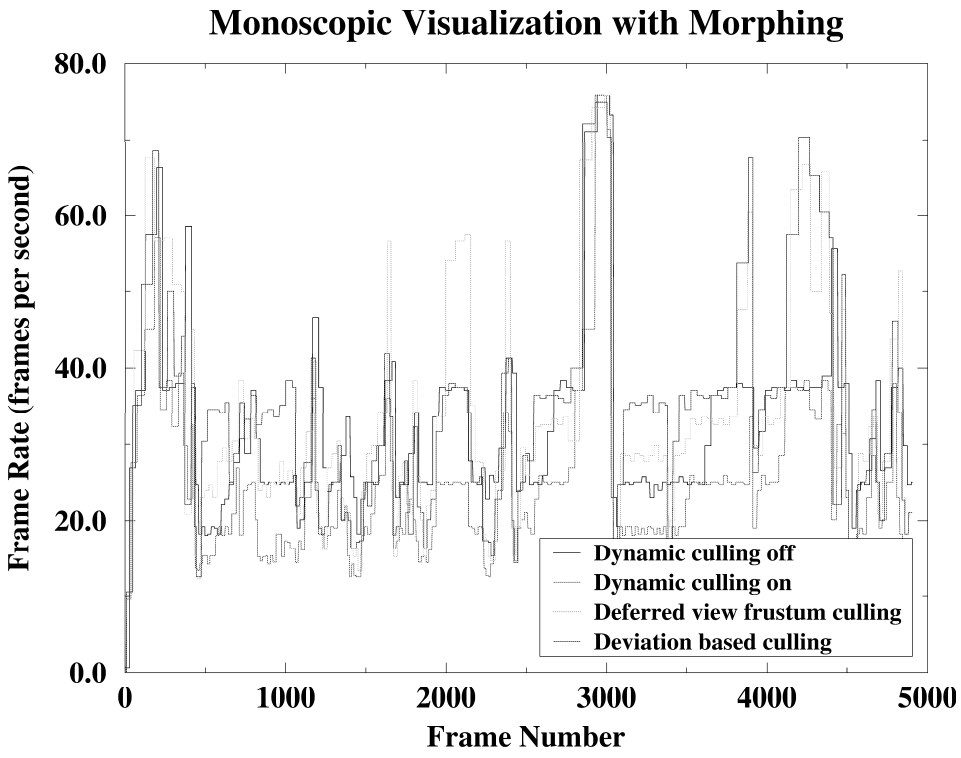
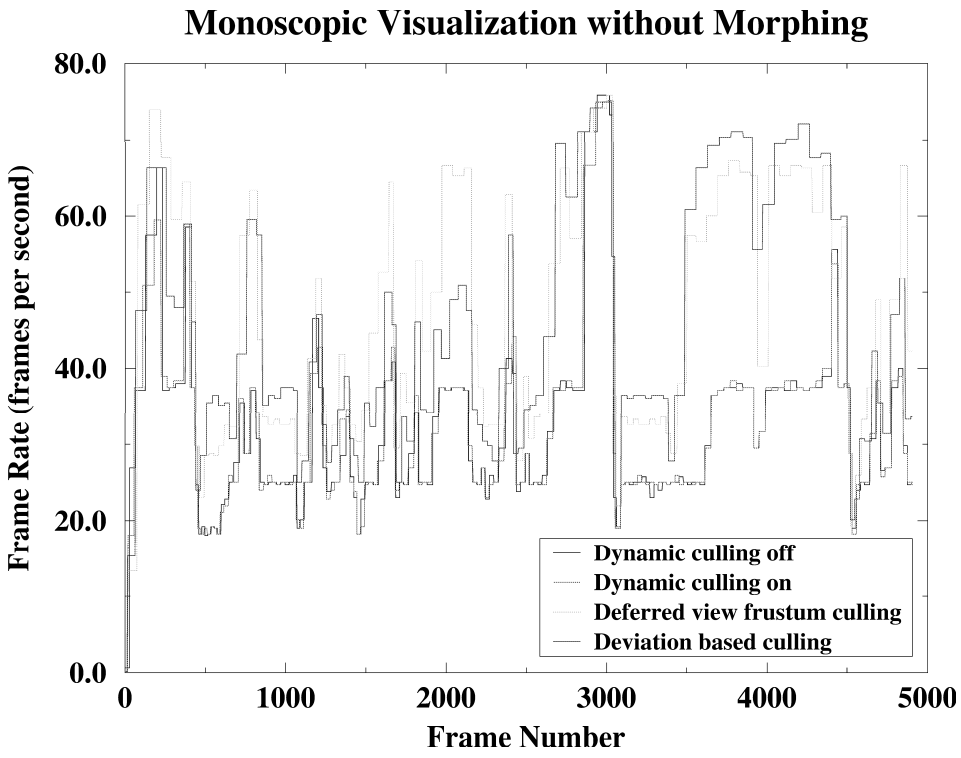


Figure E.2: Number of polygons for the experimental visualization (ACTUAL)

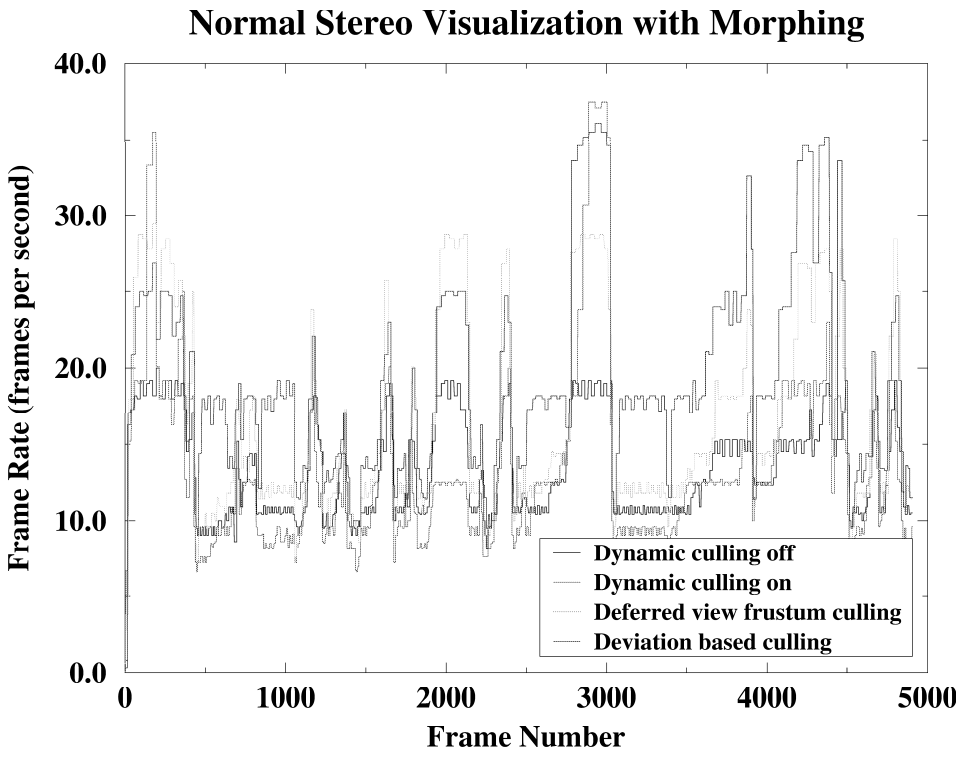


(a)

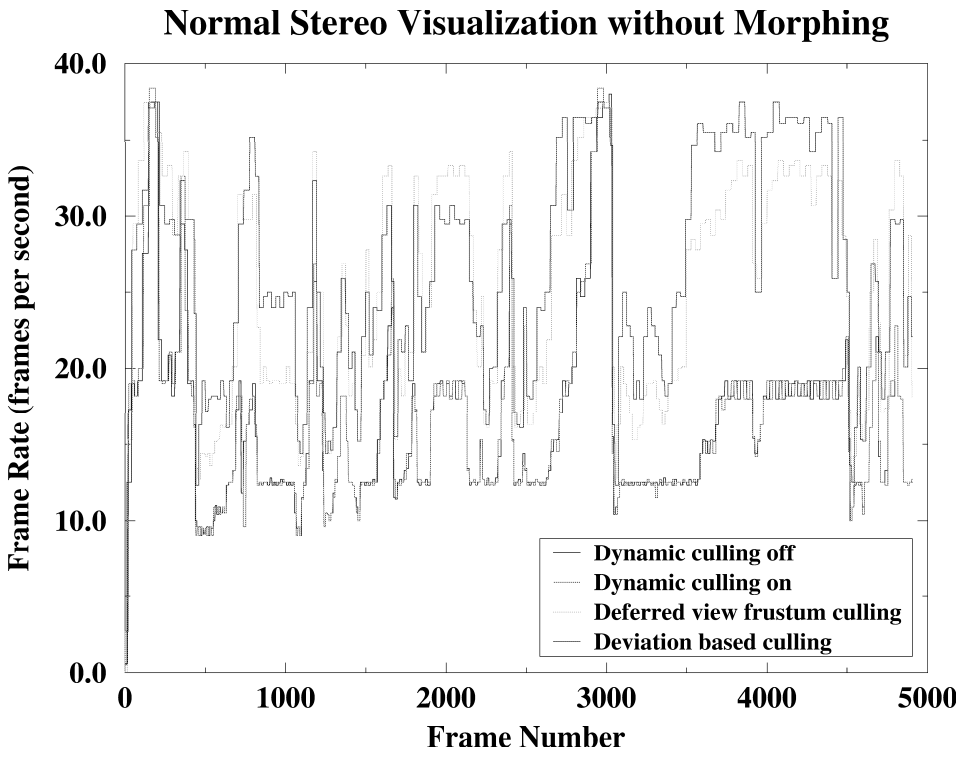


(b)

Figure E.3: Comparison of the frame rates for visualization types with different morphing/culling options (ACTUAL): (a) *monoscopic visualization* with morphing; (b) *monoscopic visualization* without morphing.

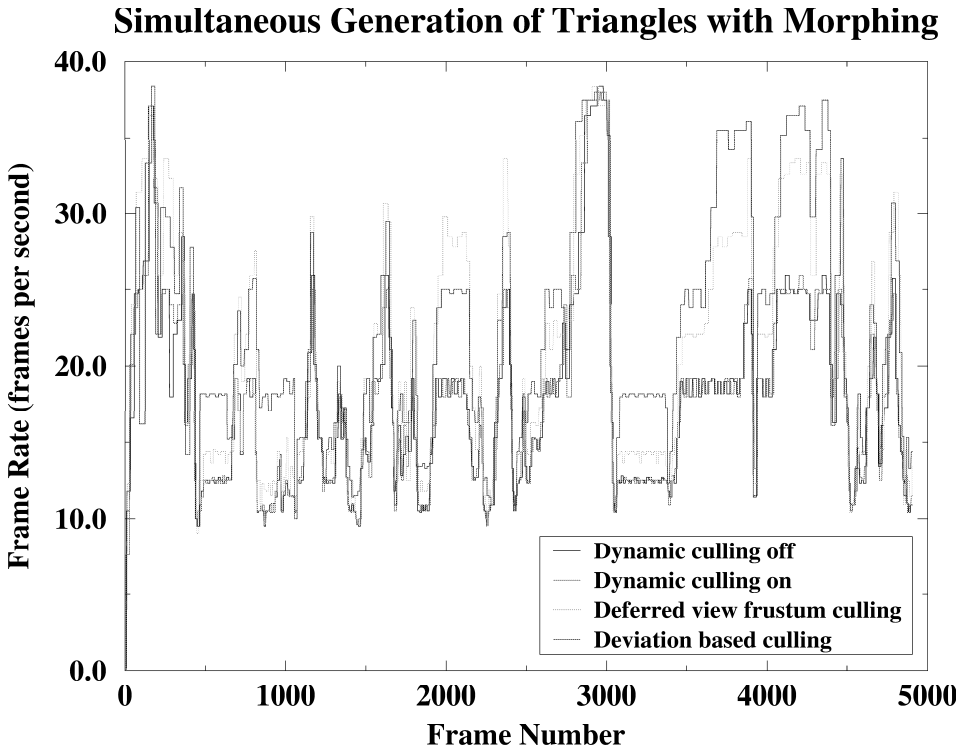


(a)

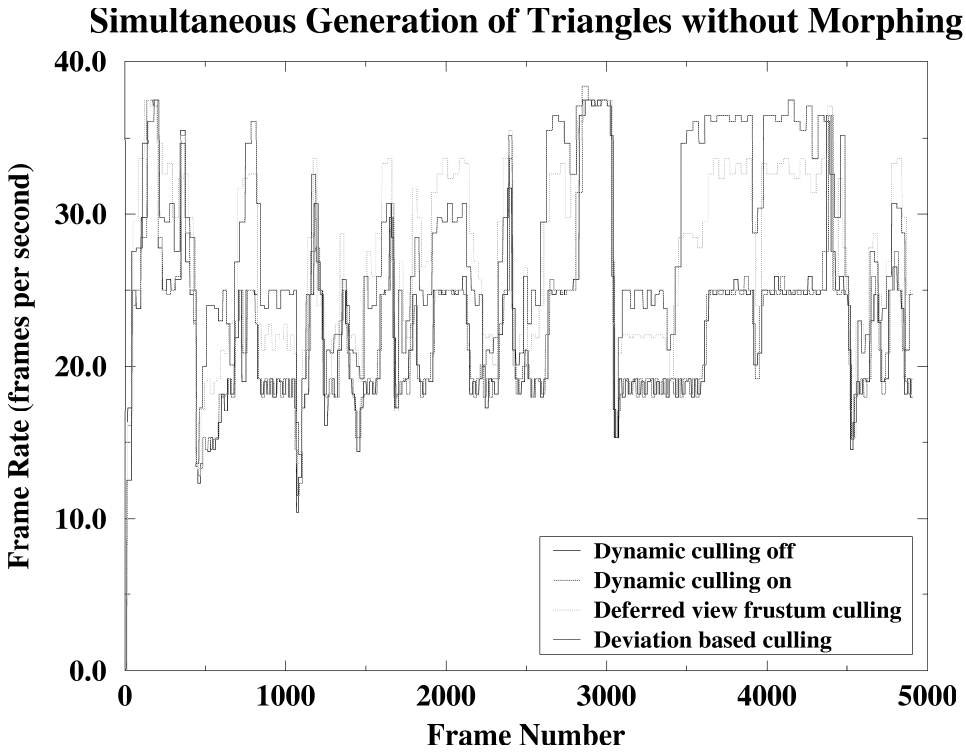


(b)

Figure E.4: Comparison of the frame rates for visualization types with different morphing/culling options (ACTUAL): (a) *standard stereoscopic visualization* with morphing; (b) *standard stereoscopic visualization* without morphing.

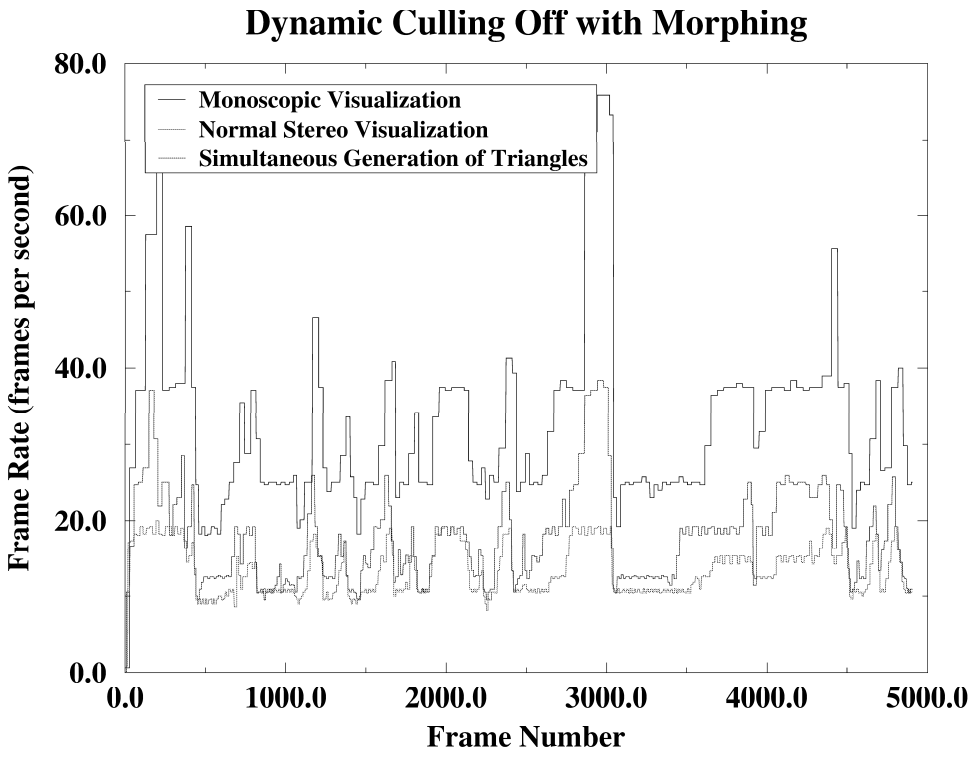


(a)

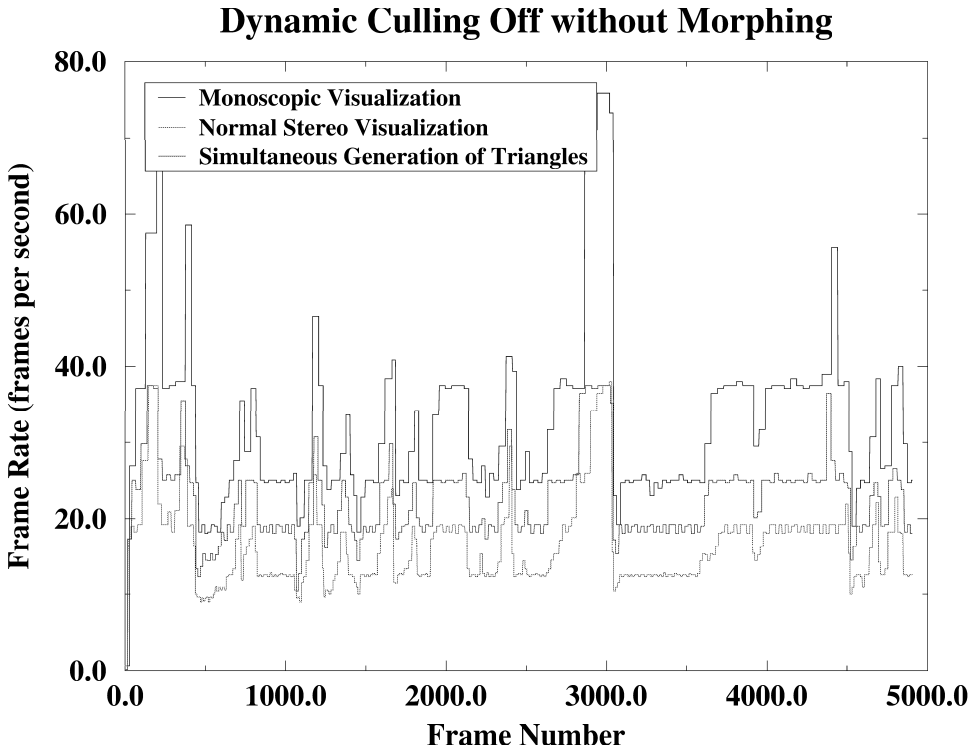


(b)

Figure E.5: Comparison of the frame rates for visualization types with different morphing/culling options (ACTUAL): (a) *simultaneous generation of triangles* with morphing; (b) *simultaneous generation of triangles* without morphing.

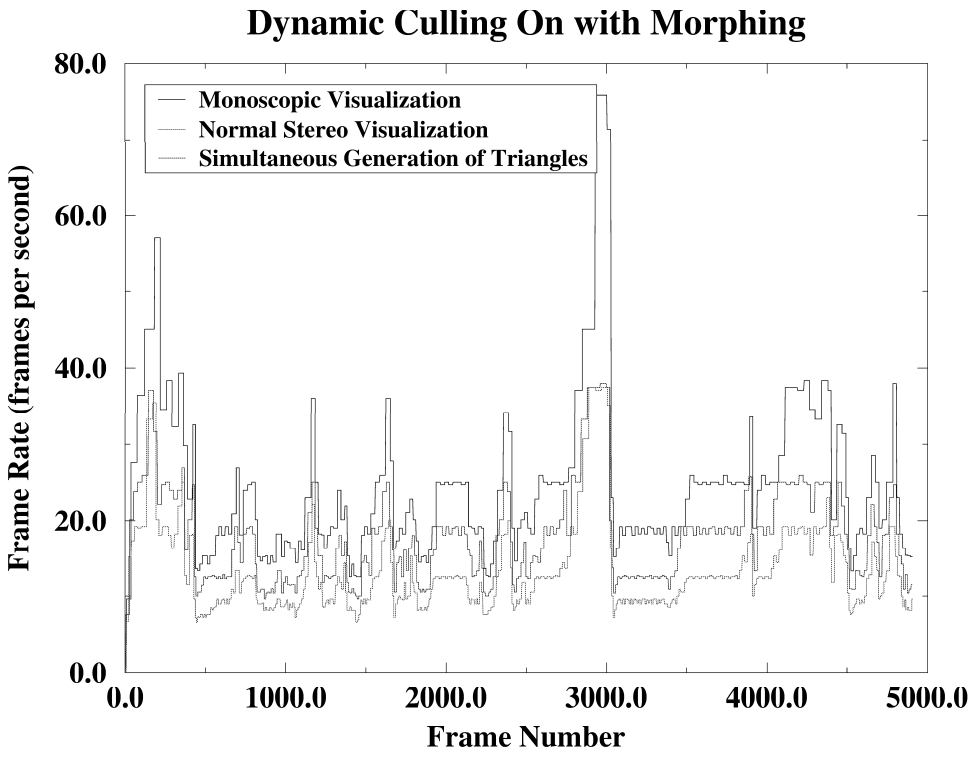


(a)

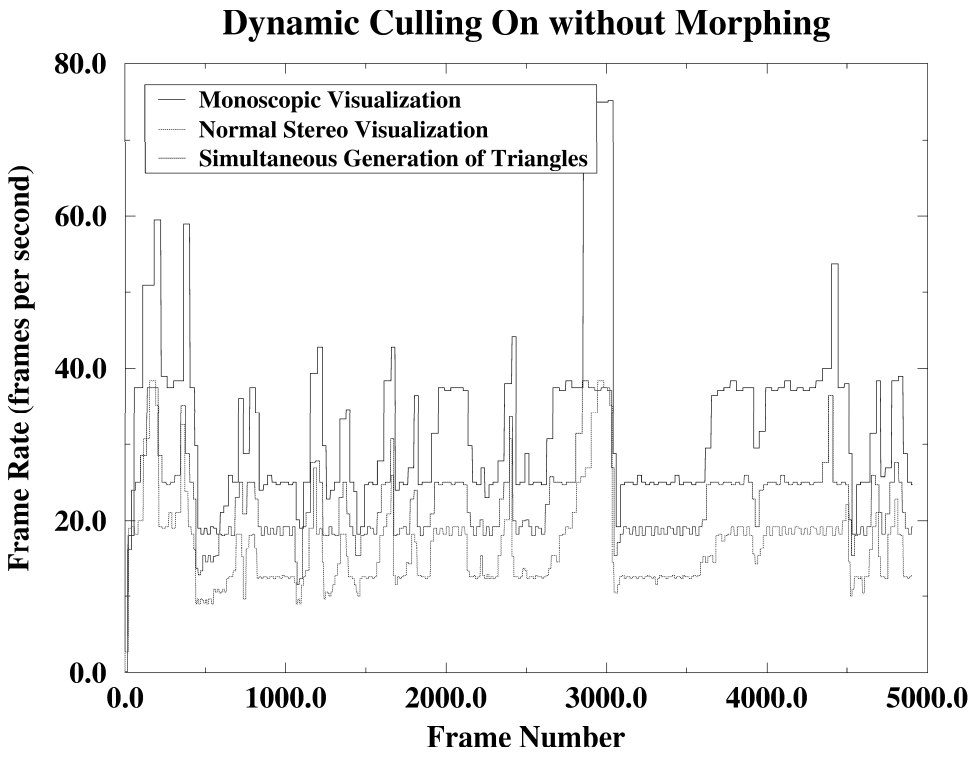


(b)

Figure E.6: Comparison of different culling schemes for visualization types (ACTUAL): (a) *dynamic culling off* with morphing; (b) *dynamic culling off* without morphing.

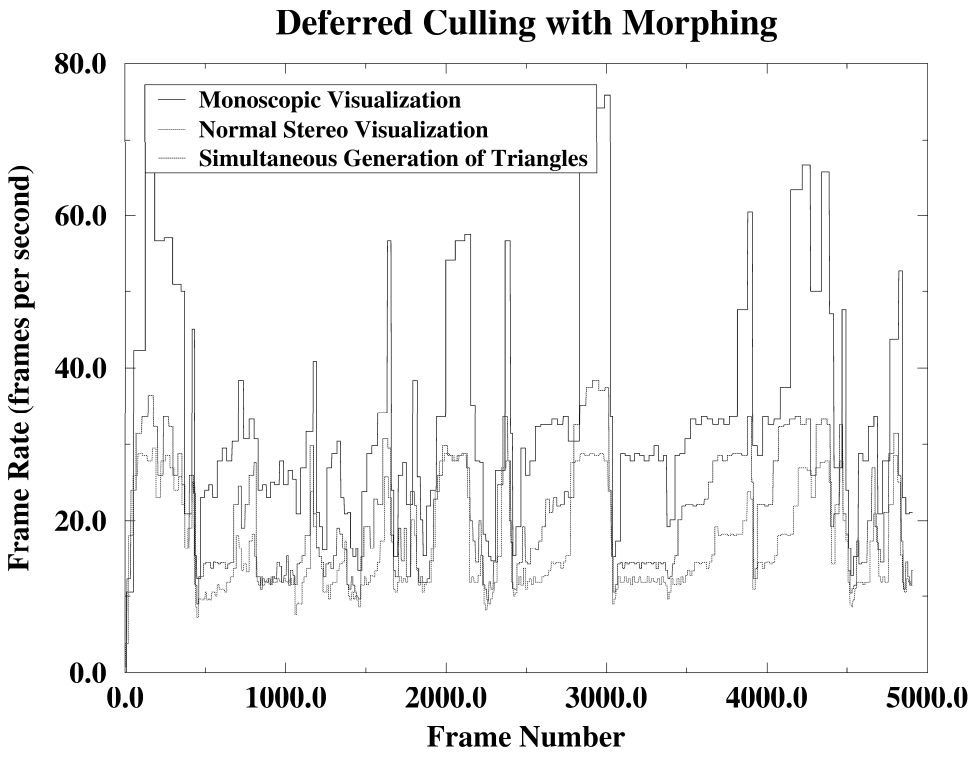


(a)

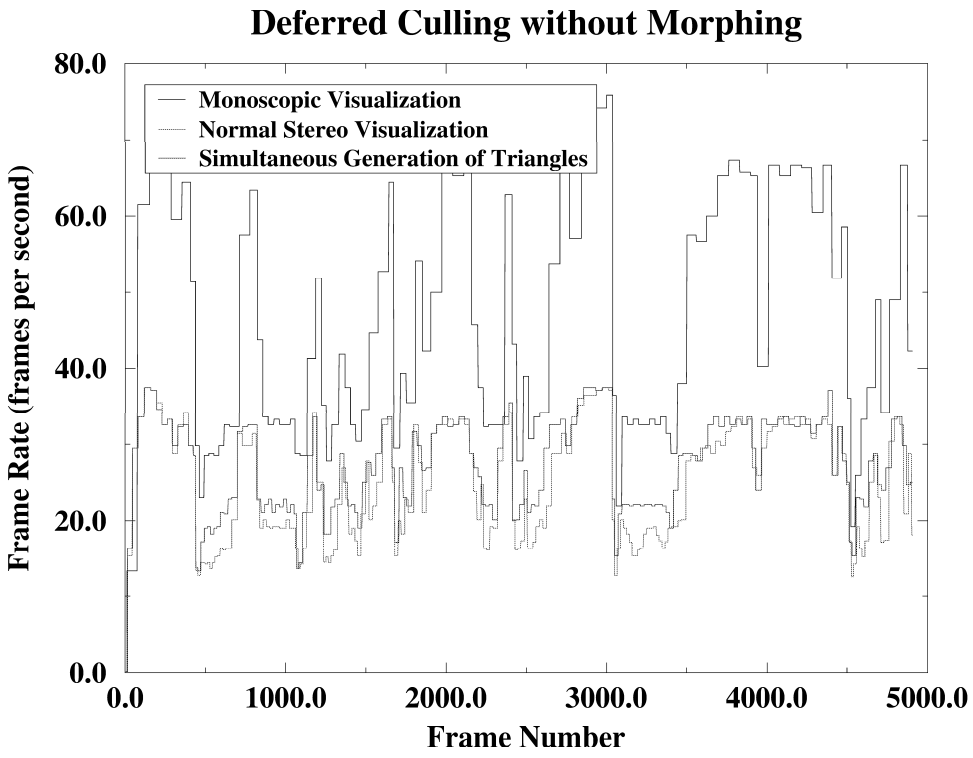


(b)

Figure E.7: Comparison of different culling schemes for visualization types (ACTUAL): (a) *dynamic culling on* with morphing; (b) *dynamic culling on* without morphing.

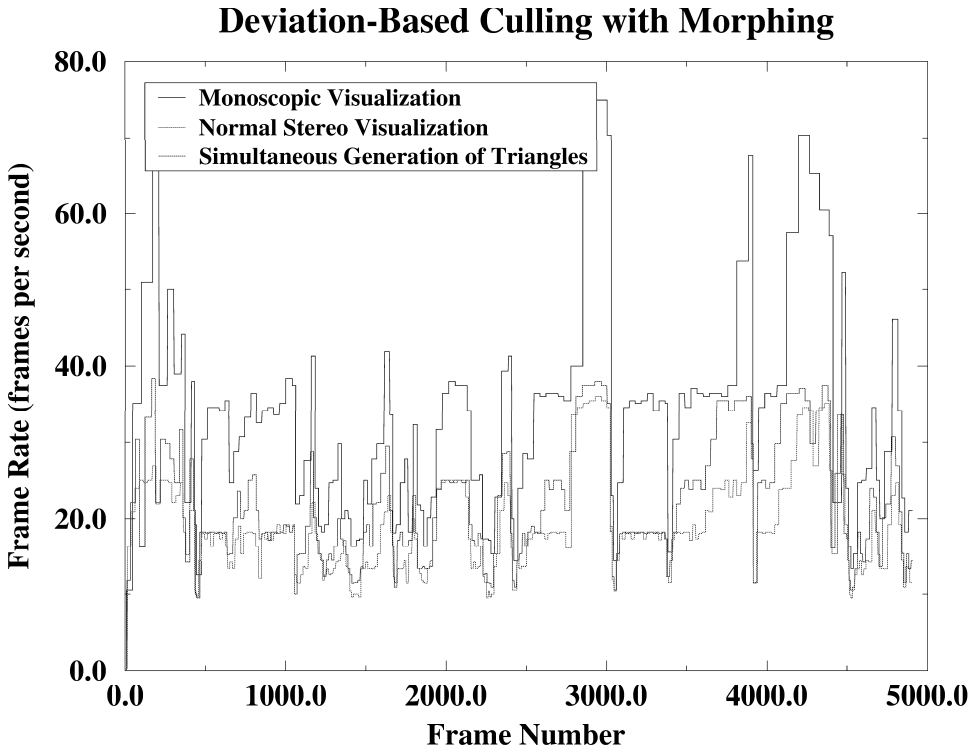


(a)

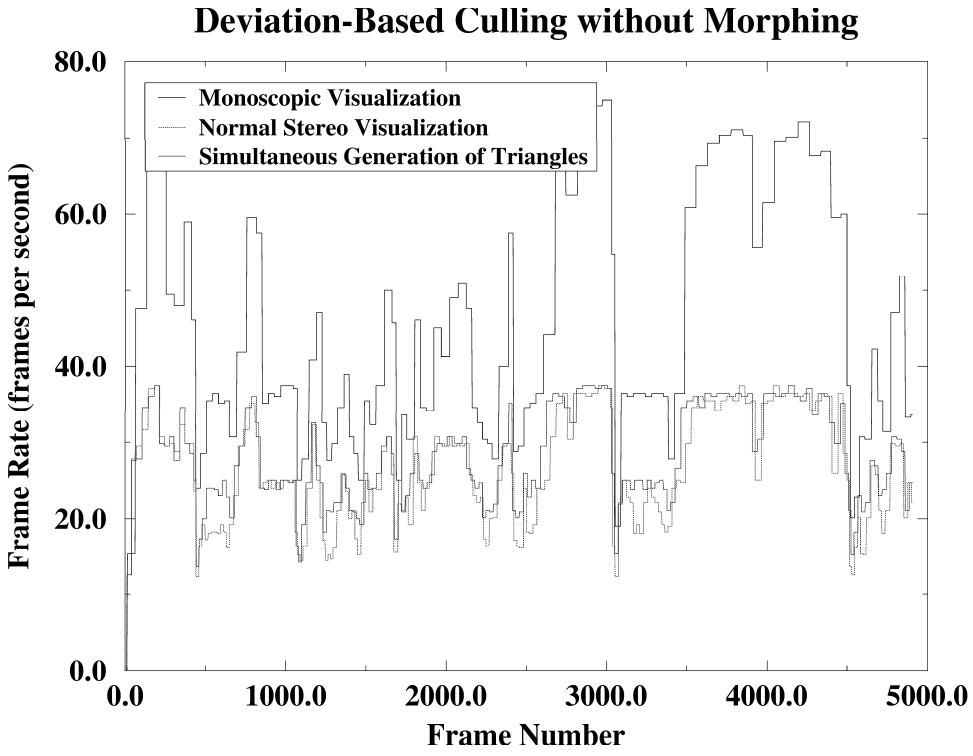


(b)

Figure E.8: Comparison of different culling schemes for visualization types (ACTUAL): (a) *deferred culling* with morphing; (b) *deferred culling* without morphing.



(a)



(b)

Figure E.9: Comparison of different culling schemes for visualization types (ACTUAL): (a) *deviation-based culling* with morphing; (b) *deviation-based culling* without morphing.

Bibliography

[1] S. Adelson, J. Bentley, I. Chong, L. Hodges, and J. Winograd. Simultaneous generation of stereographic views. *Computer Graphics Forum*, 10:3–10, 1991.

[2] S. Adelson and C. Hansen. Fast stereoscopic images with ray traced volume rendering. In *Proc. of Symposium on Volume Visualization*, pages 3–9, 1994.

[3] S. Adelson and L. Hodges. Stereoscopic ray tracing. *the Visual Computer*, 10(3):127–144, 1993.

[4] U. Assarsson and T. Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1):9–22, 2000.

[5] D. Bartz, M. Meisner, and T. Hüttner. OpenGL-assisted occlusion culling for large polygonal models. *Computers & Graphics*, 23(5):667–679, 1999.

[6] J. Ezell and L. Hodges. Some preliminary results on using spatial locality to speed-up raytracing of stereoscopic images. In *Proc. of SPIE 1256, Stereoscopic Dispaly and Applications I*, pages 298–306, 1990.

[7] L. Hodges. Tutorial: Time-multiplexed stereoscopic computer graphics. *IEEE Computer Graphics and Applications*, 12(2):20–30, 1992.

[8] L. Hodges and D. McAllister. Stereo and alternating-pair techniques for display of computer-generated images. *IEEE Computer Graphics and Applications*, 5(9):38–45, 1985.

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

[10] R. Hubbold, D. Hancock, and C. Moore. Stereoscopic volume rendering. In *Proc. Visualization in Scientific Computing'98*, pages 105–115, 1998.

[11] P. Lindstrom. 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 Institute of Technology, GA, USA, 1995.

[12] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. In *ACM Computer Graphics (Proc. of SIGGRAPH'96)*, pages 109–118, 1996.

[13] L. Lipton. Binocular symmetries as criteria for the successful transmission of images. In *Proc. of SPIE'84*, volume 507, 1984.

[14] L. Lipton. Factors affecting ghosting in a time-multiplexed plano-stereoscopic CRT display system. In *Proc. of SPIE'87*, volume 507, 1987.

[15] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison–Wesley, 1994.

[16] R. Nielson, D. Holliday, and T. Roxborough. Cracking the cracking problem with Coons patches. In *Proc. of IEEE Visualization'99*, pages 285–290, 1999.

[17] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proc. of IEEE Visualization'98*, pages 19–26, 1998.

[18] P. Rademacher. *GLUI – A GLUT Based User Interface Library Ver2.0*. http://www.cs.unc.edu/∼rademach/glui/, 1999.

[19] H. Samet. The quadtree and related data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[20] M. Segal and K. Akeley. *OpenGL Graphics System: A Specification Ver1.2.1*. Silicon Graphics Inc., CA, April 1999. Unpublished.

[21] C. Stereographics. *Stereographics Developer's Handbook*. Stereographics Corporation, http://www.stereographics.com, 1997.

[22] H. Taosong and A. Kaufman. Fast stereo volume rendering. In *Proc. of IEEE Visualization'96*, pages 49–56, 1996.

[23] N. Valyus. *Stereoscopy.* Focal Press, New York, first edition, 1962.

[24] I. Vrex. *How To View A Stereoscopic Image.* Vrex Incorporation, http://www.vrex.com, 1997.

[25] C. Wheatstone. On some remarkable, and hitherto unobserved, phenomena of binocular vision. *Philosophical Transactions of the Royal Society of London*, pages 371–394, 1838.