Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments

Jonathan Shade Dani Lischinski David H. Salesin Tony DeRose John Snyder†

University of Washington †Microsoft Research

Abstract

We present a new method that utilizes path coherence to accelerate walkthroughs of geometrically complex static scenes. As a preprocessing step, our method constructs a BSP-tree that hierarchically partitions the geometric primitives in the scene. In the course of a walkthrough, images of nodes at various levels of the hierarchy are cached for reuse in subsequent frames. A cached image is reused by texture-mapping it onto a single quadrilateral that is drawn instead of the geometry contained in the corresponding node. Visual artifacts are kept under control by using an error metric that quantifies the discrepancy between the appearance of the geometry contained in a node and the cached image. The new method is shown to achieve speedups of an order of magnitude for walkthroughs of a complex outdoor scene, with little or no loss in rendering quality.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: BSP-tree, image-based rendering, level-of-detail (LOD), path coherence, spatial hierarchy, texture mapping.

1 Introduction

Interactive visualization of extremely complex geometric environments is becoming an increasingly important application of computer graphics. Though the throughput of graphics hardware over the past decade has improved dramatically, the demand for performance continues to outpace the supply, as virtual scenes containing many millions of polygons become increasingly common. In order to rapidly visualize truly complex scenes, rendering algorithms must intelligently limit the number of geometric primitives rendered in each frame.

This paper presents a new method for accelerating walkthroughs of geometrically complex and largely unoccluded static scenes by hierarchically caching images of scene portions. As a viewer navigates through a virtual environment, the appearance of distant parts of the scene changes little from frame to frame. We exploit this *path coherence* by caching images created in one frame for possible reuse in many subsequent frames.

Our method starts with a preprocessing stage. Given an unstructured set of objects comprising a scene, we construct a BSP-tree [6] by placing splitting planes inside gaps between objects. This construction produces a hierarchical spatial partitioning of the scene with geometry stored only at the leaves of the hierarchy. During a walkthrough of the scene, our method traverses the hierarchy and caches images of nodes at various levels to be reused in subsequent frames. An error metric that quantifies the discrepancy between the appearance of the actual geometry contained in a node and its cached image is used to estimate the number of frames for which the cached image is likely to provide an adequate approximation of the node's contents. A simple cost-benefit analysis is performed at each node in order to decide whether or not an image should be cached.

The main contribution of our approach is the successful combination of two powerful paradigms: hierarchical methods and image-based rendering. Image-based rendering is capable of drawing arbitrarily complex objects in constant time, once the image is created. Using a hierarchy of images leverages the power of image-based rendering by significantly reducing the number of images that must be drawn. Another contribution is the introduction of a new simple error metric that provides automatic quality control.

1.1 Previous work

Previous work on accelerating the rendering of complex environments can be classified into three major categories: visibility culling, level-of-detail modeling, and image-based rendering.

Visibility culling

Visibility culling algorithms attempt to avoid drawing objects that are not visible in the image. This approach was first investigated by Clark [4], who used an object hierarchy to rapidly cull surfaces that lie outside the viewing frustum. Garlick *et al.* [8] applied this idea to spatial subdivisions of scenes. View-frustum culling techniques are most effective when only a small part of the scene's geometry is inside the view frustum at any single frame. In a complex environment enough geometry remains inside the view frustum to overload the graphics pipeline, and additional acceleration techniques are required.

Airey *et al.* [1] and Teller [19] described methods for interactive walkthroughs of complex buildings that compute the potentially visible set of polygons for each room in a building. Only the potentially visible set of polygons for the room currently containing the viewer needs to be rendered at each frame. Both of these methods require a lengthy preprocessing step for large models. More recently, Luebke and Georges [11] developed a dynamic version of this algorithm that eliminates the preprocessing. Such methods can be very effective for densely occluded polyhedral environments, such as building interiors, but they are not suited for mostly un-

{shade | danix | salesin}@cs.washington.edu
derose@pixar.com, johnsny@microsoft.com

occluded outdoor scenes.

The hierarchical Z-buffer [9] is another approach to fast visibility culling that allows a region of the scene to be culled whenever its closest depth value is greater than those of the pixels that have already been drawn at its projected screen location. Like previous approaches, this method can achieve dramatic speed-ups for environments with significant occlusion but is less effective for largely unoccluded environments with high visible complexity, such as a landscape containing thousands of trees.

Level-of-detail modeling

Another approach for accelerating rendering is the use of multiresolution or *level-of-detail* (LOD) modeling. The idea is to render progressively coarser representations of a model as it moves further from the viewer. Such an approach has been used since the early days of flight simulators, and has more recently been incorporated in walkthrough systems for complex environments by Funkhouser and Séquin [7], Maciel and Shirley [12], and Chamberlain *et al.* [2].

One of the chief difficulties with the LOD approach is the problem of generating the various coarse-level representations of a model. Funkhouser and Séquin [7] created the different LOD models manually. Eck *et al.* [5] described methods based on wavelet analysis that can be used to automatically create reasonably accurate low-detail models of surfaces. Maciel and Shirley [12] used a number of LOD representations, including geometric simplifications created by hand, texture maps, and colored bounding boxes. Chamberlain *et al.* [2] partitioned the scene into a spatial hierarchy of cells and associated with each cell a colored box representing its contents. Another approach to creating LOD models is described by Rossignac and Borrel [16], in which objects of arbitrary topology are simplified by collapsing groups of nearby vertices into a single representative vertex, regardless of whether they belong to the same logical part.

Another problem with geometric LOD approaches is that the shading function becomes undersampled, as geometry is decimated. This undersampling causes shading artifacts, especially with Gouraud shading hardware, which evaluates the shading function only a the (decreasing number of) polygon vertices.

Our approach can be thought of as a technique for automatically and dynamically creating view-dependent image-based LOD models. Among the above LOD approaches, ours is closest to that of Maciel and Shirley. However, there are several important differences. First, our approach computes LOD models on demand in a view-dependent fashion, rather than precomputing a fixed set of LOD models and using them throughout the walkthrough. Thus, we incur neither the preprocessing nor the storage costs associated with precomputed LOD models. Second, we use a spatial hierarchy rather than an object hierarchy, and our LOD models represent regions of the scene rather than individual objects. Spatial partitioning allows us to correctly depth-sort the LOD models chosen for rendering at each frame, whereas an object hierarchy can suffer from occlusion artifacts where objects overlap.

Image-based rendering

A different approach for interactive scene display is based on the idea of *view interpolation*, in which different views of a scene are rendered as a pre-processing step, and intermediate views are generated by morphing between the precomputed images in real time. Chen and Williams [3] and McMillan and Bishop [13] have demonstrated two variants of this approach for restricted movement in three-dimensional environments. Although not general purpose, these algorithms provide a viable method of rendering complex environments on machines that do not have fast graphics hardware. Images provide a method of rendering arbitrarily complex scenes in a constant amount of time. This idea is central to both of these papers and to the method we present here.

Another image-based approach, described by Regan and Pose [15], renders the scene onto the faces of a cube centered around the viewer location. Their method allows the display to be updated very rapidly when the viewer is standing in place and looking about. They also use multiple display memories and image compositing with depth to allow different parts of an environment to be updated at different rates. Only parts of the environment that change or move significantly are re-rendered from one frame to the next, resulting in the majority of objects being rendered infrequently.

Our method can be thought of as a hierarchical extension to the method of Regan and Pose, but with more flexibility: instead of using a fixed number of possible update rates, our method updates each object at its own rate. Another important difference is that instead of simply reusing an object's image over several consecutive frames, we use texture-mapping hardware to compensate for motion parallax.

Schaufler and Stürzlinger [17, 18] have concurrently and independently investigated ideas similar to our own. Our approach differs from theirs mostly in the formulation of the error metric and in the cost-benefit analysis that we perform in order to decide whether or not to cache an image.

1.2 Overview

The remainder of the paper is organized as follows. In the next section, we describe our algorithm in detail. In Section 3, we present the error metric used to control the updating of cached images. In Section 4, we describe the preprocessing stage that constructs a hierarchical spatial partitioning of the environment. In Section 5, we report on the performance of our algorithm for a walkthrough of a complex outdoor scene. Section 6 closes with conclusions and future work.

2 Algorithm

As a viewer follows a continuous path through a virtual environment, there is typically considerable coherence between successive frames. The basic idea behind our algorithm is to exploit this coherence by caching images of objects rendered in one frame for possible reuse in many subsequent frames. However, instead of simply reusing the same image, we apply the image as a texture map to a fixed quadrilateral placed at the center of the object. This textured quadrilateral is then rendered instead of the original object during several successive frames, using the current viewing transformation at each frame. In this way, at each frame, the image of the object is slightly warped, approximately correcting for the slight changes in the perspective projection of the original object as the viewer moves through the scene. Compensating for motion parallax in this manner results in fewer "snapping" artifacts when the cached image is updated and increases the number of frames for which the cache yields an acceptable approximation to the object's appearance.

To gain the most from image caching, it is not enough to cache images for individual objects. If too many objects are visible, the sheer number of textured polygons that must be rendered at each frame may overwhelm the hardware. However, distant objects that require infrequent updates can be grouped into clusters, and a single image can be cached and rendered in place of the entire cluster. Thus, our algorithm operates on a hierarchical representation of the entire scene, rather than on a collection of individual objects. An image can be computed and cached for any node in the hierarchy; hence the name "hierarchical image caching".

We construct the hierarchy as a preprocessing step by computing a BSP-tree [6] partitioning of the environment, as described in Section 4. We chose to use a BSP-tree since it allows us to traverse the scene in back-to-front order, which is necessary to ensure that the partially-transparent textured quadrilaterals are composited correctly in the frame-buffer. In addition, BSP-trees are more flexible than other spatial partitioning data structures, making it is easier to avoid splitting objects.

The leaf nodes of the BSP-tree correspond to convex regions of space and have associated with them a set of geometric primitives. This set consists of all the geometric primitives contained inside the node. In addition, it also contains nearby primitives from the neighboring nodes, as will be explained in Section 4. Any node in the tree may also contain a cached image.

At each frame we traverse the BSP-tree twice. The first traversal culls nodes that are outside the view frustum and updates the image caches of the visible nodes:

```
UpdateCaches(node, viewpoint)
if node is outside the view frustum then
    node.status ← CULL
else if node.cache is valid for viewpoint then
    node.status ← DRAWCACHE
else if node is a leaf then
    UpdateNode(node, viewpoint)
else
    UpdateCaches(node.back, viewpoint)
    UpdateCaches(node.front, viewpoint)
    UpdateNode(node, viewpoint)
```

For a leaf node, the routine *UpdateNode* decides whether, for the current viewpoint, it is more cost-effective to draw the geometry stored with the node, or to compute and cache an image:

```
UpdateNode(node, viewpoint)
if viewpoint ∈ node then
    if node is a leaf then
        node.status ← DRAWGEOM
    else
        node.status ← RECURSE
    return
k ← EstimateCacheLifeSpan(node, viewpoint)
amortizedCost ← ⟨cost to create cache⟩/k + ⟨cost to draw cache⟩
if amortizedCost < ⟨cost to draw contents⟩ then
    CreateCache(node, viewpoint)
    node.status ← DRAWCACHE
    node.drawingCost ← ⟨cost to draw cache⟩
else
    if node is a leaf then
        node.status ← DRAWGEOM
        node.drawingCost ← ⟨cost to draw geometry⟩
    else
        node.status ← RECURSE
        node.drawingCost ←
            node.back.drawingCost + node.front.drawingCost
```

Geometry is always drawn if the viewpoint is inside the node. Otherwise, the routine *EstimateCacheLifeSpan* computes an estimate of the number of frames k for which we expect the cached image to remain valid, as will be described in Section 3. This estimate is used to compute an amortized cost-per-frame for this node for each of the next k frames. We compute and cache an image only if the amortized cost is smaller than the cost of simply drawing the node's contents. For a leaf node, this cost is simply the cost of drawing the contained geometry, while for an interior node, this cost is the cost of drawing the node's children. The costs to draw geometric primitives and to create a cached image are established experimentally on each platform and are given as input to our system.

The routine *CreateCache* starts by computing an axis-aligned rectangle that is guaranteed to contain the image of the node's contents on the screen. This rectangle is obtained by transforming the corners of the node's bounding box from world coordinates to screen coordinates and taking the minima and maxima along each axis. If the dimensions of the rectangle exceed those of the viewport, no image is cached. Otherwise, we redefine the viewing frustum so that it contains the entire node without changing the viewpoint or the view direction, and render the node. For a leaf node we draw all of its geometry, while for an interior node we draw its children. In many cases, the children are drawn using their cached images, if any exist. Thus, caching an image typically does not involve drawing all the geometry contained in the corresponding subtree. After drawing the contents of the node, we copy the corresponding rectangular block of pixels into the node's image cache. As mentioned earlier, we use the cached image as a texture map that is applied to a quadrilateral representing the entire node. In order to define an appropriate quadrilateral in world space, we project the corners of the image rectangle onto a plane of constant depth with respect to the viewpoint that goes through the center of the node's bounding box.

Once the cached images have been updated, we can proceed to render the scene into the frame-buffer, during a second traversal of the BSP-tree from back to front:

```
Render(node, viewpoint)
if node.status == CULL then
    return
else if node.status ∈ {DRAWCACHE, DRAWGEOM} then
    Draw(node)
else if viewpoint is in front of node.splittingPlane then
    Render(node.back, viewpoint)
    Render(node.front, viewpoint)
else
    Render(node.front, viewpoint)
    Render(node.back, viewpoint)
```

To complete the description of our algorithm, the next section describes the error metric we use to determine whether a cached image is valid with respect to a given viewpoint and to estimate the life-span of a cached image. Section 4 describes in more detail our BSP-tree construction algorithm.

3 Error metric

The algorithm described in the previous section requires answers to the following two closely-related questions:

1. Given a node with a cached image computed for some previous view, is the cached image valid for the current view?
2. Given a node in the hierarchy and the current view, if we were to compute and cache an image of this node, for how many frames is the cached image likely to remain valid?

In order to answer these questions efficiently we need to define an error metric, which, given a node in the hierarchy, its cache, and the current viewpoint, quantifies the difference between the appearance of the cached image and that of the actual geometry. If this difference is smaller than some user-specified threshold ϵ, the approximation is deemed acceptable, and the cache is considered valid. An important requirement for an acceptable error metric is that it must be fast to compute. For example, we cannot afford to analyze the geometric contents of the node, as the number of primitives contained in a node can be very large.

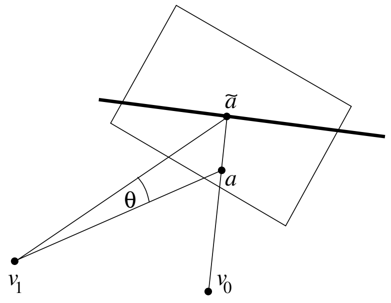


Figure 1 Angular discrepancy.

Our algorithm employs an error metric that measures the maximum angular discrepancy between a point inside a node and the point that represents it in the cached image. We shall use the 2D diagram shown in Figure 1 to define our error metric more precisely. The rectangle in this diagram represents the bounding box of a node in the hierarchy. The thick line segment crossing the bounding box represents the quadrilateral onto which the cached image is texture-mapped, as described in Section 2. The viewpoint for which the cache was computed is v_0. Let a be a point inside the node. The point that corresponds to a on the quadrilateral is $\tilde{a}$. By construction, a and $\tilde{a}$ coincide when viewed from v_0; however, for most other views, the two points subtend some angle $\theta > 0$, as illustrated by viewpoint v_1 in the diagram. Our error metric measures the maximum angular discrepancy over all points a inside the node:

$$Error\,(v, v_0) = \max_a \theta\,(a, v, \tilde{a}) \tag{1}$$

For a given view direction and field of view, the smaller the maximum angular discrepancy is allowed to be, the closer the projections of points a and $\tilde{a}$ are in the image. Thus, using a smaller error threshold results in fewer visual artifacts caused by using the cached images instead of rendering the geometry.

The right-hand side of equation (1) may be approximated by computing the angular discrepancy for each of the eight corners of a node's bounding box. This is not a conservative estimate, but it is fast to compute, and has been found to work well in practice.

In order to predict the life span of a cached image before creating it for some view v_0, we must estimate how far from v_0 we can travel while keeping the error under ϵ. If the view trajectory is known to us in advance, we can simply search along the trajectory for the farthest point for which the error is within tolerance. This is probably the best course of action for recording a walkthrough or fly-by off-line. For an interactive walkthrough, the path of the viewer is not known in advance; however, the current velocity and acceleration are known at any frame, and an upper bound on the acceleration is typically available. In this situation, for each node in the hierarchy we can attempt to find a *safety zone* around v_0, that is, a set of viewpoints v such that for each viewpoint in this set the error is less than ϵ:

$$SafetyZone(v_0) \subseteq \{v \mid Error(v, v_0) \leq \epsilon\}\,, \tag{2}$$

Given the safety zone and using bounds on velocity and acceleration, we can compute a lower bound on the number of frames for which the cache will remain valid. Alternatively, we can obtain a non-conservative estimate by extrapolating the viewer's path and intersecting it with the safety zone. Our implementation uses non-conservative estimates. Next we describe how safety zones are computed in our algorithm.

Consider the 2D diagram in Figure 2. Let v_0 be the current viewpoint, a a point inside a node, and $\tilde{a}$ its projection onto the textured quadrilateral, as in Figure 1. Note that all viewpoints v from which

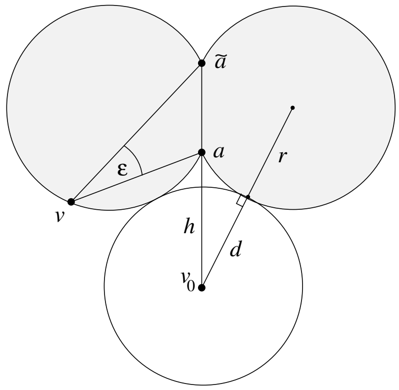


Figure 2 The shaded region contains all the viewpoints v from which a and $\tilde{a}$ subtend an angle greater than or equal to ϵ. The lower circle is a conservative safety zone.

the angle subtended by a and $\tilde{a}$ is equal to ϵ must lie on one of the two circles of radius

$$r = \frac{\|a - \tilde{a}\|}{2 \sin \epsilon} \tag{3}$$

passing through a and $\tilde{a}$. Thus, we can conservatively define a circular safety zone around v_0 (a sphere in 3D), whose radius d is given by the shortest distance between these circles and v_0:

$$d = \sqrt{h^2 + r^2 + 2hr \sin \epsilon} - r \tag{4}$$

where h is the distance between v_0 and a.

In order to approximate the safety zone for a leaf node in the hierarchy, we evaluate d for each corner of the node's bounding volume and take the smallest of these distances. We then set the safety zone to be the axis-aligned cube inscribed inside a sphere of radius d around v_0. The safety zone of an interior node is computed by first computing the safety zone using the bounding box of the node, and then taking the intersection of this safety zone with the safety zones of the children.

In our implementation, the user specifies the error threshold in pixels. This threshold is converted to an angular error threshold using the current resolution and field-of-view angle. If either the resolution or the field-of-view change in the course of a walkthrough, the angular error threshold must be adjusted accordingly.

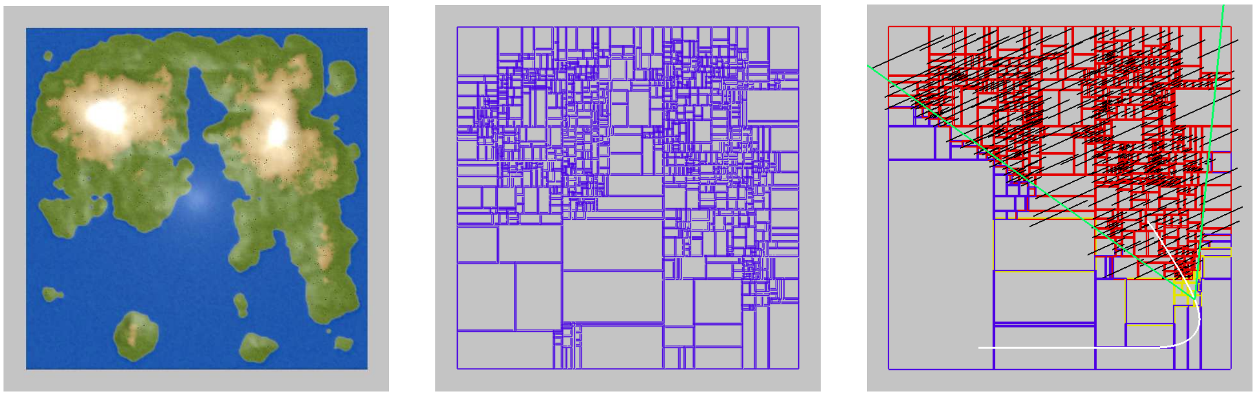
4 Partitioning

As a preprocessing step, we construct a BSP-tree [6] partitioning of the scene. The goals of the partitioning algorithm are as follows:

1. split as few objects as possible;
2. make the hierarchy as balanced as possible (in terms of the number of geometric primitives contained in each subtree);
3. make the aspect ratio of each node's bounding volume as close to 1 as possible.

The first goal aims to reduce visual artifacts. The second and third goals help improve performance: balanced trees facilitate hierarchical view-frustum culling, and cached images of nodes with good aspect ratios tend to remain valid longer. Computing the optimal BSP-tree that satisfies these potentially contradictory goals appears difficult. Therefore, our partitioning algorithm employs a simple greedy approach that is not optimal, but seems to work well in practice.

Given a list of objects to partition, we look for gaps between objects, place a splitting plane in the "best" gap we can find, and then



(a) (b) (c)

Figure 3 (a) A bird's eye view of the island scene. (b) The partitioning of the scene. (c) A viewpoint on the walkthrough path.

recurse on the lists of objects on each side of that plane. To facilitate finding the gaps between objects, we compute their extents with a method similar to the parallelepiped bounding volumes of Kay and Kajiya [10]. For each object, we compute its extent along each of N different directions on the unit sphere. Each splitting plane in the BSP-tree is constrained to be perpendicular to one of the N vectors. For example, if we chose the three coordinate axes as our direction vectors, our partitioning algorithm would yield a binary tree of axis-aligned boxes.

For each of the N directions, we create two sorted lists of objects: one, according to the lower bound of each object's extent; the other, according to the upper bound. We then scan these lists, while keeping track of the number of "active" objects (i.e., objects whose extents we are currently in). Intervals where the number of active objects is a local minimum are the gaps that we are looking for. Ideally, we are looking for a gap with zero active objects, such that the number of geometric primitives on each side of the gap is roughly equal. Such a gap does not always exist, so we compute a cost for each gap that is a function of the number of its active objects and the ratio of the number of primitives on either side of the gap. For each of the N directions, we choose the gap with the smallest cost. To create good aspect ratios, we tend to choose the best gap from the direction along which the combined extent of all the objects on the list is greatest.

The current implementation of our system is geared towards visualization of complex landscapes. Such scenes have a special structure: they essentially consist of a height-field representing land and water, and of objects such as trees and houses scattered on that height-field. Thus, assuming that the positive Y axis points up, all of the objects are spread above the XZ plane. Our partitioning algorithm takes advantage of this structure by using N direction vectors that evenly divide the unit circle perpendicular to the Y axis. As a result, all of the splitting planes of the BSP-tree are perpendicular to the XZ plane. In all of the experiments reported in Section 5, two direction vectors were used, resulting in axis-aligned boundaries between regions.

When objects are split between two or more leaf nodes, visual artifacts that look like gaps or cracks sometimes appear in the split surfaces. This problem results from approximating a single object by multiple images, with no constraint that the images match along the split boundary. Such artifacts can occur even with small error thresholds because of the discrete sampling involved in creating the caches and rendering the textured quadrilaterals. For small error thresholds, it is possible to overcome these artifacts by ensuring a small amount of overlap in the geometry contained in neighboring leaf nodes. To achieve this overlap, we construct a slightly "inflated" version of each leaf region, and associate with each leaf node the extra geometry that is contained in its inflated region, in addition to the geometry contained in the original region. In our current implementation, the amount by which regions are inflated is a user-specified parameter (typically 10 to 20 percent).

5 Results

This section demonstrates the performance of our method using a walkthrough of a complex outdoor scene. All tests were performed on a Silicon Graphics Indigo2 workstation with a 250MHz R4400 processor, 320 megabytes of RAM, and a Maximum Impact graphics board with 4 megabytes of texture memory.

The outdoor scene used in these tests is a terrain of an island populated with 1117 willow trees. The terrain consists of 131,072 triangles, and each tree consists of 36,230 triangles. The total number of triangles in the database is 40,599,982. To keep the storage requirements down the trees were instanced, and the total amount of storage for the database before any processing by our method is 20 megabytes. The amount of storage required for this scene without instancing is 3.5 gigabytes. Figure 3(a) shows a bird's eye view of the island.

Constructing the BSP-tree for this database took 46 seconds. The resulting partitioning (shown in Figure 3(b)) has 13 levels, 1072 leaf nodes, and is fairly balanced in terms of the geometric primitives contained in each subtree. Most leaf nodes contain a single tree and a portion of the terrain. The partitioning algorithm managed to avoid splitting any of the trees, and the only object split was the terrain.

Partitioning the database causes an increase in the required storage. This increase is primarily due to the need to "inflate" the leaf regions, as described in Section 4. For this database, we used an inflation factor of 17 percent, increasing the storage to 150 megabytes. Note that the increase is only 4 percent relative to the storage the original database would have required if we did not use instancing on the trees.

We recorded timings for several walkthroughs of the island. Each of the walkthroughs was along the same path, defined by a B-spline space curve shown in white in Figure 3(c). This path was designed to help us study the relative performance of image caching over a range of visible scene complexities: the camera first tracks along the edge of the model, then flies in toward the center of the island at tree-top level. Although the path was known in advance, we did not take advantage of this information, in order to get a better sense of how the algorithm would behave under interactive control.

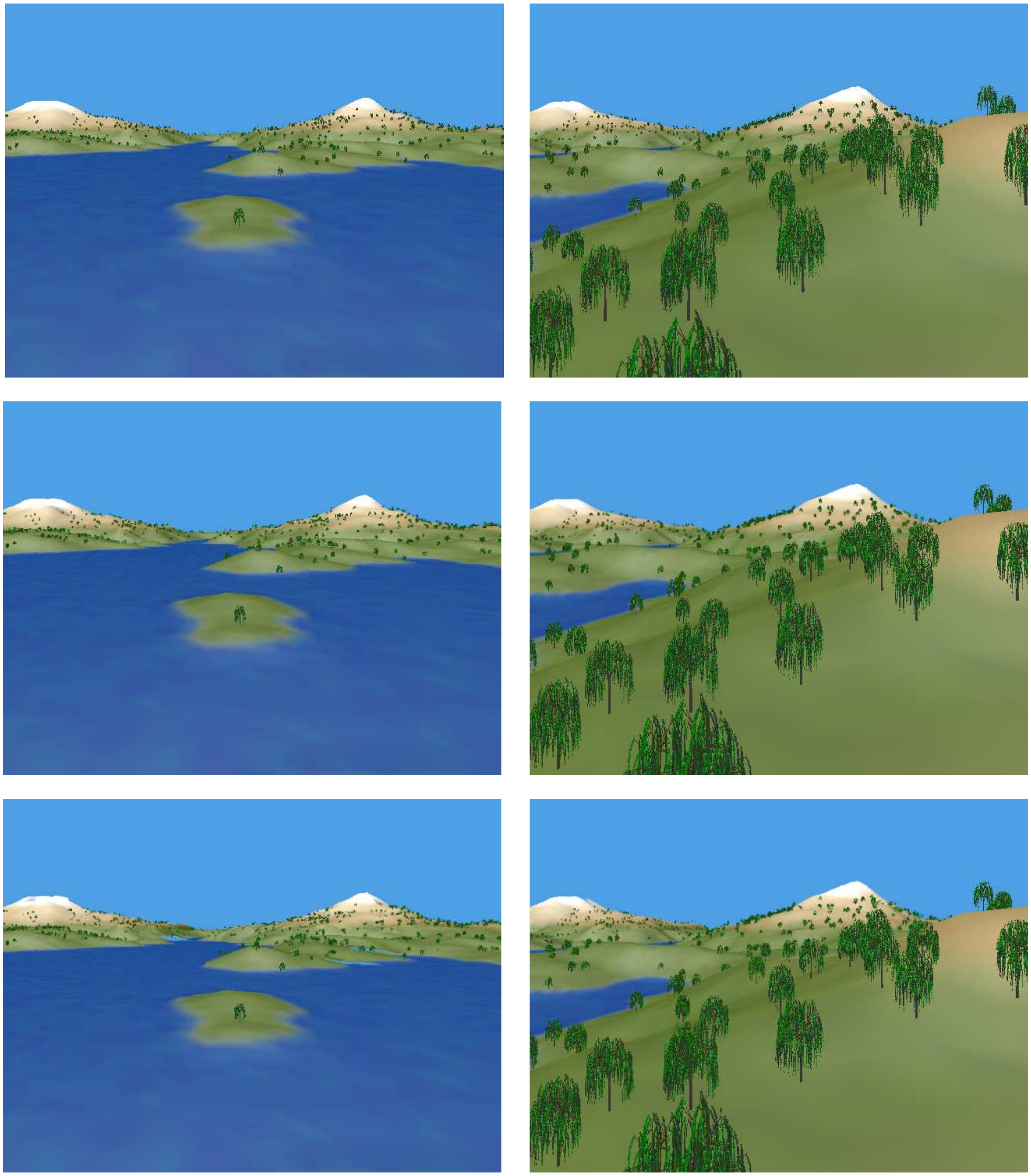


Figure 4 Frames from walkthroughs of the island. The top row shows two frames rendered using the original geometry. The second row shows the same frames rendered with image caching using an error threshold of two pixels. The third row illustrates the visual artifacts resulting from a larger error threshold (eight pixels).

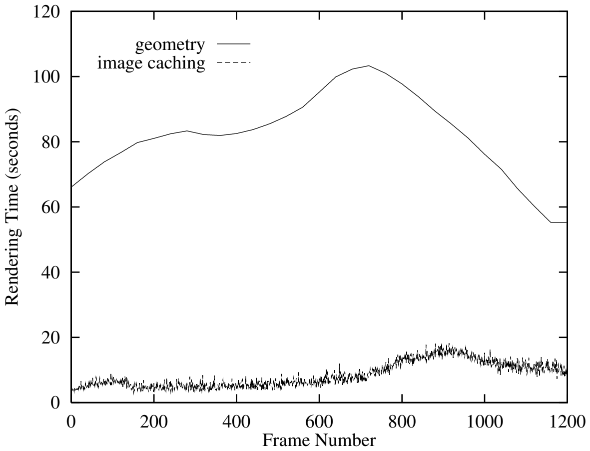


Figure 5 Image caching versus rendering geometry.

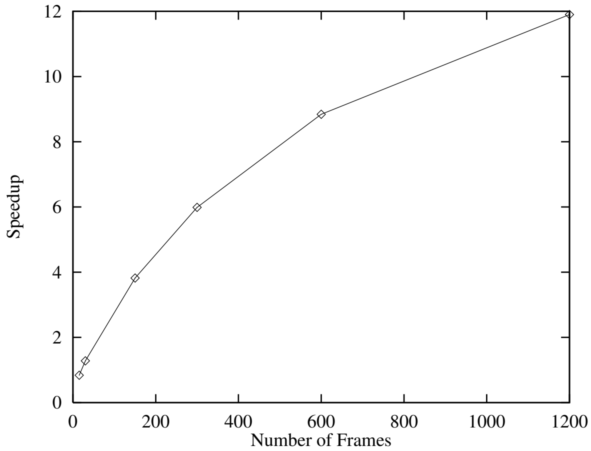


Figure 6 Speedup as a function of frame rate.

Figure 3(c) provides a snapshot illustrating our algorithm for a particular viewpoint on the path. The view frustum for that viewpoint is indicated by green lines. Nodes outlined in purple are culled, as they lie outside the view frustum. Nodes outlined in yellow are rendered using their geometry. Nodes outlined in red are rendered using their cached images. The quadrilaterals onto which these images are mapped are shown in black.

To assess the relative performance of our algorithm, we first computed two 1200-frame walkthroughs. Each frame was rendered at a resolution of 640×480. The first walkthrough was performed using an algorithm that employs hierarchical view frustum culling (using the same BSP-tree), but renders all of the original geometry contained in leaf nodes that are inside the view frustum. The second walkthrough was performed using our method with an error threshold of two pixels.

The top row of images in Figure 4 shows two different frames from the walkthrough rendered using the original geometry. The second row shows the same frames rendered by our method. The images are not identical to those in the top row, but it is very hard to tell them apart, except for the distant trees that appear slightly softer and less blocky when rendered with our method, because of the linear filtering used when rendering texture-mapped primitives.

The plot in Figure 5 shows the rendering times for the two walkthroughs. For each frame, we plot the rendering time spent by each of the two methods. It takes our method 134 seconds to compute the very first frame of the walkthrough, which is two times longer than the time required when rendering the geometry. However, once the initial image caches have been computed, subsequent frames can be rendered 4.1 to 25.2 times faster with our method, with an overall speedup factor of 11.9 for the entire sequence.

In the experiment above, our method used a fairly small error threshold: an angle subtended by roughly two pixels on the image plane. As a result, there are almost no perceptible visual artifacts in the walkthrough, as compared to rendering the geometry. If the error threshold is relaxed, more visual artifacts start to appear, but the rendering becomes faster, as cached images have longer life spans. For instance, with the error threshold set to eight pixels, the overall speedup increases to 14.1. Frames that were rendered with this error threshold are shown in the bottom row of Figure 4. Comparing these images with the ones rendered using geometry (in the top row) reveals increased "ruggedness" along the silhouette of the mountains, as well as some "cracks" in the terrain, through which the blue background shows through.

Since our method utilizes path coherence, it is interesting to examine how different frame rates along the same path affect performance. Therefore, we rendered the same walkthrough using different numbers of frames, equally spaced along the path. For example, when using two frames, the first frame is computed at the beginning of the path and the second in the middle of the path. Thus, for very small numbers of frames there is not much frame-to-frame coherence at all. For each walkthrough the overall speedup factor was computed, and the results are plotted in Figure 6. As expected, the speedup factor becomes larger, as more frames are rendered along the same path. Note that our method is faster than geometry with as few as 30 frames along the path.

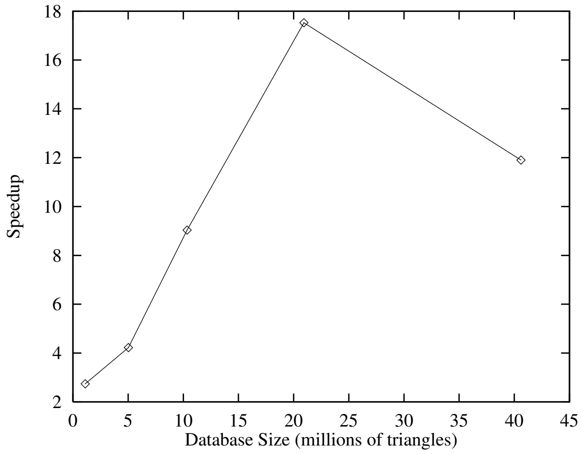


Figure 7 Speedup as a function of scene complexity.

Another interesting statistic is the behavior of our method as a function of overall scene complexity. The same walkthrough path was computed for several versions of the scene, each containing a different number of trees. Except for the number of trees, all of the scenes were identical. The overall speedup factors for these scenes (for a 1200-frame walkthrough with a two-pixel error threshold) are plotted in Figure 7. The speedup factor introduced by our algorithm first rapidly increases with the geometrical complexity of the scene, but there is a drop in the speedup when the number of triangles increases from 20 million (574 trees) to 40 million (1117 trees). The reason for this behavior is that increasing the tree density on the island causes significantly more extra geometry to be added to each leaf node when its region is inflated. This extra geometry makes the overhead of creating a cached image for the node substantially larger.

An important limiting factor on the performance of image caching is the constraint imposed by OpenGL [14] that texture maps have dimensions in powers of 2. Because of these limitations on texture size, almost half of the pixels in the textures defined by our method go unused. The handling of so many unused pixels results in a performance penalty for our image caching method.

6 Conclusions

There are many ways to extend the work presented in this paper:

- *Animation.* Although our method is currently applicable only to static scenes, it should be easy to extend it to handle a few small moving objects or animated sprites. A more challenging problem for further research is to allow scenes where many objects are capable of moving and/or deforming their geometry.
- *Pre-caching.* Our algorithm should be extended to caching images not only for nodes already in the view frustum, but also for nodes that should come into view in the next few frames. This extension would help alleviate temporary degradations in rendering performance that occur as a user travels into an area of the scene that is more complex. Pre-caching could be particularly effective if the caching computations are done in parallel by a separate thread.
- *Geometric LOD modeling.* Many of the objects drawn while creating cached images occupy only a small number of pixels in the image. Thus, instead of drawing such objects in full detail, we could draw a coarser model of the same object, using a multiresolution representation such as the one by Eck *et al.* [5] or Chamberlain *et al.* [2]. Using a multi-resolution representation could also accelerate rendering of objects for which no cached images were created.
- *Persistent caches.* As regions of the scene pass out of the view frustum, the images cached for the newly culled nodes are invalidated, and the memory is released. In the case that the viewer is simply looking around, these culled caches are still valid representations of their regions. Suspending invalidation of image caches in this case could potentially save a great deal of computation, in much the same way as the method of Regan and Pose [15].
- *Talisman.* Image caching should prove even more effective in an architecture that optimizes the reuse of rendered images as texture maps or sprites, such as the Tasliman architecture [20]. To make the best use of Talisman's capabilities, an affine warp of the cached image should be computed rather than the more general perspective warp resulting from texture-map the cached image onto a quadrilateral in 3D.

In summary, we have presented a new method for accelerating walkthroughs of complex environments by utilizing path coherence. We have demonstrated speedups of an order of magnitude on a current graphics architecture, the Indigo2 Maximum Impact. The speedups increase with the frame rate. While these speedups are significant, we believe they could be made still more dramatic through further optimizations in the underlying graphics hardware and libraries, such as improving the pixel transfer rate from the frame buffer to texture memory, relaxing the existing restrictions on texture map sizes, and providing applications with better control over texture memory management.

Acknowledgments

We would like to thank Eric Brechner, Ka Chai, Brad Chamberlain, Michael Cohen, Hugues Hoppe, and Jack Tumblin for many useful discussions during the early stages of this project.

This work was supported by an Alfred P. Sloan Research Fellowship (BR-3495), an NSF Postdoctoral Research Associates in Experimental Sciences award (CDA-9404959), an NSF Presidential Faculty Fellow award (CCR-9553199), an ONR Young Investigator award (N00014-95-1-0728), a grant from the Washington Technology Center, and industrial gifts from Interval, Microsoft, and Xerox.

References

[1] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[2] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proceedings of Graphics Interface '96*, May 1996.

[3] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Computer Graphics* Proceedings, Annual Conference Series, pp. 279–288, August 1993.

[4] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[5] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis for arbitrary meshes. In *Computer Graphics* Proceedings, Annual Conference Series, pp. 173–182, August 1995.

[6] Henry Fuchs, Zvi M. Kedem, and Bruce Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):175–181, June 1980.

[7] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics* Proceedings, Annual Conference Series, pp. 247–254, August 1993.

[8] B. Garlick, D. Baum, and J. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. SIGGRAPH '90 Course Nores: Parallel Algorithms and Architectures for 3D Image Generation, 1990.

[9] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Computer Graphics* Proceedings, Annual Conference Series, pp. 231–238, August 1993.

[10] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986.

[11] Daivid Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics*, pp. 105–106, April 1995.

[12] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, pp. 95–102, April 1995.

[13] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics* Proceedings, Annual Conference Series, pp. 39–46, August 1995.

[14] Jackie Neider, Tom Davis, and Mason Woo. *OpengGL Programming Guide*. Addison Wesley, 1993.

[15] Matthew Regan and Ronald Pose. Priority rendering with a virtual reality address recalculation pipeline. In *Computer Graphics* Proceedings, Annual Conference Series, pp. 155–162, July 1994.

[16] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. Research Report RC 17697 (#77951), IBM, Yorktown Heights, New York 10598, 1992. Also appeared in the *IFIP TC 5.WG 5.10*.

[17] Gernot Schaufler. Exploiting frame to frame coherence in a virtual reality system. In *Proceedings of VRAIS '96*, pp. 95–102, April 1996.

[18] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. In *Proceedings of Eurographics '96*, 1996. To appear.

[19] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Computer Science Division (EECS), UC Berkeley, Berkeley, California 94720, October 1992. Available as Report No. UCB/CSD-92-708.

[20] Jay Torborg and Jim Kajiya. Talisman: Commodity realtime 3D graphics for the PC. In *Computer Graphics* Proceedings, Annual Conference Series, August 1996.