# Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments

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

Department of Computer Science and Engineering
University of Washington
*Microsoft Research

Technical Report UW-CSE-96-01-06
January 1996

## Abstract

We present a new method for accelerating 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 applied as a texture map to 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 significant speedups for a walkthrough of a complex outdoor scene, with little or no loss in rendering quality.

# 1 Introduction

Interactive visualization of extremely complex geometric environments is becoming an increasingly important application of computer graphics. Advances in the throughput of graphics hardware over the past decade have admittedly been impressive, but as virtual scenes containing many millions of polygons are becoming more common, the demand for performance continues to outpace the supply. In order to rapidly render truly complex scenes, algorithms need to be developed that intelligently limit the number of geometric primitives rendered.

This paper presents a new method for efficiently rendering 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 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 the scene, we construct a BSP-tree [5] 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 we expect the cached image 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 succesful 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 use of a simple error metric to provide 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 completely occluded. This approach was first investigated by Clark [3], who used an object hierarchy to rapidly cull surfaces that lie outside the viewing frustum. Garlick *et al.* [7] applied this idea to spatial subdivisions of scenes. View-frustum culling techniques are most effective when only a small part of the geometry is inside the view frustum at any single frame. In a complex environment enough geometry falls inside the view-frustum to overload the graphics pipeline, and additional acceleration techniques are required.

Airey *et al.* [1] and Teller [16] described methods for interactive walkthoughs of complex buildings that compute for each room in the building the potentially visible set of polygons. At each frame it suffices to render only the potentially visible set of the room currently containing the viewer. Such methods can be very effective for densely occluded polyhedral environments, such as building interiors, but they are not effective for mostly unoccluded outdoor scenes. In addition, they require a lengthy precomputation step. More recently, Luebke and Georges [10] developed an online version of this algorithm that eliminated the precomputation. However, analytical computation of the potentially visible set is still fundamentally restricted to densely occluded environments.

The hierarchical Z-buffer [8] 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 use 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 "walk through" systems for complex environments by Funkhouser and Séquin [6], and Maciel and Shirley [11].

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 [6] created the different LOD models manually. Eck *et al.* [4] describe methods based on wavelet analysis that can be used to automatically create reasonably accurate low-detail models of surfaces. Maciel and Shirley [11] used a number of LOD representations, including geometric simplifications created by hand, texture maps, and colored bounding boxes. Another approach to cre-

ating LOD models is described by Rossignac and Borrel [15], 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.

Our approach can be thought of as a technique for automatically 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. Second, our LOD models represent regions of the scene, rather than individual objects.

### 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 performing image morphing on the source images in real time. Chen and Williams [2] and McMillan and Bishop [12] 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 [14], renders the scene onto the faces of a cube centered around the viewer location. This 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 is is more flexible: instead of using a fixed number of possible update rates, our method updates each object at its own rate.

## 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 hierarchi-

cal spatial partitioning of the environment. In Section 5, we report 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 navigates through a virtual environment, there is typically considerable coherence between successive frames. The basic idea behind our algorithm is to exploit frame-to-frame coherence by caching images of objects rendered in one frame for possible reuse in many subsequent frames. However, instead of simply reusing the cached image as is, we warp the image from frame to frame in order to approximate the slight changes in the perspective projection of the object as the viewer moves through the scene. This warping effect is achieved by applying the cached image as a texture map to a fixed quadrilateral placed at the center of the object and rendering the textured quadrilateral at each frame using the current viewing transformation. This correction reduces "snapping" artifacts when the cached image is updated and increases the number of frames for which it yields an acceptable approximation to the object's appearance.

To gain the most from image caching, it is not sufficient 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 [5] 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 its 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 other nodes:

```
UpdateCaches(node, viewpoint)
if node is outside the view frustum then
    node.status ← CULL
    return
if node.cache is valid for viewpoint then
    node.status ← DRAWCACHE
    return
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 children 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 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 geometry. For an interior node, the process is essentially the same, except that instead

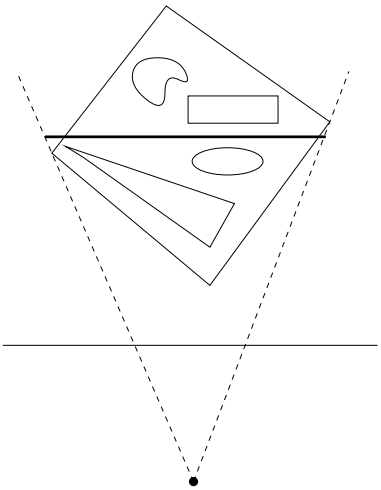


**Figure 1** Caching an image.

of considering the cost of drawing the geometry we consider the cost of drawing the 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 method.

The routine *CreateCache* points the camera towards the center of the node's bounding box and computes a rectangle on the screen that contains the node's image. This rectangle is obtained by transforming the corners of the bounding box from world coordinates to screen coordinates. If the rectangle does not fit into the viewport no image is cached. Otherwise, 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. 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 that goes through the center of the node's bounding box and whose normal is pointed at the viewpoint, as illustrated by the 2D diagram in Figure 1.

Once the cached images have been updated, we can proceed to render the scene into the frame buffer, traversing the BSP-tree back-to-front:

```
Render(node, viewpoint)
if node.status == CULL then
    return
if node.status ∈ {DRAWCACHE, DRAWGEOM} then
    Draw(node)
    return
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 that is used to determine whether a cached image is valid with respect to a given viewpoint and to estimate cache life-span. 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, how many frames can we expect the cached image to remain valid?

In order to answer these questions efficiently we need to define an error metric, which given a node in the hierachy, 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 $\epsilon$, the approximation is deemed acceptable, and the cache is considered valid. An important requirement for an acceptable error metric is that it should be fast to compute. We cannot afford to render the node and compare the result to the cached image, nor can we afford to examine the geometric content of the node, as the number of primitives contained in a node can be very large.

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

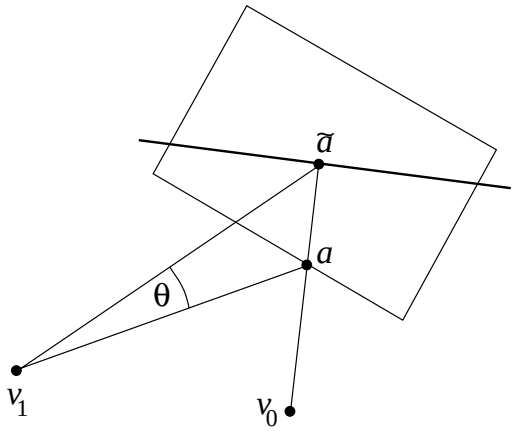



**Figure 2** Angular discrepancy.

use the 2D diagram shown in Figure 2 to define our error metric more precisely. The rectangle in this diagram represents the bounding box of a node in the hierarchy. The line segment crossing the bounding box represents the quadrilateral onto which the cached image is texture-mapped, as descibed 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 $a$:

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

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 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 $\epsilon$. Note that 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 still allowable. This is probably the best course of action for recording a walkthrough or a fly-by off-line.

In interactive walkthroughs the path of the viewer is not known in advance; however, it is reasonable to assume that the current velocity and acceleration are known at any frame, and

that an upper bound on the acceleration is 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 $\epsilon$:

$$\text{safety zone} \subseteq \{v \mid \mathit{Error}(v, v_0) \leq \epsilon\}, \tag{2}$$

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. Below we describe how safety zones are computed in our algorithm.

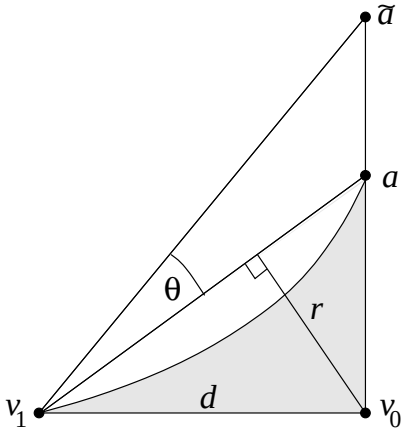



**Figure 3** Safety zones.

Consider the diagram in Figure 3. Let $v_0$ be the origin of a 2D coordinate frame in the plane with the $Y$ axis pointing toward $a$. Let $v_1$ be the farthest point on the $X$ axis, for which the angle $\theta$ subtended by $a$ and $\tilde{a}$ is equal to $\epsilon$. The viewpoints inside the triangle $(v_0, v_1, a)$ for which $\theta$ is equal to $\epsilon$ trace out an implicit second degree curve

$$x^2 + y^2 - \frac{(\tilde{A} - A)}{\tan\epsilon}x - (A + \tilde{A})y + A\tilde{A} = 0, \tag{3}$$

where $A$ and $\tilde{A}$ are the distances from $v_0$ to $a$ and $\tilde{a}$, respectively. The area under this curve (filled with grey in Figure 3) is the safety zone defined by points $a$ and $\tilde{a}$. Thus, we can conservatively define a spherical safety zone around $v_0$, whose radius is given by the shortest distance between the curve and $v_0$. The expression for this radius is quite complicated, and our implementation uses a simpler non-conservative approximation. We set the radius $r$ of the spherical safety zone to the shortest distance from $v_0$ to the line connecting $v_1$ to $a$:

$$r = \frac{Ad}{\sqrt{A^2 + d^2}} \tag{4}$$

$$d = \frac{\tilde{A} - A - \sqrt{(A - \tilde{A})^2 - 4A\tilde{A}\tan^2\epsilon}}{2\tan\epsilon} \tag{5}$$

In order to approximate the safety zone for a leaf node in the hierarchy we evaluate $r$ for each corner of the node's bounding volume, and take the smallest of the eight values. We then set the safety zone to be the axis-aligned cube inscribed inside a sphere of radius $r$ 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.

# 4 Partitioning

The current implementation of our system is geared towards navigation 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 constructing a BSP-tree [5] whose splitting planes are perpendicular to the $XZ$ plane. 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 under each subtree);
3. make the aspect ratio of each node's bounding volume as close as possible to 1, when projected onto the $XZ$ plane.

The first goal aims to reduce visual artifacts in our algorithm. The second and third goals help improve performance. Computing the optimal solution that satisfies these potentially contradictory goals appears to be very difficult. Therefore, our partitioning algorithm employs a simple greedy approach that is not optimal, but seems to work well in practice.

Given a list objects to partition, we look for gaps between the $XZ$ projections of the objects, place a splitting plane in the "best" gap we can find, and then 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 [9]. For each object, we compute its extent along each of $N$ vectors that evenly divide the unit circle perpendicular to the $Y$ axis ($N$ is a user specified parameter). Each splitting plane in the BSP-tree is constrained to be perpendicular to one of the $N$ vectors. Thus, if $N$ is 2, for example, our partitioning subdivides space into 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 (except for the ground height-field), such that the number of geometric primitives on each side of the gap is roughly equal. Sometimes such a gap does not 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.

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 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 making sure that there is a small amount of overlap in the geometry contained in neighboring leaf nodes. Thus, in addition to storing with each leaf node all of the geometric primitives contained in it, we examine a slightly "inflated" version of the node's bounding box, and add the extra geometry that is contained in the inflated box as well.

## 5 Results

This section demonstrates the performance of our method using a walkthrough of a complex outdoor scene. We tested performance on two different machines: the Onyx RealityEngine2 (RE2) with two 150 MHz R4400 processors, 256 megabytes of RAM, and 4 megabytes of texture memory; and the Indigo2 High Impact with a 200 MHz R4400 processor, 128 megabytes of RAM, and 4 megabytes of texture memory.

The outdoor scene used in these tests is a terrain of an island populated with 1,832 pine trees. The terrain consists of 32,768 triangles, and each pine tree consists of 7,779 triangles. The total number of triangles in the database is 14,283,896. 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 19 megabytes. Figure 9(a) shows a top view of the terrain (without the trees).

Constructing the BSP-tree for this database took 50 seconds on the Onyx. The resulting hierarchy has 1,826 leaf nodes and has up to 12 depth levels, which indicates that it is fairly balanced. Most leaf nodes contain a single pine 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. Figure 9(b) shows the resulting partitioning of the scene.

Partitioning the database causes an increase in the required storage. This increase is in small

part due to splitting the terrain triangles, but is primarily due to the need to "inflate" the bounding boxes of the leaf nodes, as described in Section 4. In all of the tests presented in this section, we used an inflation factor of 3 percent. This resulted in 8 percent increase in storage.

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 9 (the white squares are the control points of the B-spline). However, in order to get a better sense of how our algorithm would behave under interactive control, we did not take advantage of the fact that this path was known in advance.

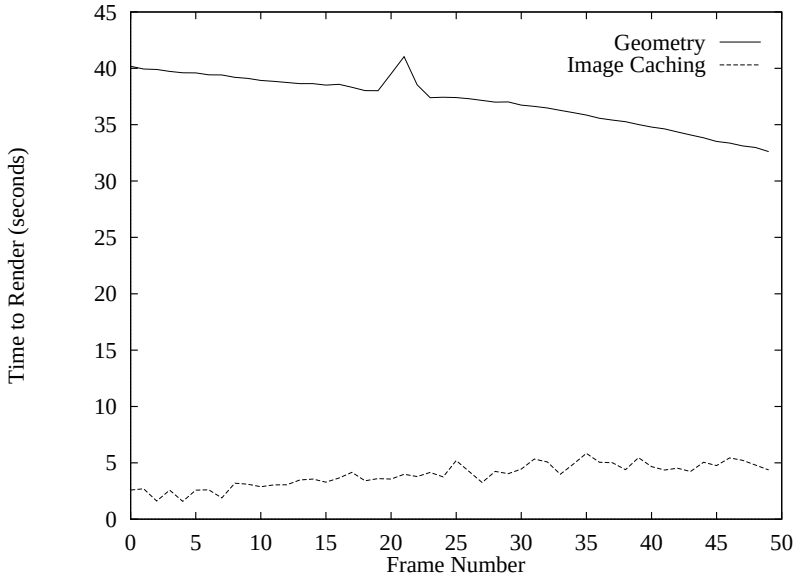
Figures 9(c) and 9(d) provide a "behind the scenes" look at our algorithm: Figure 9(c) shows, outlined in white, the textured quadrilaterals drawn by our method instead of the actual geometry, for a frame at the beginning of the walkthrough. Figure 9(d) shows a top view of the hierarchy for the same frame. The viewing frustum is indicated by green lines. Nodes outlined in red are rendered using their geometry. Nodes outlined in purple are culled, as they lie outside the view frustum. Nodes outlined in yellow have cached images. The quadrilaterals onto which these images are mapped are shown in black. Note that only a fraction of these quadrilaterals are actually drawn to the screen.

For comparison, the same walkthrough was performed using an algorithm that employs hierarchical view frustum culling (using the same BSP-tree), but renders all of the geometry contained in nodes that are inside the view frustum. Figures 10(a) and 10(b) show the same frame from the walkthrough rendered with image caching on the left and with geometry on the right. The two images are not identical, but it is very hard to tell them apart, except for the distant trees that appear slightly blockier when rendered with our method.

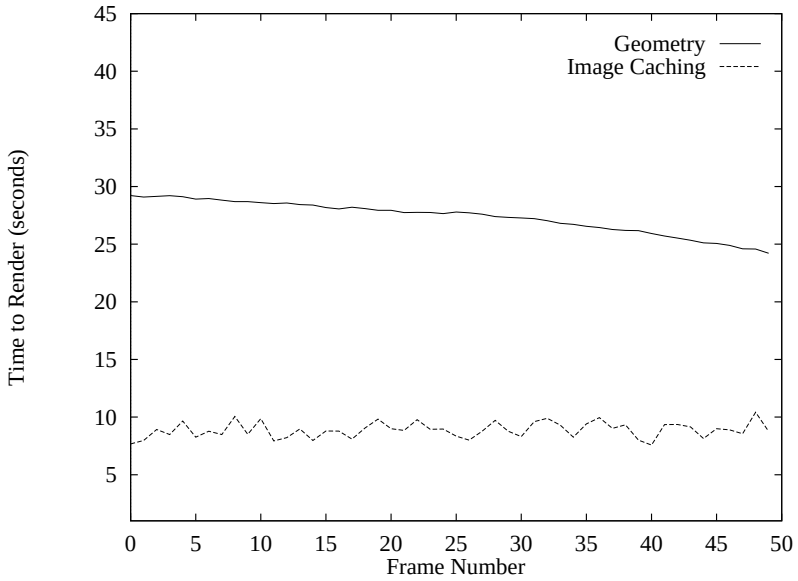
The path used in the walkthrough was designed to help us study the relative performance of image caching over a wide range of scene complexity. The walkthrough starts at the edge of the model, flies in toward the island, skirts around the land at treetop level, and then flies back out to the edge of the model.

Figures 4(a) and 4(b) show the rendering times for frames 100–150 in the walkthrough on the two different machines. Each graph plots the time to render each frame with image caching and with geometry rendering. The overall speedup for these frames on the Impact is a factor of 10.5 versus 3.1 on the RE2. The primary reason for this discrepancy is that the RE2 is about three times slower to copy a region from frame buffer to texture memory. Thus, caching an image on the RE2 is more expensive relative to the cost of drawing geometry. Because of that, fewer large cached images are found to be cost effective, resulting in reduction in the effectiveness of the hierarchy.

Unfortunately, a memory leak in the OpenGL server on the Impact did not allow us to ren-




(a) on Indigo2 High Impact




(b) on Onyx RealityEngine2

**Figure 4** Image caching versus rendering geometry

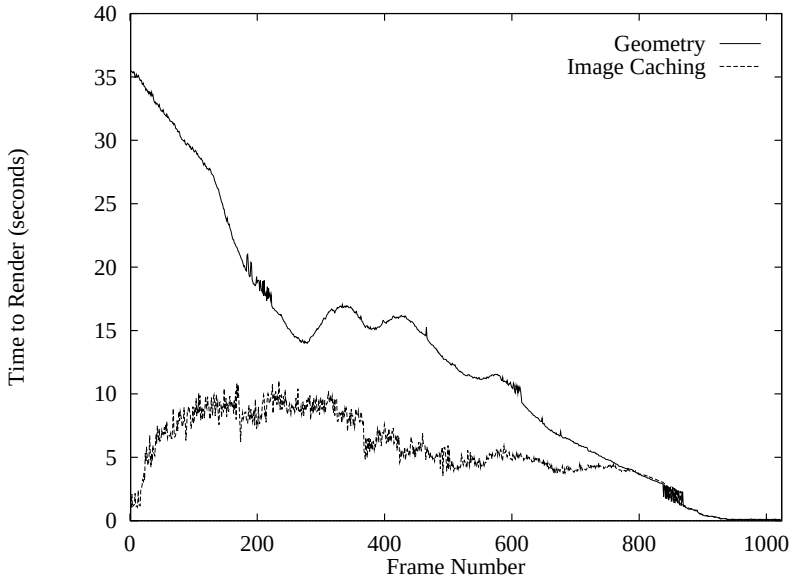



**Figure 5** Image caching versus rendering geometry (full walkthrough)

der the entire walkthrough on that machine. Therefore, statistics for the entire walkthrough could only be collected on the RE2. The overall speedup on this machine is 2.5. The times for the entire walkthrough are plotted in Figure 5. Our method's time to compute the very first frame of the walkthrough is off the vertical scale of the graph: it takes 120 seconds to compute. However, once the initial image caches have been computed, subsequent frames can be rendered very rapidly, with speedups between 30 and 40. The most geometrically complex frames in the walkthrough are those between 0 and 300. After that, the complexity slowly subsides until, beyond frame 900, the model becomes simple enough that image caching degenerates to simple view frustum culling.

An important limiting factor on the performance of image caching is the constraint imposed by OpenGL [13] that texture maps have dimensions in powers of 2, and that the smallest textures allowed are 64x64 pixels. As the camera approaches the island, increasing numbers of caches are becoming out of date and need refreshing. This results in many calls to OpenGL to define texture maps. Because of the texture size limitations of OpenGL, the vast majority of the pixels of the defined texture maps go unused. This overhead reaches a factor of 12 at the beginning of the walkthrough, as illustrated by the plot in Figure 6. The handling of so many unused pixels, results in a severe performance penalty for our image caching method.

Since our method utilizes frame-coherence, it is interesting to examine how different frame rates along the same path affect performance. Therefore, we rendered the entire walkthrough on the RE2 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

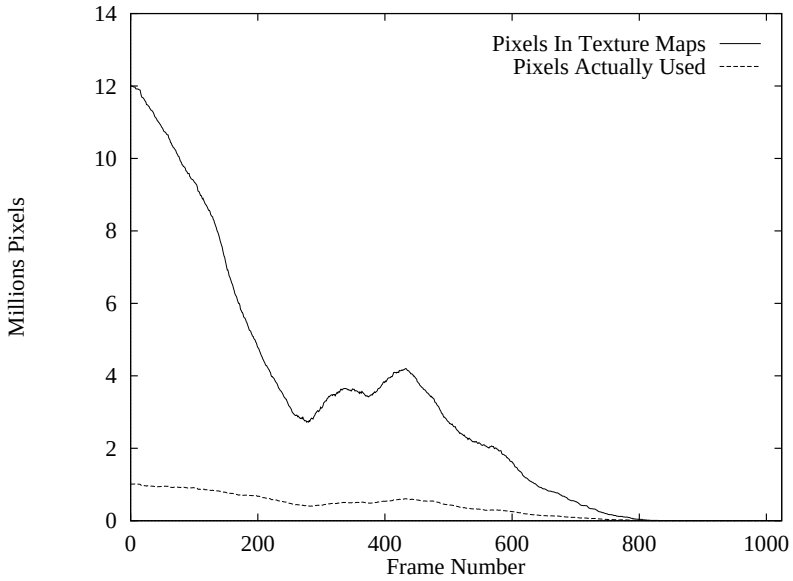



**Figure 6** Texture memory utilization

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 average speedup factor was computed, and the results are plotted in Figure 7. As expected, the speedup factor becomes larger, as more frames are rendered. Note that our method begins to exhibit a speedup with as few as 64 frames along the path.

Another interesting statistic is the behavior of our method as a function of overall scene complexity. The same walkthrough path was computed for two "decimated" versions of the scene, with 464 and 928 trees, instead of 1,832. Except for the number of trees, the three databases were identical. The overall speedup factors (computed on the RE2 for the entire walkthrough) for these three databases are plotted in Figure 8. As expected, the speedup factor introduced by our algorithm increases with the geometrical complexity of the scene.

In all of the tests 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. Figures 10(c) shows a part of a frame computed with an error threshold roughly corresponding to four pixels. Comparing this image with the one rendered using geometry (Figure 10(d)) reveals increased "ruggedness" along the silhouette of the mountains, as well as some "cracks" in the terrain, through which the blue background shows through. However, the speedup factor for this error threshold has increased to 18.3 (averaged over frames 100–150, on the Impact). A further doubling of the error threshold resulted in a speedup factor of 40.4.

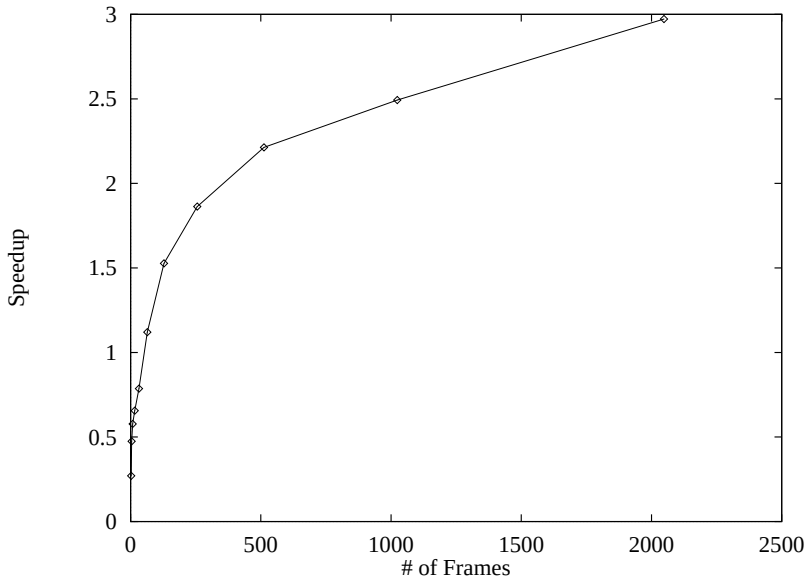



**Figure 7** Speedup as a function of frame rate

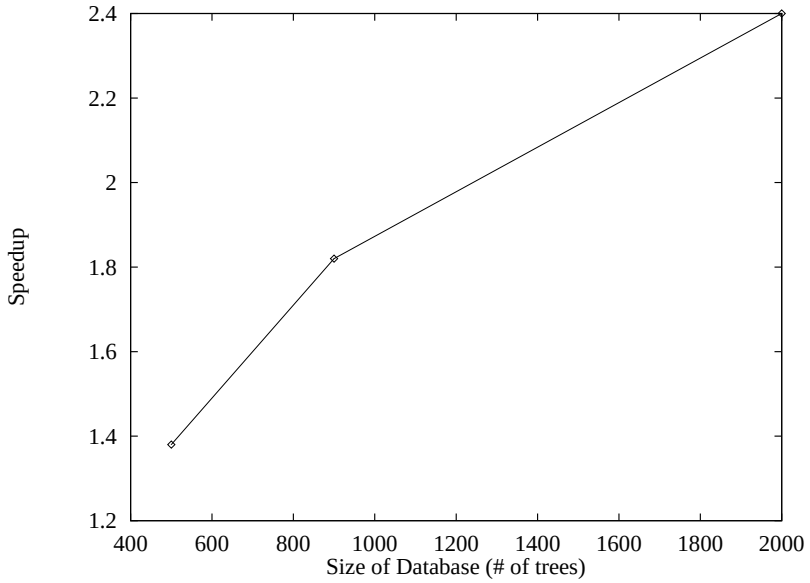



**Figure 8** Speedup as a function of scene complexity

## 6 Conclusions

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

**Animation** Our method is currently applicable only to static scenes. A challenging problem for further research is to allow scenes to contain objects that are capable of moving and/or deforming their geometry.

**Motion prediction** Our algorithm should consider caching images not only for nodes currently contained in the view frustum, but perhaps also for nodes that should come into view in the next few frames. This would help alleviate temporary degradations in rendering performance that may occur as a user travels into an area of the scene that is more complex. This 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 multi-resolution representation such as the one by Eck *et al.* [4]. 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, much in the same way as in the method of Regan and Pose [14].

In summary, we have presented a new method for accelerating walkthroughs of complex environments by utilizing frame coherence. We have demonstrated speedups of an order of magnitude on a current graphics architecture, the Indigo2 High Impact. The speedups increase with the complexity of the scene, as well as 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.

## 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] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Computer Graphics* Proceedings, Annual Conference Series, pages 279–288, ACM SIGGRAPH, August 1993.

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

[4] 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, pages 173–182, ACM SIGGRAPH, August 1995.

[5] 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.

[6] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics* Proceedings, pages 247–254, ACM SIGGRAPH, August 1993.

[7] 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.

[8] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Computer Graphics* Proceedings, pages 231–238, ACM SIGGRAPH, August 1993.

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

[10] Daivid Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, April 1995.

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

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

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

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

[15] 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*.

[16] 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.

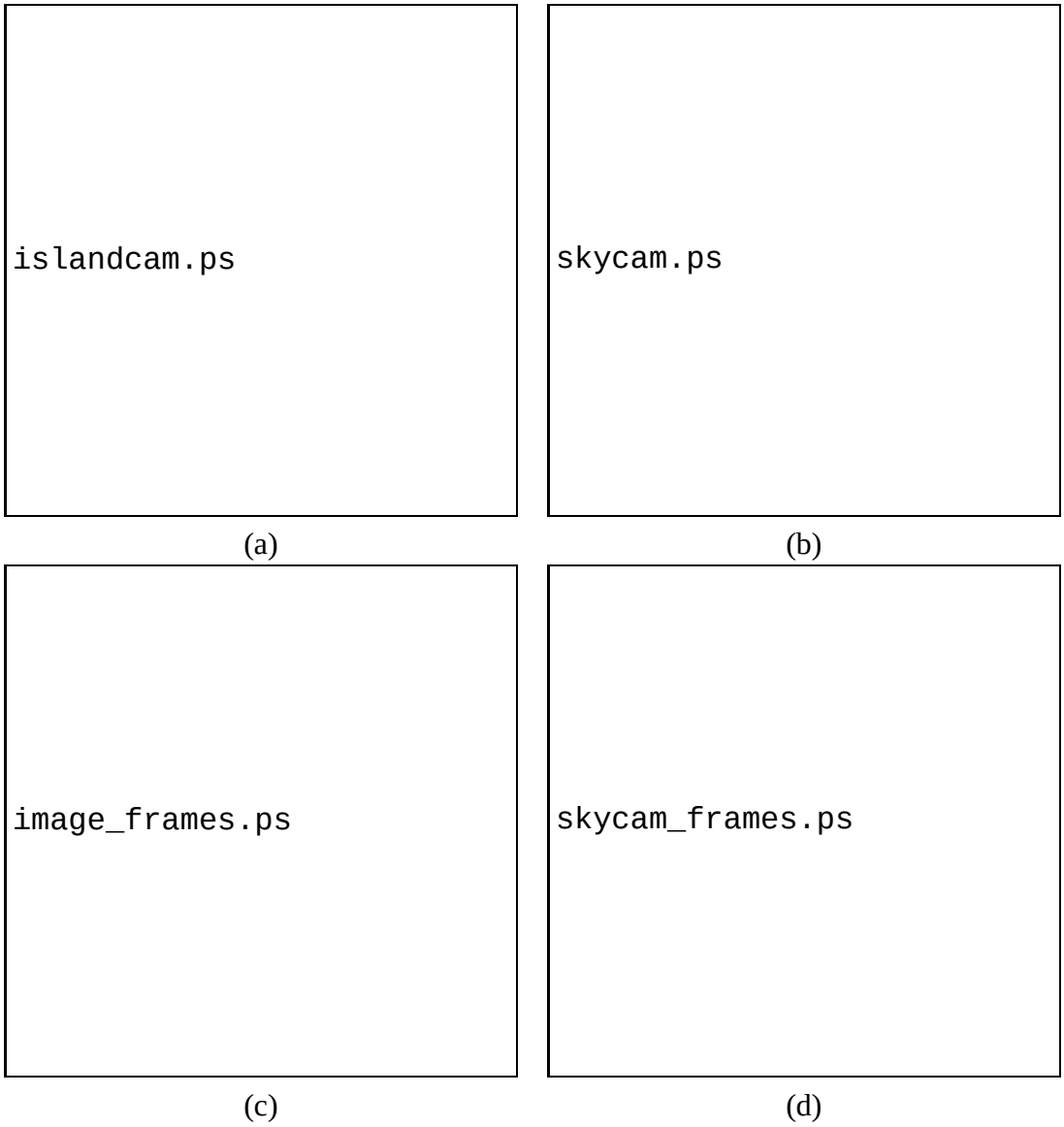



**Figure 9** Color plates

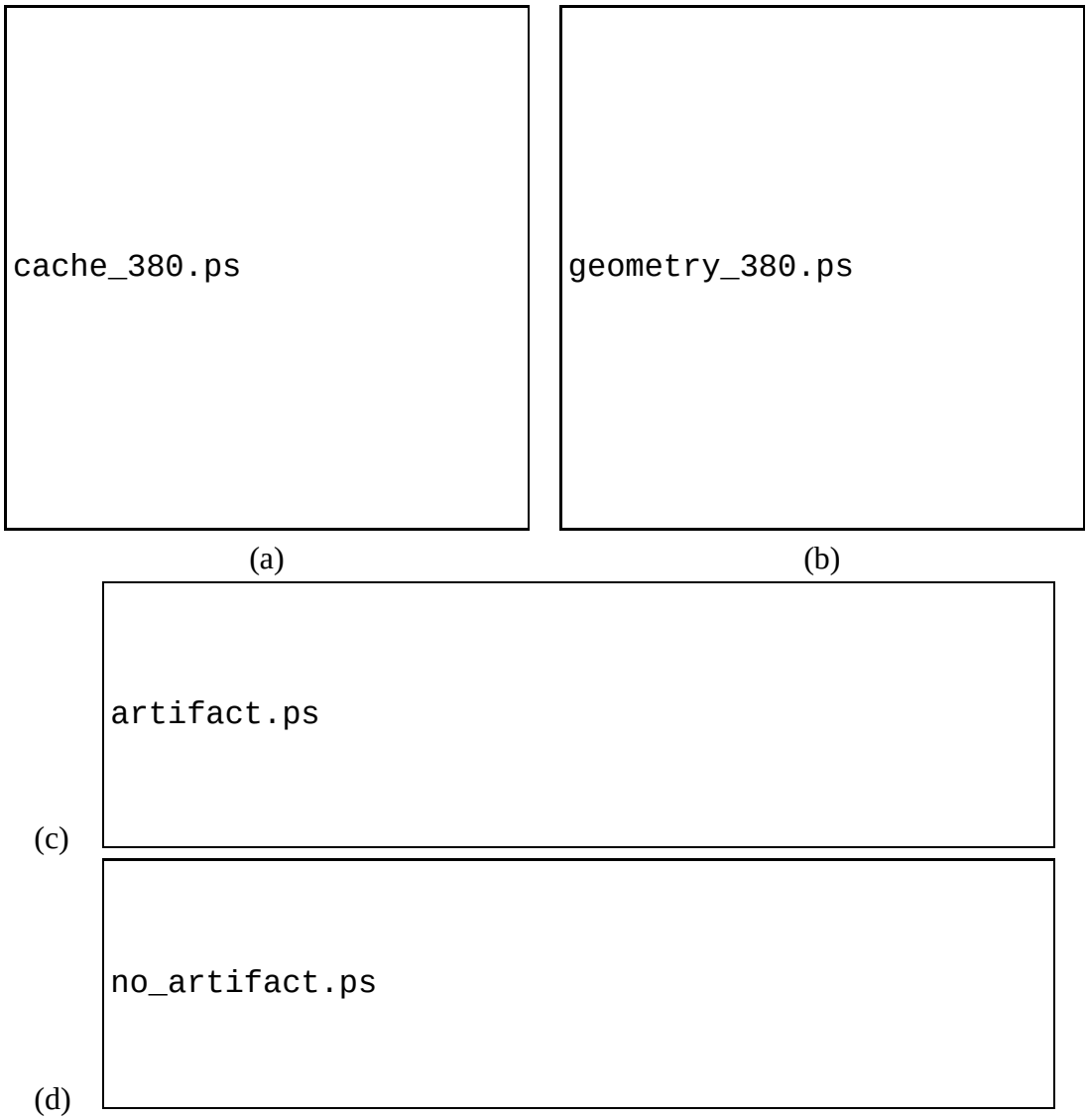



**Figure 10** Color plates