AUTOMATICALLY REDUCING AND BOUNDING GEOMETRIC COMPLEXITY BY USING IMAGES

by

Daniel G. Aliaga

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1998

ABSTRACT

Daniel G. Aliaga. Automatically Reducing and Bounding Geometric Complexity by Using Images.
(Under the direction of Anselmo A. Lastra)

Large and complex 3D models are required for computer-aided design, architectural visualizations, flight simulation, and many types of virtual environments. Often, it is not possible to render all the geometry of these complex scenes at interactive rates, even with high-end computer graphics systems. This has led to extensive work in 3D model simplification methods.

We have been investigating dynamically replacing portions of 3D models with images. This approach is similar to the old stage trick of draping cloth backgrounds in order to generate the illusion of presence in a scene too complex to actually construct on stage. An image (or backdrop) once created, can be rendered in time independent of the geometric complexity it represents. Consequently, significant frame rate increases can be achieved at the expense of storage space and visual quality. Properly handling these last two tradeoffs is crucial to a fast rendering system.

In this dissertation, we present a preprocessing and run-time algorithm for creating a constant-frame-rate rendering system that replaces selected geometry with images enhanced with depth at each pixel. First, we describe the preprocessing algorithm to automatically bound the geometric complexity of arbitrary 3D models by using images. Second, we explore two general approaches to displaying images, surrounded by geometry, at run time. Third, we present a system tailored to architectural models.

ACKNOWLEDGMENTS

I owe a tremendous amount of gratitude to the many people that have helped me during my graduate school years. In particular, I would like to thank my advisor, and friend, Anselmo Lastra. He kept me heading in the right direction and has spent many long hours helping me with this research. I also greatly appreciate the wisdom and guidance of the rest of my dissertation committee: Gary Bishop, Fred Brooks, Henry Fuchs, Steve Molnar, and Henry Rich.

I am also grateful to the long list of keen graduate students that have surrounded me. The enthusiasm and energy I received from them has helped me throughout the dissertation process. I especially acknowledge Bill Mark, my officemate and friend, for always lending an ear to me, and Yunshan Zhu, my roommate, friend and colleague for several years. Furthermore, I wish to thank Jon Cohen, David Harrison, Todd Gaul, Steve Hutson, David Luebke, Carl Mueller, Michael North, Marc Olano, Voicu Popescu, Matthew Rafferty, Keith Sykes, and many others.

I appreciate the support of the entire Computer Science Department, especially that of the Walkthrough Project, led by Fred Brooks, Dinesh Manocha and Mary Whitton, and that of the entire PixelFlow team. I also thank Henry Fuchs for introducing me into the department and Andries vam Dam for first bringing me into the field of Computer Graphics.

Finally, I thank Leigh "pooquita" Atkinson for her love and support during this long marathon as well as my parents and sister for their endless encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

1.1 Thesis Statement

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 databases currently contain hundreds of thousands to millions of primitives; more than high-end computer graphics systems can render at interactive rates. We would like to improve our rendering strategies so that even mid-range systems can render these massive models.

This demand for interactive rendering has brought about algorithms for model simplification. For example, techniques have been presented for geometric levels of detail, visibility culling, and for representing objects using images.

In this dissertation, we present a preprocessing and run-time algorithm for creating constant-frame-rate rendering systems by dynamically replacing selected geometry with images. 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 rendering cost and given sufficient resolution maintains the apparent visual detail.

First, we describe the preprocessing algorithm to automatically compute a data structure that determines which subsets of an arbitrary 3D model to replace with images in order to limit the number of primitives to render. Second, we present two general approaches for displaying images surrounded by geometry at run time. We apply our algorithms to accelerate the interactive walkthrough of several large CAD models. We also take advantage of the inherent structure of buildings and present a system tailored to architectural walkthroughs.

We summarize this work in the following thesis statement:

> *"We can accelerate the interactive rendering of complex 3D models by replacing subsets of the geometry with images. Furthermore, we can guarantee a specified level of performance by bounding the amount of geometry that must be rendered each frame."*

1.2 Models and Applications

Computer graphics [Foley90] and interactive rendering systems are used to assist in the design, visualization, and construction of complex objects such as submarines, ships, airplanes, buildings, and power plants. The models are often created by a large number of structural and mechanical designers divided into teams. A complete model is obtained by combining the latest versions of all objects in a common model database. The teams then view the model, assign new tasks, and periodically perform computations on the model, for example interference detection and computational fluid dynamics. The whole design process can take from months to years, employing hundreds of designers. Any effort to reduce the length of the design cycle or improve the model quality is a profitable investment.

Our laboratory worked together with a simulation-based design team to create an interactive walkthrough of compartments of a nuclear submarine. A model of the torpedo room contains over 850,000 primitives (Figure 1-1a). The auxiliary machine room in a similar submarine has 525,000 primitives (Figure 1-1b). The latter model contains an irregular distribution of piping, batteries, and other mechanical and electrical components. Both of these models are a challenge for simplification algorithms.

Architectural walkthroughs allow us to see a building before it is constructed or to visualize famous structures. To make the walkthroughs realistic, a large amount of detail is required as well as sophisticated lighting methods (e.g. radiosity illumination [Cohen93]). A model of a single room of the FallingWater house designed by Frank Lloyd Wright contains over one million primitives (Figure 1-2a). A complete one-story model of Dr. Fred Brooks' house contains 1.7 million triangles (Figure 1-2b). In some situations, we use head-mounted displays coupled with a position and orientation tracking system to immerse ourselves in

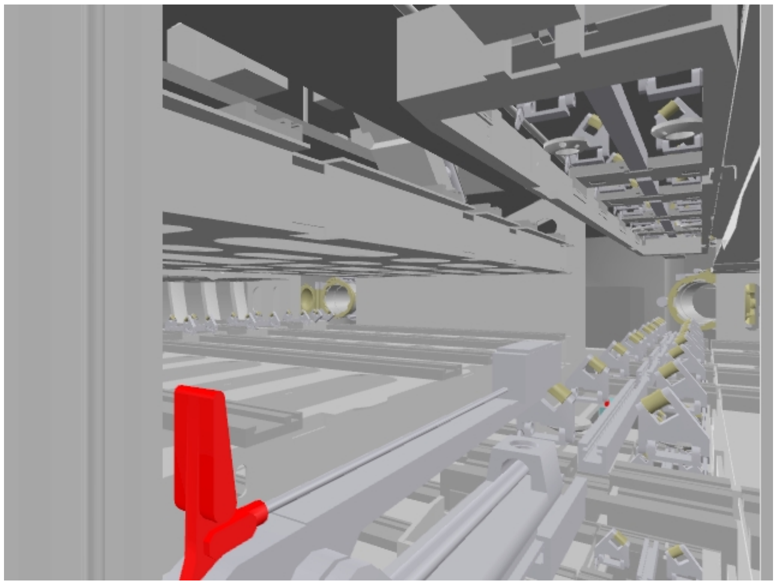

Figure 1-1a. *Torpedo Room. This is a 850,000 triangle model of the torpedo room of a notional nuclear submarine. In view are the rollers to load the torpedoes into the launching tube. Courtesy of Electric Boat Division of General Dynamics.*

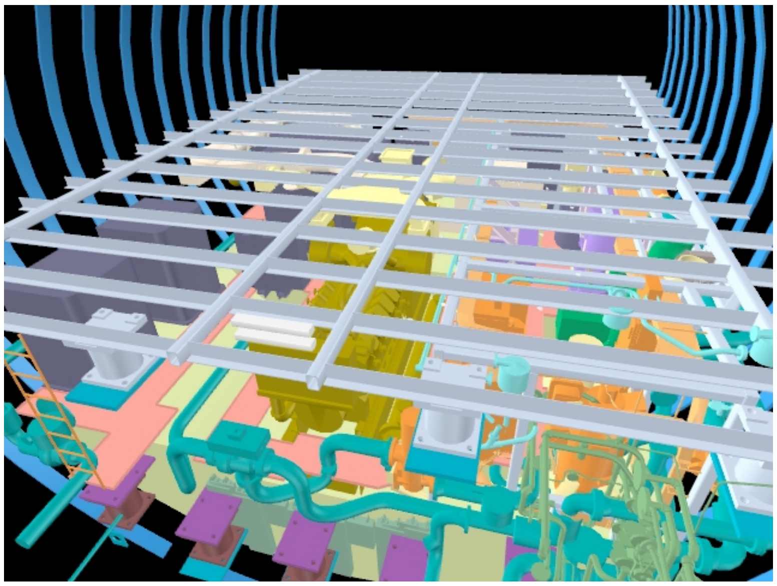

Figure 1-1b. *Auxiliary Machine Room. This is a 525,000 triangle model of the auxiliary machine room in a notional nuclear submarine. Courtesy of Electric Boat Division of General Dynamics.*

these virtual buildings. We require at least twelve updates per second [Tom Piantanida, personal communications] to give us a sense of presence.

Very large-scale (or massive) models can be too large to store in main memory. They require significant rendering acceleration as well as a compact database representation and efficient disk paging. For example, the prototype model of a coal-fired power plant, shown in Figure 1-3, consists of 13 million triangles and requires 1.3 GB of disk space.

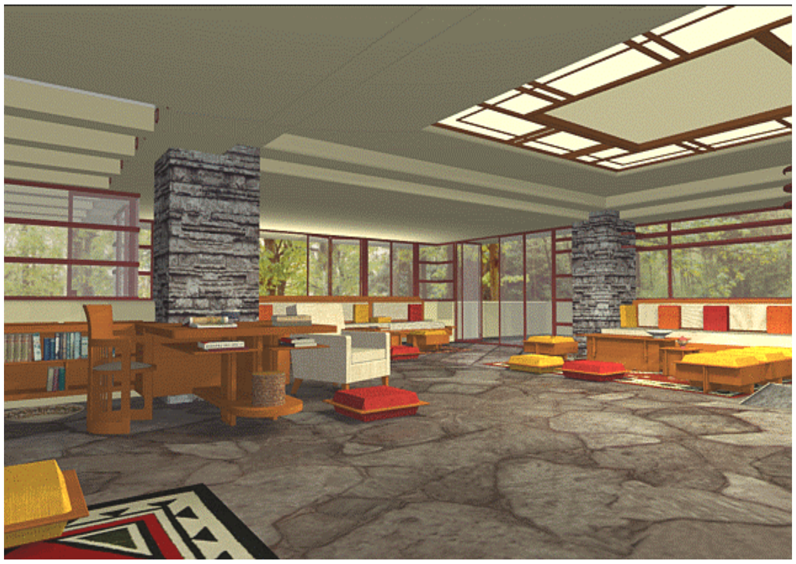

***Figure 1-2a**. Living room of the FallingWater House, a famous house designed by Frank Lloyd Wright. This room is modeled using one million geometric primitives. Courtesy Program of Computer Graphics, Cornell University.*

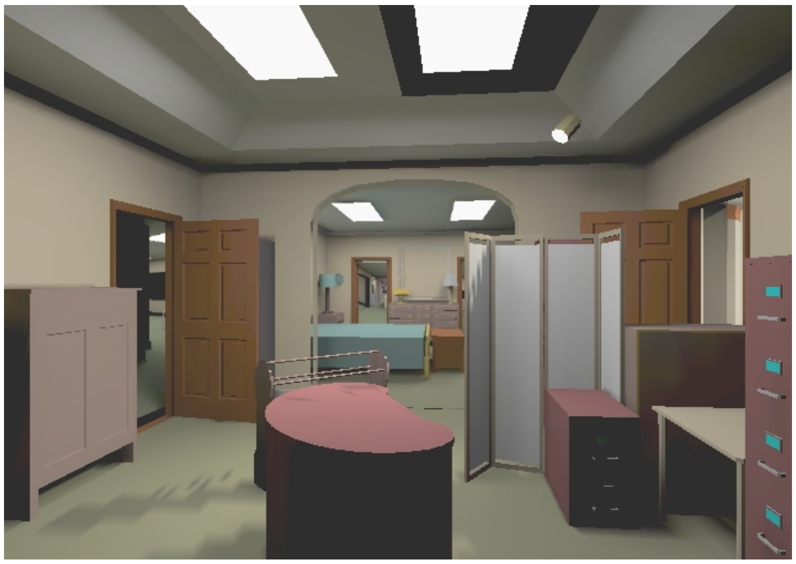

***Figure 1-2b**. Brooks House. A 1.7 million triangle model of Dr. Fred Brooks' house. This view is looking inwards from one corner of the study. Courtesy of the UNC Walkthrough Project.*

1.3 Rendering Acceleration Methods

It is not possible to render all the geometry of these complex models at highly interactive rates. Hence, a large body of literature exists that describes algorithms to reduce the amount of geometry to render (Chapter 2). We can divide such techniques into three general categories:

- *Geometric Simplification*: these methods reduce the complexity of geometry that is rendered. Often the reduction is performed as a preprocess by generating multiple levels of detail of objects in the scene (e.g. [Turk92][Eck95][Cohen96][Garland97]

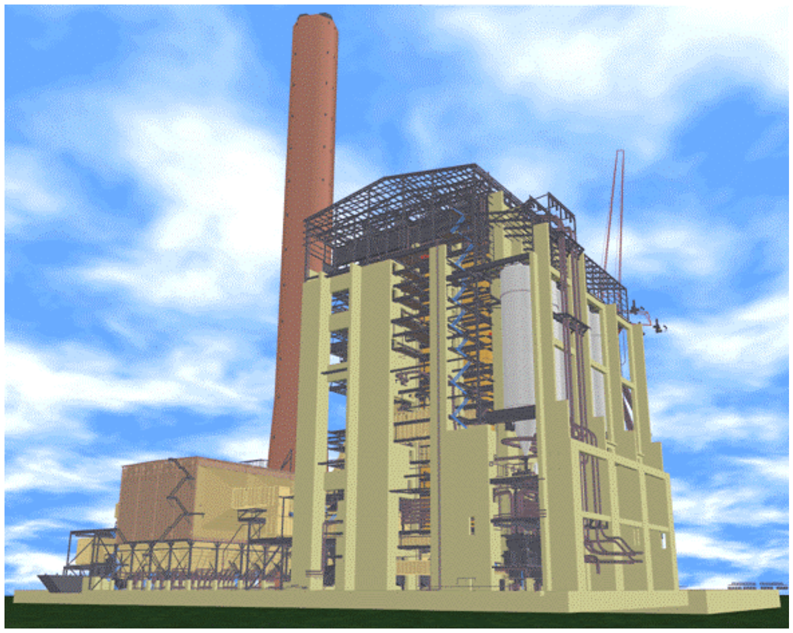

Figure 1-3. *Coal-fired Power Plant. This is a very large model (over 13 million triangles) of a coal-fired power plant. The main building is over 250 feet high (50 levels) and contains large arrays of piping, generators, and other machinery. Courtesy of ABB Engineering.*

and many others). Another approach involves dynamically simplifying the scene using view-dependent simplification [Xia96][Hoppe97][Luebke97]. This latter approach can achieve better simplification for the same quality but requires significant run-time overhead.

- *Visibility culling*: these algorithms compute, conservatively, the subset of the model that is visible [Airey90][Teller91][Luebke95] or, conversely, what conservative subset is occluded [Hudson97][Zhang97][Coorg97]. These methods (usually) do not alter the final rendered image, instead they avoid processing geometry known to be hidden.

- *Textures and Image Caching*: the third general category involves replacing geometry in the scene with images. Flight simulators have used images to represent terrain and other specialized datasets (e.g. [Clark90][Pratt92] and many others). Recently, several systems have been developed for models that replace objects with precomputed images [Maciel95][Ebbesmeyer98] or a hierarchy of spatially adjacent objects with dynamically generated images [Shade96][Schaufler96].

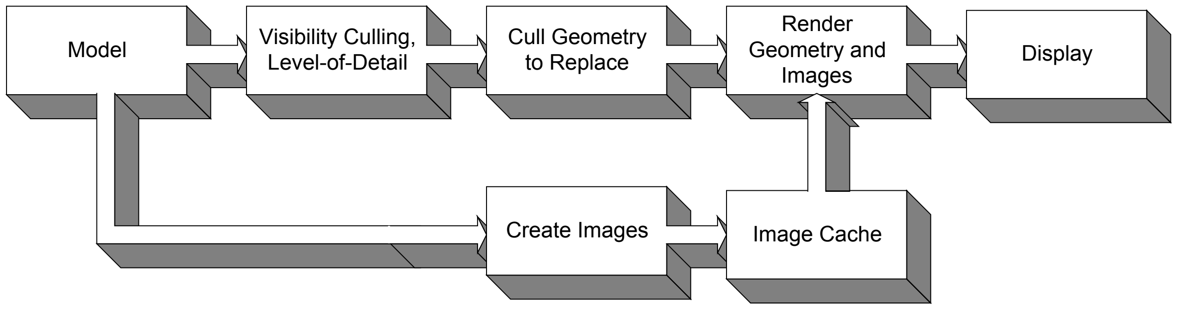
Model
Visibility Culling, Level-of-Detail
Cull Geometry to Replace
Render Geometry and Images
Display
Create Images
Image Cache

Figure 1-4. *Pipeline for Replacing Geometry with Images. Starting with a geometric model, we apply visibility culling and level-of-detail algorithms. We replace some of the remaining geometry with images. These images are obtained from an image cache. In principle, they can be precomputed or dynamically computed.*

1.4 Replacing Geometry with Images

The work presented in this dissertation builds upon the concepts of the third rendering-acceleration category. Figure 1-4 outlines a general pipeline for replacing geometry with images. The algorithms we present can be applied alone or together with visibility culling and level-of-detail methods. In this dissertation, we focus on the image aspect. The applications outlined in Section 1.2 are typically limited by the number of transformations performed each frame. Consequently, the number of primitives rendered mostly determines frame rate. In our approach, we automatically select distant geometry to replace with images so that what geometry remains does not exceed a specified number of primitives per frame.

Distant geometry is dynamically culled from the model and an image is displayed. We define an image to be a quadrilateral on which we place a view of the geometry it replaces. We assume the approximately constant cost of displaying images can be afforded each frame. Moreover, to maintain a constant frame rate, an image must be created before we reach that part of the model. The images can be precomputed or generated at run time and then stored in main memory. If the total number of precomputed images exceeds available main memory, we use auxiliary processors or idle time to load the images from secondary storage.

1.4.1 Overall System Design

In order to guide our task of devising an efficient system to display images, we have identified four important design issues:

- *Automatic Image-Placement*: the images should be inserted into the model at predetermined locations or an algorithm should be used to determine which subsets of the model to replace with images.
- *Temporal Continuity*: the system should take into consideration that an image is a two-dimensional (2D) snapshot of a 3D model. The geometry represented by the image will only have the correct perspective projection and occlusion for a single viewpoint, namely the *image-sampling viewpoint*. We want to avoid a visual *popping* artifact as we move through a model and switch between images created at discrete locations.
- *Geometric Continuity*: the system should control or eliminate artifacts that occur at the geometry-image border. For example, if we represent an object partially as geometry and partially as image, then we do not want to see a visual discontinuity at the geometry-image border as we move from the image-sampling viewpoint. By addressing this issue, a system can provide much more freedom for image placement than if the images must encompass whole objects.
- *Smooth Transitions*: the system should be able to provide smooth transitions for switching between geometry and images. This allows the rendering acceleration using images to be transparent to the viewer. Moreover, we can always choose to render a subset of the model (e.g. the portion we wish to interact with) as conventional geometry.

In our work, we create a preprocessing algorithm to automatically determine the location of images for an arbitrary 3D model. Then, with the remaining issues in mind, we must display the images at run time, together with the remaining geometry. We could use one of a variety of methods listed below:

- *Texture-mapped quadrilaterals*: use an image applied as a texture map onto a quadrilateral. This simple method does not provide geometric continuity, temporal continuity nor smooth transitions.
- *Textured depth meshes*: use a (simplified) depth mesh that represents the scene from the single viewpoint used to create the mesh [Darsa97][Sillion97]. Apply an image as a

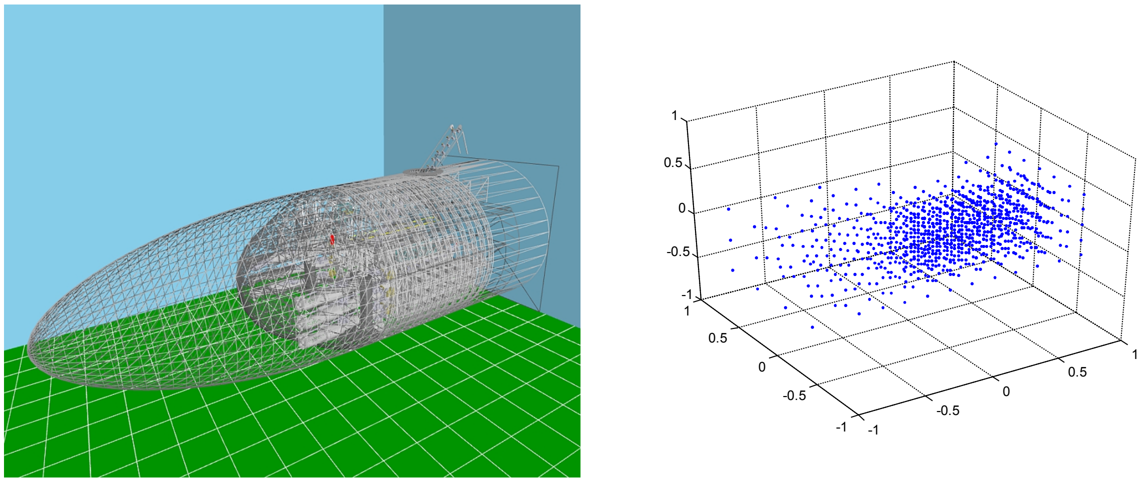

Figure 1-5. *Example Automatic-Image Placement. This figure illustrates an automatically computed viewpoint grid for the torpedo room model. The left snapshot shows an exterior view of the model rendered in wireframe. The right plot shows a grid of 1557 viewpoints from where images are computed to limit 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.*

projective texture onto the mesh. This approach provides approximate geometric continuity, temporal continuity, and smooth transitions, but consumes graphics resources.

- *Image warping*: use per-pixel depth values to warp an image to compensate for changes in viewpoint [Chen93][McMillan95b]. This approach addresses all three design issues.
- *Geometry warping*: use conventional images and warp the surrounding geometry to provide geometric continuity and smooth transitions; but, do not provide temporal continuity.

While all of these are viable approaches, we chose to investigate the latter two methods. Of the two, image warping provides the most promising results. Image warping reprojects pixels to their correct location in the image plane. By the use of multiple samples per pixel (sorted in depth), we can reconstruct initially hidden surfaces that become visible. Thus, given sufficient resolution and surface samples, we can produce high quality renderings. We combine our automatic image-placement algorithm with image warping to create a fast rendering system for large, complex models.

Geometry warping does not render the scene accurately, but surprisingly is useful for applications that do not require temporal continuity or for applications that sample multiple

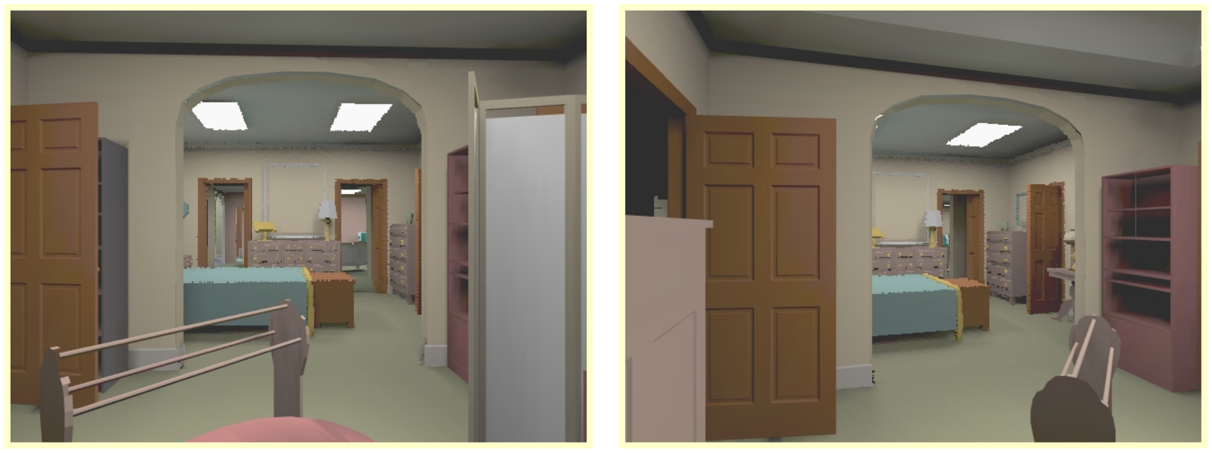

Figure 1-6. *Image Warping. The view through the doorway is rendered as an image. In the right snapshot, we have translated and rotated the viewpoint to one side. Per-pixel depth values are used to "warp" the image to the new viewpoint. We employ a layered depth image to store information for surfaces visible in the right snapshot that are not visible in the original snapshot.*

images to provide approximate temporal continuity. Moreover, we are able to use conventional, static images and achieve higher frame rates.

In the following sections, we summarize our automatic image-placement algorithm and run-time system for arbitrary models, our image-warping implementation, and our geometry-warping algorithm. We also summarize two versions of a walkthrough system tailored to architectural models. One version uses geometry warping to produce a low-memory-budget system while the other uses image warping to provide better scene quality.

1.4.2 Automatic Image-Placement

We have designed a preprocessing algorithm that takes as input a 3D model and automatically bounds the geometric complexity for all viewpoints and view directions in the model (Chapter 3). The algorithm creates a grid of viewpoints adapted to the local model complexity (Figure 1-5). At each grid viewpoint, an image-placement process selects the smallest and farthest subset of the model to remove from rendering to meet a fixed geometry budget. This process takes advantage of hierarchical spatial data structures [Clark76], used for view-frustum culling, to reduce the infinite set of viewpoints and view directions to a finite, manageable set. An image with depth values is then created to represent each selected subset.

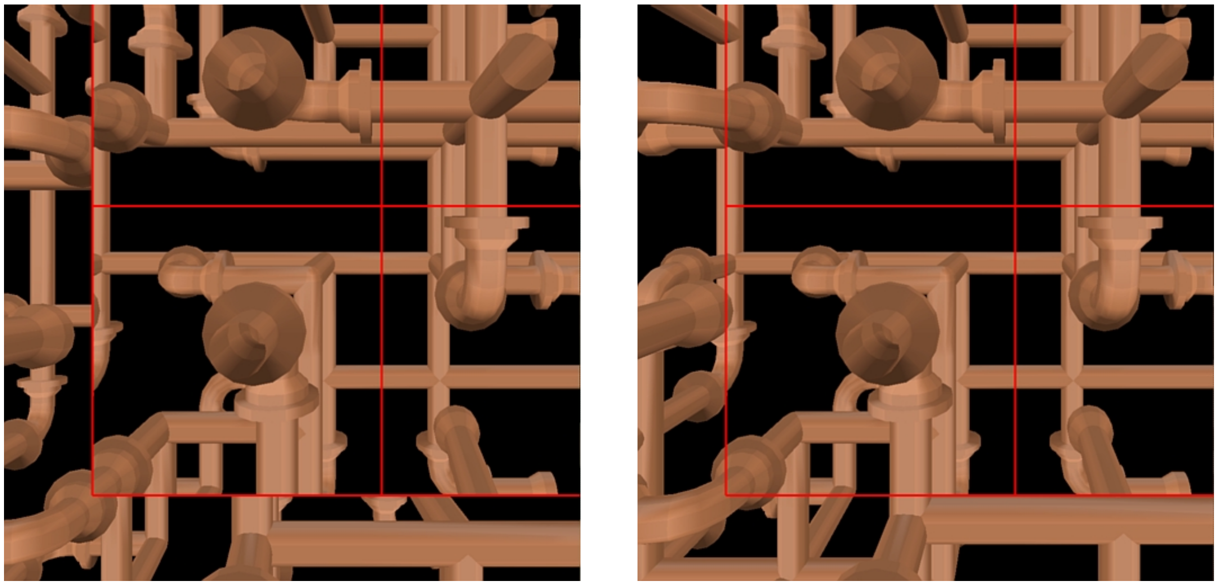
Figure 1-7. *Geometry Warping. The upper right portion of each snapshot (outlined in red) is an image. We have translated (slightly) from the center of projection of the image. In the left snapshot, the geometry bordering the image appears discontinuous with the image. In the right snapshot, we have warped the geometry to match the image.*

Our run-time component displays the selected subset in constant time by warping the corresponding depth image to the current viewpoint (as will be described in the next section). The remaining geometry is rendered normally. Thus, if we can afford the time it takes to warp an image, we can render a 3D model of any size at a predetermined frame rate.

1.4.3 Image Warping

Once we have determined image locations, we must now address how to display images surrounded by geometry. Our preferred approach to displaying images surrounded by geometry is to correct the image to the current viewpoint (Figure 1-6) [McMillan95a]. The geometry behind the image is culled. By altering the image, we prevent temporal and geometric discontinuities. Moreover, the smooth-transition problem is greatly alleviated. The cost of image warping is approximately constant, but introduces a set of visual artifacts. We will present more details on image warping and layered depth images [Max95][Shade98] in Chapter 4.

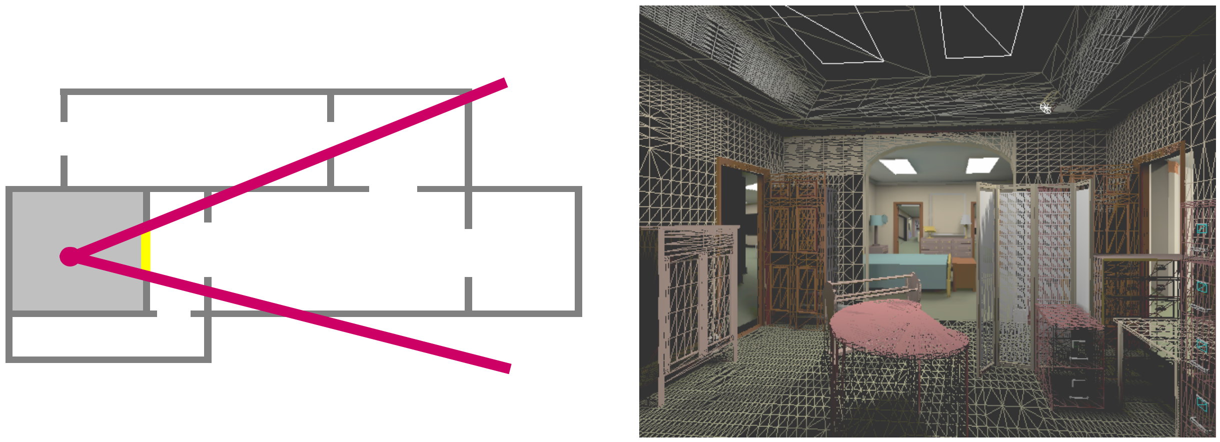

Figure 1-8. *Portal-Image Culling. The portals (doorways, windows, etc.) of the current room are replaced with images, optionally warped to the current viewpoint. Consequently, the system never needs to render more than one room's worth of geometry, even if a long sequence of rooms is visible. The right snapshot shows the portal image and geometry rendered in wireframe.*

1.4.4 Geometry Warping

Our second approach to displaying images modifies the geometry to match the image. We employ a geometry-warping algorithm to maintain geometric continuity and provide smooth transitions between geometry and images. Arbitrary subsets of a model are replaced with a conventional image, texture-mapped onto a quadrilateral. The algorithm maintains positional continuity at the geometry-image border by warping vertices of surrounding geometry (Figure 1-7). When a part of the model changes from image to geometry (or vice versa), we achieve a smooth transition by warping, over several frames, the vertices of the geometry from their actual position to their apparent position in the image (and vice versa). We have observed that as long as the eye does not translate too far from the image-sampling viewpoint, the algorithm provides good visual quality. The warp operations require a linear transformation and can be performed using the graphics hardware's matrix stack. Details will be given in Chapter 5.

1.4.5 Architectural Models

The cells and portals framework [Airey90][Teller91][Luebke95], commonly used for architectural walkthroughs, provides a very natural set of locations for images. Architectural models can easily be divided into disjoint rooms (cells). Each room is connected to adjacent cells through doors and windows (portals). We render the geometry of the cell containing the viewpoint. Then, instead of rendering the geometry of the cells attached to visible portals, we replace portals with images (Figure 1-8). This approach reduces the amount of geometry to render to that of the current room – the amount of geometry to render is not bounded but in practice significant rendering acceleration can be achieved.

We designed two variants of this system (Chapter 6). The first one is a low-memory-budget approach that uses only one image per portal. Then, as the viewpoint approaches a portal, we smoothly change back to geometry by warping the vertices of the adjacent cell from their position on the portal image to their actual position (or vice versa). The second one warps the portal image to the current viewpoint. In both, we have seen significant reductions in rendering time as compared to standard cells and portals.

1.5 Contributions

This dissertation presents several algorithms for dynamically replacing selected geometry with images. We briefly summarize the contributions.

1.5.1 Automatic Image-Placement Algorithm

We have designed a preprocessing algorithm to reduce and bound the number of primitives to render for all viewpoints and view directions within an arbitrary 3D model. The algorithm automatically determines the subsets of the model to replace with images.

1.5.2 Geometry-Warping Algorithm

We have developed an algorithm for warping the geometry surrounding an image in order to prevent discontinuities at the geometry-image border and to provide smooth transitions between geometry and images [Aliaga96][Aliaga98b]. This method can be used in conjunction with systems that use multiple images to control temporal continuity.

1.5.3 Portal Images

We have developed two architectural walkthrough systems that use images to accelerate rendering. The first one replaces portals with images and reduces the rendering load to one or two cells [Aliaga97]. We significantly improve the frame rate even when a long sequence of portals is visible.

The second system warps the portal image to the current viewpoint, thus requiring fewer reference images per portal [Rafferty98a][Rafferty98b][Popescu98]. We achieve the same reduction of geometry as with the first system, although the smoother changes between portal images is achieved at an additional constant cost (proportional to the image size).

1.5.4 Efficient Hierarchical-Culling Algorithm

In order to maximize the benefit of replacing geometry with images, we developed an efficient hierarchical culling algorithm that eliminates redundant primitives when culling a model multiple times per frame. Conventional culling would cause some primitives to be rendered more than once.

1.6 A Guide to Chapters

The rest of this dissertation is organized as follows:

- Chapter 2 presents related work. We summarize texture and image caching systems as well other rendering acceleration methods. Then, we describe relevant flight simulator work and image-based rendering.
- Chapter 3 elaborates on our automatic image-placement algorithm for 3D models. The chapter includes theoretical and empirical results of applying our algorithm to the interactive walkthrough of large 3D models.
- Chapter 4 presents our image-warping approach to displaying images. We describe an efficient layered-depth-image implementation for correcting depth images to the current viewpoint.
- Chapter 5 details our geometry-warping approach to displaying images, as well as a stand-alone rendering acceleration system used to develop the algorithm.
- Chapter 6 summarizes two rendering systems that use images to accelerate architectural walkthroughs.

- Chapter 7 describes an efficient hierarchical culling algorithm.
- Chapter 8 presents conclusions and future work.

We give implementation details and results at the end of the associated chapters.

2. Related Work

In this chapter, we present related work for accelerating interactive rendering of large and complex 3D models. First, we describe rendering acceleration methods that use textures and image caching. We partition existing algorithms into three general strategies (two of previous work and one corresponding to the strategy we adopted). Second, we summarize additional rendering acceleration techniques, namely geometric simplification and visibility culling. Third, we provide a section on flight simulators and terrain databases. The use of computer graphics for flight simulation began in the 1960's. Those systems have employed a very specialized usage of images and rendering acceleration. Fourth, we present an overview of image-based rendering as exploited later in the dissertation. Starting with a collection of images (e.g. photographs), these techniques reconstruct a 3D environment.

2.1 Textures and Image Caching

Textures have been used for some time to represent apparent geometric complexity, but usually in an ad hoc fashion [Cohen91][Haeberli93][Maillot93]. The Evans and Sutherland CT-6 machine (mid-1980's) was one of the first to use real-time photo-textures (e.g. terrain, trees, etc.). SGI Performer Town uses textures to represent the details of objects that are far away (e.g. background) and to give increased visual detail for objects (e.g. buildings).

Regan and Pose [Regan94] 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. The system also performed orientation viewport mapping after rendering, which means head orientation does not need to be known until less than a microsecond before the first pixel of an update frame is actually sent to the display. This helped to reduce the apparent rendering latency.

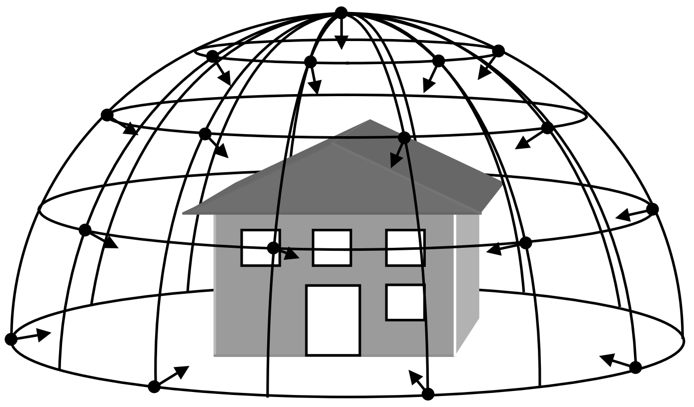

Figure 2-1. *Image-Sampling Sphere for Individual Objects. The images to represent the house are created from viewpoints on the surface of a virtual sphere (or hemisphere) surrounding the house. The radius of the sphere is the predetermined distance from which all images are created.*

Torborg and Kajiya [Torborg96] proposed the Talisman architecture. This architecture allowed rendering of geometry and images, the latter warped with affine transformations. The scene was segmented so that the rendered image of some moving objects could be warped and reused for several frames. They did not address (automatically) selecting what to display as image.

We have observed three general strategies used for replacing complex geometry with images. Images can cull and replace individual objects, nodes of a hierarchical partitioning tree or arbitrary subsets of a model. In the following sections, we describe the advantages and disadvantages of each of these.

2.1.1 Individual Objects

An image that replaces an individual object provides the easiest culling operation. Either the image or the corresponding object is displayed. Furthermore, it is straightforward to create images that represent an object from a discrete set of view directions at a fixed distance. The image-sampling viewpoints for the images are located on the surface of an *image-sampling sphere* centered on the object (Figure 2-1). The images are created as during preprocessing or on demand. Then, at run time, the image closest to the current eye location is displayed.

Representing an entire object with a single image has the advantage that it trivially provides geometric continuity at the geometry-image border (Section 1.4.1). The border

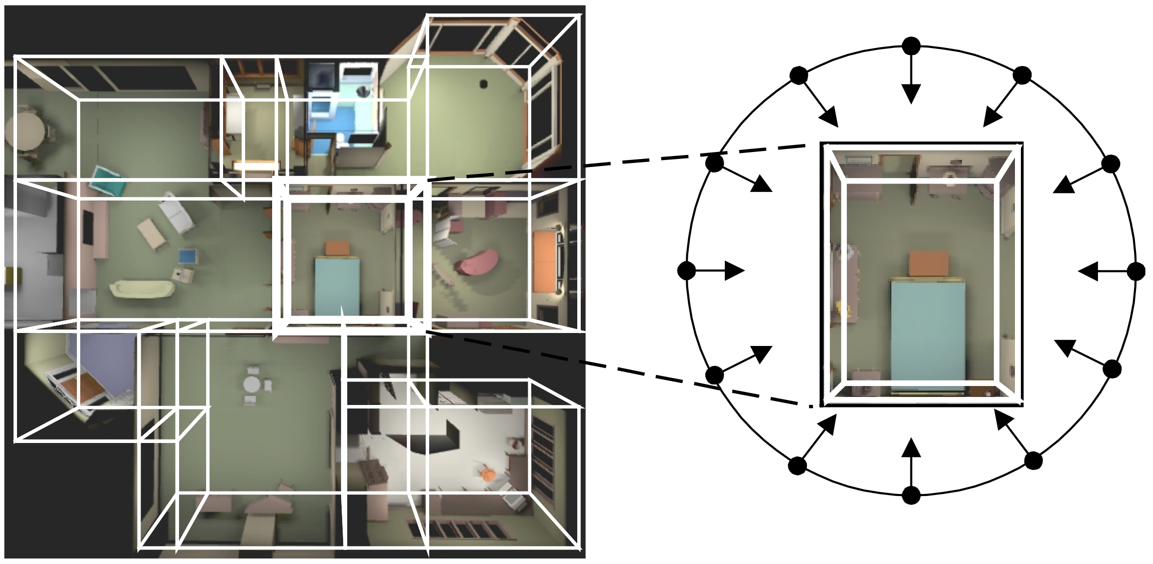

Figure 2-2. *Images for a Hierarchical Partitioning Tree. This top-down view of an architectural model outlines the boundaries of leaf nodes that partition the model space. For each leaf node, we can create images from viewpoints on its image-sampling sphere. The right half of the figure shows a 2D slice of the sphere for a particular leaf node.*

region of the image contains empty pixels (e.g. black) that are rendered with alpha=0 (fully transparent). These *billboards* are rendered in back to front order on top of any conventionally rendered geometry.

Unfortunately, the model must be divided into a set of spatially disjoint objects. Furthermore, because of nonlinear perspective foreshortening, the images cannot be linearly scaled to correctly represent the object from viewpoints inside or outside the image-sampling sphere. The first restriction is difficult to enforce on many models, such as those presented in Section 1.2. In addition, a large diversity in object complexity makes the load-balancing task difficult.

Maciel and Shirley [Maciel95] described such a system. They employed textures and other imposters to render clusters of geometry more efficiently. The system precomputed a hierarchy of representations (multiple geometrical LODs, textures sampled from different viewpoints and colored cubes) and chose at run time which representation to use [Funkhouser93]. The system worked well for outdoor environments where there were distinct clusters of geometry but was too coarse for indoor or walkthrough-type environments.

2.1.2 Nodes of a Hierarchical Partitioning Tree

The restriction to spatially disjoint objects can be removed by using an image to replace geometry contained inside a leaf node of a hierarchical spatial partitioning (e.g. octree, BSP-tree, k-D tree, etc.). Moreover, this method achieves better load balancing. The tree is created using the criteria that all leaf nodes have a maximum geometric complexity or a maximum physical size. During the run-time traversal, a cost-benefit function is used to decide whether to render the leaf node or the corresponding image. Similar to the previous strategy, images are created from viewpoints on an image-sampling sphere surrounding each leaf node (Figure 2-2).

This approach results in many overlapping images. For example, when looking inwards from the edges of the model, images for a large number of leaf nodes need to be rendered. Given two adjacent sibling nodes, their combined depth complexity could be reduced by creating an image for the parent node. If two adjacent nodes are not siblings, then the tree must be traversed upwards until a common ancestor is found. Then, the entire subtree is rendered into an image. In the limit, we could reduce the overall depth complexity by creating images from viewpoints on the image-sampling spheres surrounding all nodes in the tree.

Moreover, geometric continuity is not maintained at the geometry-image border. The images can bisect an object or even individual primitives. For example, if we use a spatial partitioning and split primitives at the node boundaries, then a primitive might appear as half geometry and half image. The halves will only be correctly aligned when the viewpoint is at the image-sampling viewpoint. If we do not split primitives, then primitives rendered as geometry will only cover the proper "hole" in the image from the image-sampling viewpoint. Also, as in the previous strategy, the images are only perspectively correct for the viewing distance from which they were created.

Shade *et al.* [Shade96] presented an image-caching system that tried to preserve object boundaries and exhibit fewer visual discontinuities. Ultimately, such a partitioning is equivalent to replacing individual objects with images. Their method replaced nodes of the hierarchical space partitioning of a model with texture-mapped quadrilaterals. Each quadrilateral contained the rendered image of a node from a specific viewpoint. As the viewpoint changed, the system maintained the error of the texture representation and a predicted lifetime for the texture. Based on this information, the system decided whether to compute a new texture or to switch the node back to geometry. The visual artifacts (e.g. popping, geometric discontinuities, etc.) were kept under control by the error metric. As with the approach of Maciel and Shirley, this does not work well for indoor environments, or for replacement of relatively near geometry. Furthermore, the Shade *et al.* system did not run at interactive rates on the hardware they used.

Simultaneously, Schaufler and Stuerzlinger [Schaufler96] developed a slightly different image-caching system. As a preprocess, they used k-D trees to hierarchically subdivide the model primitives into approximately cube-shaped boxes. At run time, the system stored with each box a texture-mapped quadrilateral that represents the geometry contained within the box. The textures of adjacent boxes were used to build the textures for boxes higher up in the k-D tree. A cost-benefit function decided when to collapse textures, when to recreate textures and when to display geometry.

2.1.3 Arbitrary Model Subsets

The third strategy allows arbitrary (contiguous) subsets of a model to be replaced with images. This freedom of image placement exacerbates the geometric continuity problem but provides great flexibility, especially if we are trying to compute the best subsets of a model to replace with images. Moreover, if these images are opaque they have a depth complexity of one. They represent all the geometry shadowed by the image from a fixed distance. Since images can be arbitrarily placed, we must assume objects will be split. Furthermore, almost all primitives along the border will be rendered as partially image and partially geometry. Thus, geometric discontinuities appear from all viewpoints other than the image-sampling viewpoints.

Ebbesmeyer [Ebbesmeyer98] described an algorithm that replaced arbitrary subsets. He manually inserted virtual textured walls into a model. The wall quadrilateral spanned the entire model area – thus eliminating the geometric discontinuity problem. An image representing the scene behind the quadrilateral from a predetermined image-sampling viewpoint was texture-mapped onto the wall. Geometry behind the wall was culled.

Sillion *et al.* [Sillion97] manually selected arbitrary subsets of a city model and replaced them with textured depth meshes. The meshes mimic parallax effects of the buildings and surrounding scenery but at a reduced rendering expense.

The algorithm we introduce in Chapter 3 automatically replaces subsets of a model with images positioned anywhere in the model space. We present in Chapter 4 an algorithm to warp depth images in order to maintain geometric continuity and in Chapter 5 we describe an approach which uses geometry warping. To compensate for the perspective foreshortening, we create images at multiple distances or warp the image.

2.2 Additional Rendering Acceleration Methods

The work presented in this dissertation is not meant to completely replace other rendering acceleration methods. On the contrary, we believe the next research step is to combine geometry-based and image-based techniques. There is a large number of rendering acceleration algorithms. All of them concentrate on reducing the number of primitives (e.g. polygons, triangles, surface patches, etc.) to be sent down the graphics pipeline. We classify them into two groups: geometric simplification and visibility culling. The following two sections summarize multiple algorithms from each group.

2.2.1 Geometric Simplification

The methods in this group use the model's geometric description to generate simplifications. A typical algorithm either eliminates or collapses primitives. If the primitives have a mathematical form (e.g. curved surfaces), they can be tessellated at run time. We further subdivide geometric simplification into *static simplification* and *dynamic simplification*.

2.2.1.1 Static Simplification

These algorithms create, as a preprocess, multiple levels-of-detail (LOD) of the geometry in the scene. The run-time cost of these methods is often negligible. For each frame, one of the LODs is simply selected for rendering according to performance requirements and distance from the viewpoint.

The reduction strategy often depends on how the model is obtained. DeHaemer and Zyda [DeHaemer91] used an adaptive polygon subdivision algorithm to reduce the number of triangles in laser-scanned objects. Given a collection of points, another approach views the problem as connecting the vertices of the model to form an input mesh and compute a simplified output mesh [Schroeder92][Hoppe93][Eck95][Certain96][Klein96].

Other sophisticated methods process individual objects in the model. For example, Turk [Turk92] presented an algorithm which used an intermediate representation (called a mutual tessellation) from which vertices are removed. The surface is locally re-triangulated in such a fashion as to maintain the original surface as well as possible. Cohen *et al.* [Cohen96, Cohen98] presented algorithms that maintain local topology and guaranteed that all approximations are within a user-specified distance of the original object. Garland and Heckbert [Garland97] presented a simplification algorithm that can rapidly produce high quality approximations of polygonal objects. Their method iteratively collapses vertex pairs and maintains surface error approximations by using quadric matrices. Rossignac and Borel [Rossignac92] and Low *et. al.* [Low97] described methods that are less dependent on an object hierarchy. They use vertex clustering to produce a continuum of representations containing a decreasing number of faces and vertices.

The algorithms for generating LODs can be combined with run-time systems to create a bounded frame rate rendering system. Funkhouser *et al.* [Funkhouser93] proposed an adaptive and predictive algorithm for LOD selection, which when combined with zero-polygon LODs, can always reach a desired frame rate.

Aliaga *et al*. [Aliaga98a] proposed a system that combined geometric and image-based acceleration techniques. They divided the model into near and far geometry. Near geometry was reduced to a desired geometric complexity by using level-of-detail simplification and visibility culling. Far geometry was rendered using an approximately constant-size image-based representation. This system also rendered models larger than main memory using viewpoint prediction and disk paging.

2.2.1.2 Dynamic Simplification

These algorithms perform the simplification at run time, although auxiliary data structures might be created during a preprocessing phase (these are sometimes referred to as continuous level-of-detail algorithms). The simplified set of primitives is typically dependent on viewpoint and view direction. Hoppe [Hoppe96, Hoppe97] introduced (view-dependent) progressive meshes as a scheme for storing and transmitting arbitrary triangle meshes. In addition, he presented a new mesh simplification procedure that tried to preserve the geometry and appearance of the original mesh. Luebke and Erikson [Luebke97] extended the basic vertex clustering idea to produce a dynamic hierarchy of simplifications. At run time, they chose the largest vertex clusters for a given screen-space error threshold. Silhouette preservation was also used. Xia *et al*. [Xia96] used a dynamic view-dependent simplification method. The algorithm also employed a larger number of geometric primitives near the object silhouette and illumination highlights.

Finally, curved surfaces lend themselves well to dynamic simplification. Kumar *et al.* [Kumar95, Kumar97] presented serial and parallel algorithms for dynamically tessellating NURBS surfaces into triangles.

2.2.2 Visibility Culling

Visibility culling algorithms determine which subset of a model is visible from the current viewpoint. A variant of these algorithms is occlusion culling, which determines the opposite, namely: the obscured portions of a model to not render. Neither algorithm, alone, can guarantee a frame rate, since some viewpoints might require a large number of primitives.

Airey [Airey90] and Teller [Teller92] did extensive work in visibility computations for densely occluded polyhedral environments (e.g. architectural walkthroughs). The methods proposed use a precomputation scheme to divide the model into cells (e.g. rooms) and portals (e.g. doorways). At run time, an online algorithm tracked visibility between the cells and produced a reduced set of geometry to render. Luebke and Georges [Luebke95] presented a fast conservative algorithm for dynamically determining visibility in a cell-partitioned model.

Zhang *et al.* [Zhang97] used a hierarchical set of occlusion maps, generated at run time, to reduce the rendering workload. For each frame, the algorithm rendered a small set of previously selected occluders to form an initial 2D screen-space occlusion map, from which a hierarchy of maps was built. The occlusion maps were used to cull away geometry not visible from the current viewpoint. Coorg and Teller [Coorg97] and Hudson *et al.* [Hudson97] also proposed occlusion-culling algorithms. Both methods performed culling in 3D object-space (as opposed to screen space) and only supported convex-shaped occluders.

2.3 Flight Simulators

Flight simulation is a specialized use of computer graphics that dates back to the 1960's [Schachter83]. Extensive work has been done in generating LODs, terrain models and terrain textures [Mueller95]. The focus is on meeting quality and performance goals for very specific datasets. Frequent manual intervention occurs to tune the dataset for performance. Nevertheless, the developers are among the first to produce rendering systems with high-fidelity imagery and 30 or 60 updates per second. The following paragraphs sample published work that has emerged from flight simulators.

As early as the seventies, Rife [Rife77] created a system with vector displays and LODs. The hardware imposed a maximum number of edges and vertices to render. This determined which LOD to use. Soon after, multiple experiments were conducted to characterize the effects of scale, shape, orientation, aerial perspective (e.g. fog), and texture gradients among other effects in low-level-high-speed flights and landing sequences [Stevens81]. One experiment used fog to desaturate the color of objects as distance increased. Another experiment used texture patterns to reflect distance [Crawford77].

Texture mapping has been widely used to add apparent complexity to flight-simulator scenery. Many systems used a terrain model (polygonal representation of the surface, often available at multiple LODs) and a terrain texture (which also became available at multiple LODs – MIP mapping). Gardner *et al.* [Gardner81] combined quadric surfaces with texture-mapping. Robinson [Robinson85] introduced the notion of texture maps, contour maps (textures used as masks to enable a subset of the pixels for rendering), modulation maps (textures used to change surface color), and combinations of these. Because of the limited storage capacity of systems of this era, Clark and Brown [Clark87] described MRIP, a process to generate self-repeating textures from non-repeating photographic images.

Scarlatos [Scarlatos90] investigated geometric simplification of terrain triangulation. Hooks *et al.* [Hooks90] was concerned with efficiently draping a photo-realistic texture over terrain. In general, flight simulator systems try to maintain a clear independence between the terrain model and the terrain texture, so that the LOD of the terrain model can change arbitrarily and the edges of the terrain texture need not coincide with the edges of the terrain model. [Ferguson90] used an enhanced triangulation to slowly subdivide the terrain to higher levels of detail while minimizing the geometric differences from one level to the next.

As the external storage capacity of systems increased, it became necessary to create more extensive texture database managers. Pratt *et al.* [Pratt92] described a fast terrain paging system that only rendered the "tiles" near the user. The tiles were rendered at different resolutions according to their distance from the user. The system was designed to work over a collection of systems connected by Ethernet. Moshell *et al.* [Moshell92] dynamically computed portions of a terrain by using physical models. Cosman [Cosman94] presented a system capable of managing very large terrain databases. He strictly maintained a separation between terrain model and terrain texture. Clark and Cosman [Clark90] also described a system, which allows for easy terrain and texture independence. In addition, these systems started using in-betweening of LODs. This process consisted of interpolating between the vertices of one level and the next. Transparency fading was used to ease the visual transition between two levels.

Flight simulator research is still very active. As computer graphics technology has become more widespread, there has been a growing overlap of rendering acceleration algorithms. The careful tuning of terrain databases used in flight simulators makes it difficult for general algorithms to produce superior results. On the other hand, automatic algorithms greatly reduce the development time and make high-fidelity visualization of arbitrary 3D models much more accessible.

2.4 Image-Based Rendering

Image-based rendering systems reconstruct a scene from a set of 2D images of the environment. Often, the images are gathered beforehand. Then, depth information is extracted from the images (e.g. stereoscopic vision) or is obtained from external sensors. One of the major goals of such systems is to avoid creating a complete geometrical model of a complex environment but to still be able to move around a synthetic visualization [Chen93][McMillan95a][McMillan95b]. For our work, we do not use real-world images, but rather computer-generated images. Hence, we trivially obtain per-pixel depth values from the hardware's *z*-buffer. We exploit the method of McMillan and Bishop [McMillan95a] to warp depth images to the current viewpoint (Chapter 4). This method uses depth values to correct the pixels of a reference image to their proper projected location for the current viewpoint.

Unfortunately, as pixels are warped, previously occluded surfaces become visible. There is no visibility information for these exposed surfaces; thus, we do not know what to display. One way to reduce these artifacts is to warp additional images. For example, Mark *et al.* [Mark97a] used a second computer-generated image created from the user's predicted position. Max and Ohsaki [Max95] described the concept of creating images with multiple samples per pixel to render trees. They warp all pixel samples, in back-to-front depth order, and are able to fill-in most of the exposures. Shade *et al.* [Shade98] expanded this idea to layered depth images for arbitrary scenes (acquired or computer-generated images). We build upon these techniques to display our images. Chapter 4 will provide more details on our implementation.

Another set of image-based methods uses a very large number of images to create a database of lighting rays for an object (*Lumigraph* [Gortler96] and *Lightfield* [Levoy96]). Subsequently, the database is indexed to render a view from any position and camera direction. In the first half of Chapter 6, we show the results of using a dense sampling of images to represent scenery. The large storage requirement is prohibitive for some systems. Thus, we also develop an image warping approach that uses less storage.

Some image-based rendering approaches do use geometry to reconstruct a synthetic scene. Darsa *et al.* [Darsa97] used cubical environment maps pre-rendered from manually selected viewing positions. As a preprocess, an image-segmentation algorithm was applied to each of the six faces of a cubical environment map. The resulting segmentations were then triangulated. Texture coordinates were assigned to the geometric primitives in order to create a textured mesh that approximates the scene from the center of the cubical environment map. At run time, the closest cubical environment map was selected for rendering. A second cubical environment map was optionally rendered using alpha-blending in order to fill-in some of the exposure artifacts. Alpha-blending of two or more cubical environment maps helps to hide artifacts for images sampled relatively closely. In a similar fashion, Sillion *et al.*'s method [Sillion97] used textured meshes specialized to a city model. While these methods are promising, we ultimately chose the image warping approach to display our images. Our image warping implementation does not require geometry rendering. Thus, we can fully utilize the graphics hardware for rendering the remaining model geometry. The main CPUs are used to perform the image warp.

3. Automatic Image Placement

Our approach to replacing geometry with images has a preprocessing component and a run-time component. In this chapter, we describe the preprocessing algorithm to determine the location of all images that may potentially be required to represent geometry during subsequent rendering.

3.1 Overview

The goal of the preprocessing algorithm is to limit the number of primitives to be sent down the graphics pipeline. The standard approach to this problem has been to reduce the scene complexity by using geometric simplification (e.g. levels of detail) and visibility culling. We have taken a different approach: we limit the number of primitives to be rendered for each viewpoint and each view direction within the model space by replacing a contiguous

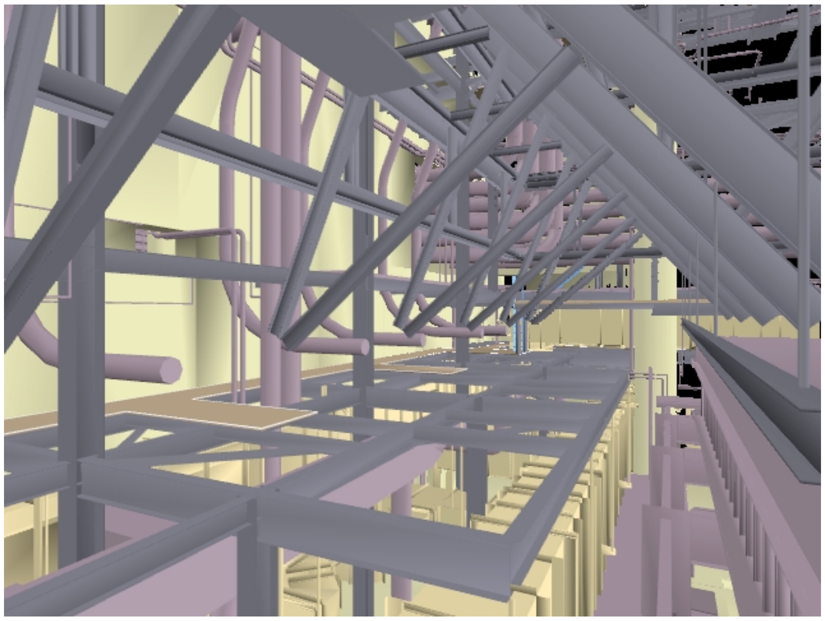

Figure 3-1. *Geometry+Image Example. In this view of the power plant, we have automatically determined what contiguous subset of the visible geometry to render as an image in order to not exceed a maximum primitive count for the complimentary subset. The piping and structures in the background are actually a warped image.*

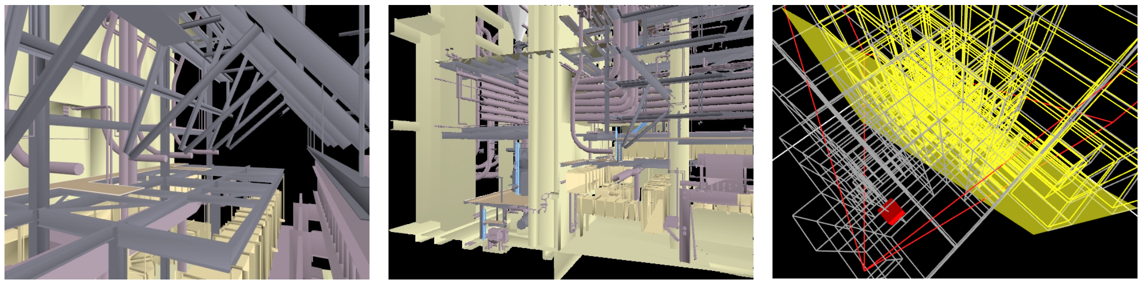

Figure 3-2. *Decomposition of Geometry+Image Example. These three snapshots illustrate how the power plant example was rendered. In the left snapshot, we render the portion of the model represented as geometry. In the middle snapshot, we render the portion of the model represented as a warped depth image. The right snapshot depicts a bird's eye view of the model, where: the four red lines outline the view frustum, the small red box represents the image-sampling viewpoint of the image, the transparent yellow plane is the image quadrilateral, the white outlined boxes are the octree-cells rendered as geometry, and the yellow outlined boxes are the octree-cells culled by the image.*

subset of the visible geometry with an image. Then, we render each replaced subset as an image displayed in constant time (Figures 3-1 and 3-2).

The image and the subset of the model it culls define a *solution* for a given viewpoint and view direction. Clearly it is impractical to compute a solution for all viewpoints and view directions. Instead, we approximate by using a finite grid of viewpoints in the space of the model. A frustum, with the same field-of-view (FOV) and view direction as the eye, centered on the closest grid viewpoint in the reverse projection of the eye's frustum will contain at least as much geometry as the frustum at the eye. Hence, we can use the same image to limit rendered geometry for both locations. Figure 3-3 illustrates this key observation. Frustum B has the same FOV and view direction as frustum A. The center-of-projection (COP) of frustum B is also the closest grid viewpoint contained within the reverse projection of frustum A. Clearly, the total number of primitives contained in frustum B is greater than or equal to the number of primitives in frustum A. Thus, if we compute images to bound the number of primitives to render from all grid viewpoints, we have also limited the number of primitives to render for any eye location within the model space. Our preprocessing task reduces to

- finding a good set of viewpoints and view directions to sample the space, and
- finding an appropriate subset of the scene to represent as an image.

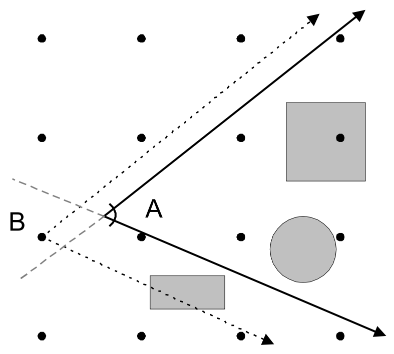

***Figure 3-3**. Key Observation for Viewpoint Grid. Frustum B has the same FOV and view direction as frustum A. Furthermore, frustum B is centered on the closest grid viewpoint contained in the reverse projection of frustum A (as indicated by the lightly dashed lines). Clearly, frustum B contains at least as much geometry as frustum A. Thus, if we compute images to bound the number of primitives to render from frustum B's grid viewpoint, we have also limited the number of primitives to render for any eye location that contains the same grid viewpoint in its reverse frustum.*

Our system stores a model in a hierarchical spatial partitioning data structure. We recursively construct an octree [Clark76] to partition the model into a hierarchy of boxes. At run time, we dynamically cull the subset of the octree occluded by the image associated with the closest grid viewpoint contained in the current reverse projection. Octree boxes on the other side of the image plane that are completely obscured can be safely culled from the model. Boxes that are partially visible can be further partitioned or not culled at all. We choose not to cull these boxes. Thus, some geometry is rendered "behind" the edges of the image quadrilateral and is never actually visible; in practice this amounts to only a small number of primitives. Figure 3-4 summarizes the entire preprocessing algorithm.

```
Enqueue all initial grid viewpoints
Repeat
   Dequeue grid viewpoint
   Compute view directions with unique culling results
   While (most expensive view > geometry budget)
      Compute octree subset to remove
      If (solution invalid) then
         Subdivide local grid
         Enqueue additional grid viewpoints
      Endif
   Endwhile
Until (queue empty)
```

***Figure 3-4.** Automatic Image-Placement Algorithm Summary.*

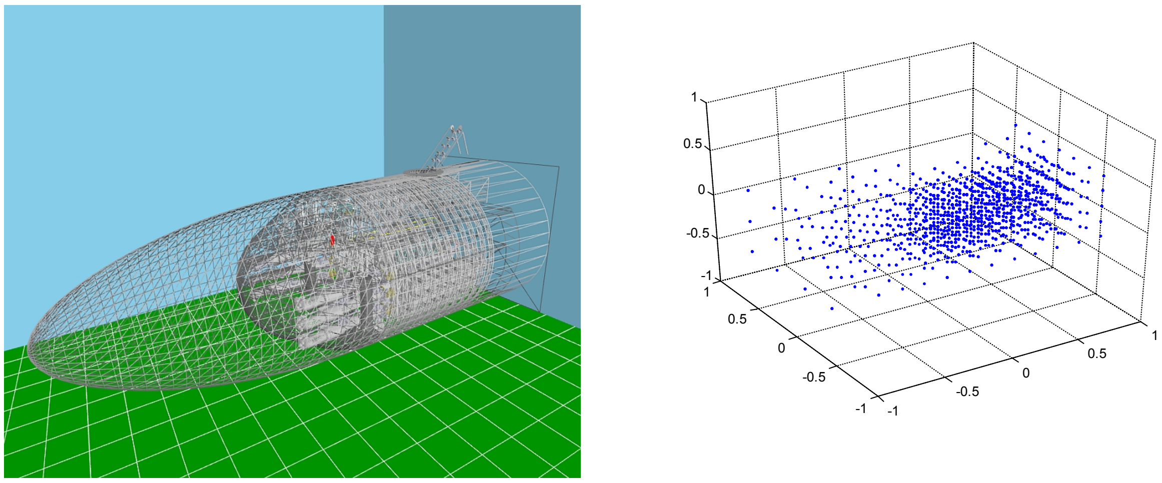
Figure 3-5. *Torpedo Room Viewpoint Grid. This figure illustrates an automatically computed viewpoint grid for the torpedo room model. The left snapshot shows an exterior view of the model rendered in wireframe. The right plot shows a grid of 1557 viewpoints from where images are computed to limit the 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.*

3.2 Viewpoint Grid

The algorithm starts by creating a uniform grid of viewpoints that spans the model space. The resolution of the grid affects not only how tightly we can bound the geometric complexity but also the final image quality. A higher resolution grid requires more preprocessing time and storage for the image data but decreases image warping artifacts (Chapter 4). We first assume a uniform grid and describe an image placement process at each viewpoint (Section 3.3). Subsequently, we locally adapt the resolution of the grid at viewpoints where we cannot produce a valid image placement (Section 3.4). The resulting grid will contain a set of viewpoints from where to place images in order to guarantee a maximum number of primitives to render anywhere inside the model space. Figure 3-5 shows an example grid for the torpedo room model.

If the eye location is near the edge of the model space and looking into the model, there might not be a grid viewpoint behind the eye. To address this, we make the viewpoint grid slightly larger than the model space (Section 3.4).

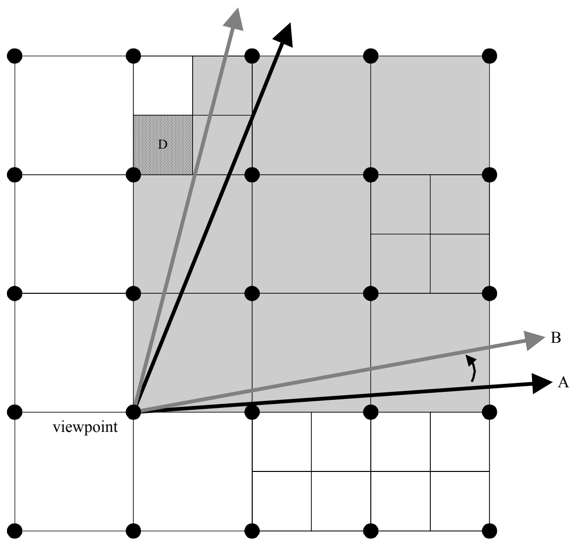

Figure 3-6. *View-Directions Set. This example depicts a 2D slice of an octree (i.e. quadtree) and two view frusta. If we rotate counter-clockwise about the viewpoint from view frustum A to view frustum B, the group of octree leaf cells in view remains the same. Only if we rotate beyond B, will cell D be marked visible, thus changing the group of visible octree cells.*

3.3 Image Placement at a Grid Viewpoint

In this section, we describe how to compute an image location that will limit the amount of geometry to render for any view direction centered on the given grid viewpoint. We start by describing an algorithm that computes a set of view directions adapted to the model complexity. Given this set of view directions, the placement task is reduced to a discrete optimization. We present a cost-benefit function to guide our optimization, as well as additional data structures.

3.3.1 View-Directions Set

We need to limit the number of primitives to render for all view directions centered on a grid viewpoint. The basic approach we have followed is to create a sampling of view directions. Then, for each sampled view direction, we ensure that the maximum primitive count is not exceeded.

We could use a uniform sampling of view directions. To prevent accidentally missing a high complexity view, we would make sure to use view directions that produce overlapping fields-of-view. Unfortunately, a coarse sampling of view directions will make it difficult to

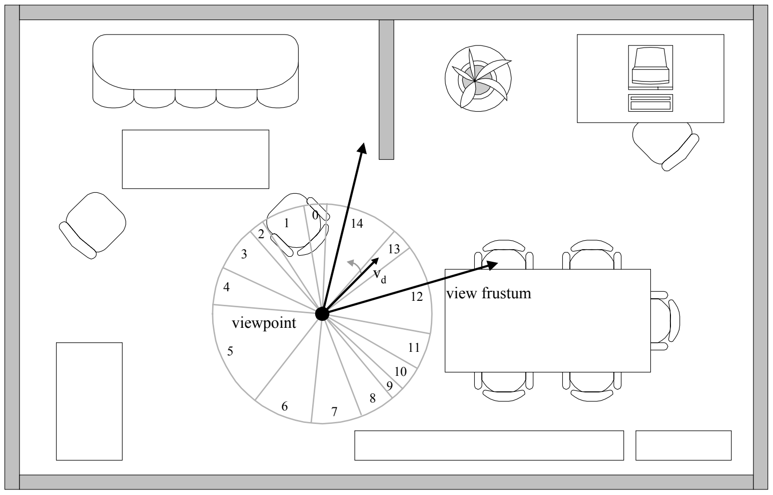

Figure 3-7. *Example Yaw-Rotation Disk. The above disk approximates 15 yaw-ranges that would produce different culling results for a hierarchical spatial subdivision of the depicted model. Thus, given a fixed viewpoint and FOV, view-frustum culling outputs a different set of geometry only when the view direction v_d rotates out from sector 13.*

place an image near its optimum location. Moreover, we would like to prevent an unnecessarily dense sampling, since that would increase the preprocessing time.

Our alternative is to create a *view-directions set* automatically adapted to the model complexity surrounding a grid viewpoint. We exploit the fact that the model is stored in an octree (or another hierarchical spatial partitioning data structure). In an octree, culling is applied to the groupings of geometry stored in the tree and not to the individual geometric primitives. Thus, as we examine view directions obtained by rotating about a grid viewpoint, the geometry to render will be the same until view-frustum culling adds or removes an octree cell. This fact turns the infinite space of view directions into a finite one, proportional to the model complexity.

The granularity of the culling operations is that of the leaves of the tree. Because of this granularity, there are varying angular ranges of movement around a grid viewpoint that will not change the results of view-frustum culling (Figure 3-6). Consider only allowing yaw rotation of a pyramidal view frustum centered on a grid viewpoint (1-D rotation). The culled set of octree cells remains constant until the left or right edge of the view frustum encounters a vertex from an octree cell. Therefore, we divide the 360-degree yaw range about a grid viewpoint into a disk with sectors (Figure 3-7). All views, using a fixed FOV, whose view direction (the vector from the viewpoint to the center of the FOV) fall within the same sector, will cull to the same set of octree cells. In our implementation, we use such a disk to represent the 1-D rotation-space. This representation contains a finite number of sampled *views.* We associate with each view the visible octree cells of the model. These data are used during the image placement process.

We could expand this representation to allow for yaw and pitch rotation of a pyramidal view frustum (the latter only has an interesting range of ±90 degrees). We would represent this 2D rotation-space using the yaw and pitch values to index a position on a manifold surface. At each sampled location, we would store the visible set of octree cells. In our experience with interactive walkthroughs, roll is a rare operation, so we would not need to account for it.

If the octree has a large number of leaf cells, we might sample a large number of views per grid viewpoint. The abundant leaf cells reduce the typical angular range a view frustum can move before changing the visible set. In order to reduce the number of views and reduce preprocessing time, we can select an arbitrary tree depth to temporarily act as leaf cells (*pseudo-leaf cells*). The shallower octree conservatively represents the complexity surrounding the grid viewpoint. Then, we use this shallow octree to determine the views around a grid viewpoint. In essence, we sacrifice granularity for preprocessing performance.

3.3.2 Image Placement: A Discrete Optimization

Our image placement process will compute octree-cell subsets to *not* render from a given grid viewpoint. Then, 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 any 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 viewpoint but increases the overall number of grid viewpoints—we discuss this tradeoff later.

The image placement process starts with the view direction containing the most primitives. If the number of primitives in this view is less than or equal to the geometry budget, then we move on to the next grid viewpoint until all have been processed. If the view exceeds our geometry budget, we compute a contiguous subset of the model to remove from rendering in order to meet the optimization budget. We compute only one contiguous subset per view because it will require at most one image per frame—this simplifies the image placement process. If, after removing the computed 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.

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 the 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. Thus, the cost is proportional to the number of primitives left to render. We divide by G_c in order to obtain a normalized value in the range [0,1].

$$Cost = g_c/G_c$$

The *benefit* is computed from the width I_w and height I_h of the screen-space bounding box of the current subset of the model and the distance *d* from the viewpoint to the nearest point of the subset. Smaller (in screen space) and farther subsets will have larger benefit values. These criteria will benefit our image quality (Chapter 4).

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

The above equation contains multiple constants. We experimented with various values for these constants and converged on a single set of constants for our test models. The constants are

- B_1, the weight for image size component,
- B_2, the weight for image depth component,
- *A*, length of the largest axis of the model,

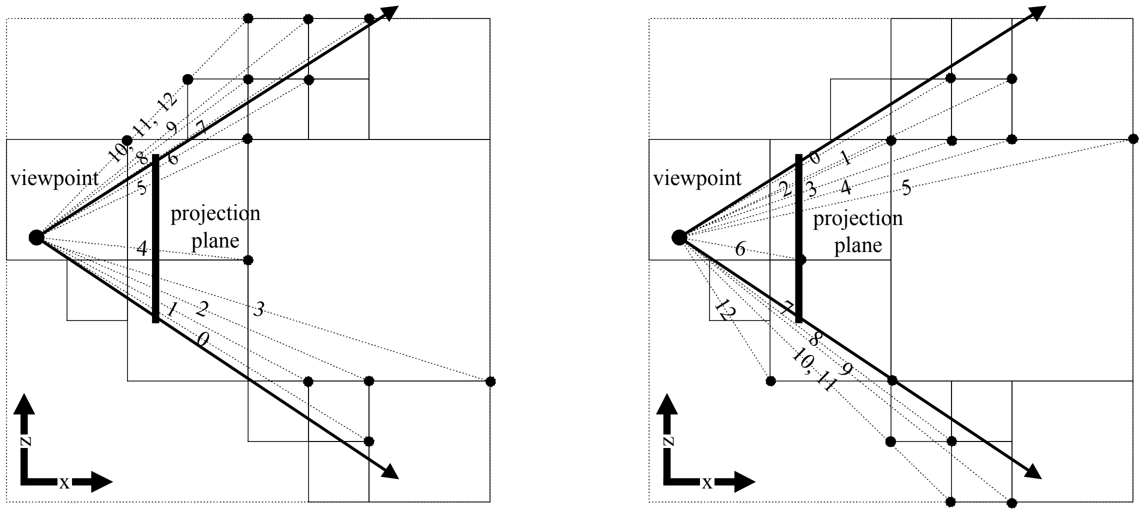

Figure 3-8. *Octree-Cell Subset Representation. These diagrams show two (of the six) sorted lists of a 2D slice of the octree cells in a view frustum (i.e. quadtree). The left diagram shows the bottom-to-top ordering of the topmost coordinates of the visible octree cells. The right diagram shows the 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.*

- S_w, screen width, and
- S_h, screen height.

The final cost-benefit function *CB* will tend to maximize the benefit component and minimize the cost. We considered using a ratio of the cost to the benefit or a weighted sum of the two. Both formulations are valid, but we chose to use a weighted sum. This gave us the flexibility to experiment with evaluations that ignored cost. As with the benefit function constants, we experimentally determined a single set of weights to combine cost and benefit evaluations for our test models. 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 viewpoint that contains all the visible geometry.

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

3.3.2.2 Representing Octree-Cell Subsets

The image placement process will search through the space of all possible contiguous 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 a binary search through the space. 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.

One approach is to choose a screen-space bounding-box and compute the octree cells that fall within it. Then, use a step factor to increment or decrement the bounding-box size and compute a new group of octree cells and so forth. Given an arbitrary model, it is difficult to choose a step factor to balance preprocessing time and search granularity.

We enhance this basic approach to a more powerful one that automatically adapts to scene complexity and easily allows binary searches through the space of all octree-cell subsets. A large number of octree leaf cells are rendered in high complexity areas. We position the screen-space bounding-box for our search to coincide with octree leaf cell boundaries and snap between cells. Consequently, we can finely change the bounding box in areas of high complexity and coarsely change the bounding box in areas of less complexity. The bounding box is exactly that of the octree cell subset it surrounds.

We represent an arbitrary, contiguous octree-cell subset with a 6-tuple of numbers. We store with each octree cell its positions in six sorted lists. The 6-tuple is a set of indices into the sorted lists representing the leftmost, rightmost, bottommost, topmost, nearest, and farthest borders of a subset (Figure 3-8). 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. With this representation, it is straightforward to incrementally update the screen-space bounding box of the subset as well as the count of geometry. We simply determine which octree cells have been added or removed from the subset and update the corresponding values.

We create the lists by first projecting the octree cells to screen space and use their minimal projected x and y values to create sorted lists 1,3 in ascending order. Then, we use their maximal projected x and y values to construct sorted lists 2,4 in descending order. Finally, we construct sorted lists 5,6 in ascending and descending order using the projected z values (i.e. distance from eye to nearest and farthest point of each octree cell).

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. If we wish 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 have an edge that projects to the same location in screen space, we set a flag in the associated sorted list to mark them as coincident. Subsequently, we move the index pointer past all coincident cells as if they were one entry.

3.3.2.3 Inner Optimization Loop

Our inner optimization loop meets the optimization budget for the current view by using the cost-benefit function and the 6-tuple subset representation to perform a binary search through the space of all contiguous subsets. The optimizer starts with the set of all octree (leaf) cells in the view frustum, e.g. [0,99,0,99,0,99]. At each iteration, moving the border along the *x*-, *y*-, and *z*-axis produces 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 we eventually create an image immediately in front of the subset, we do not 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 between subsets, preference is given to the subsets in the order they are listed. The iterations stop when the subset no longer culls enough geometry. The subset with the highest cost-benefit value is kept. Section 3.5.2 describes the time complexity of this algorithm.

We then define the image plane to be a quadrilateral perpendicular to the current view direction and exactly covering the screen-space bounding box of the computed subset. The four corners of the quadrilateral together with the current grid viewpoint determine a view frustum for creating the image to replace the subset. Chapters 4 and 5 explain 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 viewpoint.

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

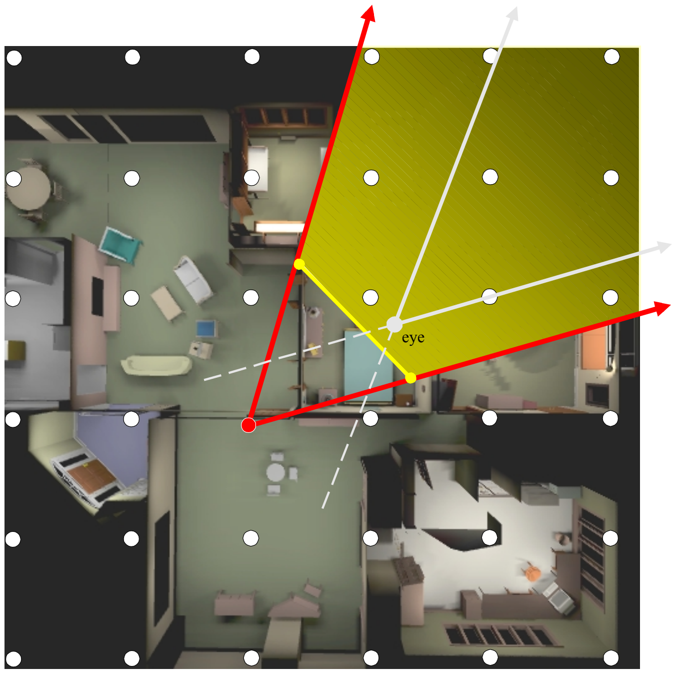

Figure 3-9. *Image Placed Behind the Eye. We show a top-down view of an architectural model. A plane of viewpoints from a uniform grid is shown. The image computed for the closest grid viewpoint in the reverse view frustum is behind the eye. This problem occurs because scene complexity forces the image to be very near its grid viewpoint. The reverse view frustum is drawn with dashed lines. The image plane and culled geometry are shaded in*

3.4 Adapting the Viewpoint Grid to the Model Complexity

3.4.1 Star-shapes

Image distance from the viewpoint is determined by the image placement process described in the previous section so as to leave no more than the specified amount of geometry. Images may be placed near their grid viewpoint. For eye locations not near a grid viewpoint, the closest grid viewpoint in the reverse projection of the eye's frustum might be *behind* the eye (Figure 3-9). In this case, no geometry or images would be rendered.

The first step to addressing this problem is to understand for what eye locations might a particular viewpoint's solution be used. Intuitively, we need to answer the question: "what is the locus of eye locations for which a given grid viewpoint might be the closest grid viewpoint in the reverse projection of the eye's frustum?". The left half of Figure 3-10 depicts a grid of viewpoints. This grid has a uniform distribution of viewpoints and is defined to be a level 0 grid -- thus an even-level. If we only allow rotations about the vertical axis (i.e.

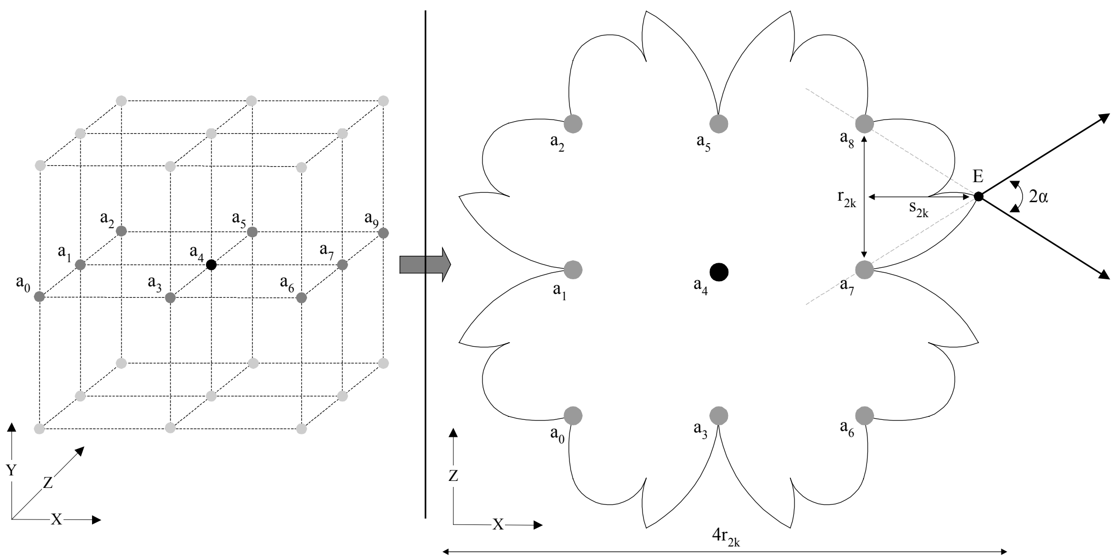

Figure 3-10. *Even-Level Star-Shape. To the left, we show a uniform grid of 3x3x3 viewpoints. The initial grid is considered to be at recursion level 0 (i.e. k=0) -- thus an even-level. To the right, we show a top-down view of the horizontal plane defined by grid viewpoints a_0-a_8. If we rotate about the vertical axis and translate a square view frustum, the star-shape represents the plane of locations that might use grid viewpoint* a_4. *The distance s_{2k} equals $r_{2k}/(2\tan\alpha)$; thus, for a FOV $2\alpha \geq 54$ degrees, $s_{2k} < r_{2k}$. We can approximate the star-shape with a sphere of diameter $4r_{2k}$.*

y-axis) and translations in the plane, the right half of the figure shows the locus of locations (*even-level star-shape*) that might contain grid viewpoint a_4 as the closest grid viewpoint in the reverse projection of the square viewing frustum. *E* is the farthest eye location from which there is a view direction that still contains a_4 as its closest grid viewpoint in the reverse view frustum. The distance s_{2k} is less than r_{2k}, the separation between grid viewpoints, as long the FOV is greater than or equal to 54 degrees. Thus, we can approximate the star-shape with a circle of diameter $4r_{2k}$. Because of the symmetry of the uniform grid and the square FOV, we construct a similar star-shape for the vertical planes and fit a sphere of diameter $4r_{2k}$.

Hence, for a practical FOV of 60 degrees, we can prevent the problematic situation by ensuring that no grid viewpoint has an image placed within its star-shape. If we superimpose the problem case of Figure 3-9 with the star-shape, we see that the image is indeed inside the star-shape.

We could reduce the star-shape to a simpler shape by allowing for images with a field-of-view wider than the view frustum's. This would potentially decrease the overall

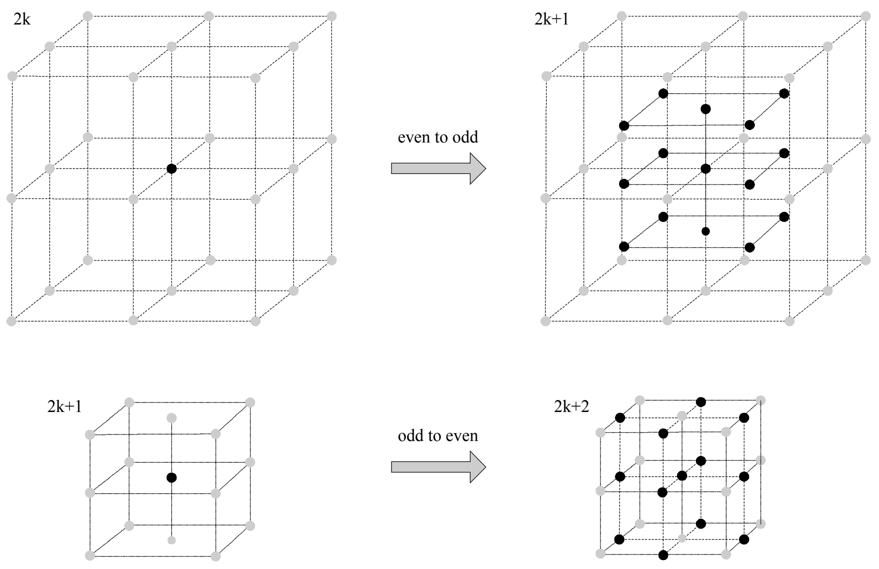

***Figure 3-11**. Even- and Odd-Level Grid Viewpoint Subdivision. We show even-to-odd and odd-to-even grid viewpoint subdivisions. In the upper half, we subdivide a level 2k viewpoint, in the middle of a grid, to produce 15 level 2k+1 viewpoints. They are placed at the midpoints between the original subdivided viewpoint and the neighboring viewpoints. In the lower half, we subdivide a level 2k+1 viewpoint to produce 13 level 2k+2 viewpoints. Thus, we have returned to an even-level grid configuration.*

number of images surrounding a grid viewpoint. But, in order to keep the same approximate image quality, we would need to increase the image resolution. It is unclear whether this approach will achieve an overall win, but the larger image size would create even more conservative solutions, thus suggesting a less desirable approach.

Eye locations near the edge of the model might not contain a grid viewpoint in the reverse projection of the frustum. Thus, we expand the grid by two viewpoints in all six directions (i.e. positive and negative *x*-, *y*-, and *z*-axis) to ensure that such eye locations have a grid viewpoint behind it.

3.4.2 Recursive Procedure

To allow images to be placed nearer to grid viewpoints, we created a recursive procedure for reducing the size of star-shapes by locally subdividing the grid. Consequently, the images selected for the current viewpoint and view direction will always be in front of the eye. The recursive procedure alternates between two sets of recursion rules: one for *even levels* (*2k*) and one for *odd levels* (*2k+1*). This two-step procedure first introduces grid viewpoints at the midpoints of the existing viewpoints and then introduces the

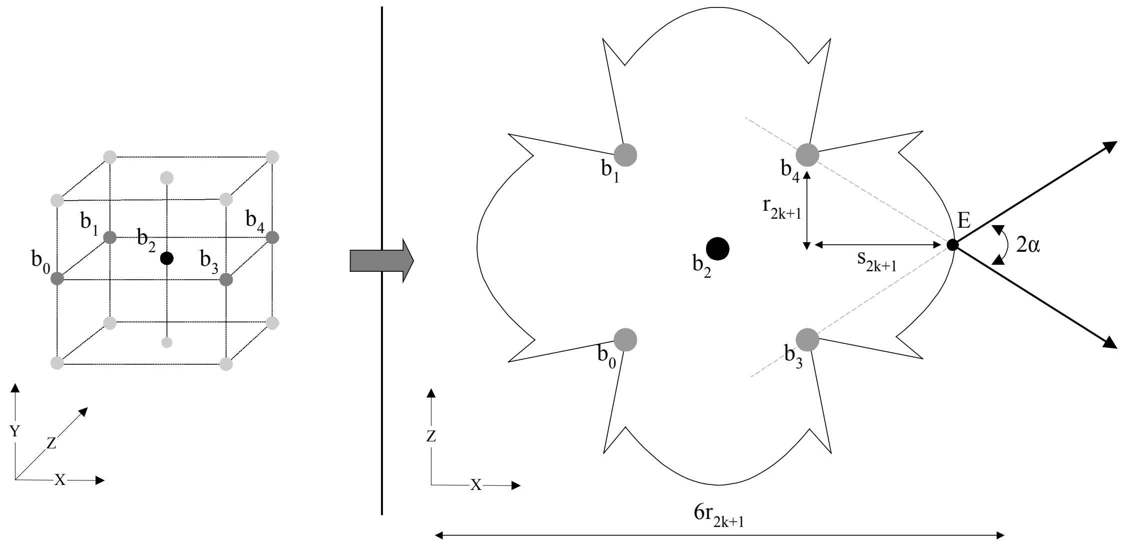

Figure 3-12. *Odd-Level Star-Shape. To the left, we show a portion of a grid subdivided to an odd-level. To the right, we show a top-down view of the horizontal plane defined by grid viewpoints b_0-b_4. If we rotate about the vertical axis and translate a square view frustum, this star-shape represents the plane of locations that might use grid viewpoint b_2. The distance s_{2k+1} equals $r_{2k+1}/\tan\alpha$; thus, for a FOV $2\alpha \geq 54$ degrees, $s_{2k+1} < 2r_{2k+1}$. We can approximate the star-shape with a sphere of diameter $6r_{2k+1}$.*

complementary viewpoints to return to a denser original configuration. At each new level, we verify that for all viewpoints a valid image placement can be produced (Section 3.3). We recursively subdivide viewpoints that fail until all viewpoints have image placements in front of any eye location that might use them.

Even-level recursion creates fifteen new viewpoints to replace the original level *2k* viewpoint (*subdivided viewpoint*). Fourteen level *2k+1* viewpoints are created at the midpoints between the subdivided viewpoint and the surrounding level *2k* viewpoints (Figure 3-11). A level *2k+1* viewpoint also replaces the subdivided viewpoint. In a similar fashion to before, we construct a star-shape that surrounds the new viewpoints. The resulting odd-level star-shape is slightly different. Figure 3-12 illustrates the star-shape as well as how the extrema are computed. In this case, we require a sphere with diameter $6r_{2k+1}$ to approximate the star-shape.

If a new level *2k+1* viewpoint still produces an image that lies within the star-shape, we proceed to an odd-level recursion. This case produces the remaining viewpoints to return to an even-level grid configuration. Twelve level *2k+2* viewpoints complete the grid

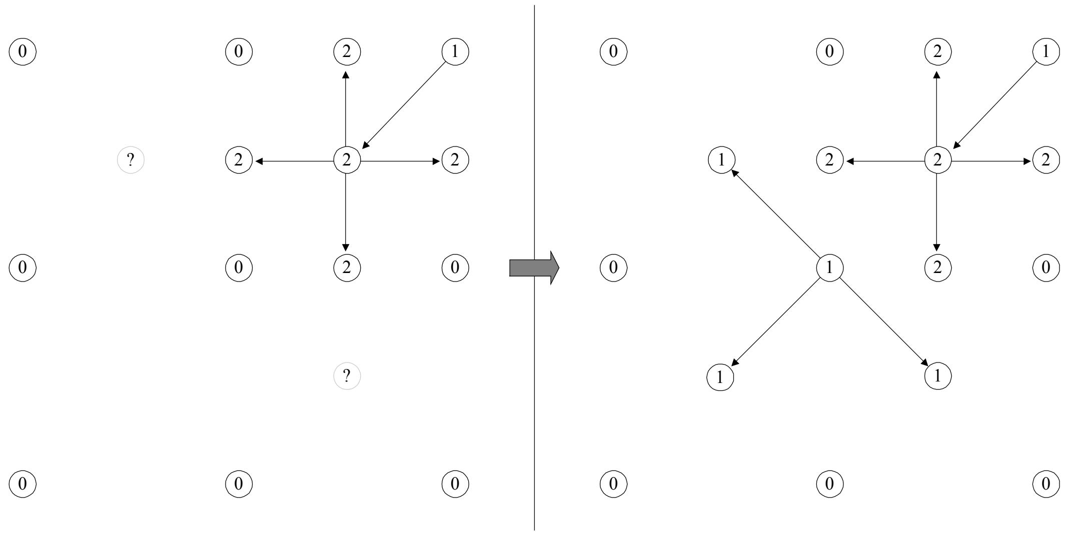

***Figure 3-13**. Star-Shape Minimal Sets. In order for the star-shapes to correctly depict the eye locations from which a grid viewpoint's solution might be used, we must ensure their minimal viewpoint set is present. The left grid depicts a subdivision that produces grid viewpoints with incomplete minimal sets. In the right grid, we have performed an additional subdivision to complete the minimal sets. We provide more details in Section 3.4.3 of the text.*

surrounding the subdivided viewpoint (Figure 3-11). As with the previous level, a level *2k+2* viewpoint also replaces the subdivided viewpoint.

3.4.3 Star-Shape Minimal Viewpoint Sets

In order for the star-shapes to represent the locus of eye locations from which a grid viewpoint's solution might be used, we must ensure that the *minimal viewpoint set* is present. The minimal viewpoint set consists of the viewpoints added to the grid during even-to-odd and odd-to-even subdivisions (Figure 3-11). We use them as a template to verify that a similar configuration of neighbors surrounds all grid viewpoints. Otherwise, the star-shapes no longer delimit the locus of eye locations. If a grid viewpoint has an incomplete minimal set, we subdivide the adjacent viewpoint that is at the lowest recursion-level. We repeat this process until a valid star-shape is constructed for every grid viewpoint.

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 boundary between the patches. We must perform an additional tessellation of the intermediate region.

We illustrate this with a 2D example. For an even-level 2D star-shape, the minimal set is viewpoint a_1, a_3, a_5, and a_7 from Figure 3-10. Without these, a sphere of diameter $4r_{2k}$ no longer approximates the eye locations from which a_4's solution might be used. Similarly, for an odd-level 2D star-shape the minimal viewpoints are b_0, b_1, b_3, and b_4 from Figure 3-12. The left grid of Figure 3-13 is constructed by a sequence of two subdivisions:

1. the upper-right level 0 viewpoint is subdivided (only the subdivided viewpoint and one of the new level 1 viewpoints fall within the grid), then
2. the new level 1 viewpoint is subdivided (and produces four level 2 viewpoints as well as replacing itself with a level 2 viewpoint).

Two of the grid viewpoints have incomplete minimal sets, as indicated by the lightly shaded dots. The right grid of the same figure shows a subdivision that produces complete minimal sets for all grid viewpoints. To construct it, we add a third subdivision:

3. we subdivide an adjacent viewpoint with the smallest recursion-level (we choose the middle viewpoint).

If we had chosen another one of the adjacent level 0 viewpoints, we would end up with a different, yet valid, solution.

3.5 Complexity Analysis

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.

We pose two questions to measure the complexity of our algorithm. For each question, we analyze our algorithm by providing bounds on best-case and worst-case scenarios. The questions are

- how many images do we need for an entire model, and
- how long does it take to compute a single image-placement?

The models we encounter in practice have complexities that fall somewhere between the two bounds. A formal specification of the average-case scenario is difficult to construct

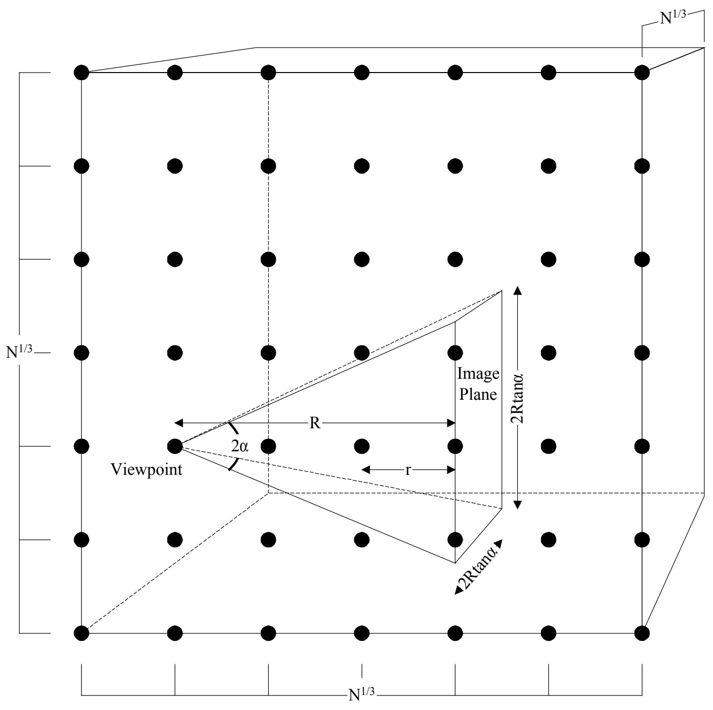

Figure 3-14. *Number of Grid Viewpoints (View-Frustum Pyramid). Our goal is to compute a conservative number of grid viewpoints N for a uniform density model. We start by finding an R that creates a view-frustum pyramid containing no more than P primitives (for a cubical model space). The pyramid is constructed by connecting the viewpoint, with a FOV of 2α, and the image plane's quadrilateral. Then, we use the fact that r equals at worst 1/3R to determine the spacing between grid viewpoints (Section 3.2) and compute the value for N.*

since it depends on the distribution of geometry within the model space. Nevertheless, we show how for a given geometric distribution, we combine the two bounds to predict an average-case performance. Furthermore, we show in Section 3.6 empirical results that reinforce our analysis.

3.5.1 How many images?

The total number of images required is equal to the number of grid viewpoints times the number of images per grid viewpoint. In the following two subsections, we present the best-case and worst-case scenarios with respect to the total number of images.

3.5.1.1 Best-Case Scenario

The algorithm performs overall best in models with a uniform distribution of geometry (by best performance, we mean that for a given frame rate, the smallest amount of space is needed). For such models, a fixed-resolution viewpoint grid is sufficient. Consequently, only even-level star-shapes are needed. If the geometric distribution were non-uniform, we would need to subdivide at least one viewpoint--thus increasing the number of the grid viewpoints.

As will become apparent later, a uniform distribution of geometry around each grid viewpoint produces the largest number of images per viewpoint. Nevertheless, the total number of images is still significantly less than the worst-case scenario. The balance point between number of viewpoints and images per viewpoint falls under the average-case scenario.

To compute the space requirement for a uniform distribution, we must first determine the fixed resolution of the viewpoint grid. We assume that for all grid viewpoints the image quadrilateral will cover the entire FOV at some distance R from the eye. The following formula computes the number of primitives P in a *view-frustum pyramid* with a constant FOV of 2α by multiplying the volume with a density term (Figure 3-14). The volume of a pyramid is equal to $1/3*A_{base}*H$. The constant term λ reflects the primitive count per unit volume:

$$P(R) = \lambda \frac{1}{3}(2R\tan(\alpha))^2 R$$

We then choose a maximum primitive count P and solve this formula for R, namely the distance between the eye and the image plane:

$$R(P) = \sqrt[3]{\frac{3P}{4\lambda \tan^2(\alpha)}}$$

The value R is equal to the radius of the associated star-shape. The radii of the star-shapes of Section 3.4 are at most 3 times larger than the separation between the grid viewpoints. We then conservatively compute the number of grid viewpoints N for a unit cube volume:

$$N(P) = \frac{3}{R(P)}\frac{3}{R(P)}\frac{3}{R(P)} = \frac{36\lambda \tan^2(\alpha)}{P}$$

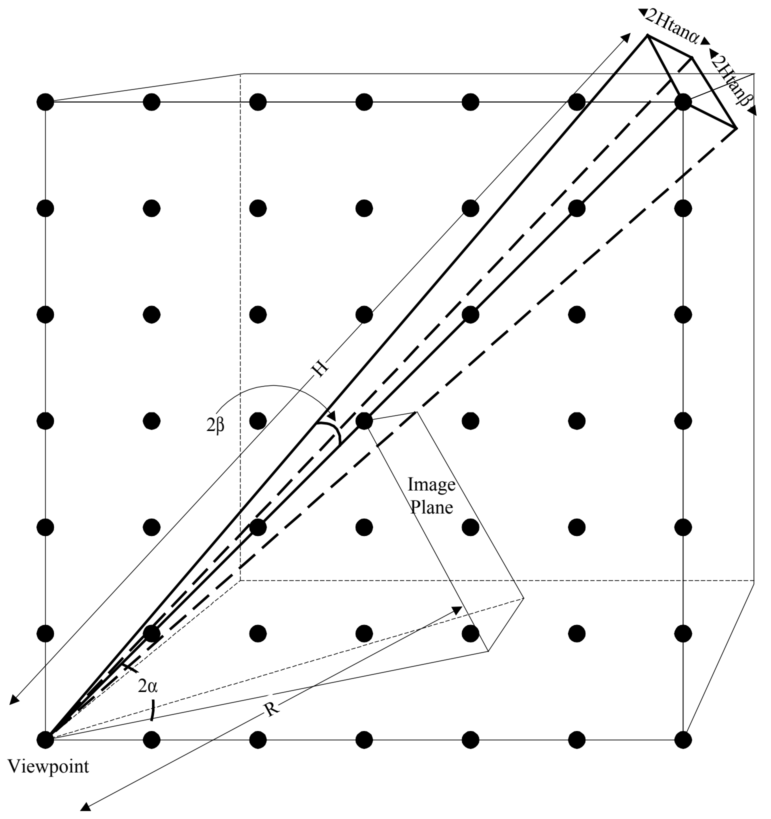

Figure 3-15*. Number of Images Per Grid Viewpoint (New-Primitive Pyramid). To compute the number of images M required at a grid viewpoint, we first estimate the number of new primitives that enter the view frustum as we rotate about the grid viewpoint. The new-primitive pyramid is defined by the viewpoint, the rotation 2β and the border of the model space. This figure depicts the pyramid with the largest possible height H, namely the pyramid constructed when the viewpoint is at the model corner looking inwards. Next, we compute the rotation 2β that yields a pyramid containing at most $D=P-P_{opt}$ primitives. Finally, we count how many such rotations we can make to determine M.*

Next, we need to determine how many images are required per grid viewpoint. We know the number of primitives between the image quadrilateral and the eye is at most the optimization budget. Figure 3-15 depicts the new primitives that enter the view frustum as we rotate counter-clockwise by 2β around a grid viewpoint. The location of the grid viewpoint, within the model space, determines the height *H* of a *new-primitive pyramid*. For a unit cube volume, Figure 3-15 shows the tallest pyramid possible. To be conservative, we assume the new-primitive pyramid's height to be equal to the largest height centered on its associated viewpoint. The shortest pyramids occur for grid viewpoints in the middle of the model. The area of the pyramid's base is determined by the FOV and by the view frustum rotation. Thus, we approximate the number of new primitives P_{new} by multiplying the volume of the new-primitive pyramid times the primitive density:

$$P_{new}(H,\beta) = \lambda \frac{1}{3}(2H\tan(\alpha))(2H\tan(\beta))H$$

The total number of primitives in a rotated view frustum is equal to the optimization budget plus the number of new primitives (in fact, this is a slight overestimation since the primitives at the apex of the pyramid are counted twice). The difference *D* between the geometry budget and the optimization budget dictates how far we can rotate until the total number of primitives exceeds the geometry budget. Thus, we assign $P_{new}=D$ and solve the previous equation for β:

$$\beta(H,D) = \tan^{-1}(\frac{3D}{4\lambda H^{3}\tan(\alpha)})$$

We trivially compute the number of images *M* needed at a grid viewpoint by counting how many 2β rotations can be made:

$$M(H,D) = \frac{2\pi}{2\beta(H,D)}$$

Finally, we combine the equations for *N* and *M* to obtain the number of images I_{total} required for the entire model space. *M* is actually a function of the model-space position of the grid viewpoint. For the formulation we have given, viewpoints near the middle of the model will have smaller values for *H*; thus, they require fewer images than at the edge. We sum the number of images in an octant of the model. Then, by symmetry, the sum over the entire model is eight times that value:

$$I(P,D)_{total} = 8\sum_{x=0}^{K}\sum_{y=0}^{K}\sum_{z=0}^{K} M(H(K),D)$$

$$K(P) = \frac{\sqrt[3]{N(P)}}{2} \qquad H(K) = \sqrt{(1-\frac{x}{2K})^{2} + (1-\frac{y}{2K})^{2} + (1-\frac{z}{2K})^{2}}$$

P and *D* are input parameters corresponding to the maximum geometry count and the difference between the geometry count and the optimization count.

In order to obtain a more intuitive result, we perform simplifications for a potential scenario. First, we assume that the model space is a unit cube and assign *H* to equal half the length of the diagonal of the XZ plane (this is equivalent to assuming that all grid viewpoints have short pyramids and, given a large enough number of viewpoints, is roughly the typical case). Second, we assume a FOV of 60 degrees ($2\alpha=\pi/3$). Third, we assign $P=\rho P_{total}$ and $D=\delta P_{total}$, where P_{total} is the total number of primitives and $\rho,\delta \leq 1$. We now simplify the equations for N, M, and I_{total}:

$$N(\rho)=\frac{12}{\rho} \qquad M(\delta)=\frac{\pi}{\tan^{-1}(\frac{3\sqrt{3}\delta}{\sqrt{2}})} \qquad I(\rho,\delta)_{total}=N(\rho)M(\delta)$$

This result is easier to understand and to graph. The lower line in Figure 3-16 is a plot of I_{total} as a function of $\rho \in [0,½]$ with a constant δ.

3.5.1.2 Worst-Case Scenario

The algorithm requires the most images when there are large variations of geometric density throughout the model space. As opposed to the best-case scenario, this situation produces a large number of additional grid viewpoints to maintain valid star-shapes (Section 3.4.3). Consider a small-volume cluster of geometry with a large number of primitives. Assume the geometric distribution within the cluster is approximately uniform. The primitives outside the cluster might also be uniformly distributed, but we must create additional grid viewpoints to join the high-resolution grid of the cluster to the lower-resolution grid outside the cluster. This phenomenon is similar to the additional tessellation of curved surfaces that must be done to join a high-curvature region with a low-curvature region. The additional polygons are needed to connect the high- and low-curvature regions but not necessarily because of the surface curvature.

In a model, there can be any number of high-density clusters. For example, these clusters might contain 99% of the model primitives and occupy only 1% of the model space. The remaining 1% of the model primitives are evenly distributed throughout the remaining 99% of the model space. For an arbitrary model, we have no way of knowing a priori the number, size, and location of these clusters. But, we do know that the grid resolution outside the high-density clusters will never exceed the resolution within the clusters.

We compute the number of images for the worst-case scenario by assuming the grid resolution of the clusters is the dominant resolution. If we know the number and size of these clusters, we could determine this resolution quite well. Unfortunately, we do not have this information. Thus, we assume a case that produces the largest number of grid viewpoints: a single cluster containing almost all of the primitives and occupying only an infinitesimal-volume of the model space (the cluster cannot contain the entire model since that would reduce to the uniform distribution case).

For the worst-case scenario, we use the same formula for *N* as in the best-case scenario but increase the density by a factor close to 100% (e.g. 99%). Moreover, only one image is required per grid viewpoint. The image will lie between the grid viewpoint and the cluster. Thus, the variable *M*=1 everywhere outside the cluster. There is almost no volume inside the cluster, so we disregard its contribution:

$$I(P)_{total} = N(P) = \frac{36(99\lambda)\tan^2(\alpha)}{P}$$

If we perform the same simplifying assumptions as in the best-case scenario, we can rewrite I_{total} as a function of ρ:

$$I(\rho)_{total} = N(\rho) = \frac{12(99)}{\rho}$$

The upper line in Figure 3-16 is a plot of this simplified result for $\rho \in$ [0,½].

3.5.1.3 Example Average-Case Scenario

For our algorithm, it is very difficult to define the average-case scenario since it would rely on defining the average-case geometric distribution of a 3D model. Given a large number of 3D polygonal models, it might be possible to use stochastic methods to establish an average distribution, but we can easily encounter another model that does not fall within the stereotype. Despite this, we can safely state that a typical model has geometric clusters of different densities, in addition to regions of uniform distribution.

We approximate a geometric distribution by quantizing the density of portions of a model. For our purposes, we do not care about the position of the subsets but only about their relative density and volume. For example, we divide the model into four (potentially overlapping) subsets. Each one is evenly distributed within a subvolume of the model space, e.g.:

- 25% of the model primitives are distributed over 25% of the model space
- 25% of the model primitives are distributed over 50% of the model space
- 25% of the model primitives are distributed over 75% of the model space
- 25% of the model primitives are distributed over 100% of the model space

Then, we compute how many images are required for each subset and sum the results. We only need to vary the density term to reflect the distribution. For each subset, we use the same simplifying assumptions as before to obtain formulas for *N* and *M,* as a function of ρ and δ:

$$N(\rho)_1 = (\frac{1}{1})\frac{12}{\rho} \qquad M(\delta)_1 = \frac{\pi}{\tan^{-1}(\frac{3\sqrt{3}\delta}{\sqrt{2}}(\frac{1}{1}))}$$

$$N(\rho)_2 = (\frac{1}{2})\frac{12}{\rho} \qquad M(\delta)_2 = \frac{\pi}{\tan^{-1}(\frac{3\sqrt{3}\delta}{\sqrt{2}}(\frac{2}{1}))}$$

$$N(\rho)_3 = (\frac{1}{3})\frac{12}{\rho} \qquad M(\delta)_3 = \frac{\pi}{\tan^{-1}(\frac{3\sqrt{3}\delta}{\sqrt{2}}(\frac{3}{1}))}$$

$$N(\rho)_4 = (\frac{1}{4})\frac{12}{\rho} \qquad M(\delta)_4 = \frac{\pi}{\tan^{-1}(\frac{3\sqrt{3}\delta}{\sqrt{2}}(\frac{4}{1}))}$$

We can of course change the distribution and obtain a different result. But, we have outlined a method to predict an average-case performance. The final I_{total} measures the space requirement for the specified distribution:

$$I(\rho,\delta)_{total} = \sum_{i=1}^{4} N(\rho)_i M(\delta)_i$$

3.5.2 How much time does it take?

The preprocessing time can be obtained by multiplying the optimization complexity times the number of images and adding the grid-adaptation complexity. In the following paragraphs, we summarize the grid-adaptation complexity and the optimization complexity.

The preprocessing time for subdividing and adapting the viewpoint grid is proportional to the number of viewpoints. Using the property that the number of internal nodes of a tree never exceeds the number of leaf nodes, we know that no more than twice the final number of viewpoints N are created. The computation is dominated by the time it takes to construct each grid viewpoint's view-directions set. For each view, we perform a $O(L)$ view-frustum culling operation to obtain the visible octree cells, where L is the number of visible octree leaf cells. Hence, given a maximum of V views per grid viewpoint, the grid-adaptation complexity is $O(NVL)$.

The preprocessing time for an image-placement optimization is proportional to the number of visible octree leaf cells. At each iteration, the optimization algorithm creates five new subsets of octree leaf cells by halving the original subset along each axis (Section 3.3.2.3). To decide which new subset to use, the algorithm recursively computes, for each new subset, the cost-benefit value that would result in Q iterations. Thus, the resulting optimization complexity is $O(5^{Q+1} \log L)$.

The number of octree cells L in the current view depends on the field-of-view, the view direction, the viewpoint position, and the stopping criteria when the octree was built. The *best-case* performance occurs when the viewpoint is located at the corner of the model and looking outwards: there are no octree cells in view. On the other hand, the *worst-case* occurs when the view direction turns to face the middle of the model (similar to Figure 3-15). In this case, the FOV determines what subset of the octree cells is visible. Given a FOV of 2α and a total count of T octree leaf cells, we can compute the maximum value for L:

$$L = \frac{8T\sqrt{2}}{3}\tan^2(\alpha)$$

3.6 Results

3.6.1 Implementation

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

- the octree of the 3D model,
- the geometry and optimization budget,
- the FOV,
- resolution of the initial viewpoint grid (minimum 3x3x3, i.e. the size of an even-level star-shape),
- tree depth to use for defining the octree (pseudo) leaf cells, and
- cost-benefit constants (we use $B_1 = 0.2$, $B_2 = 0.8$, $B = 0.6$, and $C = 0.4$).

The preprocessing algorithm uses a single processor to create and adapt the grid; afterwards, multiple processors are used to simultaneously compute the image-placement optimizations. We divide the 360-degree range of yaw about a grid viewpoint into a disk with a finite number of view directions. 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 viewpoint is subdivided. Once the grid has been adapted, the viewpoints are divided among n processors (e.g. $n=3$). Each processor performs the discrete optimizations to compute subsets to replace with images.

Our optimization algorithm queries a grid viewpoint for its most expensive view prior to computing each image. To accelerate this, we cache the primitive count of each view using a simple array. Subsequently, we mark the dirty-bit of views that change because of culled geometry. Furthermore, we usually employ octree pseudo-leaf cells to limit the number of cells for preprocessing. For our test models, we empirically determined an octree depth of 5 yields a reasonable balance between granularity and performance (thus, a maximum of 32,768 octree leaf cells per view).

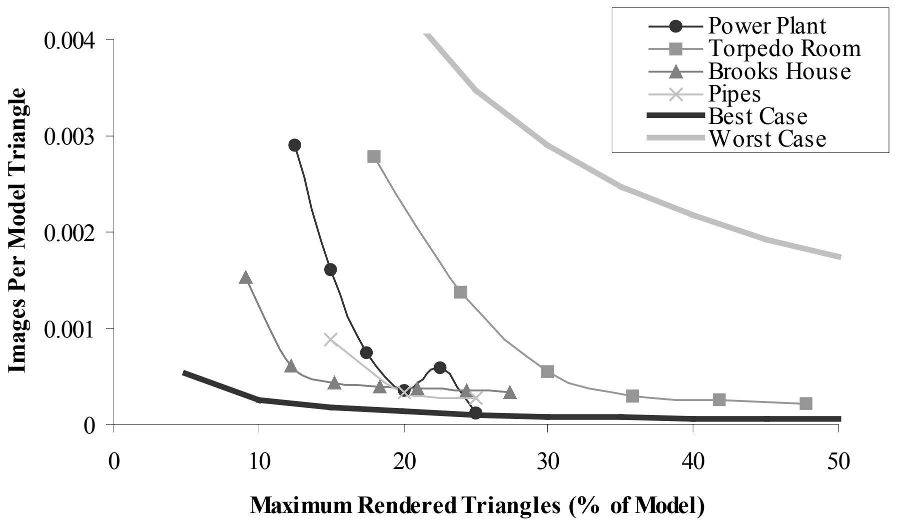

Figure 3-16. *Storage Performance. The upper gray line represents the performance of the worst-case scenario. The lower black line represents the best-case scenario. The four test models fall in between these two bounds and in fact tend towards the best-case scenario.*

3.6.2 Performance

We report the performance of our algorithm on four test models: power plant, torpedo room, Brooks House, and pipes. The first three of these models are listed in Section 1.2. We use a subset of 2M primitives of the power plant. This is the largest model we can fit in memory that leaves space for an image cache and does not require us to page model geometry dynamically. The fourth model was procedurally generated by replication and instancing of an approximately uniform distribution of pipes (courtesy of Lee Westover). It contains about 1M triangles.

Figure 3-16 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 axis. One way of doing this is to use the horizontal axis to represent 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 optimizer 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 visible geometry that was large enough to meet the target primitive count but not necessarily the smallest and farthest subset. This occurrence is

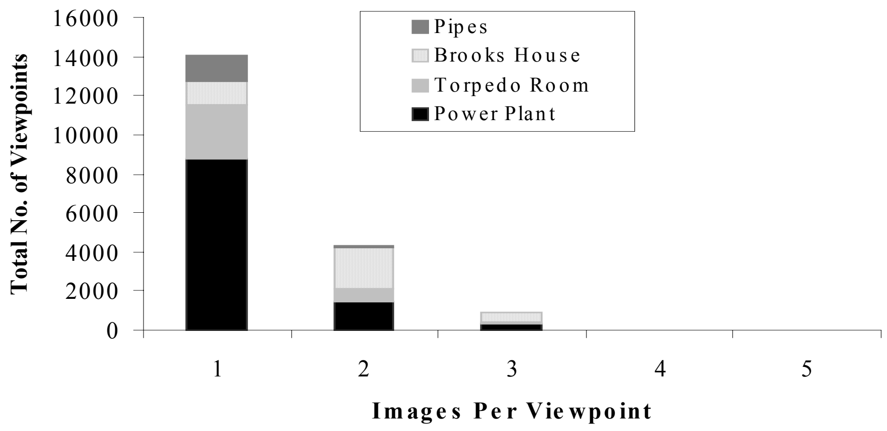

Figure 3-17. *Histogram of Images Per Viewpoint. We use D equals 5-10% of the model primitives and obtain M < 5 for all grid viewpoints. There are 14,012 viewpoints with M=1, 4311 viewpoints with M=2, 862 with M=3, 40 viewpoints with M=4 and 0 with M≥5.*

common with optimization algorithms. A potential improvement could be achieved by using a technique such as simulated annealing to move the solution to a "better" local minimum.

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 (i.e., *D* equals 5 to 10%) yields only a few images per grid viewpoint. Figure 3-17 shows a histogram of the number of grid viewpoints with the images per viewpoint *M* varying from 1 to 5 images. Figure 3-18 graphs the theoretical result of plotting *M* as a function of *D* from 0 to 100% for a perfectly uniform

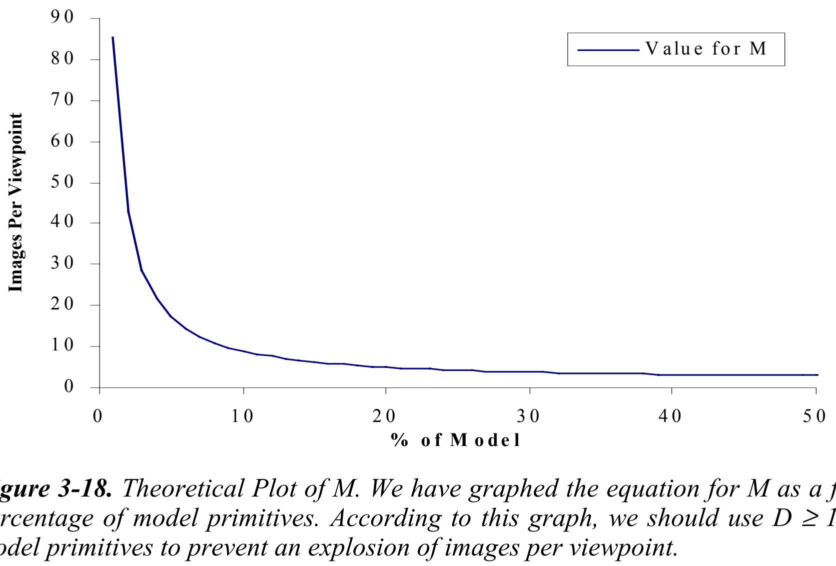

Figure 3-18. *Theoretical Plot of M. We have graphed the equation for M as a function of percentage of model primitives. According to this graph, we should use D ≥ 10% of the model primitives to prevent an explosion of images per viewpoint.*

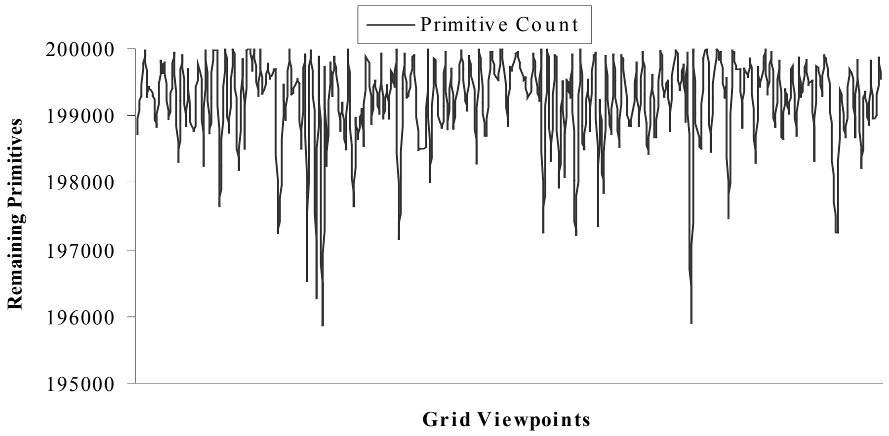

Figure 3-19. *Optimization Results for Power Plant with P_{opt}=200k primitives. The image-placement optimization computes image locations that, for all grid viewpoints, produce primitive counts within 2% of the desired value (P_{opt}). The results plotted here are fairly typical for our test models.*

distribution model. According to this graph, we should use *D* greater than 10% of the model primitives to prevent an explosion of images per viewpoint. The varying geometric distribution of real-world models tends to reduce *M*--thus the empirical and theoretical results are consistent.

Figure 3-19 illustrates how close the solutions computed by the image-placement process (Section 3.3.2.3) are to the desired optimization budget. For a given grid-viewpoint

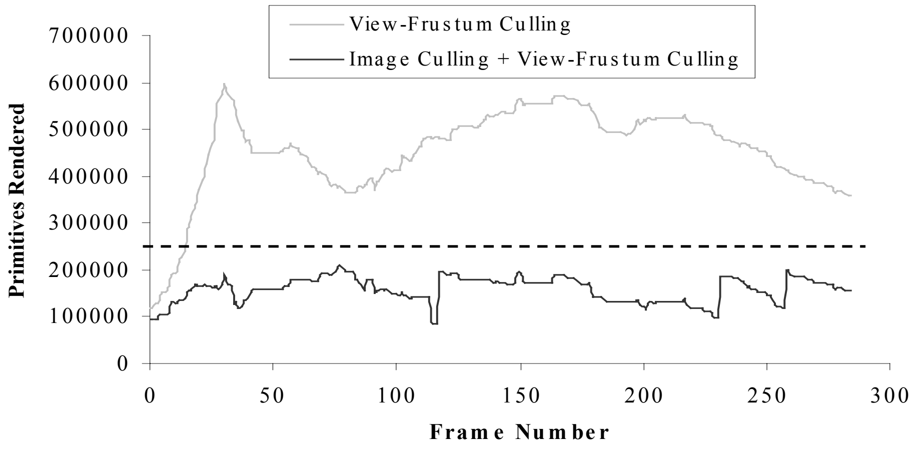

Figure 3-20. *Path through Power Plant Model. This graph shows the number of primitives rendered for a sample path through the power plant using a geometry budget of 250,000. We show the results using only view-frustum culling and using image culling plus view-frustum culling. Notice that the primitive count never exceeds our geometry budget; in fact, for this path, it almost never exceeds P_{opt}=200,000.*

view, we compute image placements that are conservative and typically fall within 2% of the optimization budget.

Figure 3-20 shows the number of primitives rendered per frame for a path through the power plant. We have automatically inserted images into the model using a geometry budget of P = 250,000 primitives and an optimization budget of P_{opt} = 200,000 primitives.

Table 3-1 (next page) summarizes the preprocessing results. For each test model, we show the number of image placements computed and the preprocessing time for grid adaptation and optimizations (Chapter 4 details rendering and construction of images). In addition, we show the estimated space requirement. To determine this, we use the average image storage size listed in Table 4-1 (Chapter 4). We then compress the images using *gzip* and use a separate processor to decompress them at run time—from this information we extrapolate space requirements (at present, we can decompress an image in under one second). The multiprocessing component of our algorithm uses three of the four processors (the fourth one is left to other users). The use of additional processors for the optimizations as well as parallelizing the grid adaptation would decrease the preprocessing time.

Model	**Maximum No. of Primitives**	**No. of Images**	**Preprocessing Time (hours)**	**Estimated Space (MB)**
Power Plant	250,000	5815	21.7	3802
2M triangles	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 Room	150,000	2333	11.8	933
850K triangles	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
Brooks House	150,000	2492	28.4	1725
1.7M triangles	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 triangles	200,000	331	2.8	205
	250,000	282	2.4	175

***Table 3-1**. Preprocessing Summary for Test Models.*

4. Depth-Image Warping

This chapter describes how we create and warp depth images at the locations computed by the preprocessing algorithm of the previous chapter. First, we present a summary of the image-warping algorithms used. We include a discussion of major artifacts introduced by warping and how we address these. Then, we outline our final image-warping algorithm and present implementation details as well as performance data.

4.1 Overview

Our preprocessing algorithm has determined the location of all the images to replace geometry—we must now create and display them. At each grid viewpoint, we have the necessary (camera) parameters to create a reference image that accurately depicts the geometry from the image-sampling viewpoint. But each image must potentially represent the selected geometry for any viewpoint within the associated star-shape.

In this chapter, we implement an image-warping method for dynamically correcting depth images to viewpoints within the current star-shape. We first summarize the basic formulation, sample reconstruction, and limitations of image warping. Subsequently, we take advantage of how the star-shapes limit the viewpoints and view directions from which an image is viewed to optimize a layered-depth-image approach. With this approach, we are able to eliminate most of the artifacts prone to occur in image warping and thus to produce high-quality visual results.

4.2 Image Warping

4.2.1 Formulation

We implemented the McMillan and Bishop image-warping equation, best described in [McMillan97]. This formulation takes as input a reference image with per-pixel *disparity* values. The disparity term is related to the classical stereo disparity measure and is proportional to the distance from the center-of-projection (COP) of the reference image to a pixel, divided by the range to the surface represented by that pixel. Thus, the disparity is inversely proportional to distance and measures how far a pixel will flow as the viewpoint changes — closer objects will move farther. We obtain the disparity value of a pixel x_i from the OpenGL *z*-buffer. The value stored in the *z*-buffer is normalized to [0,1] and corresponds to the inverse range to the near plane (with the far plane being the maximum range). We compute the disparity values of a reference image as follows:

$$\Delta(x_i) = 1 - z(x_i) * (f - n) / f$$

where,

- $z(x_i)$ is the OpenGL *z*-buffer value for pixel x_i,
- f is the distance from the reference viewpoint to the far clipping plane, and
- n is the distance from the reference viewpoint to the near clipping plane.

Given an image with per-pixel disparity values, the pixels are then warped to their correct location for the current viewpoint (Figure 4-1). We represent this operation as

$$x_2 = \Delta(x_1)P_2^{-1}(c_1-c_2) + P_2^{-1} P_1 x_1$$

where,

- x_1 is a set of coordinates for a reference image point,
- x_2 is a set of coordinates locating the corresponding point in the desired image,
- c_1 is the COP of the reference image (i.e. image-sampling viewpoint),
- c_2 is the COP of the desired image (i.e. current viewpoint),
- P_1 is the backprojection matrix of the reference image,
- P_2^{-1} is inverse of the projection matrix of the desired image, and
- $\Delta(x_1)$ is the disparity of the reference image pixel at x_1.

The amount of work for image warping is proportional to the number of pixels in the image and independent of scene complexity. Furthermore, since the reference image is on a

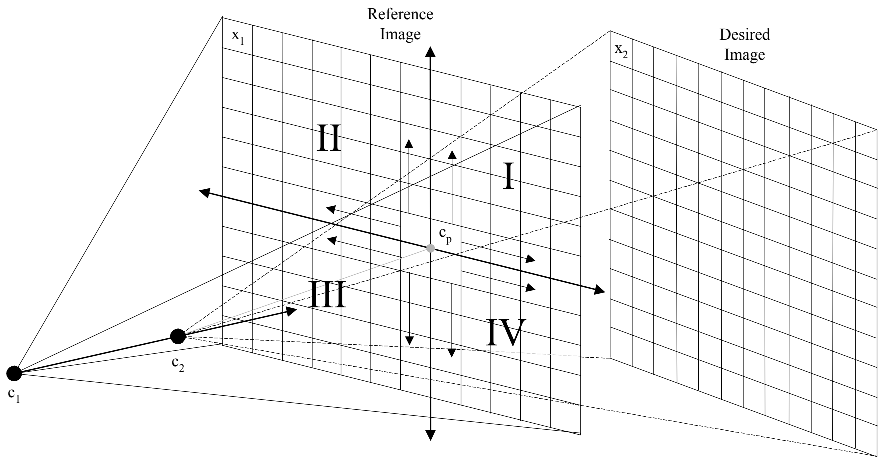

Figure 4-1. *Image Warping. The set of reference image pixels x_1 are warped from their position as seen from c_1 to their position x_2 on the desired image as seen from c_2. In this example, the projection c_p of the desired viewpoint c_2 onto the reference image divides the reference image into 4 sheets. For example, traversing the upper left sheet from right to left and bottom to top produces a correct visibility ordering of the warped pixels.*

regular grid, many of these computations are incremental and fast. The work is similar to that required by traditional texture-mapping.

The results of this warp are not one to one: multiple points in the reference image may be warped to a single point in the desired image. This raises the issue of visibility resolution. We must somehow ensure that when multiple pixels from the reference image warp to the same pixel in the desired image, the one representing the closest of the points to the current viewpoint is the one that "wins". We could use a *z*-buffer to resolve visibility, but in our case it's faster to use the back-to-front occlusion-compatible order described in [McMillan95a].

This algorithm is similar to a painter's algorithm. We first determine the projection of the COP of the desired image in the reference image. We use that point (c_p) to divide the reference image into a number of *sheets* (Figure 4-1). There are four, two, or one, depending on whether both, one, or neither of the coordinates of c_p lie in the image domain. We then determine whether we must warp the pixels in the sheets towards or away from the projected point, depending on whether the desired COP is in front of or behind the reference COP. Subsequently, we parallelize the implementation of the warp by taking advantage of the fact that sheets can be warped and rendered independently with correct occlusion guaranteed.

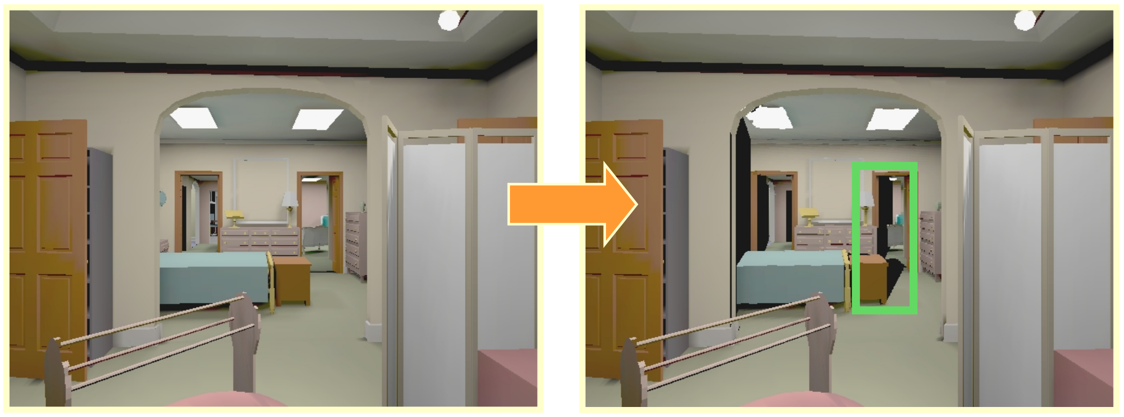

Figure 4-2. *Limitation of Single-Image Warping. If we move in three-dimensional space, regions previously occluded become visible. A single reference image cannot capture the additional visibility data. In both snapshots, the scene visible through the central doorway is rendered as a warped image (the rest is rendered using conventional geometry). The left snapshot is the view from the image-sampling viewpoint. In the right snapshot, we have slightly translated our viewpoint. Notice the "tears" or sharp shadows that have appeared because we do not have pixel samples for the previously occluded surfaces (note: we have outlined the tear to the right of the bed).*

4.2.2 Reconstruction

We've considered two methods for resampling of the desired image: bilinearly interpolated surfaces and splatting [Westover91]. [McMillan97] includes a good discussion of the reconstruction issues involved in image warping. Surface patches consume rendering resources. For our interactive rendering goals, we wish to devote rendering resources to displaying the remaining geometry. Thus, we use the main CPU to perform splats.

We have experimented with computing an approximation to the projected size of each pixel (for a more accurate splat) or using a fixed-size footprint. The former is done by treating four adjacent pixels as corners of a quadrilateral. Then, warping the corners and drawing a splat that corresponds to the rectangular bounding box of the quadrilateral. The latter option, a fixed-size splat, is cheaper to compute and still provides a visually pleasing result. In our systems, we employ a three by three footprint in preference to the more accurate solution. This approach provides satisfactory results as long as the reference image resolution is similar to the apparent resolution of the warped image in the frame buffer.

4.2.3 Limitation of Single-Image Warping

An image represents an environment from only a single viewpoint (namely, the image-sampling viewpoint). Therefore, there is information about only a single surface at each pixel, the one nearest to the COP (ignoring clipping by a hither plane). As we move in three-dimensional space warping a single reference image, we see areas of the environment that were not sampled in the image. We have no information about the exposed surfaces, so we don't know what to render. The effect of these missing samples in warped images is illustrated in Figure 4-2. If nothing is done to correct for the problem, the lack of information appears as "tears" or sharp shadows in the images.

We investigated three methods to reduce and eliminate tears. In the remainder of this section, we briefly describe the first two methods. Section 4.3 will detail the third, more comprehensive solution.

1. The simplest solution is to decrease the distance between the image-sampling viewpoints of reference images (i.e. increase the resolution of the viewpoint grid). Thus when the viewpoint moves, the reference image being warped is closer to the desired image and the widths of the tears are proportionally reduced. However, it is not always a practical solution, because it may cause a large number of reference images to be created. In our preliminary attempts, we saw up to two orders of magnitude increase in the number of required images.
2. A better solution is to warp multiple reference images [Mark97a] created from nearby viewpoints, expecting that surfaces exposed in one reference image will have been sampled in another image. We do not decide explicitly which warped pixels are best. Rather, we warp the farthest reference image first and the nearest reference image last. Moreover, we let pixels from previous frames persist [McMillan, personal communications]. This last part, although a "hack", helps to fill-in the remaining minor tears with plausible colors. In practice, warping the two nearest reference images seems to give very acceptable quality. We could warp more reference images in order to try to reconstruct the environment in greater detail. However, this is not practical because of the time involved and not particularly rewarding given the small amount of detail that is lost.

When warping the two nearest reference images, we encounter one of three visibility cases for each pixel: when both reference images contain information about the same geometry, when one image contains information absent from the other image, and when information about some geometry is missing from both images. For pixels of the first case,

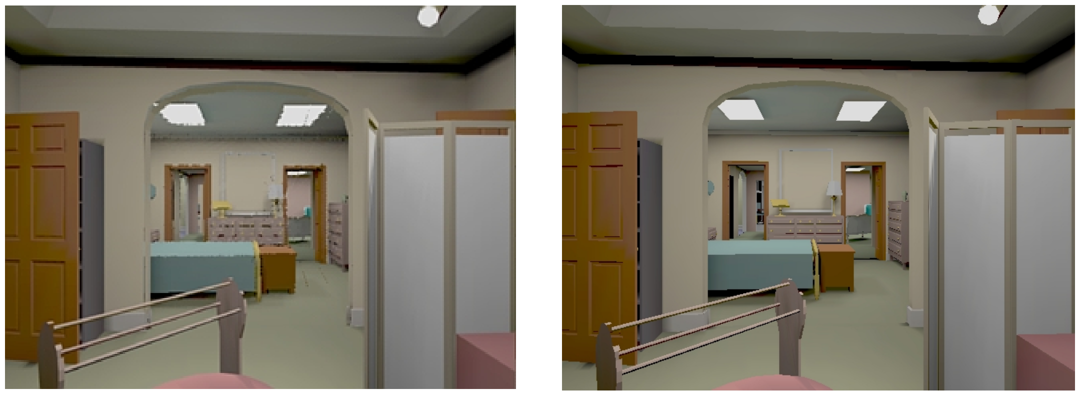

***Figure 4-3**. Example Layered Depth-Image. The doorway of the left snapshot is rendered as a layered depth-image. Notice the absence of the "tears" of Figure 4-2. For comparison, the right snapshot is rendered with conventional geometry.*

we are potentially storing the same sample multiple times thus performing redundant work (and storage). The second case becomes a problem if the last warped image is the one that is missing data from a certain location, and it contains information from a greater depth that obscures correct detail by warping to the same location. This problem manifests itself as a flashing effect, as correct detail appears and disappears under incorrectly overlaid imagery. A solution would require a decision process to choose the best source for each pixel in the composite image – a cost that might be too computationally expensive. Finally, the third case can only be eliminated by warping more reference images.

We wish to achieve the same effect of warping multiple reference images but without the redundant work and storage. Furthermore, we would like to prevent the computationally expensive process of explicitly deciding the source reference image for each pixel. This leads to the following approach.

4.3 Layered Depth Images

Layered Depth Images (or LDIs) [Max95][Shade98] 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 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

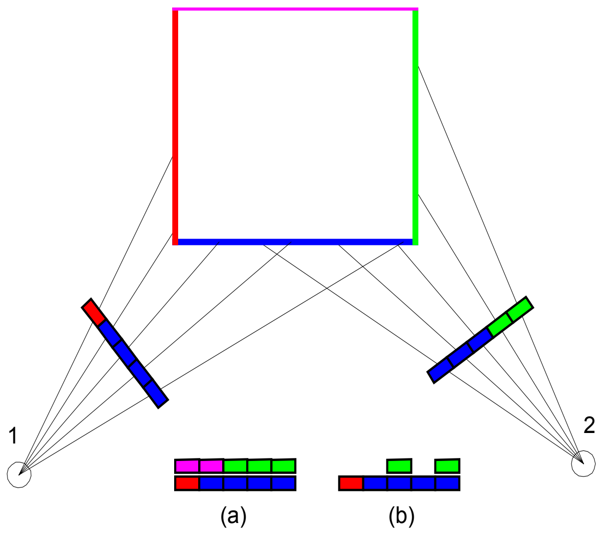

Figure 4-4. *Construction of a Layered Depth-Image. The LDI is constructed from viewpoint 1. Each sample of the LDI stores not only color, but also disparity (depth). (a) shows a LDI built by raycasting. The rays sample surfaces beyond those that would be seen from viewpoint 1. (b) shows a LDI constructed by warping a second image from viewpoint 2 to viewpoint 1. Redundant samples are discarded, others are stored at deeper layers. Note that the undersampling of the green surface from viewpoint 1 is not a problem because it's never seen from that direction.*

will naturally fill in the gaps that would otherwise appear (Figure 4-3). As will become clear in the rest of this section, most of the work is done during preprocessing, when the LDIs are constructed.

LDIs store each visible surface sample exactly once, thus eliminating redundant work (and storage) as opposed to multiple reference images. An additional benefit is that LDIs can be warped in McMillan's occlusion-compatible order [McMillan95a]. This ordering performs a raster scan of the reference image and guarantees correct visibility ordering in the warped image.

4.3.1 Constructing and Warping LDIs

Figure 4-4 shows construction of an LDI using two methods. The first, using raycasting, stores multiple samples hit by the rays of each pixel of the LDI. The second method consists of warping secondary images to the viewpoint of the primary image, then selecting samples that have not already been stored in the LDI (this is an approximate determination). During LDI construction, the main concern is to collect samples from all surfaces potentially visible from viewpoints (and view directions) within the associated star-

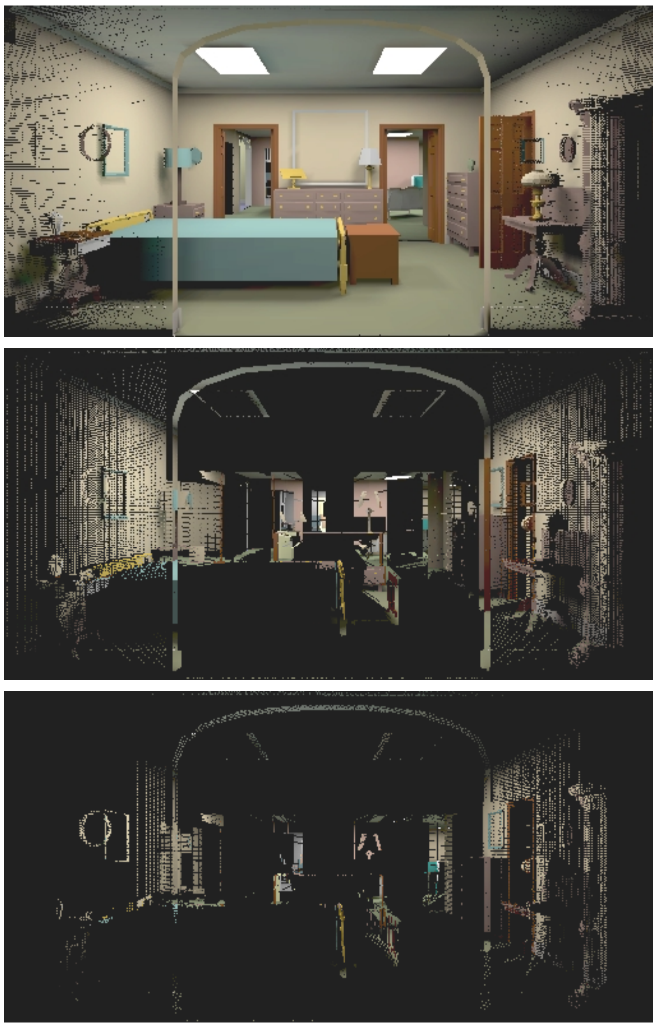

Figure 4-5. *Layers of a LDI. The images show, from top to bottom, the samples at depth 0, 1 and 2 of a LDI. The side walls are present in more than one layer not because of visibility, but rather because they are not adequately sampled by the central view; when the LDI is warped to a view where better sampling is necessary, the successive layers will unfold, ensuring better reconstruction.*

shape. Thus, we use the second method. There is no guarantee that all potentially visible surfaces are captured. However, by sampling images from the restricted space for which a particular LDI will be used, we lower the probability of missing a sample and obtain very acceptable image quality.

Briefly, to construct an LDI we first define a central image. Then, we manually select nearby construction images and warp them to the view of the central image. If multiple samples fall within the same pixel, we store them sorted by depth. If their depth values are within a small tolerance value of each other, we resolve the conflict by either tossing one sample out or by storing both samples provided they have dissimilar color values (Figure 4-5). The resulting image has multiple color plus depth pairs at each pixel. We summarize the construction algorithm in Figure 4-6.

```
Create empty LDI with view parameters of the central LDI image
For each construction image
    For each pixel (sample)
         Warp to the central LDI image
         Inspect all samples of the LDI location
         If there is a sample at similar depth,
           Resolve conflict
         Else
           Install sample in a per-pixel list at the correct location
           depth order
         Endif
    Endfor
Endfor
```

Figure 4-6. *Summary of General LDI Construction Algorithm*

The warping of the LDI proceeds in a similar fashion to single-image warping. The pixels are traversed in occlusion-compatible order thus eliminating the need for a *z*-buffer. At each pixel, the multiple samples are warped in back-to-front order. Some areas might be oversampled, causing us to warp more samples than strictly necessary. However, better sampling ensures that there will be enough samples for every visible surface from all views. Moreover, it enables us to use a small, constant-sized stamp (similar to Section 4.2.2). A more detailed discussion of issues involved with the construction and warping of LDIs (for architectural models) can be found in [Popescu98].

4.3.2 Optimizing LDIs for the Viewpoint Grid

We create the reference images for constructing an LDI from viewpoints within the star-shape surrounding each grid viewpoint. Thus, we can do a good job of sampling all potentially visible surfaces. In order to determine a good set of image-sampling viewpoints for a particular grid viewpoint, we first place an image *A* just outside an even-level star shape. Then, all view positions from which a view frustum can contain the image quadrilateral is represented by an approximate hemi-ellipsoid centered in front of the grid viewpoint. Figure 4-7 depicts a 2D slice of this configuration. For a more distant image *B*, the region will look more elongated (e.g. the dotted triangle of the same figure). A similar situation occurs with an odd-level star shape. To construct the LDI, we select reference image COPs that populate this space.

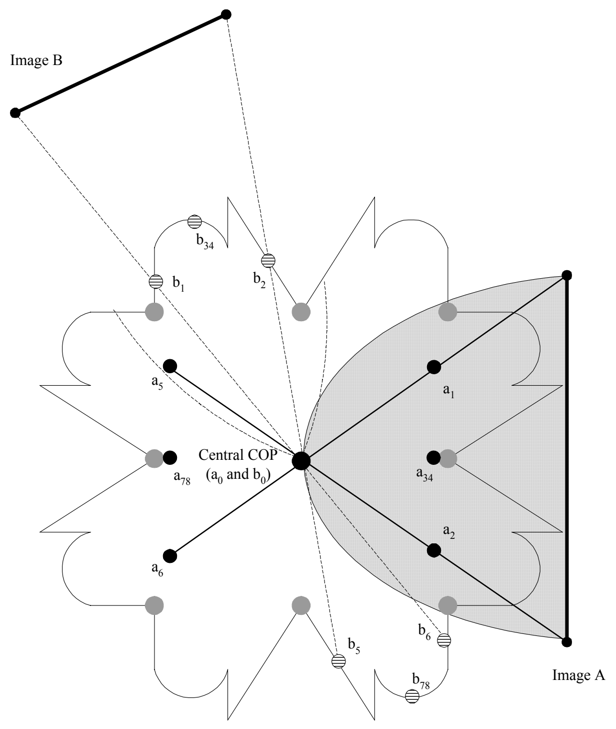

Figure 4-7. *Selecting Construction Images for a LDI. Image A is placed immediately outside an even-level star-shape. Given a fixed FOV, the hemi-ellipsoid of viewpoints within the star-shape from where there exists a view direction that contains the image quadrilateral is depicted by the shaded semi-ellipsoid. We define the central COP to be at the grid viewpoint (a_0). Four construction images a_{1-4} are placed at the middle of the vectors joining the grid viewpoint and the midpoints of each of the four edges of the image quadrilateral (a_{34} is actually the projection of two construction COPs onto the semicircle). Four more construction images are similarly defined but extending behind the central COP (a_{5-8}). b_0 and b_{1-8} are the central COP and construction COPs for a farther away image B.*

We choose a total of eight construction images and one central image to create an LDI and to eliminate most visibility artifacts. The central LDI image is created using the grid viewpoint itself as the COP (a_0 and b_0 in Figure 4-7). Four construction images are created from COPs at the middle of the vectors joining the grid viewpoint and the midpoints of each of the four edges of the image quadrilateral (a_{1-4} and b_{1-4} in Figure 4-7). An additional set of four construction images is defined in a similar way but extending behind the grid viewpoint (a_{5-8} and b_{5-8} in Figure 4-7). It would not to be advantageous to create construction images from COPs on the boundary of the hemi-ellipsoid, since surfaces visible from within the hemi-ellipsoid would be sparsely sampled. We warp the pixels of the nearest construction image first to the central LDI. 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 evenly populate the locus of eye positions from where the LDI can be viewed. Thus, they sample most of the potentially visible surfaces. The images behind the grid viewpoint help to sample visibility of objects in the periphery of the FOV. The images nearer to the LDI image plane do not sample the periphery well. An alternative is to translate laterally from the central COP, but by moving back slightly we achieve some desired oversampling. In practice, this heuristic method does a good job.

4.4 Implementation

We have implemented the image-warping algorithms in C++, on a SGI Onyx2 with 4 R10000 @ 195 MHz processors. The following sections summarize our image cache, image creation time and run-time performance.

4.4.1 Image Cache

We create a host memory cache to store image data and allow us to precompute or dynamically-compute images for an interactive session. Since single-image warping has little preprocessing, we usually render the (one) reference image on the fly. On the other hand, we typically render and construct LDIs as a preprocess and dynamically load them. If the total number of precomputed LDIs exceeds our online memory, we employ a simple prefetching algorithm. 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.

We observe that the applications outlined in Section 1.2 have viewing styles that typically fall into one of two categories: flying through a model or inspecting, in detail, one area of the model then moving on to another portion (e.g. design review of CAD models). The first style benefits more from precomputing all required images because otherwise rendering-time glitches might occur if flying speed exceeds the rate at which images can be created. If images are loaded from disk, prefetching would need to keep up with the flying speed unless we limit the maximum flying speed to the maximum rate at which we can page from disk. The latter style exhibits viewer-locality that we can exploit with demand-based computing. Creating images on demand greatly reduces storage requirements. If we employ a least-recently-used replacement policy then by simply visiting the same area twice, we essentially revert to precomputed images. With regards to prefetching, we only need to load the relatively small working set.

Model	**Image Size (pixels)**	**Image Storage (MB)**	**Per-Pixel Storage (bytes)**
Power Plant	256x256	0.8	12.2
	512x384	1.7	8.6
	512x512	2.7	10.3
Torpedo Room	512x384	1.6	8.1
Brooks House	512x384	1.8	9.1
Pipes	512x384	3.6	18.3

Table 4-1. *LDI Storage Requirements.*

4.4.2 Preprocessing

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 from the rendered eight construction images 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. To determine the average size of the power plant LDIs, we divide the total storage used by the number of images. Table 4-1 shows the average storage size of 256x256, 512x384, and 512x512 pixel LDIs. The second resolution has the same aspect ratio as a NTSC frame buffer and is the one we typically use for interactive demonstrations. For the remaining models, we create LDIs from several locations in the model and compute the average storage size. The average per-pixel storage requirement varies because of different depth complexities of our models. The procedurally generated pipes model contains many layers of pipes. Consequently, more surface samples must be stored per pixel. A single pixel sample requires three bytes for color and one byte for disparity. Thus, our LDIs typically have 2 to 4 samples per pixel.

We have experimented with simple *gzip* compression of LDIs. The power plant and Brooks House LDIs are reduced by an average factor of 2.6, while the torpedo room compresses to approximately of factor of 4. The procedurally generated pipes model, of higher average depth complexity, compresses by 5.8. We expect that more sophisticated compression methods can further reduce overall LDI storage requirements.

Method	**Image Size (pixels)**	**Updates Per Second**
Power Plant	256x256	24
	512x384	10
	512x512	9
Torpedo Room	512x384	11
Brooks House	512x384	11
Pipes	512x384	7

Table 4-2. *Run-Time Image-Warping Performance (3 processors).*

4.4.3 Run Time

The image-warping overhead is proportional to the number of pixels per image and to the number of images needed per frame. Our automatic image-placement algorithm will require at most one image per frame–thus image size is the determining factor for performance.

Our run-time system uses additional processors to warp images while culling and rendering a model. We simultaneously cull frame *n+1*, image-warp frame *n+1*, and render frame *n*. Then, at the beginning of frame *n+1*, we copy image data to the frame buffer and start rendering geometry on top of it. This scheme adds a frame of latency to the overall rendering but achieves a better CPU utilization.

To reduce traffic between the host and graphics engine, we warp all images to a common host-memory buffer (*warp buffer*). Then, at the beginning of the next frame, we scale and copy the warp buffer to the frame buffer. On our SGI Onyx^2, it takes only 2.7ms to copy a 32-bit 512x384 image from host memory to the frame buffer.

The average warp times of our layered-depth-image warper are shown in Table 4-2. We show timings for a three-processor implementation using 256x256, 512x384, and 512x512 pixel images. The LDI algorithm can use any number of processors to evenly distribute the work, but our workstation only has four processors and we must leave one for the remaining run-time operations (e.g. culling, rendering, etc.). We have found that warping is often bound by memory bandwidth; thus, image warping does not scale exactly with image size. Moreover, when fetching geometry from main memory to render in immediate mode, the reduction in memory bandwidth can affect warping performance by as much as 10 to 20%. The warp time of LDIs for the pipes model is larger because of the greater average number of layers to warp (Section 4.4.2).

We look to additional processors and dedicated hardware as means to further accelerate image warping. In addition, we lack the per-pixel information to perform view-dependent shading (e.g. specularities), thus we are currently limited to precomputed diffuse illumination. Mark *et al.* [Mark97b] present a memory architecture compatible with image-warping requirements. The UNC-Chapel Hill ImageFlow project is investigating new graphics architectures that will permit faster, more sophisticated image-warping algorithms.

5. Geometry Warping

This chapter presents our second approach to displaying images surrounded by geometry. We describe a geometry-warping algorithm to prevent geometric discontinuities and provide smooth transitions. First, we explain the overall algorithm. Then, we summarize how the algorithm is used with multiple images. Finally, we show the algorithm in a stand-alone system.

5.1 Algorithm

The image we use to replace a subset of a model will only be aligned with the remaining geometry when the eye is at the image-sampling viewpoint. For all other eye locations, we warp the vertices of surrounding geometry to match the geometry displayed by the image – the image itself does not change. The final rendering, although perspectively incorrect, is surprisingly pleasing. Moreover, we guarantee that the geometry near the viewpoint is unaffected. The error in the image is proportional to the distance between the current viewpoint and the image-sampling viewpoint. Figure 5-1 shows a sequence of frames with an image surrounded by warped geometry. For comparison, Figure 5-2 shows the same sequence but without warping the surrounding geometry.

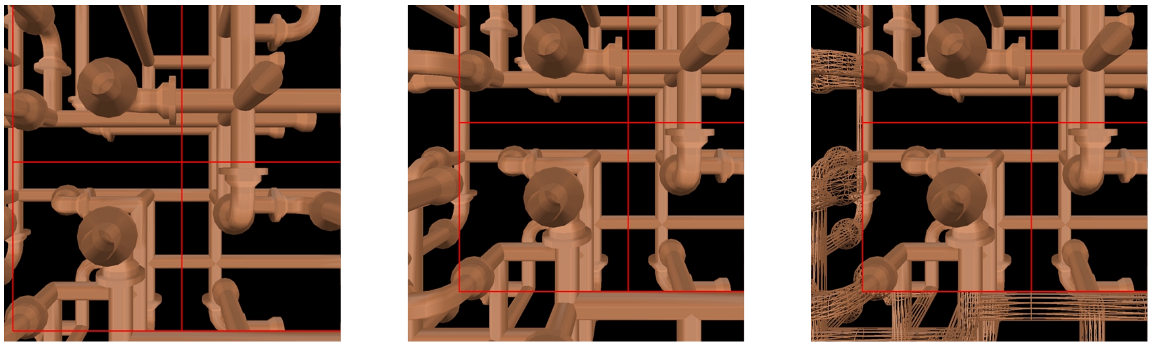

Figure 5-1. *Geometry-Warping Example. We show a sequence of frames with an image (outlined in red) surrounded by warped geometry. In the left snapshot, we are near the image-sampling viewpoint. In the middle snapshot, we have translated farther to the left yet no geometric discontinuities are visible. In the right snapshot, we render geometry in wireframe.*

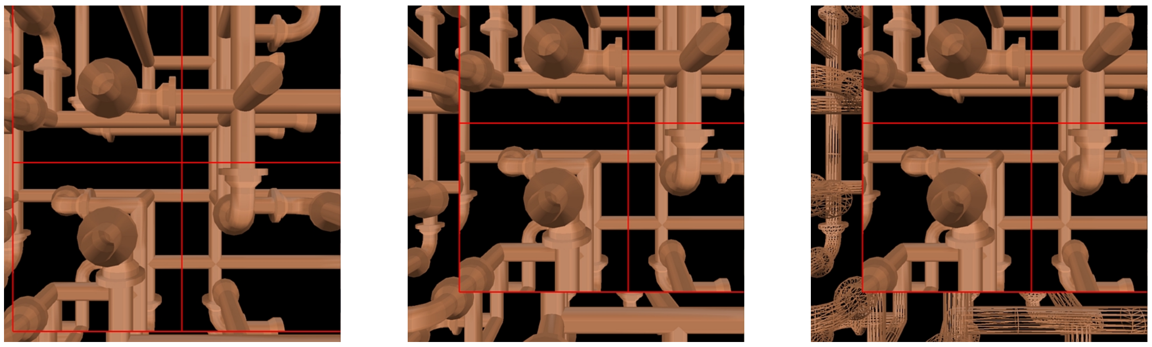

Figure 5-2. *No Geometry-Warping Example. We show the same sequence of viewpoints as before but geometry surrounding the image is rendered normally. In the middle and right snapshots, geometric discontinuities are easily seen at the geometry-image border.*

In addition to continuous borders between geometry and images, the algorithm can provide smooth transitions from image to geometry (and vice versa). For a smooth transition, we first replace an image with its corresponding geometry projected onto the image plane. Then, we morph the geometry, over several frames, back to its correct position (Figure 5-3).

The geometry-warping approach is attractive for four reasons: (a) texturing hardware is efficiently used, (b) images do not change every frame, (c) geometry warping is efficiently performed using the graphics hardware's transformation engine, (d) as the viewpoint changes, exposure artifacts are not introduced, as they may be with image warping. Although geometry warping could be considered less "realistic" than warping the image, it takes advantage of the fact that geometry is re-rendered every frame anyway, so by slightly modifying the geometry we are able to use static textures and achieve higher frame rates.

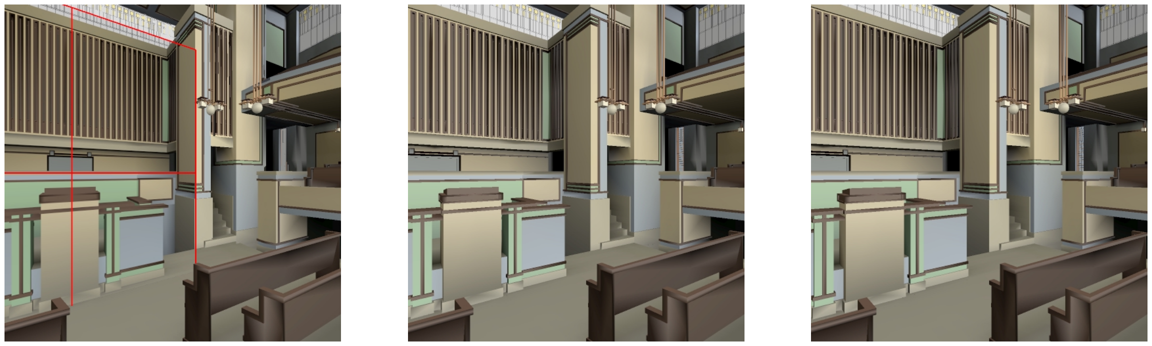

Figure 5-3. *Smooth-Transition Example. We show a sequence of frames that illustrates smoothly changing an image to geometry. In the left snapshot, the altar is an image (outlined in red). In the middle and right snapshots, we warp geometry, over several frames, from its projected positioned on the image to its correct position. The right snapshot is rendered as geometry.*

5.1.1 Partitioning the Geometry

The image quadrilateral partitions the model into 3 subsets of geometry*: near geometry* (unaffected by the warping operation), *image geometry* (geometry behind the image, which will be culled), and *surrounding geometry* (geometry surrounding all four edges of the quadrilateral). Figures 5-4 illustrates the model partitioning. The warp operation changes the surrounding geometry to maintain positional continuity (i.e. C0) with the geometry displayed by the image. The geometry between the image-sampling viewpoint and the image plane is rendered normally; thus, near geometry at the depth of the image will also maintain positional continuity with the image.

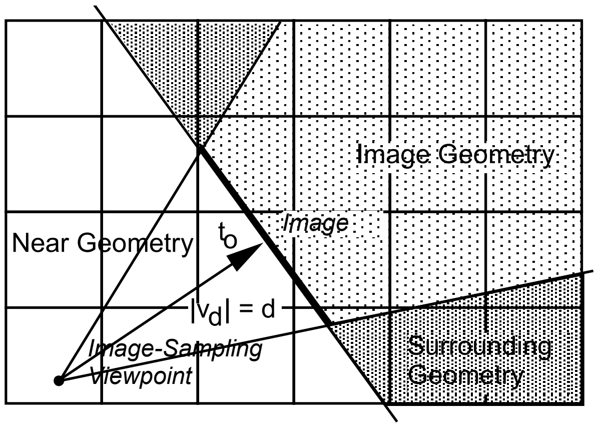

Figure 5-4. *Model Partitioning for Geometry Warping. Each box corresponds to a space-partitioning box. The boxes are classified: near, image, and surrounding. Intersected boxes can be optionally partitioned.*

5.1.2 Geometric Continuity

The geometry-warping algorithm applies a double-projection scheme to the vertices of the geometry surrounding an image to produce a continuous geometry-image border. First, we define a view frustum with the four vertices of the image quadrilateral and the image-sampling viewpoint. We denote this view frustum by [v_0-v_3, p_a] (Figure 5-5). Then, in a similar fashion, we represent the current view frustum with [v_0-v_3, p_b], where p_b is the current viewpoint (Figure 5-5). Next, despite having our eye at point p_b, our algorithm projects the surrounding geometry onto the image plane as if we were at p_a.

We accomplish this by using an inferred perspective transformation. Wolberg [Wolberg90] describes such a transformation. We adapt it to warp the projection plane, defined by the frustum [v_0-v_3, p_a], where p_a is the center-of-projection and v_0-v_3 are the four corner points of the image plane, to appear as if it were seen from the current viewpoint p_b. The image's view frustum can also be defined by a model-space transformation M_a and a projection P_a. Similarly, the current view frustum, [v_0-v_3, p_b], can also be expressed using a model-space transformation M_b and a projection P_b. We define the matrices of the final warped frustum to be $M_w = P_aM_a$ and $P_w = W_{ab}$, where W_{ab} is an inferred perspective warp from p_a to p_b.

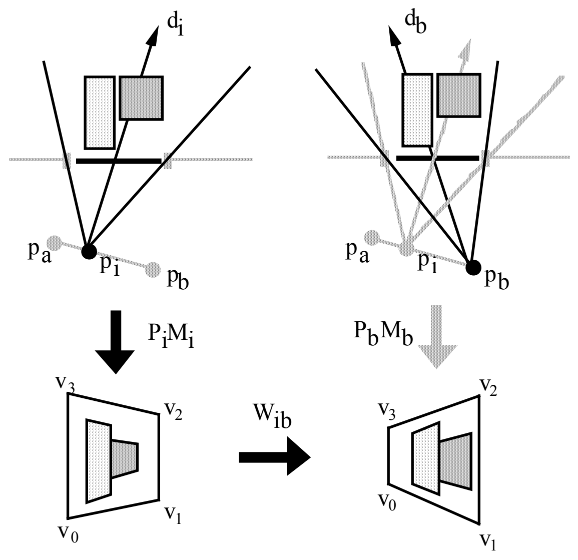

Figure 5-5. *Sequence of Transformations for Smooth Transitions. To create the intermediate projection from p_i, we project along d_i onto the image using P_iM_i. Then, we re-project from p_i to p_b using W_{ib}, where p_b is our current viewpoint.*

To construct the warp matrix W_{ab}, we multiply the vertices v_0-v_3 by P_aM_a and also by P_bM_b. The resulting eight projected *xy*-positions are used to construct a four corner mapping. This mapping creates a correspondence between the four corner points of the current view frustum and the four corner points of the image's view frustum. The warp matrix will interpolate intermediate points. In order to resolve occlusion properly, we must set up the matrix W_{ab} so that the final transformation matrix will produce *z*-values that correspond to the projection onto the original image plane [v_0-v_3, p_a]. In essence, we let the projected *z*-value pass through the warp unaffected. Placing the coefficients of the warp matrix into a 4x4 matrix as follows accomplishes this:

$$\begin{bmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ g & h & 0 & i \end{bmatrix}$$

5.1.3 Smooth Transitions

If we wish to change an image to geometry (or vice versa), we smoothly interpolate, over time, the image geometry from its projected position on the image plane to its correct position. We accomplish this by augmenting the warp operation to use intermediate view frustums. More precisely, we re-project the image geometry using the interpolated view frustum [v_0-v_3, p_i] where p_i is a point along the line segment p_a-p_b. Then, we apply an inferred perspective transformation to warp the projection plane, defined by frustum [v_0-v_3, p_i], to appear as if it were seen from the current viewpoint p_b. This sequence of transformations is illustrated in Figure 5-5.

5.1.4 Artifacts

Geometry warping distorts the geometry surrounding an image to match the rendering displayed by the image. The scene captured by the image is only correct from the image-sampling viewpoint. Consequently, the image and surrounding geometry have incorrect occlusion and perspective for all other viewpoints. Furthermore, near geometry that intersects the image or the surrounding geometry will only show positional continuity (i.e. C0). The amount of distortion introduced is proportional to the distance of the current eye position from the image-sampling viewpoint. In practice, we have observed that as long as this distance is kept relatively small the distortion is not particularly noticeable. Nevertheless, there are applications where even a small distortion might be undesirable–in this case, geometry warping is not appropriate.

5.2 Multiple Images

So far we have described the geometry-warping algorithm in the context of a single image. Using multiple images can simultaneously represent several subsets of a model. The following sections discuss the issues involved with having multiple images present and how it affects geometric continuity and smooth transitions. We divide the images into those with a common image-sampling viewpoint and those with different ones. Each of these categories is explored below.

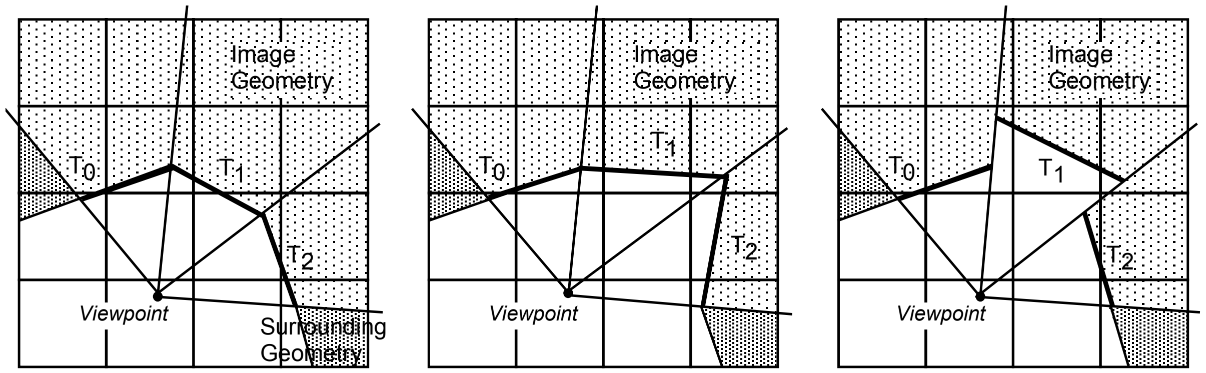

Figure 5-6. *Common Viewpoint, Adjacent Images. (Left) Images at equal view depth thus will maintain geometric continuity. (Middle) Varying view depth but same at the seams, geometric continuity maintained. (Right) Discontinuous view depth, geometric continuity not maintained. This case should be avoided.*

5.2.1 Common Viewpoint

Images with a common image-sampling viewpoint can still have different view directions and *view depths* (view depth is the distance between the eye and the image plane). Furthermore, since we have defined our images to be opaque (Section 2.1.3), we assume images do not overlap. With these criteria is mind, we can further subdivide this category into adjacent images and images with gaps between them.

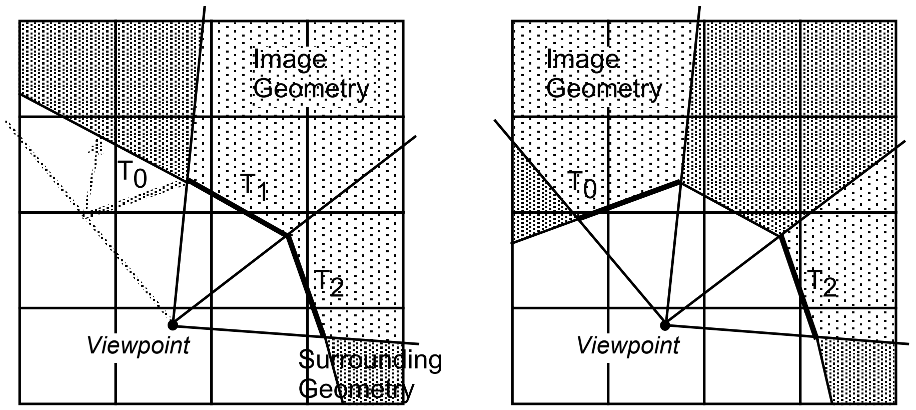

Figure 5-7*. Smooth Transitions. (Left) Removing an image at the edge of a string of adjacent images, interpolation required. (Right) Removing an image in the middle of a string of adjacent images, no interpolation.*

First, we address adjacent images (Figure 5-6). Adjacent images form a single large image with piecewise planar components. They can be used to replace a complex region of the model or even completely surround the viewpoint. Geometric continuity can easily be maintained (except for the third case in Figure 5-6, which should be avoided).

Maintaining smooth transitions, on the other hand, is slightly different than with a single image. If an image, at the edge of a string of adjacent images, is returned to geometry, the vertices must be interpolated (Figure 5-7, left) between the previous-edge image plane T_0 and the new-edge image plane T_1. If an image in the middle of a string of adjacent images is returned to geometry, the vertices need to be warped to match the projection of the image being removed--no interpolation occurs (Figure 5-7, right).

For images with a gap between them, a virtual image needs to be added to span the gap. The virtual image is actually geometry rendered using the double-projection scheme (Section 5.1.3) so that it appears as if an image were present. Now we can treat the original images as adjacent ones. It is worth noting that for two images (a "left" image and a "right" image) with a gap between them and very different view depth values, the geometry can be warped to match both images, but the distortion introduced might be very apparent from certain view directions. For example, assume the left image was defined at a significantly closer distance to the image-sampling viewpoint than the right image. Thus, viewing the left image from the left side might occlude some of the right image and all of the geometry in between both images.

5.2.2 Different Viewpoints

If multiple images are created from different image-sampling viewpoints with different view directions and view depths, the geometry surrounding each image must be warped before continuing on to create the next image. The first image is created just as in the previous section. But subsequent images, created from viewpoints and view directions that contain geometry in the plane of an existing image, will re-warp previously distorted geometry (Figure 5-8). Images created by using geometry that does not surround previous images will contain conventionally rendered geometry and cause no difficulty. Thus, it might be the case that the distortion introduced by warping operations will be magnified after several images are created from different viewpoints. At present, we have no metric to control this distortion. Fortunately, this is not the typical case since the number of images needed to surround the local view area is small. If the view area migrates to another portion of the model (by a series of transitions), a new set of images is used.

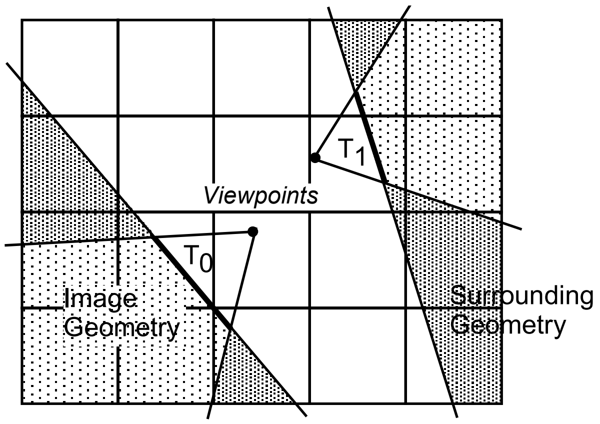

Figure 5-8. *Multiple Images. We illustrate the model partitioning that occurs when using two images. Each image is created from its own viewpoint and view depth. In this case, neither image contains previously warped geometry.*

The case where a subset of the model geometry, warped for a particular image, intersects with another subset of warped geometry is similar to the case of two images using a common viewpoint but different view depth values. Both subsets of geometry will have to be warped to match the images simultaneously. This results in a nonlinear warp operation that is difficult to (efficiently) implement. We avoid this problem by only using images rendered from a common image-sampling viewpoint.

5.3 Implementation

In this section, we describe a stand-alone system that implements the geometry-warping algorithm. The automatic image-placement issue is not addressed. Instead, the user selects distant (and visible) subsets of the model to replace with images. The image geometry is culled from the model and, consequently, significant increases in rendering performance are obtained. We use the geometry-warp operation to maintain geometric continuity at all times. Geometry near the viewpoint is rendered normally. The algorithm is written in C/C++, on a SGI Onyx with R4400 @ 250MHz processors and Reality Engine II graphics.

When we create an image, we assign it a unique ID and store the image data and image-sampling viewpoint in a host memory cache (similar to Section 4.4.1). The cache has a fixed size and uses a least-recently-used replacement policy to maintain images in memory. During the rendering of a frame, we use the ID to access all required images from the host memory cache. In order to perform texture mapping, image data must be in the graphics hardware's texture store.

Texture memory is typically much smaller than host memory. Our SGI Onyx has 16MB of texture store. Thus, for example, we can fit up to 256 images of 256x256 32-bit pixels. We have experimented with fewer bits per pixel but have found them to yield insufficient visual quality. Additionally, we have timed the host-to-texture-memory transfer rate and found it to be sufficient to copy over 30 images a second (~1 ms to copy a 32-bit 256x256 image from host to texture memory).

Images are used to represent complex subsets of the model from the current viewing area. After replacing a subset of the model with an image (*geometry-to-image transition*), the user cannot walk forward beyond the image plane without returning the subset to geometry. In order to return the image to geometry, a smooth transition from image back to geometry (*image-to-geometry transition*) is performed over the next few frames (Figure 5-9). The following two sections describe transitions in more detail.

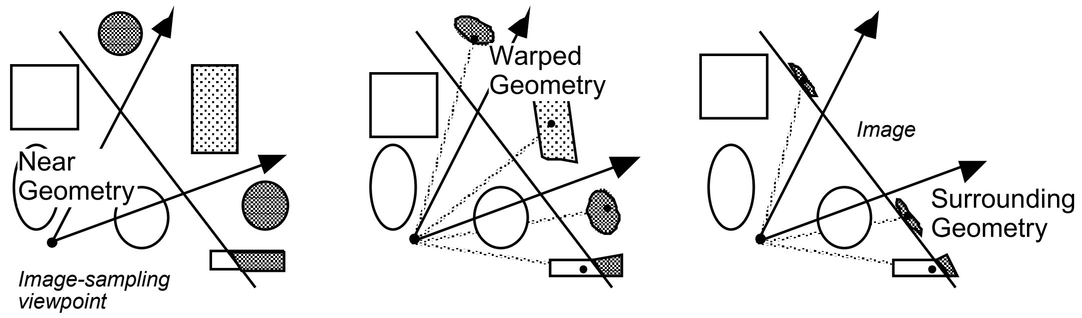

Figure 5-9. *Geometry-To-Image Transitions. A geometry-to-image transition goes from left to right. At the end of the transition, the image is introduced. An image-to-geometry transition goes from right to left. From the image-sampling viewpoint, the objects look the same at all times.*

5.3.1 Geometry-To-Image Transition

First, the user selects a subset of the model to be replaced with an image. This can be done in various ways. We adopt the following simple strategy: select all geometry inside the view frustum and beyond a distance *d* from the viewpoint. The image plane is defined as the plane whose normal is the current view direction v_d and which contains the point t_o, at a distance *d* from the viewpoint along the view direction (Figure 5-4). The subset of the model behind the image will be the image geometry.

Then, we push the near clipping plane back to coincide with the image plane and render the image geometry. We copy the rendered image from the frame buffer to texture memory. A texture-mapped quadrilateral covering the subset of the model being replaced is added to the model. The image geometry is removed from the set of rendered geometry.

The images can be precomputed or dynamically computed. If they are computed on demand and the eye location and image-sampling viewpoint coincide, there is no need to perform the first geometry-to-image transition. On the other hand, if images have been precomputed, the eye will most likely not be at the image-sampling viewpoint, thus geometry-to-image transitions are needed. In this case, we employ the geometry warp operation over several frames (e.g. five) to change the geometry behind the image plane (that has not been culled) and the surrounding geometry to match the image. The intermediate view frustums are created using viewpoints along the line between the current viewpoint and the image-sampling viewpoint. At the end of the transition, the image is displayed instead of the warped geometry.

In all cases, the geometry in front of the image is rendered normally.

5.3.2 Image-to-Geometry Transition

Initially, the image geometry is reintroduced into the model but the vertices are set to their projected position on the image plane. The image geometry and surrounding geometry are warped from their projected position on the image plane to their correct position over several frames. If the image plane is currently not in the view frustum, an instantaneous transition can be performed.

Once an image has been computed, it might undergo various geometry-to-image and image-to-geometry transitions. As mentioned before, all subsequent transitions (after the first geometry-to-image transition) will generally be from viewpoints other than the image-sampling viewpoint. Thus, the surrounding geometry is gradually warped from its correct

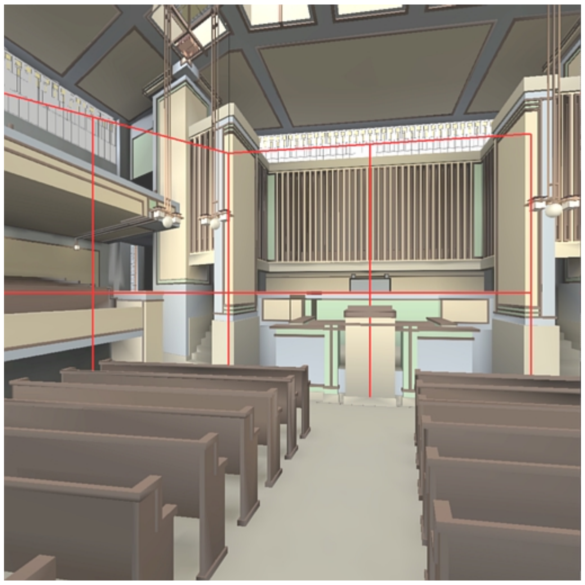

Figure 5-10. *Images and Warped Geometry in the Church Model. We show two images (outlined in red) surrounded by warped geometry. This frame is rendered from an eye location far behind the image-sampling viewpoint. The geometric distortion, particularly in the ceiling, is very noticeable. Nevertheless, there are no discontinuities at the geometry-image border.*

position to its projected position on the image plane. In any case, since view frustum culling is used, only the visible geometry is actually warped.

5.3.3 Results

We use three models to show this system: a procedurally generated pipes model (205,000 triangles, similar to the one used in Chapter 3, courtesy of Lee Westover), a radiosity-illuminated model of Frank Lloyd Wright's famous Unity Church in Oak Park, Illinois (158,000 triangles, courtesy of Lightscape Technologies Inc.) and the submarine auxiliary machine room model (525,000 triangles, courtesy of Electric Boat Division of General Dynamics).

For each model, the user predetermined the location of several images. We recorded various paths through the models (spline-interpolated paths and captured user-motion paths). Figure 5-10 shows a frame from a path through the church model. Two images have replaced selected geometry. No discontinuities are perceivable at the geometry-image border.

A typical (single-pipe) graphics system can be fed by at most one process at a time. Since our system is usually render-bound and we use the graphics system to warp geometry, there is little to gain from multiprocessing.

Table 5-1 summarizes the speedups we obtained. At any time, the images can be smoothly changed back to geometry (with the corresponding decreases in performance). We computed speedups based on average frame rates in different parts of the model.

Model	Average Speedup	Comments
Pipes	9.2	3 images in path
Church	3.3	3 images in path
AMR	6.5	2 images in path

Table 5-1. *Performance Summary of Stand-Alone Geometry-Warping System.*

6. Architectural Models

We have described a preprocessing algorithm and two approaches to displaying images applicable to arbitrary 3D models. In this chapter, we detail a specialized solution for architectural models. The system uses either geometry or image warping and we discuss the advantages and disadvantages of each.

6.1 Image Placement in Architectural Models

The image-placement problem for arbitrary 3D models is difficult. However, for architectural models, we take advantage of their inherent structure to define a set of image locations. Walls that occlude everything on the other side naturally partition architectural spaces. Adjacent areas are only visible through certain openings (doors, windows, etc.). Past research has focused on dividing a model into cells (rooms or predetermined subsections of the model) and portals (doors, windows, and other openings). Visibility culling algorithms are used to determine which cells are visible from a particular viewpoint, and view direction. Rendering is limited to the geometry of the visible cells. Exact pre-processing algorithms [Airey90, Teller91, Teller92] as well as conservative run-time algorithms [Luebke95] have been developed.

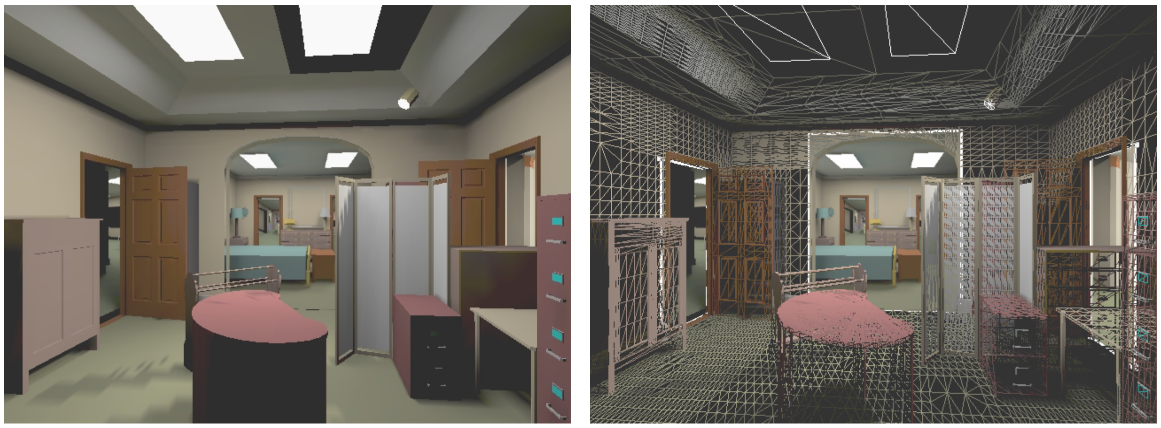

Figure 6-1. *Portal Images. Both of these snapshots are from within the Brooks House model. The rooms visible through the three doorways are images. In the right snapshot, we have rendered geometry in wireframe and outlined the images in white.*

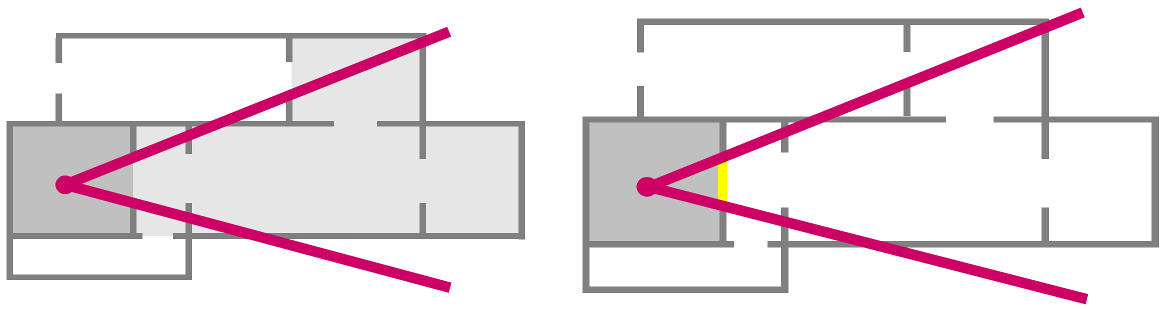

Figure 6-2. Portal-Image Culling. By conditionally replacing the cells visible through a portal with an image, we only need to render the geometry of the room containing the eye. The pair of lines represent a view frustum positioned inside the view cell (dark gray). The left floor plan depicts the cells that are marked visible (light gray) by conventional portal culling. The right floor plan shows the reduced set of cells to render when a portal image (thick, vertical line at the opening of the view cell) is present.

We reduce the image-placement problem to one of conditionally replacing the cells visible through a portal with an image (Figure 6-1). Consequently, the system must only render the geometry of the cell containing the viewpoint, and a few image quadrilaterals. We cannot guarantee a specific number of primitives per frame but in practice are able to significantly reduce the rendering requirements for architectural models. Furthermore, this approach alleviates the sudden decreases in performance when a complex cluster of cells becomes visible. If the viewpoint approaches a portal, the *portal image* will return to geometry, allowing the viewpoint to move into the adjacent cell.

6.2 Portal Images

This section first reviews conventional portal culling, and then details how portal images more aggressively cull an architectural model. Then, we describe general strategies for creating portal images, and detail our approach.

6.2.1 Portal Culling vs. Portal-Image Culling

We can partition a model into cells by identifying the location of walls, and other opaque surfaces [Airey90, Teller91, Teller92]. Each cell contains a list of portals, each of which defines an opening through which an adjacent cell may be seen. The left floor plan in Figure 6-2 shows a cell-partitioned model. The viewpoint is inside the *view cell*. The view frustum only intersects a subset of the portals of the view cell, thus cells attached to each visible portal are recursively traversed to compute all of the visible cells.

Since the model contains the location of all portals, we can compute images to be placed at the location of the otherwise transparent portal openings. At run time, we render the view cell normally. All visible portals of the view cell are rendered as images, and no adjacent cells are actually rendered, despite being visible. The right floor plan in Figure 6-2 illustrates the reduced set of cells to render. As the viewpoint approaches a portal, we switch to rendering the geometry of the cell behind the portal. Once the viewpoint enters the adjacent cell, it becomes the view cell, and the previous cell will now be rendered as a portal image.

6.2.2 Creating Portal Images

A portal can be viewed from multiple view directions as well as from multiple viewpoints. Since a single image only produces a perspectively correct image from its image-sampling viewpoint, we need to do some additional work to control temporal continuity. We divide the overall strategies for choosing image-sampling viewpoints to create portal images into three categories:

- *Model-independent viewpoints*: these define a regularly spaced set of viewpoints spanning the space on the front side of a portal without regard to particular model characteristics.
- *Model-dependent viewpoints*: these define a subset of viewpoints that do not necessarily span the entire front side of a portal. This approach requires some knowledge about model characteristics, such as the typical portal viewing directions. For example, consider a hallway with a portal on the side. The portal will typically only be viewed from acute angles. By the time the viewpoint is in front of the portal, the portal will be rendered using geometry.
- *Single viewpoint*: a single viewpoint, usually facing the portal. This method is economical, and works well when the possible or likely view directions to the portal are restricted (e.g. a narrow hall with a portal at the end).

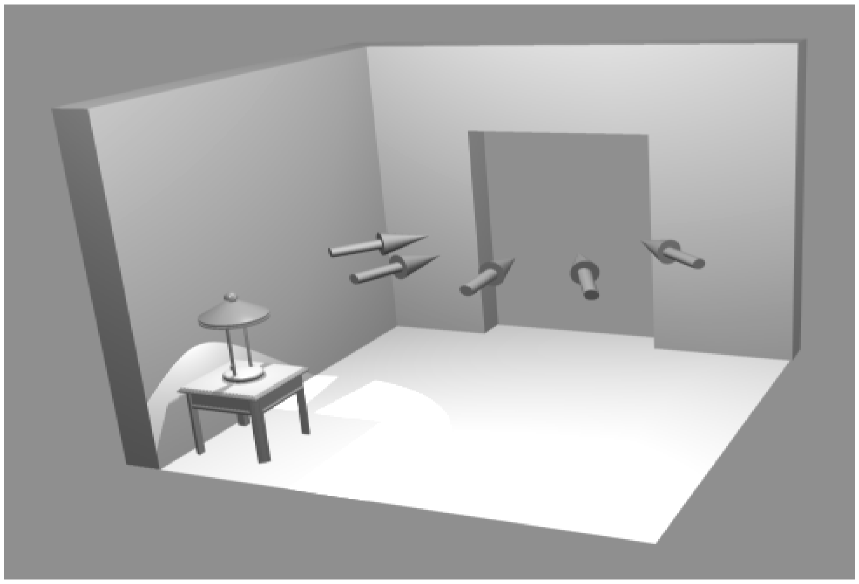

Figure 6-3. *Constrained Model-Dependent Viewpoints. We evenly space the image-sampling viewpoints for a portal image along a semicircle placed in front of the portal at a typical eye height.*

In general, model-dependent viewpoints produce better results than model-independent viewpoints because they take advantage of the user's domain knowledge of the model to reduce the number of images. Moreover, when visualizing architectural models, we typically walk at about the same height (although we perhaps change floors).

Our systems take advantage of this fact, and uses *constrained model-dependent viewpoints* to reduce the number of necessary images. We assume that our head movement is typically left-right, and forward-backward. We may also gaze up or down at any time. For each portal, the modeler can define a set of image-sampling viewpoints constrained to lie on a semicircle of some radius on the front-side of the portal. We also assume that portals are perpendicular to the “floor” of the model, and thus fix the semicircle to lie at some typical eye height for each portal (Figure 6-3). At run time, we select the image that most closely represents the view of the geometry behind the portal from the current eye location.

For each portal of the model, we need to define (or assume reasonable default) values for the following parameters:

(a) *Viewing Height*: the typical viewing height of the portal.

(b) *Sampling Distance*: the radius of the constraining semicircle from the portal.

(c) *Transition Distance*: the distance from the portal at which to perform a portal-image to geometry transition (or vice versa).

(d) *Viewing Angles*: the set of points on the semicircle to use as image-sampling viewpoints. We have found that regularly spacing the viewpoints along a semicircle yields good results.

We create the portal images either as a preprocess or at run time (Section 4.4.1). Creating portal images on demand works quite well for the common application of architectural walkthroughs. In practice, users do not fly quickly through the model. They usually go to a room, and examine an area in detail before proceeding to another portion of the model. Demand rendering of images will result in slower performance when the user first enters a cell. However, as the user works in an area, that area will "sweeten", and performance will increase. An alternative is anticipate the viewer’s movements, and create additional portal images using idle time or separate processors.

6.3 Geometry-Warping System

The first of our two architectural walkthrough systems uses geometry warping to display images at run time. We present an overview of the system, followed by implementation details. Then, we present performance results, and observations.

6.3.1 Overview

Our geometry-warping system replaces geometry behind portals with portal images created from a constrained set of model-dependent viewpoints. Warping the geometry from its projected position (on the portal image) to the projection for the current viewpoint provides smooth transitions. Geometric continuity is not a problem because the (opaque) wall surrounding a portal prevents us from seeing the geometry-image border. On the other hand, temporal discontinuities (i.e. popping) may occur as we switch to the best image for each portal. One way to partially address the temporal continuity issue is to increase the number of images per portal, thus reducing the extent of the popping. We have experimented with blending portal images but have not found the results to be visually pleasing.

Geometry warping is also useful when on a tight memory budget. The single portal-image case, when only one image is used to represent the portal from all directions, has very low memory cost but may result in very noticeable transitions from image to geometry (or vice versa). This is one of the worst examples of popping. We eliminate this abrupt transition completely by smoothly warping the geometry represented by the single portal-image.

6.3.2 Implementation

We implemented this system on a SGI Onyx2, R10000 @ 195MHz processors with Infinite Reality graphics. The system is coded in C++, uses the OpenGL graphics library, and employs a user-configurable amount of host memory and texture memory. For simplicity, all images are 256x256 pixels in size and 8 bits per color component.

The contents of each cell are maintained as a collection of geometric primitives organized in an octree. When a cell is flagged as visible, its contents are culled to the current frustum, and rendered. Portals are culled to the current frustum using their screen-space bounding rectangle. The overall visibility algorithm is summarized in Figure 6-4.

```
Visibility(cell, frustum) {
      Mark cell visible
      Cull cell to frustum
      Foreach portal {
            Cull portal to frustum
            if (portal is visible) {
                  if (portal in transition)
                        Initialize transition
                  else if (portal is an image)
                        Choose best image
                  else if (portal is geometry)
                        Visibility(portal's adjacent cell,
                                   culled frustum)
            }

            if (portal in transition) {
                  Next transition step
                  if (portal-to-image finished)
                        Choose best image
                  if (image-to-portal finished)
                        Visibility(portal's adjacent cell,
                                   culled frustum)
            }
      }
}
```

Figure 6-4. *Portal-Image Visibility Culling using Geometry Warping.*

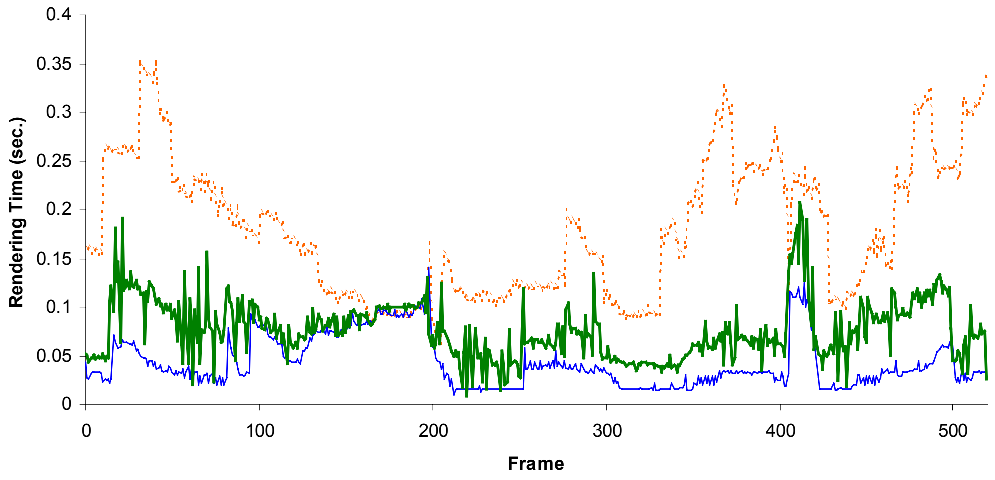

Figure 6-5. *Portal Images Rendering Times. Rendering times of a path through Brooks House showing portal culling (upper line), LDI warping (middle line), described in Section 6.4, and portal images (lower line). The spikes are caused by increases in the amount of rendered geometry when a neighboring cell becomes visible and is sufficiently close to be rendered as geometry. In the case of LDI warping, multiple portals are sometimes visible and increase the overall warp time.*

6.3.3 Results and Observations

6.3.3.1 Performance

We tested our system using two architectural models. The first model, Brooks House (Section 1.2), has 1.7 million triangles, 19 cells, and 52 portals. The second model, the *Haunted House* (courtesy David Luebke and Michael Goslin), is of a two-story house and consists of 214,000 polygons with 7 cells and 12 portals.

We present the rendering times for a path through the Brooks House model. Figure 6-5 shows the performance along the path as compared to view-frustum culling. To obtain high visual fidelity, we create portal images for every degree, over viewing directions ranging from 30 to 120 degrees in front of the portals, and precompute the images (the next section has some observations about the cost of portal image rendering).

Notice how the large variations in rendering performance have been significantly reduced. This is due primarily to the fact that an image can be rendered in time independent of the complexity of the geometry it represents.

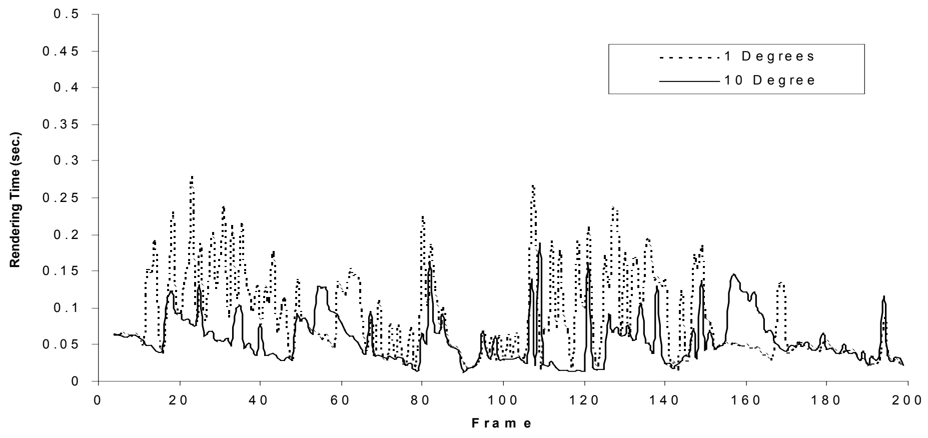

Figure 6-6. *Cold-Cache Rendering Times. Cold-cache rendering times for the Brooks House model using one portal texture sample for every 10 degrees vs. one per degree.*

6.3.3.2 Portal Images on Demand

What if we decide to compute portal images on demand? Figure 6-6 shows frame times (on a SGI Onyx with R4400 @ 250MHz processors and Reality Engine II graphics) with images always computed on demand (equivalent to starting with an empty cache) and sampling ranges of one and ten degrees per portal image. We use the Brooks House model and a similar path to before. Although the performance is still quite good, we have lost the steadier frame rate achieved with precomputed portal-images. An alternative might be to predict the viewer's position and distributed portal-image rendering over several frames.

We can sometimes peak above traditional portal culling because some scenes in the model force us to render images for several rooms in one frame (a room visible in a doorway that was visible in another doorway, etc.). However, on average our performance does not exceed that of traditional portal culling, plus some extra overhead.

6.3.3.3 Single Portal-Image Case

The use of a single portal-image per portal gives us the best and steadiest performance, albeit with lowest image fidelity. However, we have observed that with warping at transitions, the user feels very comfortable interacting with the model. Since it is trivial to precompute all of the portal images when loading a model, variations in frame rate are small. Thus, this is the best choice for a tight memory budget.

```
Visibility(cell, frustum) {
   Mark cell visible
   Cull cell to frustum
   Foreach portal {
      Cull portal to frustum
      if (portal is visible) {
         if (portal is image) {
            Choose best reference image(s)
            Warp reference image(s)
         } else
            Visibility(portal's adjacent cell,
                       culled frustum)
      }
   }
}
```

Figure 6-7*. Portal-Image Visibility Culling using Image Warping.*

6.3.3.4 Image Quality

Increased performance is achieved at the expense of some image quality. In general, a greater number of images, together with properly configured portal parameters and the use of geometry warping will improve the image quality. One image per degree gives excellent quality but may be too expensive for some applications.

The amount of texture memory on the graphics accelerator has not proven to be a problem since we only need one to three images per frame. Moreover, the cost of copying images to texture memory is quite low (Section 5.3).

6.4 Image-Warping System

Our second architectural walkthrough system uses image warping. First, we describe how the system differs from the previous one. Next, we detail our multiprocessor implementation and present performance results and observations.

6.4.1 Overview

Our image-warping system also replaces portals with images, but we warp the images to the current viewpoint every frame. Thus, we are able to greatly reduce the number of images required per portal, at the cost of an approximately constant overhead. In addition, portal-image warping provides temporal continuity and smooth transitions (as in the previous system, geometric continuity is not an issue). Unfortunately, portal-image warping introduces

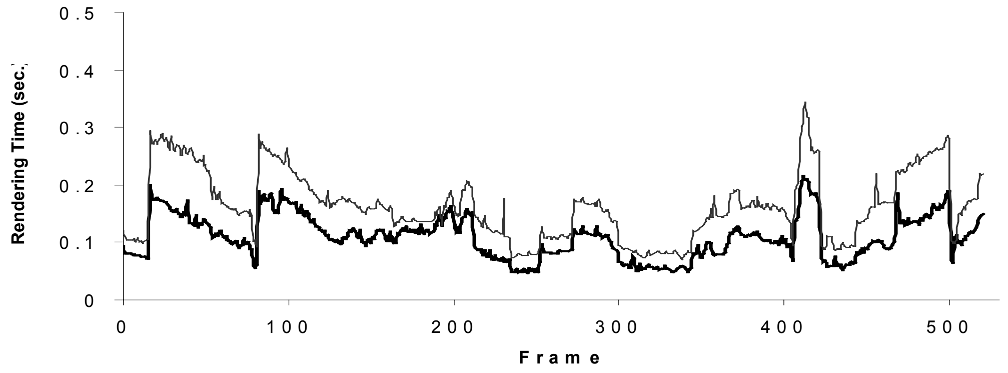

Figure 6-8. *Rendering times (of a similar path through Brooks House) warping one and two reference images. Note that warping a single image takes roughly the same amount of time as LDI warping.*

exposure artifacts. We address the artifacts by warping two or more reference images per portal (Section 4.2.3) or by warping layered depth images (Section 4.3).

6.4.2 Implementation

We implemented our system on a SGI Onyx2, 4 R10000's @ 195Mhz processors with Infinite Reality graphics. The system is coded in C++ and uses the OpenGL graphics library.

As in the geometry-warping system, our reference portal-images are sampled along a semicircle in front of each portal located at the typical viewing height. The semicircle does not need to cover the full halfspace in front of a portal, but only the span from which the portal will be seen. In our examples, we typically generate reference images every 10 degrees in front of the portal over an angular range of 60 or 120 degrees (for a total of 7 or 13 images per portal). At run time, our visibility algorithm determines which portals are visible. Then, we make sure the reference images are created and warp them (Figure 6-7). We chose a nominal image size of 256x256 pixels.

6.4.3 Results and Observations

We tested our system with the same two architectural models: Brooks House and Haunted House. For this application, we typically look at the portal when approaching it and keep it in the center of the FOV. Consequently, the COP of the desired image almost always projects onto the reference image, producing four roughly equal-sized sheets (Chapter 4). We take advantage of this to parallelize single-image warping. As discussed in Section 4.3, our LDI implementation already takes advantage of the available processors.

Figure 6-5 shows the rendering time using layered depth images for the portals. Figure 6-8 shows the performance warping one and two reference images. Recall that an advantage to the use of image warping is that warping time is independent of the amount of geometry beyond the portal. The rendering load is reduced to that of the current cell; that and the size of the images to be warped determine the total rendering time. Thus, we should see higher speedups for more complex models.

Since we require few reference images per portal, we precompute the images and store them in host memory. Thus, the total number of reference images stored per portal does not affect performance. The results in Figures 6-5 and 6-8 use precomputed images. We have experimented with dynamically computing images. We obtain the same speedups except for spikes whenever an image is created. We feel that the amount of storage needed to store the portal images is not unreasonable (for example, in the Brooks House model there are 52 portals for up to a total of 676 images or 127 MB).

7. Efficient Hierarchical Culling

This chapter describes a hierarchical culling algorithm to provide tighter culling of geometry by images. First, we provide an overview of conventional hierarchical culling. Second, we detail our improved culling algorithm. At the end of the chapter, we compare culling results on a test model.

7.1 Hierarchical Culling

In order to perform geometry warping or even simple image culling, we must perform more culling operations as compared to view-frustum culling alone. The geometry in the view frustum is divided into a set of sub view frusta. Geometry in each sub frustum is either warped or removed from rendering (Figure 7-1). If we use conventional hierarchical culling for this configuration, primitives that span more than one sub view frustum are rendered multiple times. In particular, some of these primitives are actually not visible but we lack the

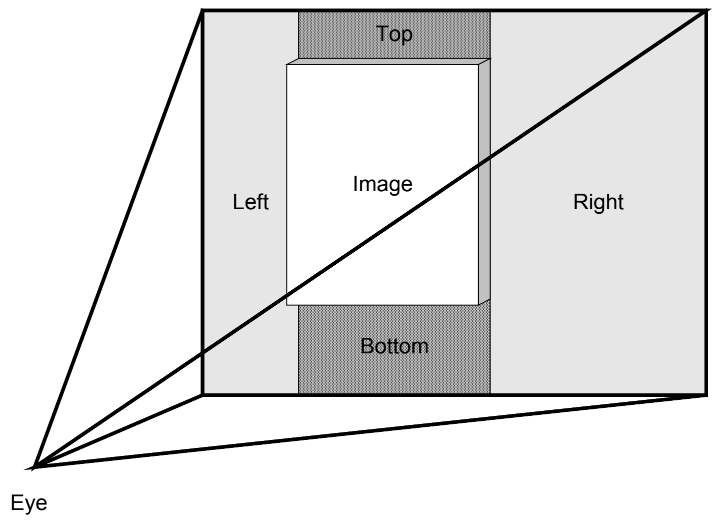

***Figure 7-1**. Example Tiling of Screen Geometry. This figure illustrates how an image inserted into the scene can tile the surrounding geometry. Particularly, in the case of geometry warping (Chapter 5), the left and right regions are culled to subview frusta that are warped to match the image. A different pair of subview frusta and warp operations is used for the top and bottom regions. The geometry behind the image is always culled.*

additional visibility information to know this. We have found the extra overhead to become significant when several sub view frusta are used. Therefore, we present a culling algorithm to eliminate this overhead. We implemented this algorithm using an *octree* -- but the same idea can be used in other hierarchical spatial partitioning algorithms (e.g. k-D tree, BSP-tree, etc).

7.1.1 Conventional Hierarchical Culling

Since the cost of culling individual primitives is prohibitive, we store sets of primitives in an octree. This tree structure is constructed by recursively dividing the model space into axis-aligned boxes and sets of geometry. The recursion continues until a minimum box size or primitive count is reached. The root node contains the bounding box of the entire model. We can store the primitives either in the intermediate cells or only in the leaf cells. The former requires multiple references to a single primitive but the latter requires us to always traverse the tree down to the leaf cells. To get the maximum performance, we assume primitives are stored in the intermediate cells. We classify the primitives stored in the tree into two categories:

- *Contained-primitives* are those totally contained inside an octree cell, and
- *Cross-primitives* are those that intersect two or more cells at the same level.

Cross-primitives must be handled slightly differently than contained primitives. We can either split them or explicitly store them in the parent of the cells they intersect. Splitting cross-primitives increases the overall number of primitives (in the worst-case, doubling the primitive count). In practice, we have observed that a 30% increase in primitives is not unusual. On the other hand, storing cross-primitives in intermediate cells does not increase the model complexity but makes the tree traversal slightly more complicated. At run time, we must mark for rendering all cross-primitives sets encountered during tree traversal, in addition to visible contained-primitives sets.

If we cull the model to one viewing frustum, the conventional tree structure works well. The tree is only traversed once from top to bottom. Thus, all contained- and cross-primitives sets will be visited (and rendered) once.

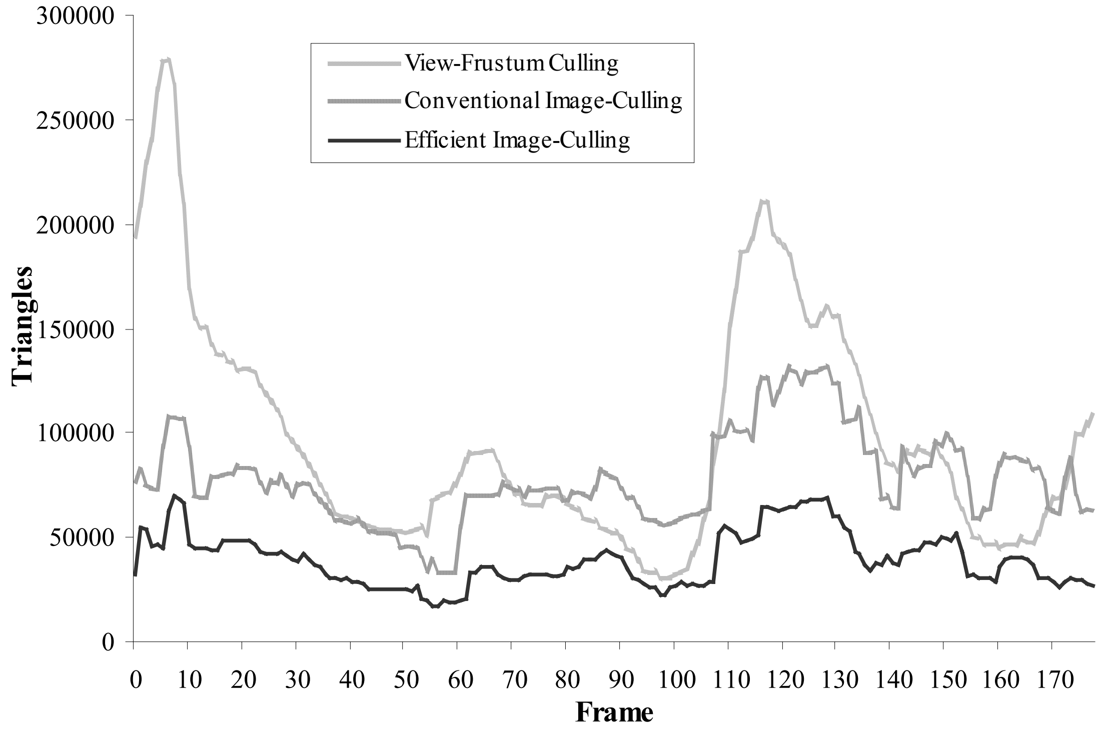

Figure 7-2. *Culling Comparison. This graph shows the number of triangles rendered for a path through the auxiliary machine room model. The top (light gray) line is the triangle count using only view-frustum culling. The middle (medium gray) line is the count with one image present that divides the visible geometry into four tiles and culls another tile of geometry using conventional hierarchical culling. The bottom (black) line is the count using the same image but the efficient hierarchical culling algorithm described in this section.*

If we cull to multiple viewing frusta, the tree does not perform as well. We will unnecessarily render some cross-primitives multiple times. Contained-primitives are, by definition, disjoint. We can simply use a visited flag to prevent rendering them twice. For example, assume we cull to two sub view frusta. During the traversal for the first view frustum, we encounter a cross-primitives set. We mark it for rendering. But, one of the primitives it contains is not actually visible. It had intersected two octree cells at an intermediate level of the tree and thus was stored as a cross-primitive. One of those octree cells is now partially visible—the other is not, but because we do not know exactly where the primitive is, we must render it. During the traversal for the second view frustum, a similar situation arises and we mark the same cross-primitives set for rendering. Thus, some primitives that are not actually visible are rendered multiple times.

We show the number of primitives rendered for a path through the auxiliary machine room model (Figure 7-2). The top line represents the number of primitives rendered after applying view-frustum culling. The middle line shows the primitive count when we replace complex visible geometry with one or two images and use conventional culling. In the case of geometry warping, the geometry surrounding the images is divided into at least five tiles (Figure 7-1), which we cull, warp and render separately. For image warping, the geometry behind the image is culled. In both cases, cross-primitives not contained in the view frustum are unnecessarily rendered, perhaps multiple times.

7.1.2 Efficient Hierarchical Culling

In order to reduce the unnecessary rendering of cross-primitives, we construct the tree slightly differently. We store, as the cross-primitives set, geometry that is partially contained in a cell's bounding box and at least one of a cell's adjacent *siblings* (as opposed to storing primitives that intersect any two or more *children* cells). Furthermore, when constructing the tree, we treat cross-primitives as additional contained-primitives. Consequently, a single cross-primitive might propagate down the tree and be referenced by cells at successive levels.

During a culling traversal, we mark for rendering contained-primitives sets and their corresponding cross-primitives sets. As opposed to conventional hierarchical culling, we do not need mark for rendering all cross-primitives sets encountered during tree traversal. Moreover, since we only render cross-primitives associated with visible cells, we greatly reduce the number of non-visible primitives rendered.

We can improve the culling even further by marking exactly which cross-primitives have been rendered. Consider selecting two adjacent sibling cells for rendering. Their cross-primitives sets will share some geometry (i.e. the primitives intersecting the common boundary of the two cells). At the beginning of each frame, we increment a frame count. During rendering, we check a frame count array with an entry for each cross-primitive selected for rendering. If the frame count is already that of the current frame, we do not render the primitive again. Otherwise, we set the entry to the current frame count and render the primitive. Since we have a relatively small number of unique cross-primitives per frame, the overhead is minimal.

Our new culling algorithm increases storage by a constant factor. If there are *c* cross-primitives in the spatial partitioning, a conventional octree uses *O(c)* to store them. Our modified octree uses up to *O(cd)*, where *d* is the maximum depth of the octree.

We can reduce the additional overhead by limiting how far a cross-primitive is propagated down the tree. When the limit is reached, we store the primitives in a *stopped-primitives* set at the current cell. The additional storage now becomes *O(cs)* where *s* is the number of tree levels allowed before stopping propagation. During a culling traversal, all stopped-primitives sets must be selected for rendering. Hence, more non-visible primitives will be rendered. Since, in our systems, the additional storage for full cross-primitive propagation is relatively small, we do not use stopped-primitives sets.

We compare the improved culling results to conventional culling in Figure 7-2. We employ our modified cross-primitives algorithm and mark rendered cross-primitives to prevent drawing them twice. For this path, we now render only 60% of the primitives as compared to conventional culling. The run time cost of the extra culling is insignificant (even more so when we pipeline culling and rendering using two processors). The improvement is dependent on the number of extra culling traversals and on the model itself, but we found it to be well worth the extra work for the scenarios presented in Section 1.2.

8. Conclusions and Future Work

8.1 Summary

We introduced a preprocessing and run-time algorithm 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 predetermined frame rate. We identified four major issues that must be addressed by any similar system: automatic image-placement, temporal continuity, geometric continuity, and smooth transitions. First, we detailed our preprocessing component to automatically compute which subsets of an arbitrary 3D model to replace with images in order to meet a geometry budget. Subsequently, we presented two approaches to displaying images: (a) an optimized implementation of layered-depth-image warping, and (b) a geometry-warping scheme to provide geometric continuity and smooth transitions by slightly changing the geometry surrounding an image. We then applied our algorithms to several complex 3D models.

Our preprocessing algorithm trades off storage for rendering performance and reduces the 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). At present, we only guarantee a rendering performance for translation and yaw rotation. Furthermore, we have seen preprocessing times of up to 28 hours. We have empirically determined the set of constants and weights required by our algorithm. They have worked well for our test models, but further automation or elimination of these constants would facilitate the processing of new models.

We demonstrated an optimized layered-depth-image approach that provides nearly artifact-free image warping. We can reduce the (large) storage requirement by more sophisticated image representations and by image compression methods. Currently, we cannot perform view-dependent shading with the images at reasonable frame rates; thus, we use precomputed diffuse illumination. Our software warper runs at interactive rates on a 4-node multiprocessor system but introduces one frame of latency. We used near NTSC resolution images (512x384). Higher resolution images (for multisample antialiasing) 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.

We described a heuristical method for selecting the reference images to construct our LDIs. Some surfaces are still not sampled and can cause exposure artifacts. Furthermore, for performance reasons we choose fixed-size splats. This simple reconstruction kernel can work well if the reference-image resolution is at least that of the warped image in the frame buffer. In general, we need more sophisticated reconstruction methods to further improve the warp quality.

We introduced a geometry-warping algorithm for displaying images that does not generate significant overhead on today's graphics hardware. The algorithm provides smooth transitions and removes geometric discontinuities at the geometry-image border but introduces a distortion proportional to the distance of the eye from the original image-sampling viewpoint. This scheme is particularly useful for systems on a tight image-storage budget or for systems that do not require temporal continuity. We demonstrated the algorithm in a stand-alone system and in a system designed for architectural walkthroughs. This architectural walkthrough system takes advantage of the inherent structure of buildings to specialize our algorithms to architectural models by building upon cells-and-portals methods. We obtained significant speedup over conventional portal culling. We have also employed single-image warping and layered-depth-image warping to display portal images.

Finally, we presented an effective image-caching paradigm and an efficient hierarchical culling algorithm. We still need to do further investigation of prefetching algorithms for the image data and for 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.

Thus, if we can afford the cost of storing and displaying images, we have successfully proven our thesis statement:

> *"We can accelerate the interactive rendering of complex 3D models by replacing subsets of the geometry with images. Furthermore, we can guarantee a specified level of performance by bounding the amount of geometry that must be rendered each frame."*

8.2 Extensions to Automatic Image-Placement

Our algorithm creates and warps at most one image at a time. Thus, if a complex part of the model is visible from a particular grid viewpoint, the preprocessing algorithm might need to generate separate images for several nearly adjacent view directions. These images will contain overlapping subsets of the model. We could exploit the overlap to perform image compression or to warp multiple non-overlapping images per frame. To maintain a constant warping cost, we would have to distribute the pixel budget among the images of a single frame. With either of these modifications, it might be possible to reduce the overall storage requirement. Moreover, a tighter fit of images over the far geometry, accomplished by the several smaller images per frame, might help to increase the average distance between images and image-sampling viewpoints – thus reducing potential warping artifacts.

In our current implementation, we only guarantee a maximum number of primitives to render for a horizontal gaze. We could extend the preprocessing algorithm to take pitch into account. It is unclear whether the extra complexity (in both implementation and storage) this adds is worth the effort. We observe that with interactive walkthroughs, the gaze is often kept nearly horizontal. Viewers do glance up and down at times, so the simplification we have already performed might be sufficient.

In addition, we plan to refine our run-time system to make sure that we actually maintain a constant frame rate (as opposed to a minimum frame rate). This includes dividing the culling and warping tasks across different processors in order to eliminate the cost of culling.

8.3 Extensions to Image Warping

For either single-image warping or layered-depth-image warping, we need to create reference images. An avenue of future work is to find the optimal set of reference images. The meaning of optimal is not well defined. For example, we could try to produce the smallest set of images for some approximate quality; or, we could try to produce reference images that sample all potentially visible surfaces. Furthermore, we can hopefully improve our reconstruction techniques by using more sophisticated splats.

A known problem in image warping (and image processing in general) is defining a visual quality metric. In geometric levels of detail, the screen-space deviation of the simplified surface from the original surface is generally regarded as an acceptable metric. There is no equivalent for image warping or for systems that render geometry and images simultaneously. If we had such a metric, we could measure the visual results of our system in a quantitative fashion. Moreover, we could drive our preprocessing algorithm or the selection of reference images to guarantee a specified visual quality.

A limitation of displaying images on current hardware is that we must assume precomputed diffuse illumination. There is not enough per-pixel information to perform view-dependent shading on the image. This is not a problem for many walkthrough environments (e.g. functionally colored CAD models, architectural spaces, etc.), but we hope that in the future per-pixel normals and deferred shading [Whitted81][Deering88] [Schneider88][Molnar92] might be viable solutions.

Another area of future work is to reduce the storage requirement of images and to reduce the cost of warping. Our current images have a large amount of redundant information. We expect image compression methods to work well on the color component. As we mentioned at the end of Chapter 4, we look to dedicated image-warping hardware as a way to reduce the constant warping cost (during preprocessing and run time). A more immediate technique is to dedicate a larger number of processors to warping (e.g. a 16 or 32 processor machine).

8.4 Extensions to Portal Images

The extensions of most immediate benefit to the architectural system are those that might automatically compute the best portal viewpoints and view directions to use for creating the images. Possibly through the use of exact cell and portal visibility calculations [Airey90, Teller91], we could locate areas of the model from which each portal is visible and sample only from those areas. Once we gather more experience with portal images, we may decide to reduce the constraint that image-sampling viewpoints must lie on a single semicircle. The modelers may wish to have more freedom to place image-sampling viewpoints. An interactive portal-image placement program would be useful.

At the moment, we are not taking advantage of idle time to render portal images (or the LDIs for general models) that may be needed in the future. We could enhance the system to perform incremental rendering. For example, if our system is equipped with an additional graphics pipe, then we can use a simple prediction algorithm to create images expected in the near future without affecting the main pipe. Alternately, we can devote a fraction of the frame time to incrementally create predicted images.

References

[Airey90] Airey J., “Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments”, *Symposium on Interactive 3D Graphics*, 41-50 (1990).

[Aliaga96] Aliaga D., “Visualization of Complex Models Using Dynamic Texture-Based Simplification”, *IEEE Visualization*, 101-106 (1996).

[Aliaga97] Aliaga D. and Lastra A., “Architectural Walkthroughs Using Portal Textures”*,* *IEEE Visualization*, 355-362 (1997).

[Aliaga98a] Aliaga D., Cohen J., Wilson A., Zhang H., Erikson C., Hoff K., Hudson T., Stuerzlinger W., Baker E., Bastos R., Whitton M., Brooks F., Manocha D., "A Framework for the Real-time Walkthrough of Massive Models", *Computer Science Technical Report TR98-013*, University of North Carolina at Chapel Hill (1998).

[Aliaga98b] Aliaga D., Lastra A., "Smooth Transitions in Texture-based Simplification", Computer & Graphics, Elsevier Science, Vol 22:1, 71-81 (1998).

[Certain96] Certain A., Popovic J., DeRose T., Duchamp T., Salesin D., Stuetzle W., “Interactice Multiresolution Surface Viewing”, *Computer Graphics (SIGGRAPH ‘96),* 91-98 (1996).

[Chen93] Chen S. E. and Williams L., “View Interpolation for Image Synthesis”, *Computer Graphics (SIGGRAPH ‘93),* 279-288 (1993).

[Chen95] Chen S. E, “QuickTime VR - An Image-Based Approach to Virtual Environment Navigation”, *Computer Graphics (SIGGRAPH ‘95),* 29-38 (1995).

[Clark76] Clark J., “Hierarchical Geometric Models for Visible Surface Algorithms”, *CACM*, Vol. 19(10), 547-554 (1976).

[Clark87] Clark C., Brown T., "Photographic Texture and CIG: Modeling Strategies for Production Databases*", Interservice/Industry Training Systems Conference*, 274-283 (1987).

[Clark90] Clark C., Cosman M., "Terrain Independent Feature Modeling", *Interservice/Industry Training Systems Conference*, 7-17 (1990).

[Cohen91] Cohen F., Patel M., “Modeling and Synthesis of Images of 3D Textured Surfaces”, *Graphical Modeling and Image Processing*, Vol. 53, Iss. 6, 501-510 (1991).

[Cohen93] Cohen M., Wallace, J. Radiosity and Realistic Image Synthesis. Academic Press (1993).

[Cohen96] 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).

[Cohen98] Cohen J., Olano M. Manocha D., “Appearance-Preserving Simplification”, *Computer Graphics (SIGGRAPH ’98*), 115-122 (1998).

[Coorg97] Coorg S. and Teller S., “Real-Time Occlusion Culling for Models with Large Occluders”, *Symposium on Interactive 3D Graphics*, 83-90 (1997).

[Cosman94] Cosman M., "Global Terrain Texture: Lowering the Cost", *Image Society, VII Conference*, 52-64 (1994).

[Crawford77] Crawford B., Topmiller D., "Effects of Variation in Computer Generated Display Features on the Perception of Distance*", Image Society, I Conference*, 271-288 (1977).

[Darsa97] 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).

[Deering88] Deering M., Winner S., Schediwy B., Duffy C. and Hunt N., “The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics”, *Computer Graphics (SIGGRAPH ’88),* 21-30 (1988).

[DeHaemer91] DeHaemer M. and Zyda M., “Simplification of Objects Rendered by Polygonal Approximations”*, Computer Graphics*, Vol. 15(2), 175-184 (1991).

[Ebbesmeyer98] Ebbesmeyer P., “Textured Virtual Walls - Achieving Interactive Frame Rates During Walkthroughs of Complex Indoor Environments”, *VRAIS ‘98,* 220-227 (1998).

[Eck95] Eck M., DeRose T., Duchamp T., Hoppe H., Lounsbery M., Stuetzle W., "Multiresolution Analysis of Arbitrary Meshes", *Computer Graphics (SIGGRAPH ‘95),* 173-182 (1995).

[Ferguson90] Ferguson R. L., "Continuous Terrain Level of Detail For Visual Simulation", *Image Society, V Conference*, 144-151 (1990).

[Foley90] Foley J., van Dam A., Feiner S. and Hughes J., Computer Graphics: Principles and Practices, Addison Wesley (1990).

[Funkhouser93] Funkhouser T. and Sequin C., “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments”, *Computer Graphics (SIGGRAPH ‘93),* 247-254 (1993).

[Gardner81] Gardner G. *et al*. "A Real-Time Computer Image Generation System Using Textured Curved Surfaces", *Image Society, II Conference,* 60-76 (1981).

[Garland97] Garland M., Heckbert P., “Surface Simplification using Quadric Error Bounds”, *Computer Graphics (SIGGRAPH ‘97),* 209-216 (1997).

[Gortler96] Gortler S., Grzeszczuk R., Szeliski R. and Cohen M., “The Lumigraph”, *Computer Graphics (SIGGRAPH ‘96)*, 43-54 (1996).

[Hooks90] Hooks J., *et al*. "On 3-D Perspective Generation From a Multi-Resolution Photo Mosaic Database", *Image Society, V Conference*, 133-142 (1990).

[Haeberli93] Haeberli P., Segal M., *et al*., “Texture Mapping as a Fundamental Drawing Primitive”, *Fourth Eurographics Workshop on Rendering*, Paris, France (1993).

[Hoppe93] Hoppe H., DeRose T., Duchamp T., McDonald J., Stuetzle W., “Mesh Optimization”, *Computer Graphics (SIGGRAPH ‘93),* 19-26 (1993).

[Hoppe96] Hoppe H., “Progressive Meshes”, *Computer Graphics (SIGGRAPH ‘96)*, 99-108 (1996).

[Hoppe97] Hoppe H., “View-Dependent Refinement of Progressive Meshes”, *Computer Graphics (SIGGRAPH ‘97)*, 189-198 (1997).

[Hudson97] Hudson T., Manocha D., Cohen J., Lin M., Hoff K., Zhang H., “Accelerated Occlusion Culling using Shadow Frusta”, *ACM Symposium on Computational Geometry*, (1997).

[Klein96] Klein R., Liebich G., Strasser W., “Mesh Reduction with Error Control”, *IEEE Visualization*, (1996).

[Kumar95] Kumar S., Manocha D., Lastra A., “Interactive Display of Large Sacle NURBS Models”, *Symposium on Interactive 3D Graphics*, 51-58 (1995).

[Kumar97] Kumar S., Manocha D., Zhang H. and Hoff K., “Accelerated Walkthrough of Large Spline Models”, *Symposium on Interactive 3D Graphics*, 91-101 (1997).

[Lastra95] Lastra A., Molnar S., Olano M. and Wang Y., “Real-Time Programmable Shading”, *Symposium on Interactive 3D Graphics*, 59-66 (1995).

[Levoy96] Levoy M. and Hanrahan P., “Light Field Rendering”, *Computer Graphics (SIGGRAPH ’96)*, 31-42 (1996).

[Low97] Low K., Tan T., “Model Simplification using Vertex-Clustering”, *Symposium on Interactive 3D Graphics*, 75-82 (1997).

[Luebke95] Luebke D. and Georges C., “Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets”, *Symposium on Interactive 3D Graphics*, 105-106 (1995).

[Luebke97] Luebke D. and Erikson C., “View-Dependent Simplification of Arbitrary Polygonal Environments”, *Computer Graphics (SIGGRAPH ‘97)*, 199-208 (1997).

[Maciel95] Maciel P. and Shirley P., “Visual Navigation of Large Environments Using Textured Clusters”, *Symposium on Interactive 3D Graphics*, 95-102 (1995).

[Maillot93] Maillot J., Yahia H., Veroust A., “Interactive Texture Mapping”, *Computer Graphics (SIGGRAPH ‘93),* 27-34 (1993).

[Mark97a] Mark W., McMillan L. and Bishop G., “Post-Rendering 3D Warping”, *Symposium on Interactive 3D Graphics*, 7-16 (1997).

[Mark97b] Mark W., Bishop G., "Memory Access Patterns of Occlusion-Compatible 3D Image Warping", *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 35-44 (1997).

[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).

[McMillan95a] McMillan L. and Bishop G., “Head-Tracked Stereo Display Using Image Warping”, *Stereoscopic Displays and Virtual Reality Systems II*, Scott S. Fisher, John O. Merritt, Mark T. Bolas, ed., SPIE Proceedings 2409, 21-30 (1995).

[McMillan95b] McMillan L. and Bishop G., “Plenoptic Modeling: An Image-Based Rendering System”, *Computer Graphics (SIGGRAPH '95)*, 39-46 (1995).

[McMillan97] McMillan L., “An Image-Based Approach to Three-Dimensional Computer Graphics”, *Ph.D. Dissertation*, University of North Carolina at Chapel Hill (1997).

[Molnar92] Molnar S., Eyles J., Poulton J., PixelFlow: High-Speed Rendering Using Image Composition, *Computer Graphics (SIGGRAPH ’92),* 231-240 (1992).

[Moshell92] Moshell J. M., *et al*. "Dynamic Terrain Databases for Networked Visual Simulators", *Image Society, VI Conference*, 98-112 (1992).

[Mueller95] Mueller C., “Architectures of Image Generators for Flight Simulators”, *Computer Science Technical Report TR95-015*, University of North Carolina at Chapel Hill (1995).

[Popescu98] Popescu V., Lastra A., Aliaga D., and Oliveira Neto M., “Efficient Warping for Architectural Walkthroughs using Layered Depth Images”, *IEEE Visualization*, 211-215 (1998).

[Pratt92] Pratt D., Zyda M., *et al*., "NPSNET: A Networked Vehicle Simulation with Hierarchical Data Structures", *Image Society, VI Conference*, 216-226 (1992).

[Rafferty98a] Rafferty M., Aliaga D. and Lastra A., “3D Image Warping in Architectural Walkthroughs”, *VRAIS ‘98*, 228-233 (1998).

[Rafferty98b] Rafferty M., Aliaga D., Popescu V. and Lastra A., “Images to Accelerate Architectural Walkthroughs”, *Computer Graphics & Applications*, Vol. 18, No. 6, November-December, 38-45 (1998).

[Regan94] Regan M., Pose R., “Priority Rendering with a Virtual Reality Address Recalculation Pipeline”, *Computer Graphics (SIGGRAPH ‘94),* 155-162 (1994).

[Rife77] Rife R., "Level-of-Detail Control Considerations for Computer Image Generation Systems", *Image Society, I Conference*, 142-159 (1977).

[Robinson85] Robinson J., "Exploiting Texture in an Integrated Training Environment", *Interservice/Industry Training Systems Conference*, 113-121 (1985).

[Rossignac92] Rossignac J. and Borrel P., “Multi-resolution 3D Approximations for Rendering Complex Scenes”, *Technical Report*, IBM T.J. Watson Research Center, Yorktown Heights, NY (1992).

[Scarlatos90] Scarlatos L., "A Refined Triangulation Hierarchy for Multiple Levels of Terrain Detail", *Image Society, V Conference*, 115-122 (1990).

[Schachter83] Bruce Schachter (*ed*.), Computer Image Generation, John Wiley and Sons (1983).

[Schaufler96] Schaufler G. and Stuerzlinger W., “Three Dimensional Image Cache for Virtual Reality”, *Computer Graphics Forum (Eurographics ‘96),* Vol. 15(3), 227-235 (1996).

[Schneider88] Schneider B.O., Claussen U., “PROOF: An Architecture for Rendering in Object-Space”, *Advances in Computer Graphics Hardware III, Eurographics Seminars,* 121-140 (1988).

[Schroeder92] Schroeder W. J., Zarge A., Lorensen W. E., “Decimation of Triangle Meshes”, *Computer Graphics (SIGGRAPH ‘92),* 65-70 (1992).

[Shade96] 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).

[Shade98] Shade J., Gortler S., He L., and Szeliski R., Layered Depth Images, *Computer Graphics (SIGGRAPH ’98),* 231-242 (1998).

[Sillion97] 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).

[Stevens81] Stevens K., "Computational Analysis: A Technique for Improving the Visual Simulation of Terrain", *Image Society, II Conference*, 5-24 (1981).

[Teller91] Teller S., Séquin C., “Visibility Preprocessing For Interactive Walkthroughs”, *Computer Graphics (SIGGRAPH ’91),* 61-69 (1991).

[Teller92] Teller S., Visibility Computation in Densely Occluded Polyhedral Environments, *Ph.D. Dissertation (also Computer Science Department TR92-708),* University of California at Berkeley (1992).

[Torborg96] Torborg J., Kajiya J., “Talisman: Commodity Real-time 3D Graphics for the PC”, *Computer Graphics (SIGGRAPH ‘96),* 353-364 (1996).

[Turk92] Turk G., “Re-Tiling Polygonal Surfaces”, *Computer Graphics (SIGGRAPH ‘92),* 55-64, (1992).

[Westover91] Westover L., Splatting: A Feed-Forward Volume Rendering Algorithm*, Ph.D. Dissertation*, University of North Carolina at Chapel Hill (1991).

[Whitted81] Whitted T., Weimer D. M., “A Software Test-bed for the Development of 3-D Raster Graphics Systems”, *Computer Graphics (SIGGRAPH ‘81),* 271-277 (1981).

[Wolberg90] Wolberg G., Digital Image Warping, IEEE Computer Society Press (1990).

[Xia96] Xia J., Varshney A., “Dynamic View-Dependent Simplification for Polygonal Models”, *IEEE Visualization* (1996).

[Zhang97] Zhang H., Manocha D., Hudson T. and Hoff K., “Visibility Culling Using Hierarchical Occlusion Maps”, *Computer Graphics* (*SIGGRAPH ‘97)*, 77-88 (1997).