Automatic Image Placement to Provide A Guaranteed Frame Rate

Daniel G. Aliaga
Lucent Technologies Bell Laboratories

Anselmo Lastra
University of North Carolina at Chapel Hill

Abstract

We present a preprocessing algorithm and run-time system for rendering 3D geometric models at a guaranteed frame rate. Our approach trades off space for frame rate by using images to replace distant geometry. The preprocessing algorithm automatically chooses a subset of the model to display as an image so as to render no more than a specified number of geometric primitives. We also summarize an optimized layered-depth-image warper to display images surrounded by geometry at run time. Furthermore, we show the results of applying our method to accelerate the interactive walkthrough of several complex models.

1. INTRODUCTION

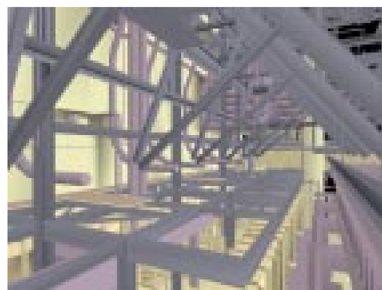
Large and complex three-dimensional (3D) models are required for applications such as computer-aided design (CAD), architectural visualizations, flight simulation, and virtual environments. These models currently contain hundreds of thousands to millions of primitives; more than high-end computer graphics systems can render at interactive rates. Often, the bottleneck for these applications is the geometric transformations required each frame. Thus, rendering acceleration methods endeavor to reduce the number of primitives sent to the graphics pipeline. For our work, we assume that by providing a bound on geometric complexity we can achieve a desired frame rate.

The demand for interactive rendering has brought about many algorithms for model simplification. For example, techniques have been presented for levels of detail [DeH91, Tur92, Coh96, Gar97, Hop97, Lue97], visibility culling [Air90, Tel91, Coo97, Zha97], and replacing objects with images [Mac95, Sha96, Scf96].

In this paper, we present an algorithm for limiting the maximum number of geometric primitives to render from all viewpoints and view directions by dynamically replacing selected geometry with images (Figure 1). We demonstrate our algorithm in a walkthrough system that allows for translation and yaw-rotation of a 60-degree or greater view frustum through several large models. Using images is desirable because we can render them in time proportional to the number of pixels. In addition, increasingly simplified geometric levels of detail, viewed from the same distance, eventually lose shape and color information. A fixed resolution image, on the other hand, maintains an approximately constant display cost and, given sufficient resolution, maintains the apparent visual detail.

Email: aliaga@research.bell-labs.com, lastra@cs.unc.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGGRAPH 99, Los Angeles, CA USA
Copyright ACM 1999 0-201-48560-5/99/08 . . . $5.00



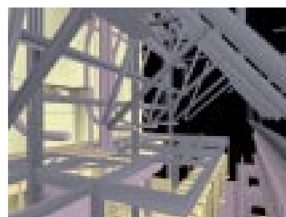


Figure 1. *Geometry+Image Example. These three snapshots illustrate an example rendering of a power plant model. The top snapshot is what the viewer actually sees. In the bottom left snapshot, we render the portion represented as geometry. In the bottom right snapshot, we render the portion represented as a warped image.*

Our results show that we are able to visualize geometric models ranging from 850K to 2M triangles using as little as one-tenth of the geometry and no more than 3.8GB of image data. Based on our empirical results and an analysis of our algorithm, we also predict the expected best case and a near worst-case performance for additional models.

We believe that ultimately a rendering system should combine algorithms such as ours with geometric simplification and other rendering acceleration methods [Ali99]. The system should automatically choose the most appropriate method(s) to use.

1.1 Overview

Our approach consists of a preprocessing component to determine the subsets of a model to replace with images and a run-time component for displaying images and conventional geometry.

The preprocessing takes as input a 3D model, stored in a hierarchical spatial partitioning data structure (e.g. octree) [Cla76] and creates a non-uniform grid of points adapted to the local model complexity. At each grid point, an image-placement process selects the smallest and farthest subsets of the model to remove from rendering to meet a fixed geometry budget from that location. Images are then created to represent each selected subset.

At run-time, we select an image from a grid point near the current viewpoint. The geometry behind the projection plane of the image is culled while the remaining geometry is rendered normally. Our grid-point selection algorithm guarantees that we always meet our bound on the amount of geometry to render. We could display the

image using texture mapping, but this approach would only yield the correct perspective from the viewpoint where the image was created. Instead, we adopted the strategy of McMillan and Bishop [McM95] to warp images, enhanced with depth, to get proper perspective. Furthermore, to reduce the number of disocclusions that occur because of this technique, we warp layered depth images (LDIs) [Max95, Sha98]. We have observed that, on average, most of the pixel samples of our LDIs are in the first two to four layers [Pop98]. Thus, if we can afford the approximately constant time it takes to warp an image, we can render any size 3D model at a guaranteed frame rate.

2. RELATED WORK

A large body of literature has been written on how to reduce the geometric complexity of 3D models. For the purposes of this paper, we can classify related work into three main approaches:

- frame-rate control,
- view-dependent simplification, and
- image caching.

Funkhouser and Sèquin [Fun93] presented a system that, at run-time, selects levels-of-detail (LODs) and shading algorithms, in order to meet a target frame rate. The system maintains a hierarchy of the objects in the environment. It computes cost and benefit metrics for all of the alternative representations of objects and uses a knapsack-style algorithm to find the best set for each frame. If too much geometry is present, detail elision is used.

Maciel and Shirley [Mac95] expanded upon this and increased the representations available for the objects. A set of *impostors*, which include LODs, texture-based representations and colored cubes, can be used to meet the target frame rate.

Flight simulators use several techniques to achieve high frame rates [Sch83, Mue95]. For example, during each frame the system evaluates scene complexity in order to determine the LODs and terrain texture resolutions. When the current selection takes too much time to render, the LOD switching distance and texture resolution are reduced.

View-dependent simplification algorithms support maintaining constant geometric complexity every frame [Hop97, Lue97]. Alternately, they can maintain a bounded screen-space error during simplification. Unfortunately, depending on the amount of simplification needed and on the scene complexity, objects will be merged and details will eventually be lost.

Various systems have been presented that use image-based representations (typically texture-mapped quadrilaterals) to replace subsets of the model. The source images are either pre-computed [Mac95, Ali96, Ebb98] or computed on the fly [Sha96, Scf96]. Metrics are used to roughly control image quality but not the amount of geometry to render. Aliaga and Lastra [Ali97] and Rafferty *et al.* [Raf98] used images to accelerate rendering in architectural models. Doorways (i.e. portals) are replaced with images and only the geometry of the current room is rendered. Both Darsa *et al.* [Dar97] and Sillion *et al.* [Sil97] constructed a simplified mesh to represent the far scene. In the worst case, the complexity of the mesh is proportional to the screen resolution. However, neither system provided control of the number of primitives required to draw the mesh or any nearby geometry.

Regan and Pose [Reg94] created a hardware system that employed large textures as a backdrop. The foreground objects were given a higher priority and rendered at faster update rates. Image composition was used to combine the renderings. Their approach helped to reduce the apparent rendering latency but did not control the number of primitives rendered.

3. AUTOMATIC IMAGE PLACEMENT

The goal of our preprocessing is to automatically compute what geometry to replace with images so as to limit the number of primitives to render for an arbitrary 3D model. An image and the subset of the model it culls define a *solution* for a given viewpoint and view direction. We refer to the position of a quadrilateral, corresponding to both the projection and near plane for rendering a model subset, as the *location* of the associated image. Clearly it is impractical to compute a solution for all viewpoints and view directions. Instead, we exploit a property of overlapping view frusta to limit the number of positions and take advantage of hierarchical spatial data structures, used for view-frustum culling, to conservatively sample the view directions. We assume that during preprocessing and run time the same field-of-view (FOV) is used. In this section, we present a recursive method to create a *grid of solution points,* and a process to place the images associated with each of these grid points. Figure 2 provides a summary of the preprocessing pipeline.

```
1. Enqueue all grid points of a uniform grid
2. Repeat
3.   Dequeue grid point
4.   Create view-directions set
5.   While (view direction with most number of
           primitives > geometry budget)
6.     Compute smallest and farthest octree cell
         subset to remove from rendering to meet
         the target geometry budget
7.     If (the resulting image is outside the
           star-shape for the given grid point)
         Discard solution
         Subdivide local grid
         Enqueue new grid points
8.     Else
         Compute a layered-depth image to
           represent the octree-cell subset
       Endif
     Endwhile
9. Until (no more unprocessed grid points)
```

Figure 2. *Preprocessing Algorithm Summary.*

3.1 Enclosed View Frusta

Our algorithm exploits the fact that a semi-infinite frustum (A in Figure 3) completely enclosed by another (B in Figure 3), with the same FOV, contains no more geometry than that included in the enclosing frustum. For any viewpoint, such as A, we select the closest grid point contained within the reverse projection of the view frustum (shown in dashed lines). If we have a solution that bounds the total amount of geometry for B, we also have a sufficient solution for a viewpoint such as A (an enclosed frustum with the same FOV and view direction). Since we are considering the total amount of geometry in the view frustum, occlusions are not an issue. Thus, a finite grid of solutions is sufficient to limit the complexity for all viewpoints within the grid. Our preprocessing task reduces to

- finding a good set of the aforementioned grid solution-points to sample the model space (Section 3.2), and
- finding a solution (e.g. the appropriate subsets of the scene to represent as image) for the infinite number of view directions, at each grid point (Section 3.3).

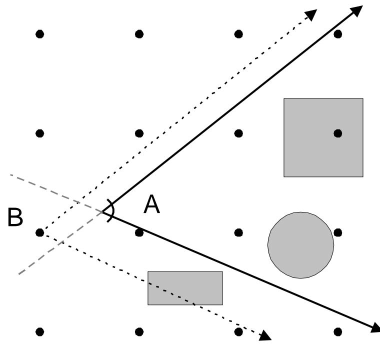


Figure 3. *Enclosed View Frusta. Frustum B has the same FOV and view direction as frustum A. Furthermore, frustum B is centered on the closest grid point contained in the reverse projection of frustum A (as indicated by the lightly dashed lines). Frustum A contains no more geometry than that in frustum B. Hence, we can use an image computed for B to limit rendered geometry from viewpoints such as A.*

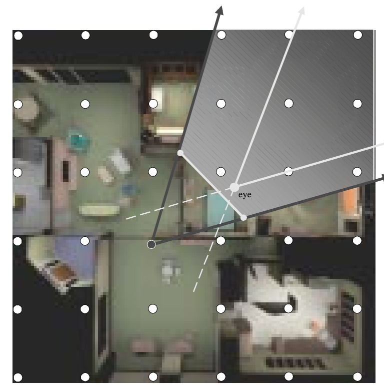


Figure 4. *Image Placed Behind the Eye. We show a top-down view of an architectural model. A plane of points from a uniform grid is shown. The projection plane of the image (yellow line) computed for the closest grid point in the reverse view frustum (dashed lines) is behind the eye. This problem occurs because scene complexity forces the image to be very near its grid point. Geometry replaced by the image is shaded in yellow.*

3.2 Solution Grid

The preprocessing begins by creating a sparse, uniform grid of points that spans the model space. The problem we encounter with this sparse grid is that the image placement may not be valid. Recall that, for a particular point, we choose a solution whose grid point is behind us. It is possible, as shown in Figure 4, that the projection plane used to create the image was placed nearer to its grid point than our current viewpoint (this occurs in areas of the model with complex geometry). We need to ensure that the grid is dense enough to guarantee that the projection plane of selected images will always be in front of any eye location.

3.2.1 Star-Shapes

The first step is to determine the locus of eye locations for which a given grid point might be selected (because it's the closest grid point in the reverse projection of the eye's frustum). The left half of Figure 5 depicts a grid of points. This grid has a uniform distribution of points and is defined to be a level 0 grid -- thus an even-level grid. If we allow rotations only about the vertical axis (i.e. *y*-axis) and translations only in the plane, the right half of Figure 5 shows the locus of viewpoints (we refer to this as a *star-shape*) that might have grid point a_4 as the closest grid point in the reverse projection of a square view frustum of FOV 2α. *E* is the farthest eye location from which there is a view direction that still contains a_4 as its closest grid point in the reverse view frustum. The distance s_{2k} is equal to $r_{2k}/(2\tan\alpha)$ where r_{2k} is the separation between grid points. As long as the FOV is greater than or equal to 54 degrees (i.e. $2\alpha \geq 54$) s_{2k} is less than or equal to r_{2k}. Thus, we can approximate the star-shape with a circle of diameter $4r_{2k}$. Using symmetry, we can conservatively estimate the locus of eye locations with a sphere of diameter $4r_{2k}$.

Hence, for a practical FOV of 60 degrees or greater, we can prevent the problematic situation by ensuring that no grid point has an image placed within its star-shape. If we superimpose the star-shape on the problem case of Figure 4, we see that the image is indeed inside the star-shape. Our algorithm will not work well with narrower fields-of-view because the selected grid point might be too far behind the eye and the star-shape excessively large.

Eye positions near the edge of the model might not contain a grid point in the reverse frustum. We inflate the grid by two points in all the six directions (i.e. positive and negative *x*-, *y*-, and *z*-axes) so that such eye positions have a grid point behind them.

3.2.2 Recursive Subdivision

To ensure that all viewpoints in the model have a valid image solution, we recursively reduce the size of star-shapes by locally subdividing the grid. Our goal is to subdivide until the images computed for all of the grid points are always in front of any possible eye position. The recursion alternates between two sets of rules: one for *even levels* (*2k*) and one for *odd levels* (*2k+1*). We first introduce grid points at the midpoints of the existing points. Then, we introduce the complementary points to return to a denser original configuration. Figure 6 depicts a grid point subdivided through two levels. At each new level, we verify that, for all points, a valid image placement can be produced (Section 3.3). We recursively subdivide points that fail until all have image placements in front of any eye position that can use them. For the odd-level grid, we use a slightly different star-shape that we can approximate with a sphere of diameter $6r_{2k+1}$ (for more details, see [Ali98]). Figure 7 shows a grid automatically computed for one of our test models.

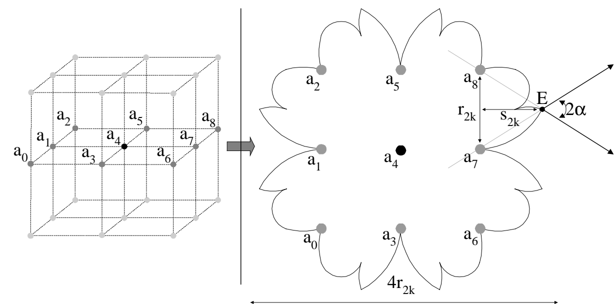


Figure 5. *Star-Shape. To the left, we show a uniform grid of 3x3x3 viewpoints. To the right, we show a top-down view of the horizontal plane defined by grid points a_0-a_8. If we rotate about the vertical axis and translate a square view frustum, the star-shape represents the plane of locations that might use grid point a_4. The distance s_{2k} equals $r_{2k}/(2\tan\alpha)$; thus, for a FOV $2\alpha \geq 54$ degrees, $s_{2k} \leq r_{2k}$. We can approximate the star-shape with a sphere of diameter $4r_{2k}$.*

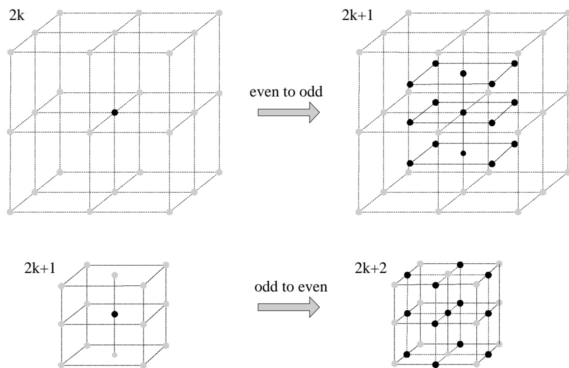


Figure 6. Even- and Odd-Level Grid Subdivision. We show even-to-odd and odd-to-even grid point subdivisions. In the upper half, we subdivide a level 2k point, in the middle of a grid, to produce 15 level 2k+1 points. In the lower half, we subdivide a level 2k+1 point to produce 13 level 2k+2 points -- thus returning to an even-level grid.

To maintain valid star-shapes, we ensure that all neighboring points have a difference of at most one recursion level. This is similar to a problem that occurs when tessellating curved surfaces. If two adjacent patches are tessellated to different resolutions, T-junctions (and cracks) occur at the patch boundary. We must perform an additional tessellation of the intermediate region.

3.3 Image Placement at a Grid Point

The goal of our image-placement process is to limit the number of rendered primitives for all view directions centered on a given grid point. The basic approach we have followed is to create a conservative sampling of view directions. Then, for each sampled direction, we ensure that the target primitive count is not exceeded.

3.3.1 View-Directions Set

The first step of our image-placement process creates a *view-directions set* for the view directions surrounding a given grid point. The simplest set could be created using a constant sampling of view directions and the same FOV as at run time. But, since there might be a large variation in the amount of geometry surrounding a particular grid point, it is not clear how many samples to produce. Thus, we exploit the fact that the model is stored in an octree (or another hierarchical spatial-partitioning data structure) and create a sampling using the same FOV as at run time but that adapts to the local model complexity.

In an octree, culling is applied on a cell by cell basis, not to the individual geometric primitives. Consider only allowing yaw rotation of a pyramidal view frustum centered on a grid point. The visible set of octree cells remains constant until the left or right edge of the view frustum encounters a vertex from an octree cell, at which point view-frustum culling adds or removes the corresponding octree cell from the set (Figure 8).

The above fact turns the infinite space of view directions into a finite one, consisting of a set of angular ranges. Each angular range is inversely proportional to the model complexity in the view frustum: more complexity will generate more octree cells, hence the visible set of cells will typically remain constant for a smaller angular range. We compute the angular ranges for all grid points by starting with a common initial direction (z=-1 axis). Then, we rotate clockwise until the visible set of octree cells changes. We represent the ranges by the view direction at which

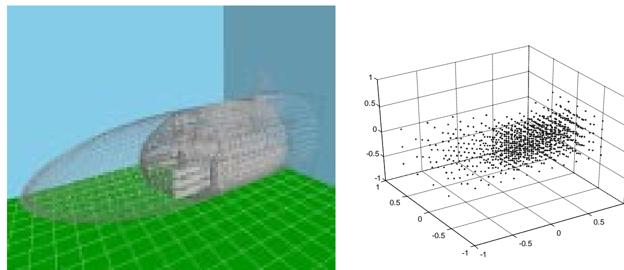


***Figure 7.** Torpedo Room Grid. This figure illustrates an automatically computed solution grid for a torpedo room model. The left snapshot shows an exterior view of the model rendered in wireframe. The right plot shows a grid of 1557 points from where 2333 LDIs are computed to limit the number of primitives per frame to at most 150,000 triangles. Note the cluster of geometry in the middle of the left snapshot and the corresponding cluster in the grid.*

this occurs, thus creating a sampling of view directions with more samples in areas with more model complexity in the view frustum.

If the octree has a large number of leaf cells, we might sample a large number of view directions per grid point, thus increasing the overall number of views and the preprocessing time. Therefore, we select an arbitrary tree depth to act as leaf cells (*pseudo-leaf cells*) and to conservatively represent the views around a grid point. This shallower octree will cause geometry to be more aggressively culled and generate slightly larger and nearer images than strictly necessary.

3.3.2 Image Placement

The second step of our image-placement process computes octree-cell subsets to *not* render from a given grid point. Then, at run time images are placed immediately in front of these subsets and the subsets themselves are culled. We define

- a *geometry budget P* – this value represents the maximum number of geometric primitives to render during a frame, and
- an *optimization budget* P_{opt} – this value is slightly less than the actual geometry budget. A larger difference between these two budgets requires fewer images per grid point but increases the overall number of grid points.

For each grid point, the image-placement process starts with the view direction containing the most primitives. If the view exceeds

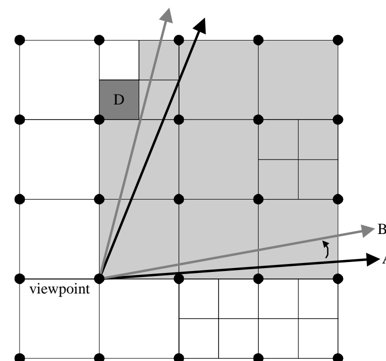


***Figure 8.** View-Directions Set. This example depicts a 2D slice of an octree (i.e. quadtree) and two view frusta. If we rotate counter-clockwise about the viewpoint from view frustum A to view frustum B, the group of octree leaf cells in view remains the same. Only if we rotate beyond B, will cell D be marked visible, thus changing the group of visible octree cells.*

our geometry budget, we perform a binary search through the space of contiguous, visible octree-cell subsets and employ a cost-benefit function to try to select the best subset to remove from rendering in order to meet the optimization budget. We compute one contiguous subset per view because it will require at most one image per frame—this simplifies the run-time system. If, after removing the subset, there is another view that violates the geometry budget, we compute a different subset for that view. The process is repeated until the geometry budget is met for all views.

In the following three sections, we provide more details on our cost-benefit function, our representation scheme for octree cell subsets, and our inner image-placement loop.

3.3.2.1 Cost-Benefit Function

In order to determine which subset of the model to omit from rendering, we define a cost-benefit function *CB*. The function is composed of a weighted sum of the cost and benefit of selecting a given subset. It returns a value in the range [0,1].

The *cost* is defined as the ratio of the number of primitives g_c to render after removing the current subset, to the total number of primitives G_c in the view frustum.

$$Cost = g_c/G_c$$

The *benefit* is computed from the width I_w and height I_h of the screen-space bounding box of the current model subset and the distance d from the grid point to the nearest vertex of the subset.

$$Benefit = B_1*(1-max(I_w,I_h)/max(S_w,S_h)) + B_2*d/A$$

The final cost-benefit function *CB* will tend to maximize the benefit component and minimize the cost. A function value near 0 implies a very large-area subset placed directly in front of the eye that contains almost no geometry; 1 implies a subset with small screen area placed far from the grid point that contains all the visible geometry.

$$CB = C*(1-Cost(g_c, G_c)) + B*Benefit(I_w,I_h,d)$$

The constants of the above equation are

- *C, B:* weights of the cost and benefit components
- B_1, B_2: weights of the image size and depth components,
- *A*: length of the largest axis of the model space, and
- S_w, S_h: screen width and height.

3.3.2.2 Representing Octree-Cell Subsets

The image-placement process will search through the space of all contiguous, visible subsets of octree cells associated with the current view. This process does not need to create all subsets but does need to enumerate them in order to perform the binary search. Thus, we need a fast and efficient method to represent and enumerate contiguous subsets. Furthermore, the cost-benefit function needs to compute the screen-space bounding box and primitive count of octree-cell subsets.

Our approach is to position a screen-space bounding box to exactly surround the projection of a contiguous group of octree cells. Since a larger number of octree leaf cells are rendered in high complexity areas, we snap between cells to finely change the bounding box in areas of high complexity and to coarsely change the bounding box in areas of low complexity.

We represent an arbitrary, contiguous octree-cell subset with a 6-tuple of numbers. Each number is an index into one of 6 sorted lists representing the leftmost, rightmost, bottommost, topmost, nearest, and farthest borders of a subset (Figure 9). All octree cells whose indices lie within the ranges defined by a 6-tuple are members of the subset. We can change one of the bounding planes of the subset to its next significant value by simply changing an index in the 6-tuple. Furthermore, it is straightforward to incrementally update the screen-space bounding box of the subset as well as the count of geometry.

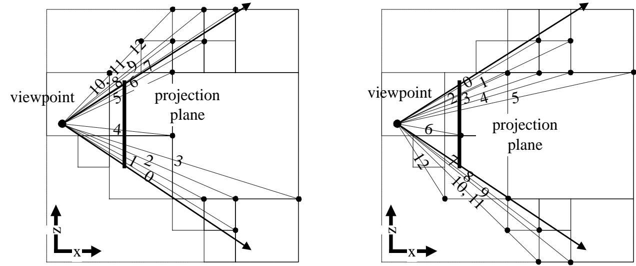


Figure 9. *Octree-Cell Subset Representation. These diagrams show two (of the six) sorted lists of a 2D slice of the octree cells in a view frustum (i.e. quadtree). The left diagram shows the bottom-to-top ordering of the topmost coordinates of the visible octree cells. The right diagram shows the top-to-bottom ordering of the bottommost coordinates. A subset of the visible octree cells can be represented by a minimal index m from the left diagram and a maximal index M from the right diagram. All cells that have a minimal index ≥ m and a maximal index < M are part of the 2-tuple [m,M]. By using this same notation in the XY plane and YZ plane, we can represent an arbitrary contiguous subset in 3D using a 6-tuple of such indices.*

For example, consider a view with 100 octree cells (each cell is labeled from 0 to 99). The 6-tuple [0,99,0,99,0,99] represents the entire set. To obtain a subset whose screen-space projection is slimmer in the *x*-axis, we increment the "left border" index, e.g. [1,99,0,99,0,99], or decrement the "right border" index, e.g. [0,98,0,99,0,99]. If two or more octree cells share a screen space edge, we consider them as one entry.

3.3.2.3 Inner Image-Placement Loop

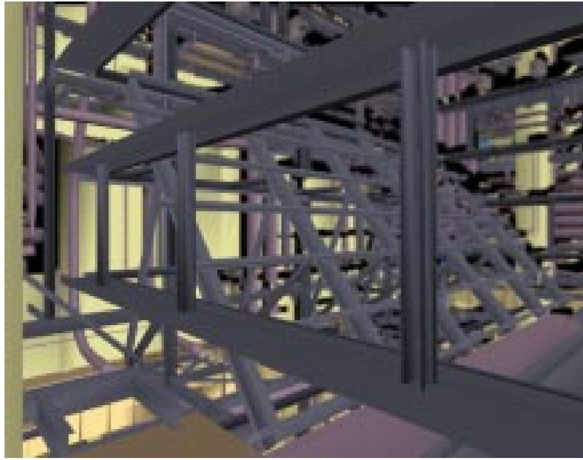
Our inner loop uses the cost-benefit function and the 6-tuple subset representation to select the octree-cell subset to remove from rendering in order to meet the optimization budget. The loop starts with the set of all octree (leaf) cells in the view frustum, e.g. [0,99,0,99,0,99]. At each iteration, by moving the border along the *x*-, *y*-, and *z*-axes, we produce five new subsets. Specifically, the

- near border is moved halfway back (e.g. [0,99,0,99,50,99]),
- top border is moved halfway down (e.g. [0,99,0,50,0,99]),
- bottom border is moved halfway up (e.g. [0,99,50,99,0,99]),
- right border is moved halfway left (e.g. [0,50,0,99,0,99]), and
- left border is moved halfway right (e.g. [50,99,0,99,0,99]).

(note: since an image is meant to replace geometry behind the image's projection and near plane, we don't change the far border, the sixth-tuple value, because it will not affect image placement)

To decide which of these subsets to use next, we recurse ahead a few iterations with each of the five subsets. We then choose the subset that returned the largest cost-benefit value. In case of a tie, preference is given to the subsets in the order listed. Iterations stop when the subset no longer culls enough geometry.

We then define the projection plane for the image to be a quadrilateral perpendicular to the current view direction and



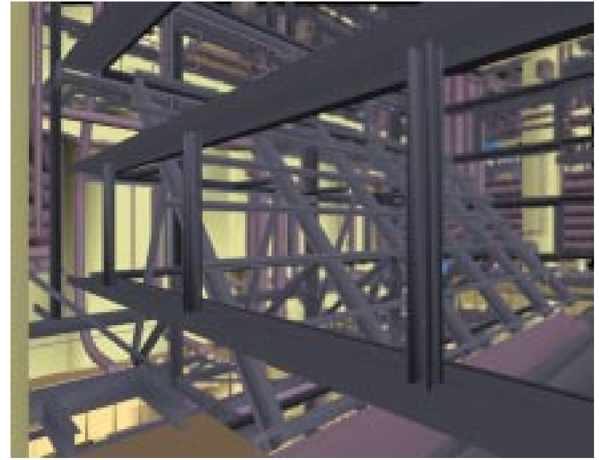


***Figure 10**. Single- and Multi-Reference-Image LDI. (Left) In this snapshot, we see geometry (foreground) and a 512x384-pixel LDI (background) created from a single reference image. The viewpoint is as far as possible from the center-of-projection before switching to another image. Notice the presence of disocclusions that appear as black gaps. (Right) In this snapshot, we are at the same viewpoint, but we use 9 reference images to construct the LDI. In both cases, we apply a 3x3 convolution-kernel to smooth the warped image.*

exactly covering the screen-space bounding box of the computed subset. The four corners of the quadrilateral, together with the current grid point, determine a view frustum for creating the image to replace the subset. Section 4 explains in more detail how we create the images and display them at run time. For now, we simply associate the computed subset with this view direction and grid point.

Next, we temporarily cull the subset from the model and move on to the next most expensive view from the current grid point. If the total number of primitives in the view frustum is within the geometry budget, we are done with this grid point. Otherwise, we restore the subset to the model and compute another solution for the new view. By using the full model during each pass, we enforce solutions that contain exactly one subset, thus enabling us to warp no more than one image per frame.

4. IMAGE WARPING

The preprocessing component has determined the subsets of the model to replace with images—we must now create and display these images. At each grid point, we have the necessary (camera) parameters to create a reference image that accurately depicts the geometry from that position. But each image must potentially represent the selected geometry for any viewpoint within the associated star-shape.

One alternative is to use per-pixel depth values to dynamically warp images to the current viewpoint [McM95]. Unfortunately, warping a single depth image has the limitation that surfaces not visible in the original reference image appear as gaps in the rendered image (Figure 10, left).

Layered Depth Images (or LDIs) [Max95][Sha98] are the best solution to date for the visibility errors to which image warping is prone. They are a generalization of images with depth since, like regular images, they have only one set of view parameters but, unlike regular images, they can store more than one sample per pixel. The additional samples at a pixel belong to surfaces, which are not visible from the original center-of-projection (COP) of the image, along the same ray from the viewpoint. Whenever the LDI is warped to a view that reveals the (initially) hidden surfaces, the samples from deeper layers will not be overwritten and they will naturally fill in the gaps that otherwise appear (Figure 10, right).

LDIs store each visible surface sample once, thus eliminating redundant work (and storage) as compared to warping multiple reference images. An additional benefit is that LDIs can be warped in McMillan's occlusion-compatible order [McM95]. This order guarantees correct visibility resolution in the warped image.

4.1 Optimizing LDIs for the Solution Grid

We create the reference images for constructing an LDI from viewpoints within the star-shape surrounding each grid point. Thus, we can do a good job of sampling all potentially visible surfaces. Consider a solution image, *A*. All outward-looking viewpoints from which a view frustum can contain the image quadrilateral are represented approximately by an hemi-ellipsoid centered in front of the grid point. Figure 11 depicts a 2D slice of this configuration. For a more distant image *B*, the region is more elongated (e.g. the dashed hemi-ellipsoid). To construct the LDI, we select reference image COPs that populate this space.

We choose a total of eight construction images and one central image to create a LDI and to eliminate most visibility artifacts. The central LDI image is created using the grid point itself as the

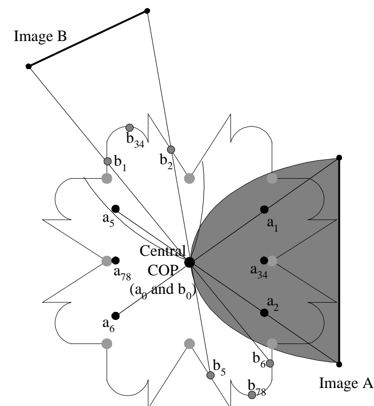


***Figure 11.** Construction Images for a LDI. Image A is placed immediately outside a star-shape. Given a fixed FOV, the locus of outward-looking viewpoints within the star-shape from where there exists a view direction that contains the image quadrilateral is depicted by the shaded hemi-ellipsoid. The central COP is placed at grid point a_0; the 8 construction-images a_1-a_8 are placed as indicated. Similarly, b_0 and b_1-b_8 are the COPs for a farther away image B.*

COP (a_0 and b_0 in Figure 11). Four construction images are created from COPs at the middle of the vectors joining the grid point and the midpoints of each of the four edges of the image quadrilateral (a_{1-4} and b_{1-4} in Figure 11). An additional set of four construction images is defined in a similar way but extending behind the grid point (a_{5-8} and b_{5-8} in Figure 11). We warp the pixels of the nearest construction image first. This prioritizes the higher quality samples of the nearer images.

Most of the visibility information is obtained from the central image and the first four construction images. They sample most of the potentially visible surfaces. The images behind the grid point help to sample visibility of objects in the periphery of the FOV. In practice, this heuristic method does a good job.

5. IMPLEMENTATION

We implemented our program in C++, on a Silicon Graphics (SGI) Onyx2, 4 R10000's @ 195 MHz and Infinite Reality graphics. The program takes as input the

- octree of the 3D model,
- geometry and optimization budget,
- FOV to use for both preprocessing and run time,
- resolution of the initial viewpoint grid (minimum 3x3x3, i.e. the size of an even-level star-shape),
- tree depth to use for the octree (pseudo) leaf cells, and
- cost-benefit constants ($C = 0.4$, $B = 0.6$, $B_1 = 0.1$, $B_2 = 0.9$).

The preprocessing program uses a single processor to create and subdivide the grid; afterwards, multiple processors are used to simultaneously compute the image placements. We use spheres to approximate the star-shapes. If for any view direction, the amount of geometry inside the FOV and within the sphere exceeds the geometry budget, the grid point is subdivided. Once the grid has been created, the grid points are divided among three (of the four) processors. Each processor performs the inner image-placement loop to compute subsets to replace with images.

We empirically determined the constants for the cost-benefit function. In general, we found that LDIs work better the more distant they are (as expected); thus, we bias the function to prefer distant images. Furthermore, the C and B constants are set so that we slightly prefer higher-benefit solutions (i.e. the more distant ones) to ones that cull a little more geometry.

We employ octree pseudo-leaf cells to limit the number of cells for preprocessing. For our test models, we determined that an octree depth of 5 yields a reasonable balance between granularity and performance (thus, a maximum of 32,768 leaf cells per view).

At run time, we find the closest grid point in the reverse view frustum, select the view direction sample for the angular range that contains the current view direction, and check for an image placement. If one was computed, we warp the associated LDI. Our software-based warper distributes the work among three processors and is able to warp near NTSC-resolution LDIs (512x384) at about 8 Hz. We have also pipelined the culling and rendering phases of the system, therefore introducing one frame of latency. Furthermore, we use a 3x3 convolution-kernel to smooth the warped image. Figure 12 summarizes the run-time algorithm.

We create a least-recently-used cache to store image data and to allow us to precompute or dynamically-compute images for an interactive session. All images within a pre-specified radius of the current viewpoint are loaded from disk in near to far order. We either load the additional image data during idle time or use a separate processor to load image data.

```
1. For each frame
2.   Compute the reverse view frustum for the
       current view position and direction
3.   Find the closest grid point contained
       within the reverse frustum
4.   Find the sampled view for the angular range
       that contains the current view direction
5.   If (image was computed)
       Cull octree-cell subset from model
       Cull remaining geometry to view frustum
       Render geometry and warp image to current
         viewpoint
6.   Else
       Cull geometry to view frustum
       Render geometry
     Endif
   Endfor
```

Figure 12. *Run-time Algorithm Summary.*

6. PERFORMANCE

We report the performance of our algorithm on four test models:

- a 2M triangle model of a coal-fired power plant (this is the largest model we can fit in memory that leaves space for the image cache and does not require us to page geometry),
- a 850K triangle model of the torpedo room of a notional nuclear submarine,
- a 1.7M triangle architectural model of a house, and
- a 1M triangle model of an array of pipes (procedurally generated by replication and instancing of pipes).

Figure 13 shows the amount of storage required for several maximum primitive counts. In order to display the results in a single graph, we chose to normalize the values to a common pair of axes. We use the horizontal axis to represent the geometry budget as a percentage of model size and the vertical axis to represent the total number of images divided by the total number of model primitives. The non-monotonic behavior of the power plant curve is because our algorithm found a local minimum farther away from the global minimum than the neighboring solutions. The solution at a geometry budget of 23% converged to a cluster of geometry that was large enough to meet the target primitive count but not necessarily the smallest and farthest subset. This occurrence is common with optimization algorithms.

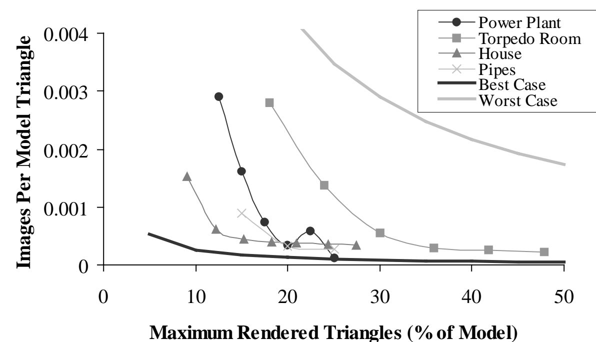


***Figure 13**. Storage Performance. The upper gray line represents the performance of the worst-case scenario. The lower black line represents the best-case scenario. The four test models fall in between these two bounds and in fact tend towards the best-case scenario.*

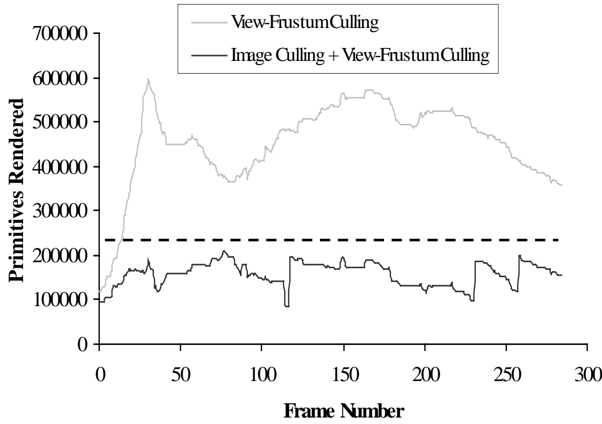


Figure 14. Path through Power Plant Model. This graph shows the number of primitives rendered for a sample path through the power plant using a geometry budget of 250,000. We show the results using only view-frustum culling and using image culling plus view-frustum culling. Notice that the primitive count never exceeds our geometry budget; in fact, for this path, it almost never exceeds P_{opt}=200,000.

An improvement could be achieved by using a technique such as simulated annealing to move the solution to a "better" minimum.

For comparison with our empirical results, we also show in Figure 13 a curve that corresponds to the theoretical best-case performance of our algorithm. This occurs in models with a uniform distribution of geometry. Figure 13 also shows a curve that corresponds to a theoretical near worst-case performance, as is the case in models with large variations of geometric density. In practice, models fall somewhere in between these two extremes. For more details, we refer you to [Ali98].

Figure 14 shows the number of primitives rendered per frame for a path through the power plant using the solution set for a geometry budget of P = 250,000 primitives (and an optimization budget of P_{opt} = 200,000 primitives). We have observed that for our test models, a difference between the geometry budget and optimization budget of 5 to 10% of the model primitives yields 1 to 4 images per grid point, on average.

Figure 15 shows a histogram of the number of grid points with the number of images varying from 1 to 5 images. Grid points near the edge usually have fewer images and are generally facing inwards towards the model center. Although images of neighboring grid points are similar, we do not share them. The similarity could be exploited for image compression purposes.

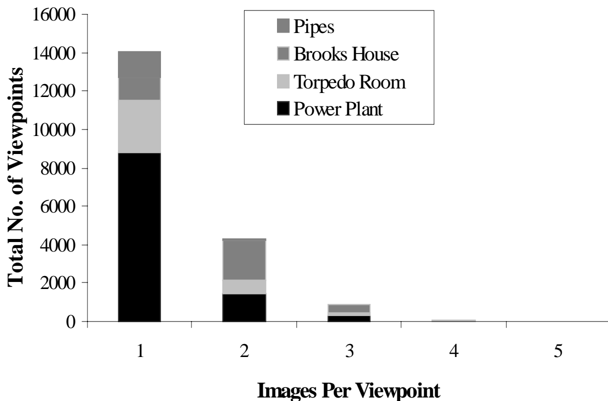


Figure 15. Histogram of Images Per Grid Point. Most grid points have between 1 and 3 images; specifically: 14,012 grid points with M=1, 4311 grid points with M=2, 862 with M=3, 40 grid points with M=4.

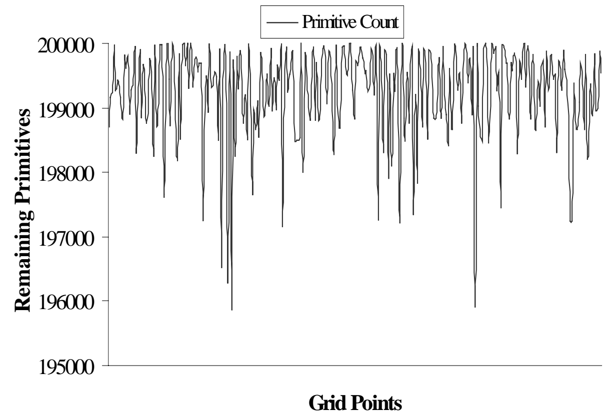


Figure 16. Primitive Counts for Solutions at Grid Points. The image-placement process computes image locations that typically produce primitive counts within 2% of the desired value P_{opt} = 200,000.

Figure 16 illustrates how close the solutions computed by the image-placement process (Section 3.3) are to the desired optimization budget. For a given grid-point view, we compute image placements that are conservative and typically fall within 2% of the optimization budget.

Figure 17 compares higher-resolution LDIs to an all-geometry rendering. Both the geometry and NTSC-resolution (640x512) LDI are rendered using 2x2 multi-sampling. This LDI resolution is beyond what we can do interactively today on our SGI workstation, nevertheless we show in our video an animation with these LDIs. To achieve the visual quality of Figure 17, at a frame rate of 30Hz, we would require a graphics performance of at most 7.5M triangles per second plus the ability to warp a multi-sampled NTSC-resolution LDI at 30Hz. The all-geometry approach would require at most 60M triangles per second processing power.

Table 1 summarizes the image-placement results. For each test model, we show the number of images computed and the preprocessing time for grid adaptation and image placement. In addition, we show the estimated space requirement. To determine this, we use an empirically determined average image size. We compress images using *gzip* and use a separate processor to uncompress them at run time—from this information we extrapolate space requirements (at present, we can uncompress an image in under one second).

The preprocessing time of a LDI is dependent on the number of construction images and the model complexity per construction image. First, we render eight construction images and one central image using only view-frustum culling. Second, we create a LDI in time proportional to the number of construction images. For our test models, our (unoptimized) LDI creation process takes 7 to 23 seconds. The total rendering and construction time of 3100 512x384-pixel power plant LDIs is approximately ten hours.

7. LIMITATIONS AND FUTURE WORK

Our current implementation only guarantees a rendering performance for translation and yaw rotation. The view-directions set can be easily expanded to include pitch, but it is unclear whether is it worth the extra effort and storage. We have observed that with interactive walkthroughs, gaze is kept nearly horizontal.

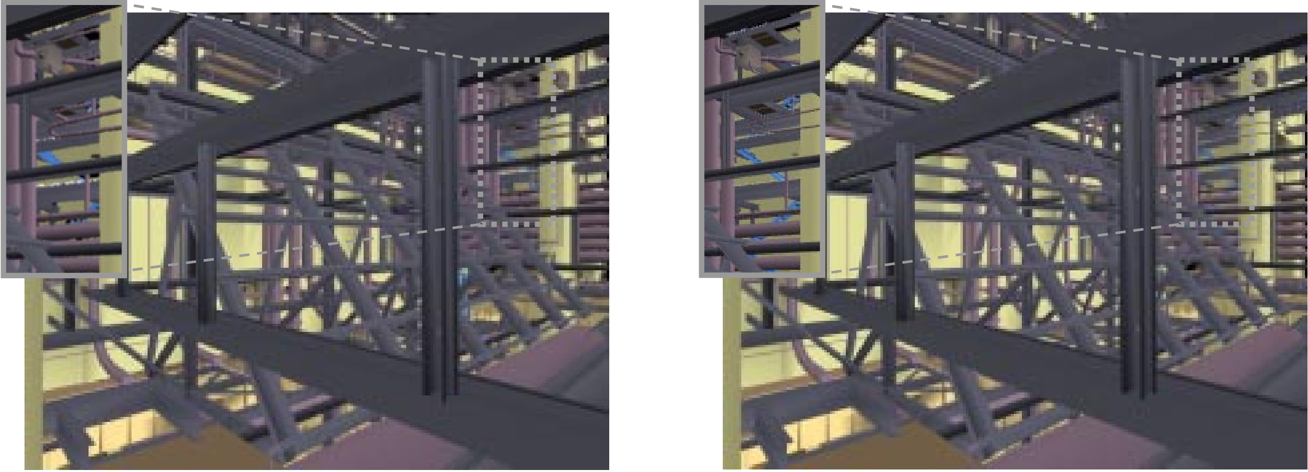


Figure 17. *LDI+Geometry vs. All-Geometry Comparison. (Left) Snapshot using the same viewpoint as in Figure 10, but with a NTSC-resolution 2x2 multi-sampled LDI. The geometry is rendered using the graphics hardware's single-pass 2x2 multi-sample mode. (Right) For comparison purposes, we show a snapshot of an all-geometry rendering. The LDI does not perfectly reconstruct all surfaces, as can be observed by the pair of insets.*

We have seen a wide range of preprocessing times (from one to 28 hours). Furthermore, we have empirically determined the set of constants and weights required during preprocessing. They have worked well for our test models, but further parallelization (e.g. of the grid creation) and more automatic methods for determining these constants would improve the preprocessing.

Our cost function ignores fill rate. To more accurately achieve a constant frame rate, in particular on midrange systems, we need to take rasterization costs into account. In addition, we could measure the depth complexity (or the total pixel count) of the LDIs to more precisely trade off images for geometry.

Model (triangles)	Max. No. Tris	No. of LDIs	Preprocess (hours)	Estimated Space (MB)
Power Plant	250,000	5815	21.7	3802
(2M)	300,000	3224	12.4	2108
	350,000	1485	6.1	971
	400,000	706	6.5	462
	450,000	1169	5.9	764
	500,000	239	1.2	156
Torpedo Rm.	150,000	2333	11.8	933
(850k)	200,000	1160	6.0	464
	250,000	462	2.8	185
	300,000	243	1.6	97
	350,000	212	1.3	85
	400,000	181	1.1	72
House	150,000	2492	28.4	1725
(1.7M)	200,000	994	22.0	688
	250,000	714	10.6	494
	300,000	662	10.5	458
	350,000	629	11.2	435
	400,000	593	12.5	410
	450,000	561	11.4	388
Pipes	150,000	893	4.6	554
(1M)	200,000	331	2.8	205
	250,000	282	2.4	175

Table 1. *Preprocessing Summary for Test Models*

Currently, we cannot perform view-dependent shading with the images at reasonable frame rates; thus, we use precomputed diffuse illumination. Our interactive software warper uses near NTSC resolution images. Higher resolution LDIs (for multi-sample anti-aliasing) are feasible but require proportionally more compute power or processors. Hence, because of our limited warping speed today, we cannot reduce geometric complexity to an arbitrary amount and achieve a high quality rendering.

In general, image warping demands good memory bandwidth. Every frame, we must perform pixel operations on the entire LDI, copy the warped image to the graphics engine and fetch future LDIs. We can transfer a 512x384-pixel image to the frame buffer in less than 3ms. Furthermore, since a single LDI is typically reused for several frames, we expect pixel operations to be performed from cache. The paging of image data from disk and from main memory is the slowest part.

We need to do further investigation of prefetching algorithms for the image data as well as the model geometry. In our current system, we assume the entire model fits in main memory. Moreover, our walking speed is limited by the rate at which we can page data from disk. In addition, we expect to be able to reduce the storage requirement by more sophisticated image representations and by image compression methods.

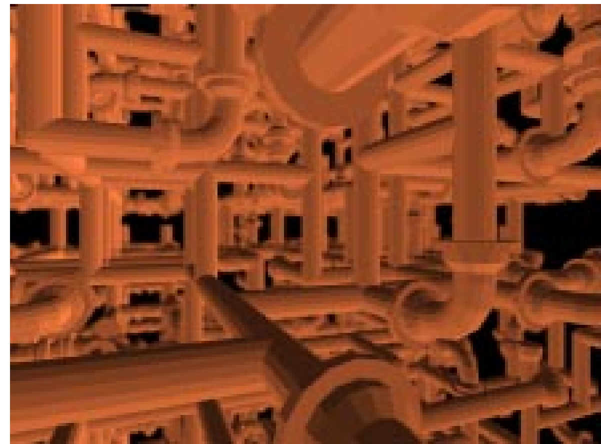


Figure 18. *Example View of Pipes Model. Foreground pipes are geometry. Most of the background pipes are a 512x384-pixel LDI.*

8. CONCLUSIONS

We introduced a preprocessing algorithm and run-time system for reducing and bounding the geometric complexity of 3D models by dynamically replacing subsets of the geometry with (depth) images. Therefore, if we can afford the approximately constant cost of displaying images and the number of primitives to render dominates our application's rendering performance, we can achieve a guaranteed (minimum) frame rate. We also demonstrated an optimized layered-depth-image approach that yields good visual results and applied our algorithms to several complex 3D models (Figure 18).

The automatic image-placement algorithm we have presented allows us to trade off *space* for *frame rate*. In our case, space is proportional to the total number of images needed to replace geometry and the image size. Higher frame rate is equivalent to reducing the maximum number of primitives to render. Our results, both empirical and theoretical, indicate we can reduce geometric complexity by approximately an order of magnitude using a practical amount of storage (by today's standards).

9. ACKNOWLEDGMENTS

We would like to acknowledge the anonymous reviewers for their generous comments and suggestions. We also greatly appreciate the help received from Voicu Popescu, Matthew Rafferty, Bill Mark and the UNC Walkthrough and PixelFlow group.

The power plant model is courtesy of James Close and Combustion Engineering. The Brooks' House model is courtesy of many generations of UNC graduate students. The torpedo room model is courtesy of Electric Boat Division of General Dynamics. The pipes model was created from code written by Lee Westover.

This research was supported in part by grants from the NIH National Center for Research Resources (RR02170), DARPA (E278), NSF (MIP-9612643) and a UNC Dissertation Fellowship. In addition, we thank Intel for their generous equipment support.

References

[Air90] Airey J., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments", *Symposium on Interactive 3D Graphics*, 41-50 (1990).

[Ali96] Aliaga D., "Visualization of Complex Models Using Dynamic Texture-Based Simplification", *IEEE Visualization*, 101-106 (1996).

[Ali97] Aliaga D. and Lastra A., "Architectural Walkthroughs Using Portal Textures", *IEEE Visualization*, 355-362 (1997).

[Ali98] Aliaga D., "Automatically Reducing and Bounding Geometric Complexity by Using Images", *Ph.D. Dissertation,* University of North Carolina at Chapel Hill, Computer Science Dept., October (1998).

[Ali99] Aliaga D., Cohen J., Wilson A., Baker E., Zhang H., Erikson C., Hoff K., Hudson T., Stuerzlinger W., Bastos R., Whitton M., Brooks F., Manocha D., "MMR: An Interactive Massive Model Rendering System Using Geometric and Image-based Acceleration", *Symposium on Interactive 3D Graphics,* 199-206 (1999).

[Cla76] Clark J., "Hierarchical Geometric Models for Visible Surface Algorithms", *CACM*, Vol. 19(10), 547-554 (1976).

[Coh96] Cohen J., Varshney A., Manocha D., Turk G., Weber H., Agarwal P., Brooks F. and Wright W., "Simplification Envelopes", *Computer Graphics (SIGGRAPH '96)*, 119-128 (1996).

[Coo97] Coorg S. and Teller S., "Real-Time Occlusion Culling for Models with Large Occluders", *Symposium on Interactive 3D Graphics*, 83-90 (1997).

[Dar97] Darsa L., Costa Silva B., and Varshney A., "Navigating Static Environments Using Image-Space Simplification and Morphing", *Symposium on Interactive 3D Graphics,* 25-34 (1997).

[DeH91] DeHaemer M. and Zyda M., "Simplification of Objects Rendered by Polygonal Approximations", *Computer Graphics*, Vol. 15(2), 175-184 (1991).

[Ebb98] Ebbesmeyer P., "Textured Virtual Walls - Achieving Interactive Frame Rates During Walkthroughs of Complex Indoor Environments", *VRAIS '98,* 220-227 (1998).

[Fun93] Funkhouser T., Sequin C., "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments", *Computer Graphics (SIGGRAPH '93),* 247-254 (1993).

[Gar97] Garland M., Heckbert P., "Surface Simplification using Quadric Error Bounds", *Computer Graphics (SIGGRAPH '97),* 209-216 (1997).

[Hop97] Hoppe H., "View-Dependent Refinement of Progressive Meshes", *Computer Graphics (SIGGRAPH '97)*, 189-198 (1997).

[Lue97] Luebke D. and Erikson C., "View-Dependent Simplification of Arbitrary Polygonal Environments", *Computer Graphics (SIGGRAPH '97)*, 199-208 (1997).

[Mac95] Maciel P. and Shirley P., "Visual Navigation of Large Environments Using Textured Clusters", *Symposium on Interactive 3D Graphics*, 95-102 (1995).

[Max95] Max N., Ohsaki K., "Rendering Trees from Precomputed Z-Buffer Views", *Rendering Techniques '95*: *Proceedings of the 6th Eurographics Workshop on Rendering*, 45-54 (1995).

[McM95] McMillan L. and Bishop G., "Plenoptic Modeling: An Image-Based Rendering System", *Computer Graphics (SIGGRAPH '95)*, 39-46 (1995).

[Mue95] Mueller C., "Architectures of Image Generators for Flight Simulators", *Computer Science Technical Report TR95-015*, University of North Carolina at Chapel Hill (1995).

[Pop98] Popescu V., Lastra A., Aliaga D., and Oliveira Neto M., "Efficient Warping for Architectural Walkthroughs using Layered Depth Images", *IEEE Visualization*, (1998).

[Raf98] Rafferty M., Aliaga D. and Lastra A., "3D Image Warping in Architectural Walkthroughs", *IEEE VRAIS*, 228-233 (1998).

[Reg94] Regan M., Pose R., "Priority Rendering with a Virtual Reality Address Recalculation Pipeline", *Computer Graphics (SIGGRAPH '94),* 155-162 (1994).

[Sch83] Bruce Schachter (ed.), Computer Image Generation, John Wiley and Sons, 1983.

[Scf96] Schaufler G. and Stuerzlinger W., "Three Dimensional Image Cache for Virtual Reality", *Computer Graphics Forum (Eurographics '96),* Vol. 15(3), 227-235 (1996).

[Sha96] Shade J., Lischinski D., Salesin D., DeRose T., Snyder J., "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments", *Computer Graphics (SIGGRAPH '96),* 75-82 (1996).

[Sha98] Shade J., Gortler S., He L., and Szeliski R., Layered Depth Images, *Computer Graphics (SIGGRAPH '98),* 231-242 (1998).

[Sil97] Sillion F., Drettakis G. and Bodelet B., "Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery", *Computer Graphics Forum Vol. 16 No. 3 (Eurographics),* 207-218 (1997).

[Tel91] Teller S., Séquin C., "Visibility Preprocessing For Interactive Walkthroughs", *Computer Graphics (SIGGRAPH '91),* 61-69 (1991).

[Tur92] Turk G., "Re-Tiling Polygonal Surfaces", *Computer Graphics (SIGGRAPH '92),* 55-64, (1992).

[Zha97] Zhang H., Manocha D., Hudson T. and Hoff K., "Visibility Culling Using Hierarchical Occlusion Maps", *Computer Graphics (SIGGRAPH '97),* 77-88 (1997).