LOD-Sprite Technique for Accelerated Terrain Rendering

Baoquan Chen[1]
SUNY at Stony Brook

J. Edward Swan II[2]
Naval Research Lab

Eddy Kuo [2]
Naval Research Lab

Arie Kaufman[1]
SUNY at Stony Brook

Abstract

We present a new rendering technique, termed *LOD-sprite* rendering, which uses a combination of a level-of-detail (LOD) representation of the scene together with reusing image *sprites* (previously rendered images). Our primary application is accelerating terrain rendering. The LOD-sprite technique renders an initial frame using a high-resolution model of the scene geometry. It renders subsequent frames with a much lower-resolution model of the scene geometry and texture-maps each polygon with the image sprite from the initial high-resolution frame. As it renders these subsequent frames the technique measures the error associated with the divergence of the view position from the position where the initial frame was rendered. Once this error exceeds a user-defined threshold, the technique re-renders the scene from the high-resolution model. We have efficiently implemented the LOD-sprite technique with texture-mapping graphics hardware. Although to date we have only applied LOD-sprite to terrain rendering, it could easily be extended to other applications. We feel LOD-sprite holds particular promise for real-time rendering systems.

Keywords: Image-Based Modeling and Rendering, Texture Mapping, Acceleration Techniques, Multi-Resolution, Level of Detail, Terrain Rendering, Virtual Reality, Virtual Environments.

1 INTRODUCTION

As scene geometry becomes complex (into the millions of polygons), even the most advanced rendering hardware cannot provide interactive rates. Current satellite imaging technology provides terrain datasets which are well beyond this level of complexity. This presents two problems for real-time systems: 1) the provided frame rate may be insufficient, and 2) the system latency may be too high. Much of real-time computer graphics has been dedicated to finding ways to trade off image quality for frame rate and/or system latency. Many recent efforts fall into two general categories:

Level-of-detail (LOD): These techniques model the objects in the scene at different levels of detail. They select a particular LOD for each object based on various considerations such as the rendering cost and perceptual contribution to the final image.

Image-based modeling and rendering (IBMR): These techniques model (some of the) objects in the scene as image sprites. These sprites only require 2D transformations for most rendering operations, which, depending on the object, can result in substantial time savings. However, the 2D transformations eventually result in distortions which require the underlying objects to be re-rendered from their full 3D geometry. IBMR techniques typically organize
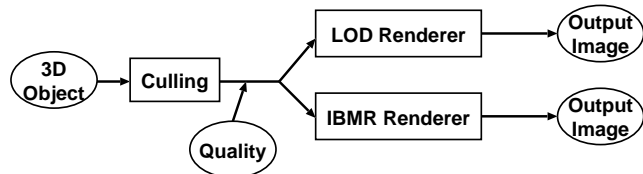
[1]Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA. Email: {baoquan|ari}@cs.sunysb.edu

[2]Virtual Reality Laboratory, Naval Research Laboratory Code 5580, 4555 Overlook Ave SW, Washington, DC, 20375-5320, USA. Email: swan@acm.org, ekuo@homemail.com

Figure 1: *Traditional hybrid LOD and IBMR techniques render each object either as a sprite or at a certain level of detail.*
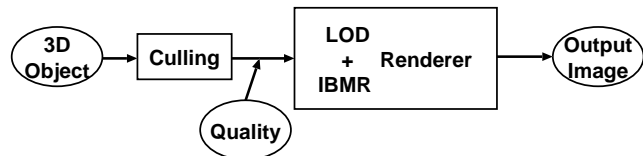

Figure 2: *The LOD-sprite technique renders each object as both a sprite and as a geometric object at a certain level of detail.*

the scene into separate non-occluding layers, where each layer consists of an object or a small group of related objects. They render each layer separately, and then alpha-channel composite them.

Some hybrid techniques use both multiple LODs and IBMR methods [16, 27, 22]. A general pipeline of these techniques is shown in Figure 1. Each 3D object is first subjected to a culling operation. Then, depending upon user-supplied quality parameters, the system either renders the object at a particular LOD, or it reuses a cached sprite of the object.

This paper presents the *LOD-sprite* rendering technique. As shown in Figure 2, the technique is similar to previous hybrid techniques in that it utilizes view frustum culling and a user-supplied quality metric. Objects are also modeled as both LOD models and sprites. However, the LOD-sprite technique differs in that the 2D sprite is *coupled* with the LOD representation; the renderer utilizes both the LOD and the sprite as the inputs to create the output image. The LOD-sprite technique first renders a frame from high-resolution 3D scene geometry, and then caches this frame as an image sprite. It renders subsequent frames by texture-mapping the cached image sprite onto a lower-resolution representation of the scene geometry. This continues until an image quality metric requires again rendering the scene from the high-resolution geometry.

We have developed the LOD-sprite technique as part of the rendering engine for a real-time, three-dimensional battlefield visualization system [9]. For this application the terrain database consumes the vast majority of the rendering resources, and therefore in this paper our focus is on terrain rendering. However, LOD-sprite is a general-purpose rendering technique and could certainly be applied to many different types of scene geometry.

The primary advantage of LOD-sprite over previous techniques is that when the sprite is transformed, if the 2D transformation is within the context of an underlying 3D structure (even if only com-

posed of a few polygons), a much larger transformation can occur before image distortions require re-rendering the sprite from the full 3D scene geometry. Thus, the LOD-sprite technique can reuse image sprites for a larger number of frames than previous techniques. In addition, because the sprite preserves object details, a lower LOD model can be used for the same image quality. These properties allow interactive frame rates for larger scene databases.

The next section of this paper places LOD-sprite in the context of previous work. Section 3 describes the LOD-sprite technique itself. Section 4 presents the results of our implementation of LOD-sprite.

2 RELATED WORK

The previous work which is most closely related to LOD-sprite can be classified into *image-based modeling and rendering* techniques and *level-of-detail* techniques. We first revisit and classify previous IBMR techniques while also considering LOD techniques, and then focus on some hybrid techniques.

2.1 Image-Based Modeling and Rendering

Previous work in image-based modeling and rendering falls primarily into three categories:

(1) The scene is modeled by 2D image sprites; no 3D geometry is used. Many previous techniques model the 3D scene by registering a number of static images [2, 18, 19, 26]. These techniques are particularly well-suited for applications where photographs are easy to take but modeling the scene would be difficult (outdoor settings, for example). Novel views of the scene are created by 2D transforming and interpolating between images [3, 18]. By adding depth [17] or even layered depth [23] to the sprites, more realistic navigation, which includes limited parallax, is possible. Another category samples the full *plenoptic function*, resulting in 3D, 4D or even 5D image sprites [13, 10], which allow the most unrestricted navigation of this class of techniques. However, all of these techniques lack the full 3D structure of the scene, and so restrict navigation to at least some degree.

(2) The scene is modeled using either 3D geometry *or* 2D image sprites. Another set of previous techniques model each object with either 3D geometry or a 2D image sprite, based on object contribution to the final image and / or viewing direction [5, 16, 20, 21, 22, 27]. The LOD-sprite technique differs from these techniques in that it integrates both 3D geometry and 2D image sprites to model and render objects.

(3) The scene is modeled using a combination of 3D geometry *and* 2D image sprites. There are a group of techniques which add very simple 3D geometry to a single 2D image [6, 7, 12, 24], which guides the subsequent image warping. Debevec et al. [7] construct a 3D model from reference images, while Sillion et al. [24] and Darsa et al. [6] use a textured depth mesh which is constructed and simplified from depth information. In general, using a depth mesh with projective texture mapping gives better image quality than using depth image warping [17], because the mesh stretches to cover regions where no pixel information is available, and thus no holes appear. The main advantage of adding 3D scene geometry to the image is that it allows the warping to approximate parallax, and therefore increases the range of novel views which are possible before image distortion becomes too severe.

Our LOD-sprite is most closely related to the techniques of Cohen et al. [4] and Soucy et al. [25]. Both create a texture map from a 3D object represented at a high geometric resolution, and then subsequently represent the object at a much lower geometric resolution, but apply the previously created texture map to the geometry.

However, the LOD-sprite technique generates texture maps (image sprites) from images rendered at run-time, while these techniques generate the texture map from the object itself.

2.2 Level-of-Detail

There is a large body of previous work in level-of-detail (LOD) techniques, which is not reviewed here. The general LOD-sprite technique requires that geometric objects be represented at various levels of detail, but it does not require any particular LOD representation or technique (although a specific implementation of LOD-sprite will need to access the underlying LOD data structures).

This paper does not cover how to create LOD representations of a terrain — there exist numerous multiresolution representations for height fields. Lindstrom et al. [14] and Hoppe [11] represent the most recent view-dependent terrain LOD methods, and Luebke and Erikson [15] can also be adapted for terrain datasets. In this paper we adopt the technique of Lindstrom et al. [14]. This algorithm organizes the terrain mesh into a hierarchical quadtree structure. To decide which quadrant level to use, the algorithm computes a screen space error for each vertex, and compares it to a pre-defined error threshold. This error measures the pixel difference between the full-resolution and lower-resolution representations of the quadrant.

2.3 Accelerated Virtual Environment Navigation

As stated above, many LOD and IBMR techniques have been applied to the problem of accelerating virtual environment navigation. Of these, LOD-sprite is most closely related to the techniques of Maciel and Shirley [16], Shade et al. [22], Schaufler and Stuerzlinger [21], and Aliaga [1]. All of these papers present similar hybrid LOD/IBMR techniques. They create a hierarchy of image sprites based on a space partition of the scene geometry. In subsequent frames, for each node the techniques either texture map the node sprite onto a polygon, or re-render the node's 3D geometry if an error metric is above a threshold. Each reused image sprite means an entire subtree of 3D geometry need not be rendered, which yields substantial speedup for navigating large virtual environments. The main limitation of these techniques is that creating a balanced space partition is not a quick operation, and it must be updated if objects move. Also, to avoid gaps between neighboring partitions, they either maintain a fairly large amount of overlap between partitions [22], or they morph geometries to guarantee a smooth transition between geometry and sprite [1]; both operations add storage and computational complexity. LOD-sprite differs from these techniques in that they interpolate the image sprite on a single 2D polygon, while LOD-sprite interpolates the image sprite on a coarse representation of the 3D scene geometry.

3 THE LOD-SPRITE TECHNIQUE

3.1 Algorithm

The general idea of the LOD-sprite technique is to cache the rendered view of a high-resolution representation of the dataset. We refer to this image as a *sprite*, and the frame where the sprite is created as a *keyframe*. LOD-sprite renders subsequent frames, referred to as *novel* views, at a lower resolution, but applies the sprite as a texture map. LOD-sprite measures the error caused by the divergence of the viewpoint from the keyframe as each novel view is rendered. When this error exceeds a threshold, LOD-sprite renders a new keyframe.

Pseudocode for the LOD-sprite algorithm is given in Figure 3. Lines 1 and 5 generate a sprite image from high-resolution scene geometry. This is necessary whenever the viewer jumps to a new viewpoint position (line 1), and when LOD-sprite generates a new

```
1  render *sprite* image from high-resolution
      scene geometry at viewpoint *vp*
2  **for** each novel viewpoint *vp*
3    let *error* = ErrorMetric( *vp*, *sprite* )
4    **if** *error* > *threshold* **then**
5       render *sprite* image from high-resolution
          scene geometry at viewpoint *vp*
6       let *polys* = set of low-resolution scene
          geometry polygons
7       **for** each *poly*
8          **if** *WasVisible*( *poly*, *sprite* ) **then**
9             render *poly*, map with *sprite*
10         **else**
11            render *poly*, map with original texture map
```

Figure 3: *Pseudocode for the LOD-Sprite algorithm.*

keyframe (line 5). At line 2 the algorithm processes each novel viewpoint. Lines 3 and 4 measure the error associated with how far the current viewpoint diverges from viewpoint at the time when the sprite was rendered; the procedure *ErrorMetric* is described in Section 3.2. At line 6 the algorithm prepares to render the frame at the current viewpoint by gathering a set of polygons from a low-resolution version of the scene geometry. Line 7 considers each polygon. Line 8 determines, for each low-resolution polygon, whether the polygon was visible when the sprite image was taken. This routine, *WasVisible* (described in Section 3.3), determines whether the polygon is texture mapped with the sprite texture (line 9) or the original texture map (line 11).

The *sprite* data structure holds both the sprite texture map and the keyframe viewing parameters; LOD-sprite uses both to map polygons with the sprite texture in line 9. Creating a new sprite (lines 1 and 5) requires copying the frame buffer into texture memory, which is efficiently implemented with the OpenGL *glCopyTexImage2D* function.

Texture mapping a keyframe could be achieved using projective texture mapping: a light placed at the keyframe camera position projects the sprite image onto the scene geometry. However, our implementation of LOD-sprite does not use projective texture mapping, because the current OpenGL implementation does not test for polygon visibility. Occluded polygons in the keyframe are mapped with wrong textures when they become visible. Therefore, our implementation detects polygon visibility on its own (line 8), and applies a different texture map depending on each polygons' visibility (lines 9 and 11).

3.2 Error Metric

LOD-sprite decides when to render a new keyframe based on an error metric which is similar to that described by Shade et al. [22]. Figure 4 gives the technique, which is drawn in 2D for clarity. Consider rendering the full-resolution dataset from viewpoint position v_1. In this case the line segments AC and CB are rendered (in 3D these are polygons). From this view, the ray passing through vertex C intersects the edge AB at point C'. After rendering the full-resolution dataset, the image from v_1 is stored as a texture map. Now consider rendering the scene from the novel viewpoint v_2, using the low-resolution representation of the dataset. In this case the line segment AB is rendered, and texture mapped with the sprite rendered from v_1. Note that this projects the vertex C to the position C' on AB. From v_1 this projection makes no visible differ-

ence. However, from v_2, vertex C' is shifted by the angle θ from its true location C. This angle can be converted to a pixel distance on the image plane of view v_2, which is our measure of the error of rendering point C from view v_2:

$$\theta < \beta \cdot \epsilon, \tag{1}$$

where β is the view angle of a single pixel (e.g., the field-of-view over the screen resolution), and ϵ is a user-specified error threshold. As long as Equation 1 is true, we render using the sprite from the most recent keyframe (e.g., line 5 in Figure 3 is skipped). Once Equation 1 becomes false, it is again necessary to render from the full-resolution dataset (e.g., line 5 in Figure 3 is executed).
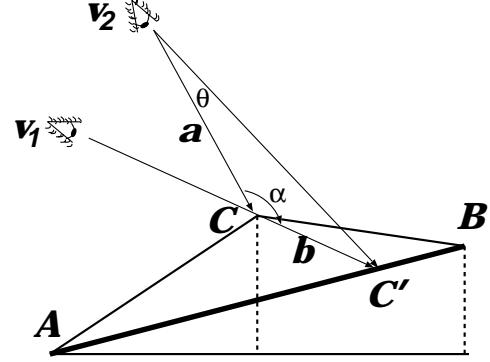

Figure 4: *Calculating the error metric.*

Theoretically, we should evaluate Equation 1 for all points in the high-resolution dataset for each novel view. Clearly this is impractical. Instead, our implementation calculates θ for the central vertex of each low-resolution quadtree quadrant. The resolution of each quadrant is determined by the number of levels we traverse down into the quadtree that is created by our LOD algorithm [14]. We calculate the central vertex by averaging the four corner vertices of the quadrant. To calculate θ, we have to know the point C'. We calculate C' by interesting the vector v_1C with the plane spanned by the estimated central vertex and two original vertices of the quadrant. Once we know C', we calculate θ from the dot product of the vectors v_2C' and v_2C.

We next calculate the average sum of squares of the error for all evaluated quadrants and compare this with $(\beta \cdot \epsilon)^2$:

$$\frac{\sum_{i=1}^{n} \theta_i^2}{n} < (\beta \cdot \epsilon)^2, \tag{2}$$

where n is the number of low-resolution quadrants. When this test fails, line 5 in Figure 3 is executed.

3.3 Visibility Changes

As the viewpoint changes, polygons which were originally occluded or culled by the view frustum may become visible. Figure 5 illustrates this problem. Let the two objects represent mountains. The light shaded region of the back mountain indicates occluded polygons in the keyframe, while the heavy shaded regions in both mountains show polygons culled by the view frustum. If these regions become visible in a novel view, there will be no sprite texture to map on them. Our solution is to map them with the same texture map we use to generate the keyframe.

We classify the visibility of each polygon with a single pass over all vertices of the low-resolution geometry. This loop is part of the process of generating a new keyframe. For novel views, the visibility of each polygon to the sprite is already flagged. This visibility
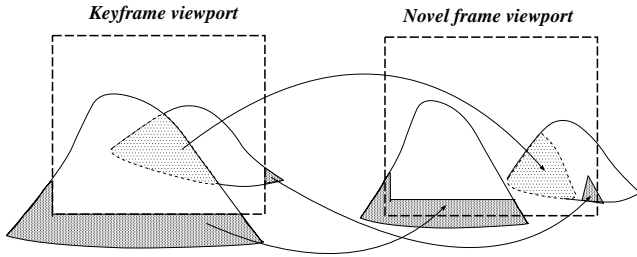
Figure 5: *The originally occluded or view frustum culled objects may become visible.*

flag controls which texture map is used for the polygon (and thus line 8 in Figure 3 is a fast table look-up). OpenGL determines the visibility of each polygon from the novel viewpoint using the hardware z-buffer.

In our implementation, the terrain is represented by a triangle mesh. We determine the visibility of each low-resolution triangle using the keyframe viewing parameters and the keyframe z-buffer. Our visibility determination for each triangle is binary, which means we consider a partially occluded triangle to be fully occluded. We do not attempt to subdivide partially occluded triangles, because achieving this would require clipping the triangle into visible and invisible sub-triangles [8]. This would not only be expensive, but would also generate too many small triangles.

To accurately detect visibility, we should scan-convert the whole triangle and detect the visibility of every pixel. This is obviously too expensive. Instead, we only perform this detection for the three triangle vertices. Only when all three vertices are visible do we flag the triangle as visible. Of course, this fails for triangles with unoccluded vertices but which are nevertheless partially occluded (e.g., a part of an edge and interior could be occluded). Such triangle will be erroneously flagged as visible. However, with terrain datasets this rarely occurs, since the projections of background triangles tend to be much smaller than foreground triangles.

We use the z-buffer to determine the visibility of each vertex. When we calculate a keyframe, we store both the z-buffer and the viewing matrix. Then, for each vertex, we calculate the (x, y) screen coordinate and the z-depth value with the keyframe viewing matrix. We compare this depth value to the z value at location (x, y) in the z-buffer. This tells us whether the vertex is occluded in the keyframe.

This raises several implementation issues. The first is that a vertex is usually not projected onto an integer grid point in the z-buffer. Using the z-buffer value at the closest grid position does not always give the correct visibility, because that z value could represent a neighboring triangle. Interpolating between neighboring z values is also inappropriate, because they could represent disconnected objects. The second issue is that the LOD mesh is not static — we compare the low-resolution geometry to the z-buffer rendered from the high-resolution geometry.

Although it does not solve either of these problems, we have obtained good results in practice by using the following equation to determine visibility:

$$|Z_{vertex} - Z_{buffer}| < \epsilon, \qquad (3)$$

where Z_{vertex} is the calculated z value of the vertex, Z_{buffer} is the z-buffer value at the closet grid point, and ϵ is the specified 'thickness' of the visible surface. When Equation 3 is true we flag the vertex as visible.

3.4 Implementation Notes

To further enhance rendering time, we have tried to optimize our implementation for the graphics hardware. For each frame, we need two texture maps — the original texture map and the current keyframe — to map all of the visible polygons. It is much too costly to load the appropriate map into texture memory on a per-polygon basis. Instead, we load both maps into texture memory, and scale the calculated texture coordinates so that each polygon accesses the correct map. In addition, we use triangle strips as our rendering primitive. The drawback of this primitive is that we can only apply one texture map to the whole strip. For strips which contain both visible and invisible triangles, we can only use the original texture map.

4 RESULTS

Results are shown in Figures 6 and 7. The input is a 512×512 height field and 512×512 texture map. Figures 6a–e compare the LOD-sprite technique to a standard LOD technique [14]. Figure 6a shows a terrain dataset rendered from a low-resolution LOD decomposition containing 1,503 triangles, while Figure 6b shows the same terrain rendered from a high-resolution decomposition with 387,973 triangles. Both figures use the same texture map. Comparing 6a to 6b, we see that, as expected, many surface features are smoothed out. Figure 6c shows the same view rendered with the LOD-sprite technique, using the same 1,503 triangles as Figure 6a but texture mapped with Figure 6b. Unlike Figure 6a the surface features are quite well preserved, yet Figures 6a and 6c take the same amount of time (10 milliseconds) to render. Figures 6d and 6e give difference images; Figure 6d gives the absolute value of the difference between the high and low resolution images, while Figure 6e between the high and LOD-sprite images. Figures 6d and e clearly show the image-quality advantage of the LOD-sprite technique. Notice, however, the bright band along the silhouette, both against the horizon as well as the edge of the dataset in the lower left-hand corner of the images. These appear because our LOD decomposition [14] is not sensitive to the edge of the dataset or to silhouette edges.

Figure 7a–e show similar results but are rendered from a viewpoint over the mountains, looking down onto the plain beyond. In this figure note that the close mountains appear very similar at low resolution (a), high resolution (b), and with the LOD-sprite technique (c). This is because these mountains are so close that even at a high resolution the polygons are large, and the LOD decomposition keeps these polygons at full resolution. The difference images (Figures 7d and e) also demonstrate this. The comments regarding the silhouette edge given above also apply to this figure, although in this case the entire silhouette edge is also the edge of the data.

Figures 9–14 give the algorithm's timing behavior for the camera path shown in Figure 8. The camera starts away from the terrain, zooms in, flies over a plain, and then over a mountain range and onto the plain beyond. This path visits most of the interesting topological features of this dataset. The animation contains 600 frames for all the figures except for Figure 12, where the frame count is varied. Each frame was rendered at a resolution of 512×512 on an SGI Onyx 2 with 6 195MHz MIPS processors and Infinite Reality graphics. We rendered the same animation for three different runs: 1) using a high-resolution LOD decomposition, 2) using a low-resolution LOD decomposition, and 3) using the LOD-sprite technique. The LOD-sprite technique used the same settings as the high-resolution run for keyframes, and the same settings as the low-resolution run for the other frames.

Figure 9 shows how the number of triangles changes as each frame is rendered. The low-resolution and LOD-sprite runs have identical triangle counts, except at the keyframes. The high-
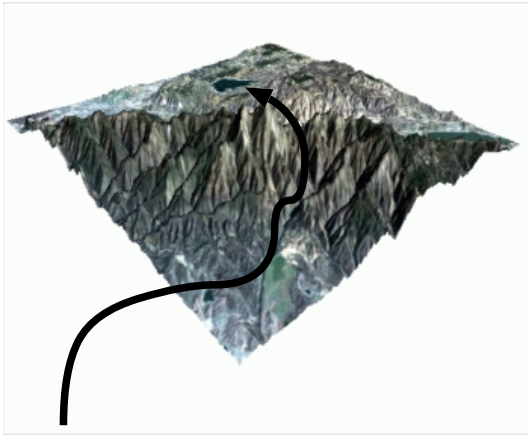
Figure 8: *The camera path for Figures 9–14.*
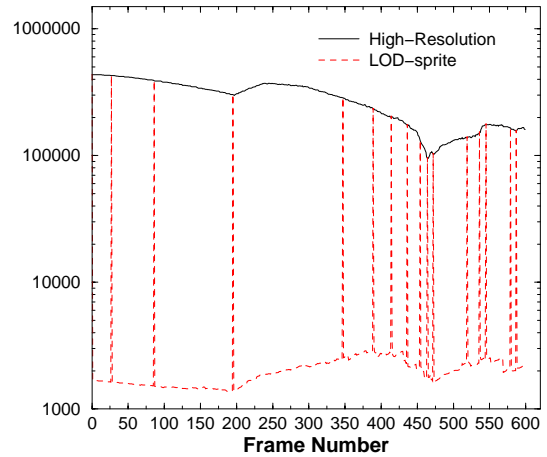

Figure 9: *The number of triangles as a function of the frame number on a semi-log plot. (600 frames; path from Figure 8.)*
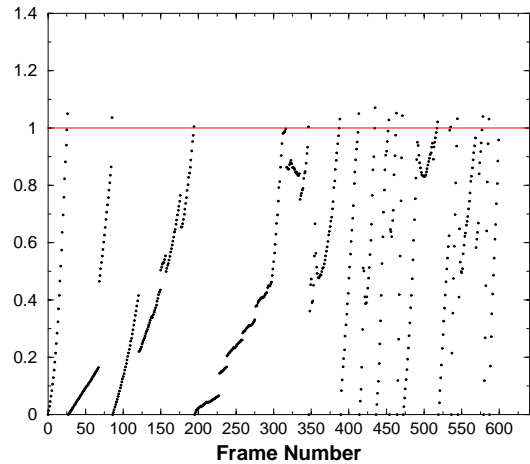

Figure 10: *The error in pixels as a function of the frame number for the LOD-sprite run. (600 frames; path from Figure 8.)*

resolution run requires about 2 orders of magnitude more triangles. The semi-log plot shows that both triangle counts have a similar variation as the animation progresses.

Figure 10 shows how the LOD-sprite error (Section 3.2) changes as each frame is rendered. The error always starts from zero for a keyframe. As more novel views are interpolated from the keyframe, the error increases. When the error exceeds 1.0 pixels, we calculate another keyframe from the high-resolution scene geometry, which again drops the error to zero.

Figure 11 shows the amount of time required to render each frame. The high-resolution time runs along the top of the graph, at an average of 526 milliseconds per frame. The low-resolution time runs along the bottom, at an average of 22 milliseconds per frame. The rendering time for the LOD-sprite frames follows the low-resolution times, except when a new keyframe is rendered. For this animation the system generated 16 keyframes, at an average time of 680 milliseconds per keyframe. The great majority of the LOD-sprite frames are shown near the bottom of the graph; these took an average of 36 milliseconds to render. The overall average for LOD-sprite was 53 milliseconds per frame.

Figure 12 shows the fraction of the total number of rendered frames which are keyframes. This is plotted against the total number of frames rendered for the path shown in Figure 8. As expected, as more frames are rendered for a fixed path, the distance moved between each frame decreases, and so there is more coherence between successive frames. This figure shows how our system takes advantage of this increasing coherence by rendering a smaller fraction of keyframes. This figure also illustrates a useful property of the LOD-sprite technique for real-time systems: as the frame update rate increases, the LOD-sprite technique becomes even more efficient in terms of reusing keyframes.

Figure 13 also shows the fraction of the total number of rendered frames which are keyframes, but this time plots the fraction against the error threshold in pixels. As expected, a larger error threshold means fewer keyframes need to be rendered. However, the shape of this curve indicates a decreasing performance benefit as the error threshold exceeds about 1.0 pixel. For a given dataset and a path which is representative of the types of maneuvers the user is expected to make, this type of analysis can help determine the best error threshold versus performance tradeoff.

The LOD-sprite technique results in a substantial speedup over rendering a full-resolution dataset. Rendering 600 frames of the full-resolution dataset along the path in Figure 8 takes 316 seconds. Rendering the same 600 frames with the LOD-sprite technique, using an error threshold of 1.0 pixel, takes 32 seconds — a speedup of 9.9. Figure 14 shows how the speedup varies as a function of the error threshold.

5   CONCLUSIONS AND FUTURE WORK

This paper has described the LOD-sprite rendering technique, and our application of the technique to accelerating terrain rendering. The technique is a combination of two rich directions in accelerated rendering for virtual environments: multiple level-of-detail (LOD) techniques, and image-based modeling and rendering (IBMR) techniques. It is a general-purpose rendering technique that could accelerate rendering for any application. It could be built upon any LOD decomposition technique. It improves the image quality of LOD techniques by preserving surface complexity, and it improves the efficiency of IBMR techniques by increasing the range of novel views that are possible. The LOD-sprite technique is particularly well-suited for real-time system architectures that decompose the
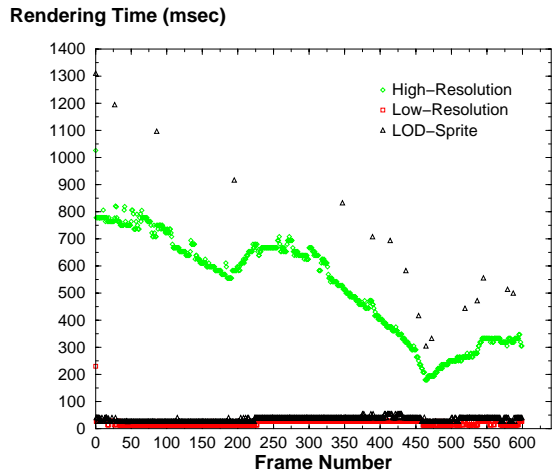
Rendering Time (msec)

Figure 11: *The rendering time in milliseconds as a function of frame number. (600 frames; path from Figure 8.)*

Fraction of Keyframes
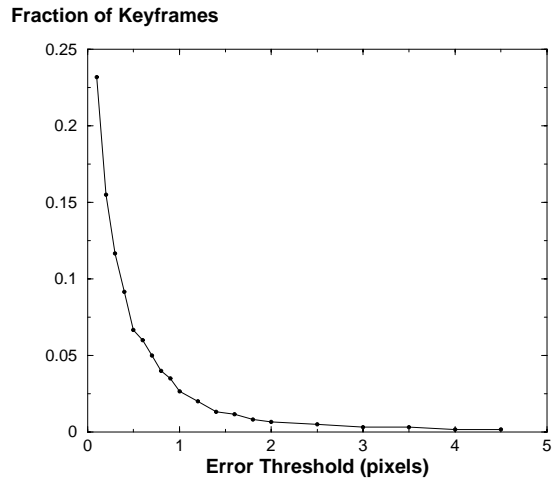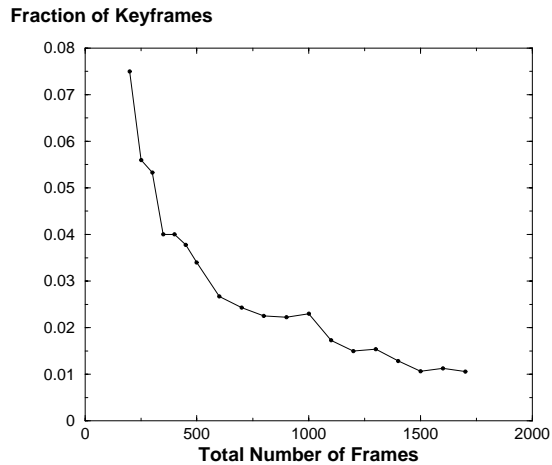

Figure 12: *The fraction of keyframes as a function of the total number of frames rendered. (Path from Figure 8.)*

Fraction of Keyframes
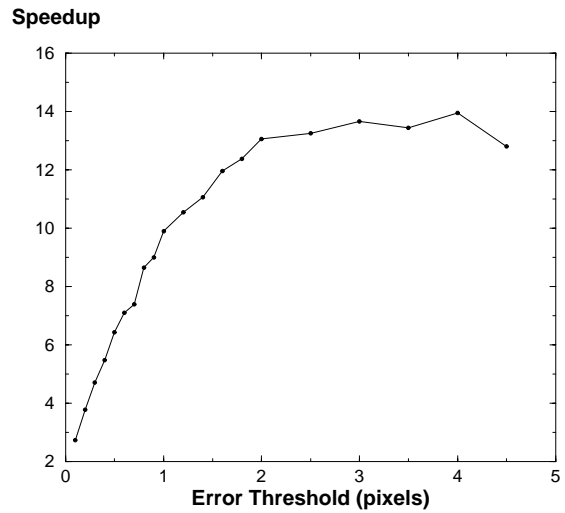

Figure 13: *The fraction of keyframes as a function of error threshold. (600 frames; path from Figure 8.)*

Speedup

Figure 14: *Speedup as a function of error threshold. 600 frames. (Path from Figure 8.)*

scene into coherent layers.

Our primary applied thrust with this work is to augment the rendering engine of a real-time, three-dimensional battlefield visualization system [9]. As this system operates in real-time, our most important item of future work is to address the variable latency caused by rendering the keyframes. One optimization is to use a dual-thread implementation, where one thread renders the keyframe while another renders each LOD-sprite frame. Another optimization is to render the keyframe in advance by predicting where the viewpoint will be when it is next time to render a keyframe. We can predict this by extrapolating from the past several viewpoint locations. Thus, we can begin rendering a new keyframe immediately after the previous keyframe has been rendered. If the system makes a bad prediction (perhaps the user makes a sudden, high-speed maneuver), two solutions are possible: 1) we could use the previous keyframe as the sprite for additional frames of LOD-sprite rendering, with the penalty that succeeding frames will have errors beyond the normal threshold. Or, 2) if the predicted viewpoint is closer to the current viewpoint than the current viewpoint is to the previous keyframe, we can use the predicted viewpoint as the keyframe in-

stead. We are also considering implementing a cache of keyframes, which would accelerate the common virtual environment navigation behavior of moving back and forth within a particular viewing region. Issues include how many previous keyframes to cache, and choosing a cache replacement policy.

The continuous LOD algorithm [14] in our implementation is well-suited for our application of real-time terrain rendering. However, the low-resolution mesh generated by this technique does not preserve silhouette edges, which as demonstrated in Figures 6 and 7, forces us to use the original texture map along the silhouette. Another problem with many continuous-LOD techniques (including [14]) is the artifact caused by sudden resolution changes, which results in a continuous popping effect during real-time flythroughs. The solution to this artifact is *geomorphing*, where the geometry is slowly changed over several frames. To address both of these issues we are currently integrating the LOD technique of Luebke and Erikson [15], which preserves silhouette edges and provides a nice framework for evaluating geomorphing techniques.

Finally, an important limiting factor for the performance of the LOD-sprite technique, as well as other image-based modeling and rendering techniques (e.g., [22]), is that OpenGL requires texture maps to have dimensions which are powers of 2. Thus, many texels in our texture maps are actually unused. The LOD-sprite technique could be more efficiently implemented with graphics hardware that did not impose this constraint.

ACKNOWLEDGMENTS

References

[1] D. G. Aliaga. Visualization of complex models using dynamic texture-based simplification. *Proceedings IEEE Visualization '96*, pages 101–106, Oct. 1996.

[2] S. E. Chen. Quicktime VR - an image-based approach to virtual environment navigation. *Computer Graphics (Proc. SIGGRAPH '95)*, pages 29–38, 1995.

[3] S. E. Chen and L. Williams. View interpolation for image synthesis. *Computer Graphics (Proc. SIGGRAPH '93)*, pages 279–288, 1993.

[4] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. *Computer Graphics (Proc. SIGGRAPH '98)*, pages 115–122, July 1998.

[5] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):255–264, Sept. 1996.

[6] L. Darsa, B. C. Silva, and A. Varshney. Navigating static environments using image-space simplification and morphing. *Symposium on Interactive 3D Graphics*, pages 25–34, Apr. 1997.

[7] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 11–20, Aug. 1996.

[8] P. E. Debevec, Y. Yu, and G. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. *Rendering Techniques'98*, pages 105–116, 1998.

[9] J. Durbin, J. E. Swan II, B. Colbert, J. Crowe, R. King, T. King, C. Scannell, Z. Wartell, and T. Welsh. Battlefield visualization on the responsive workbench. *Proceedings IEEE Visualization '98*, pages 463–466, Oct. 1998.

[10] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 43–54, 1996.

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

[12] Y. Horry, K. ichi Anjyo, and K. Arai. Tour into the picture: Using a spidery mesh interface to make animation from a single image. *Computer Graphics (Proc. SIGGRAPH '97)*, pages 225–232, Aug. 1997.

[13] M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 31–42, 1996.

[14] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hughes, N. Faust, and G. Turner. Real-Time, continuous level of detail rendering of height fields. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 109–118, Aug. 1996.

[15] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics (Proc. SIGGRAPH '97)*, pages 199–208, Aug. 1997.

[16] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. *Symposium on Interactive 3D Graphics*, pages 95–102, Apr. 1995.

[17] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. *Symposium on Interactive 3D Graphics*, pages 7–16, Apr. 1997.

[18] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics (Proc. SIGGRAPH '95)*, pages 39–46, 1995.

[19] P. Rademacher and G. Bishop. Multiple-center-of-projection images. *Computer Graphics (Proc. SIGGRAPH '98)*, pages 199–206, July 1998.

[20] M. Regan and R. Post. Priority rendering with a virtual reality address recalculation pipeline. *Computer Graphics (Proc. SIGGRAPH '94)*, pages 155–162, July 1994.

[21] G. Schaufler and W. Stuerzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum (Proc. of Eurographics '96)*, 15(3):227–235, Aug. 1996.

[22] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 75–82, Aug. 1996.

[23] J. W. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered depth images. *Computer Graphics (Proc. SIGGRAPH '98)*, pages 231–242, July 1998.

[24] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum (Proc. of Eurographics '97)*, 16(3):207–218, Sept. 1997.

[25] M. Soucy, G. Godin, and M. Rioux. A texture-mapping approach for the compression of colored 3D triangulations. *The Visual Computer*, 12(10):503–514, 1996.

[26] R. Szeliski. Video mosaics for virtual environments. *IEEE Computer Graphics and Applications*, 16(2):22–30, Mar. 1996.

[27] J. Torborg and J. Kajiya. Talisman: Commodity Real-time 3D graphics for the PC. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 353–364, Aug. 1996.
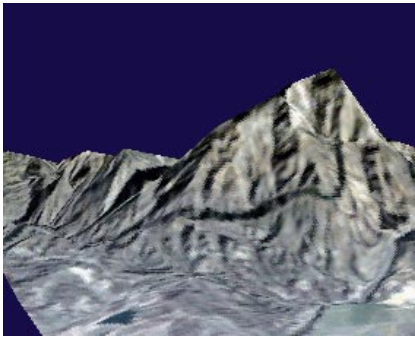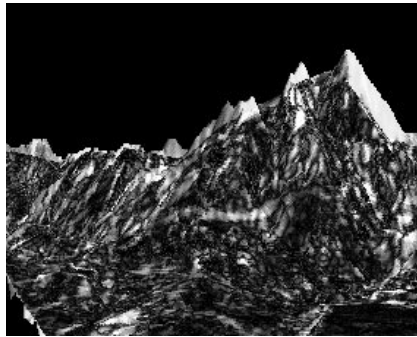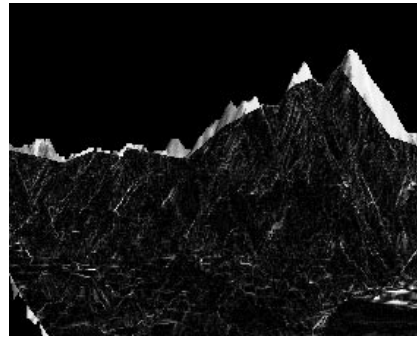
(a) *Low resolution with 1,503 triangles.*
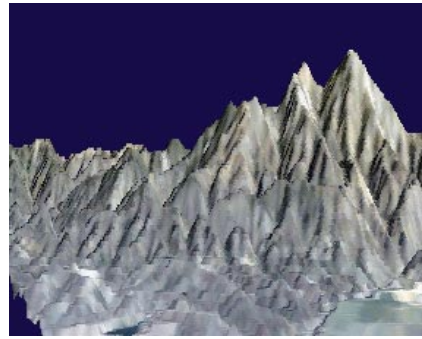
(b) *High resolution with 387,937 triangles.*
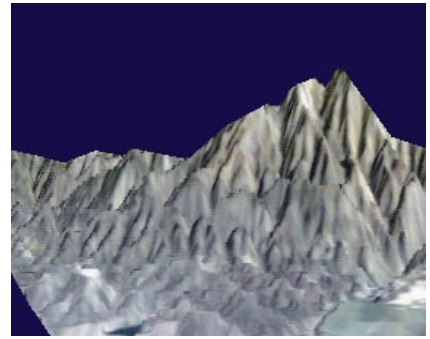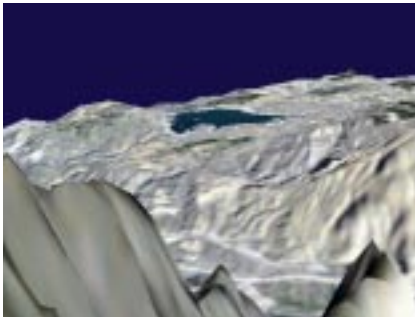
(c) *LOD-sprite with 1,503 triangles.*

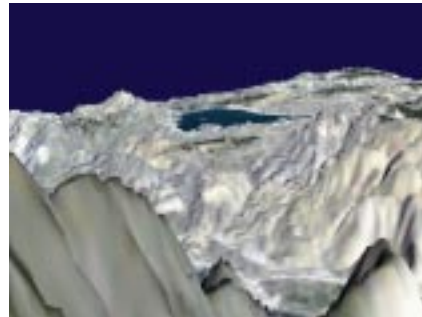(d) *Difference: low* (a) − *high* (b).

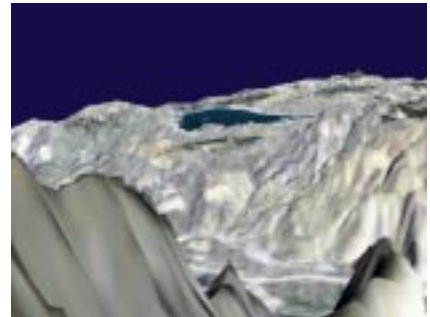(e) *Difference: high* (b) − *LOD-sprite* (c).

Figure 6: *Comparing the LOD-sprite technique to a traditional LOD technique, first view (see also color plates).*
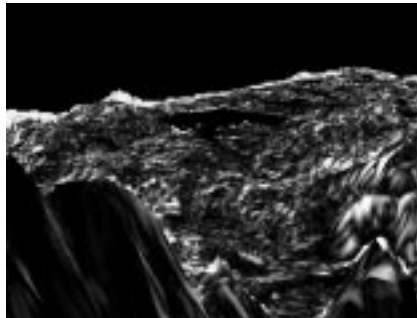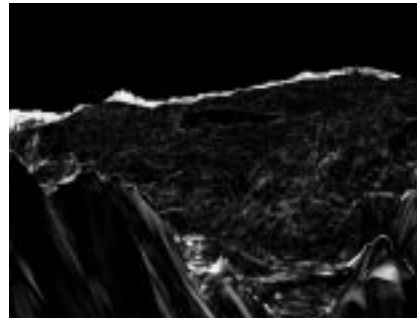

(a) *Low resolution with 2,007 triangles.*

(b) *High resolution with 156,884 triangles.*

(c) *LOD-sprite with 2,007 triangles.*

(d) *Difference: low* (a) − *high* (b).

(e) *Difference: high* (b) − *LOD-sprite* (c).

Figure 7: *Comparing the LOD-sprite technique to a traditional LOD technique, second view (see also color plates).*