Bounding Geometric Complexity using Image Replacement

Daniel G. Aliaga, Anselmo A. Lastra

Department of Computer Science

University of North Carolina at Chapel Hill

{ aliaga | lastra } @cs.unc.edu

Abstract

We present an algorithm that takes as input a three-dimensional model and automatically bounds the geometric complexity for all viewpoints and view directions in the model. The algorithm creates an adaptive set of bounding-nodes. At each bounding-node, an optimization process selects the smallest and farthest subset of the model to remove from rendering to meet a fixed geometry budget. The bounding-nodes take advantage of hierarchical spatial data structures, used for view-frustum culling, to reduce the infinite set of viewpoints and view directions to a finite, manageable set. At run-time, we render the selected subset using an image that we warp to the current view. We use a quality metric to control the final rendered image quality.

Keywords: geometry, images, simplification, automatic, bounding complexity.

1. INTRODUCTION

When interactively displaying complex 3D models, it is important that the system maintain a high frame rate. This makes movement appear smooth and fluid. Designers of dedicated vehicle simulation systems realize this, so they make it a priority to sustain a high frame rate, even at the expense of visual quality. However, very few systems provide bounds on the geometric complexity rendered every frame.

In this paper, we present an algorithm that bounds the amount of geometry rendered from any viewpoint by selecting areas of high geometric complexity that must be replaced (or perhaps simplified). We couple this preprocess selection with a run-time system that replaces geometry-based rendering of the complex areas of the model with images. We then apply 3D image warping [McMillan95][Mark97] to correct the images to the current viewpoint.

1.1 Overview

The first step of our preprocessing algorithm partitions the model space into a uniform spatial grid. At the center of each of the boxes of the spatial grid, we create a *bounding-node* (or *b-node*). We examine all view directions centered on a b-node and compute an optimum *solution* that selects the best subset of the model to remove in order to bound the maximum number of primitives that would need to be rendered for any view direction around that b-node. We present an algorithm to choose an appropriate b-node for any viewpoint and view direction within the model, and show that we can bound the geometry to be rendered for any viewpoint. We adaptively refine the original set of b-nodes to ensure a specified image quality.

At run-time, we choose a b-node (as above) for the given viewpoint and view direction. We replace the subset of the model selected as the solution for the b-node with an image, and warp it to the current viewpoint.

The preprocessing complexity is kept manageable because we take advantage of the fact that geometry is culled using a hierarchical spatial data structure (e.g., an octree). Culling is applied to the groupings of geometry stored in the spatial partitioning tree and not to the individual geometric primitives. Thus, as we examine view directions obtained by rotating about a b-node, the geometry to be sent down the graphics pipeline will be the same until view-frustum culling adds or removes an octree cell. This fact turns the infinite space of viewpoints and view directions into a finite one.

This paper is organized as follows. Section 2 describes related work. In Section 3, we provide details of our algorithm for bounding geometric rendering complexity. Section 4 presents our run-time system for replacing geometry with warped images. Section 5 describes the implementation and Section 6 presents performance results. Section 7 lists future work. Finally, we conclude with Section 8.

2. RELATED WORK

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

- Controlled Frame Rate Systems
- View-Dependent Simplification
- Image Caching Systems

In [Funkhouser93], Funkhouser and Sequin presented a system that, at run-time, selects levels-of-detail (LODs) and shading algorithms, in order to meet a constant frame rate. The system maintains a hierarchy of the objects in the environment. It computes cost and benefit metrics for all of the alternative representations of objects and uses a knapsack-style algorithm to find the best set for each frame. If too much geometry is present, detail elision is used.

Maciel and Shirley [Maciel95] expanded upon this and increased the representations available for the objects. A set of *impostors*, which include LODs, texture-based representations and colored cubes, can be used to meet the target frame rate.

Flight simulators use several techniques to reduce geometric complexity and maintain high frame rates [Schacter83, Mueller95]. Primarily, they assess scene complexity at each frame in order to determine which LODs to use for the next frame. In order to maintain a high frame rate, they operate with a target of 90% of the polygon budget. When the LOD selection takes more than 90% of the time, the LOD switching distance is reduced. A backup technique is to render polygons from front to back; when the rendering time is exhausted, the system stops rendering and displays the current image.

Alternately, view-dependent simplification algorithms support maintaining constant geometric complexity every frame [Hoppe97, Luebke97]. The algorithms can maintain a minimal screen-space error during simplification. Unfortunately, depending on the amount of simplification needed and on the scene complexity, objects will be merged and details will eventually be lost.

Various systems have been presented that use image-based representations (typically texture-mapped quadrilaterals) to replace subsets of the model. The source images are either pre-computed (as in [Maciel95] and [Aliaga96]) or computed on the fly [Shade96][Schaufler96]. Metrics are used to control image quality but not the amount of geometry to render. Rafferty [Rafferty98] uses 3D image warping to accelerate rendering in architectural models. Starting with a cells and portals [Airey90][Teller91][Luebke95] subdivision of an architectural model, the portals are replaced with pre-computed images that are warped every frame to the current viewpoint. The geometric complexity is dependent on the contents of the current cell. Both [Darsa97] and [Sillion97] construct a simplified mesh to represent the background of the scene. In the worst case, the complexity of the mesh is proportional to the screen resolution. However, neither system provides control of the number of primitives required to draw the simplified mesh and any nearby geometry.

3. BOUNDING GEOMETRIC COMPLEXITY

In this section, we describe how we can bound the amount of geometry that is sent to the graphics pipeline. We first detail the way we select which b-node to use for a given viewpoint and view direction. We then show that, given bounded solutions at the b-nodes, we can guarantee that we will meet the bound from any viewpoint and view direction within the model. Next, we describe why we only need to consider a finite number of view directions at each b-node. Finally, we detail our optimization process and explain how we enforce a minimum image quality throughout the model space.

3.1 Node Selection

For an arbitrary viewpoint V in the model, we will select and use (at run time) the closest b-node in the reverse projection of the current view frustum, as shown in Figure 1a. Create a frustum from B with the same field-of-view (FOV) and in the same view direction (shown with dashed lines). We see that it will always encompass more geometry than the one from V. Therefore, a solution from B will be conservative.

In order to make use of this method, we must show that, for a practical FOV of around 60 degrees, we can find a b-node within

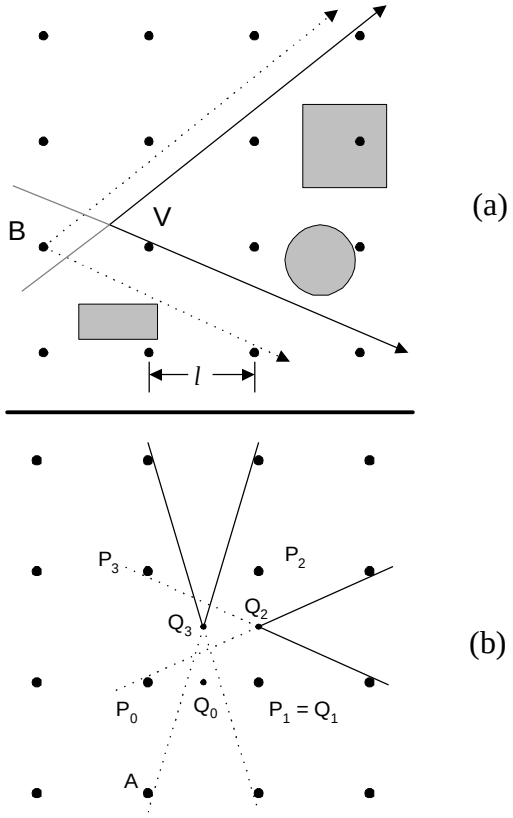


Figure 1: (a) Selection of b-node B for viewpoint V. (b) Guarantee that a b-node can be found within distance *2l* for FOV of 60 degrees or greater.

a short distance of any viewpoint. Keeping this distance small will place the images we use to replace geometry as far away as possible. We will see in Section 3.3 that placing the images far away improves our warping quality. Also, the closer the b-node is to the viewpoint, the tighter bound we have on the complexity. In our system, we would like to limit this distance to at most twice the separation between b-nodes (*l* in Figure 1a).

Consider, without loss of generality, the two-dimensional grid of b-nodes in Figure 1b. B-nodes are the larger dots. We have labeled a set of nodes P_i. The points Q_i delineate the lower-right quadrant of the quadrilateral $P_0P_1P_2P_3$. Because of symmetry, we know that, if for all viewpoints in quadrilateral $Q_0Q_1Q_2Q_3$ we can find a b-node closer than *2l*, then we can find one for all viewpoints in quadrilateral $P_0P_1P_2P_3$. We start by analyzing the corner points, Q_i. Since we are considering all view directions, the points Q_0 and Q_2 are reflections about the line Q_1Q_3. So we only need to consider one of Q_0 or Q_2. Q_1 is a node point, so the distance is 0. Hence, we only need to show that for all view directions emanating from Q_2 and Q_3 there exists a b-node within distance *2l*.

If we place the viewpoint at Q_2, a worst-case view direction is illustrated in Figure 1b. It occurs when we miss b-nodes P_0 and P_3. This cannot happen if the FOV is greater than approximately 53 degrees ($2\tan^{-1}(0.5)$). If we place the viewpoint at Q_3, the worst case occurs when we miss b-nodes such as A. This cannot happen unless the FOV is less than 37 degrees.

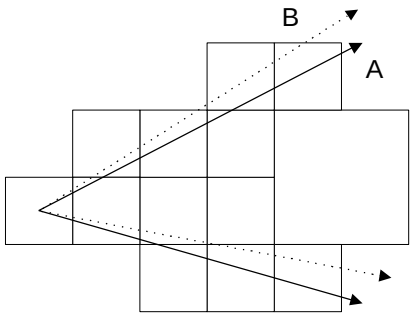


Figure 2: Discrete culling events. The set of visible octree cells will not change if we rotate between frustum A and frustum B.

We have shown that for all points Q_i, we can find a b-node within distance *2l* for FOVs greater than 53 degrees. To see that this holds true for any viewpoint and view direction, we recursively subdivide space $Q_0Q_1Q_2Q_3$, as we did $P_0P_1P_2P_3$. This space is always closer to a node than the previous, so we are guaranteed that for FOV of 60 degrees or greater, we can use a solution at most *2l* away and the view frustum will encompass at most the same amount of geometry as from the b-node. For rectangular spacing of nodes on the grid, we simply consider the longest edge to be the node distance. We could use a narrower FOV, but our solutions will suffer because the b-nodes might be farther from the viewpoints.

3.2 Discrete Culling Events

Before describing our optimization process, we explain our discrete culling events (DCE) data structure, which allows us to consider a finite number of view directions per b-node.

To efficiently view-frustum cull a model, hierarchical spatial data structures are typically used (e.g. octree). The granularity of the culling operations is that of the leaves of the octree. Because of this granularity, there are varying angular ranges of movement around a b-node that will not change the results of view-frustum culling (Figure 2).

Consider only allowing yaw rotation of a view frustum centered on a b-node. 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 can divide the 360-degree yaw range about a b-node into a disk with pie sections (Figure 3). All views, using a fixed FOV, whose view direction (the vector from the viewpoint to the center of the FOV), fall within the same pie section, will cull to the same set of octree cells.

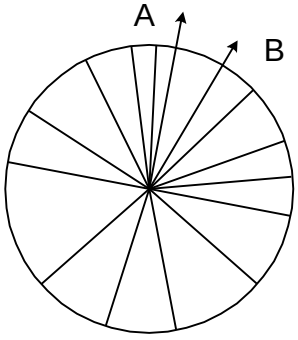


Figure 3: Culling events with yaw rotation. The same amount of geometry will be sent to the graphics pipeline for view directions A and B because the respective view frusta intersect the same octree nodes.

We can expand this to allow for yaw and pitch rotation (the latter only has an interesting range of ± 90 degrees). In this case, the culling space can be represented using a 2D manifold (i.e. the outer half of a torus) discretely tessellated into sections. Since our viewing frustum is rectangular, in general, the culling space cannot be represented using a sphere. Consider looking in a certain view direction. If the gaze is straight up, then you will not necessarily see the same frustum as if you were initially looking in a slightly different view direction and then looked up.

We associate with each disk or torus section (which we will call a *view*) the visible octree cells of the model. We will use these data during the optimization process. In our experience with interactive walkthroughs, roll is not a typical operation. Our run-time system does not disallow it, but we do not explicitly consider it during preprocessing.

If the octree is very deep, we might encounter a very large number of views per b-node. We do not necessarily need to consider the actual leaves of the octree. Instead, we can select an arbitrary tree depth to act as leaf cells. This will give us a reduced number of views per b-node. The set of views still correctly represents the complexity surrounding the b-node but we sacrifice granularity for preprocessing performance.

3.3 Optimization Process

Our optimization process iterates through the grid of b-nodes. For each b-node, we find the most expensive view. If the number of primitives contained in this view is less than or equal to our geometry budget, then we move on to the next b-node until all have been processed. If the view exceeds our geometry budget, we compute the best visible subset of the model to remove from rendering. If after removing the computed subset, there is still a view that violates our budget, we compute another subset for that view. The process is repeated until the geometry budget is met for all views.

In this subsection, we first describe our cost-benefit function, then describe how we can efficiently represent visible subsets of the model. Afterwards, we present our inner optimization loop and our method for computing multiple subsets.

3.3.1 Cost-Benefit Function

In order to quantify what we mean by the "best visible subset" of the model to remove from rendering, we define a cost-benefit function. The function is composed of a weighted sum of the cost and benefit of the given subset. It returns a value in the range [0,1].

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

$$\text{Cost} = g_c/G_c$$

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.

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

Where the constants are:

- B_1: weight for image size component
- B_2: weight for image depth component
- D: maximum dimension of the model
- S_w: screen width
- S_h: screen height

The final *cost-benefit* function will tend to maximize the benefit component and minimize the cost. We use an additional set of weights to combine the cost and benefit. 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:

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

3.3.2 Efficiently Representing Visible Subsets

The DCE data structure stores the list of octree cells that are visible from the current view (for which we are about to perform the optimization). The cost-benefit function needs to compute the screen-space bounding box and geometric complexity of subsets of the visible octree cells.

We can efficiently perform these by first projecting the octree cells to screen space and use their minimal projected *x* and *y* values to create two sorted lists in ascending order. Then, we use their maximal projected *x* and *y* values to construct two more sorted lists in descending order. Finally, we construct a fifth and sixth list in ascending and descending order using the range values (distance from eye to nearest and farthest point of each octree cell). We store with each octree cell its positions in the six sorted lists. To represent any (contiguous) subset of the visible octree cells, we use a 6-tuple of numbers. The 6-tuples are indices into the sorted lists representing the leftmost, rightmost, bottommost, topmost, nearest and farthest borders of a subset (Figure 4). All octree cells whose indices lie within the ranges defined by the 6-tuple are members of the subset. In our implementation, we do not have a sixth sorted list because for our image warping purposes all subsets are opaque and thus the farthest plane is always the end of the model.

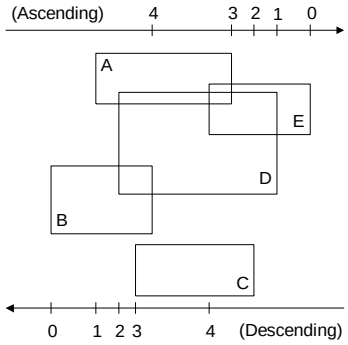


Figure 4: We show a 2D example of representing a subset of cells. The ascending and descending lists for projected x values are illustrated. We can represent the subset ACD with the tuple (1, 1). In the general 3D case, a 6-tuple is used to index 3 pairs of lists.

We can slightly modify a subset by simply changing an index in the 6-tuple, and then incrementally update the screen-space bounding box of the subset as well as the count of geometry.

3.3.3 Inner Optimization Loop

Our inner optimization loop meets the geometry budget by using the cost-benefit function to find the best subset of the current view to remove from rendering. We start by subdividing the visible set of octree cells into a M x N x P grid of initial subsets. If an initial subset is small enough that removing it from rendering does not meet our geometry budget, we classify it as an *expanding* subset. If removing an initial subset from rendering already meets our geometry budget, we label is as a *contracting* subset. We typically choose values for M, N and P that cause most subsets to be expanding. In any case, we enqueue all initial subsets into a subset queue.

We process the subset queue as follows:

1. Set the current best subset for removal to empty.
2. Dequeue a candidate subset. If the queue is empty, then stop.
3. If it is an expanding subset and meets our geometry budget, then compute its cost-benefit and update the current best subset. Loop back to step 2.
4. If it is a contracting subset and does not meet our geometry budget anymore, then loop back to step 2.
5. Use the current candidate subset to compute a new candidate. A look-ahead tree is used to recurse a few iterations into the future to find the best new subset. Based on the subset classification, one of the indices of the subset's 6-tuple will be increased or decreased, then we can incrementally update the screen-space bounding box and the count of geometry. We do not allow the subset to include an area within *2l* of the b-node (see Section 3.1).
6. The new candidate subset is enqueued into the subset queue. Loop back to step 2.

Once the subset queue is empty, we will have determined the best subset of the model to remove from rendering to meet the geometry budget.

3.3.4 Multiple Subsets

After computing the best subset to remove, we update the DCE data structure to reflect the removed geometry. If there still exists a view that contains too much geometry, we run the inner optimization loop again on that view. While computing

additional subsets, we do not allow them to overlap any existing subsets.

3.4 Adaptively Creating Nodes

Once we have computed the initial grid of b-nodes, we have stored at each b-node a subset of the model to represent using an image. In this next step, we enforce a minimal image quality throughout the model space. First, we define an *image quality metric*. Second, we explain how we recursively replace b-nodes that do not meet the quality threshold with a new set of higher quality b-nodes. Third, we present a fast replication technique for further refining optimization solutions.

3.4.1 Quality Metric

We wish to render the geometry subsets computed at each b-node using 3D image warping. A common problem in warping images is the lack of data for regions that may become exposed [McMillan95]. For example, if the image-sampling viewpoint includes an object, but we move from that sample point, we should see whatever is behind the object. Unfortunately, we do not have that information in the image, so don't know what to render. These areas typically manifest themselves as *tears* in the image. Our quality metric limits the largest tear that can appear while warping.

The metric attempts to quantify the largest error that we could encounter within the volume of viewpoints serviced by a b-node. We compute the maximum screen-space displacement of the nearest point within the subset (at the farthest viewpoint from the b-node). Similarly, we compute how much the farthest point within the subset can move in screen space. We subtract these two numbers to obtain the quality metric. This is a measure of the largest possible tear that can occur. In practice, the geometry inside the subset frequently prevents us from seeing the farthest point of the subset and we see far less tearing. Unfortunately, performing a per-pixel analysis is too expensive for our purposes, so we rely on this conservative measure.

3.4.2 Subdividing Space

We use the above metric to measure the quality of each b-node's solution. If a solution does not surpass our minimum quality requirement, we reduce the spacing between nodes in order to help improve the quality.

We center an imaginary axis-aligned box of size l on a b-node. Then, we recursively subdivide the axis-aligned box into eight sub-boxes. At the center of each sub-box, we compute a new b-node and its corresponding optimization solution. We repeat this process until all b-nodes meet the quality threshold.

3.4.3 Replication Technique

In practice, we have found large coherence between b-node solutions, especially with high quality levels. Thus, instead of running the full optimization process when subdividing for high quality levels, we replicate the current b-node to produce eight sub-nodes, changing only the image-sampling viewpoint. We use the solution of the parent b-node at each of the eight sub-nodes. The eight new sub-nodes reduce the volume within we must warp a particular image, thus reducing the potential error.

This replication technique will not violate our geometry budget. In essence, we are starting with a partially adapted grid and simply creating the nodes in the interior of the original quadrilaterals $P_0P_1P_2P_3$ as described in Section 3.1 (Figure 1b). By definition they will still meet our bound on geometry.

4. RUN-TIME SYSTEM

This section describes our run-time system that takes the output of the preprocessing algorithm and renders the computed subsets using image warping.

At the beginning of each frame, the system searches through the set of b-nodes to find the closest one contained in the reverse projection of the current view frustum (Section 3.1). This b-node specifies the subset of the model to render using image warping. We dynamically remove the subset from the model by culling the geometry behind an imaginary quadrilateral that matches the screen-space bounding box of the subset.

The same imaginary quadrilateral is used to define the reference images for warping. We use the four model-space corners of the quadrilateral and the b-node's position to construct a view frustum. Then, we render the model and save the framebuffer contents as the reference image.

We could render all of the reference images at startup time, but we have found that in practice it is not worth the extra storage. Instead, we render the images on demand. We keep a variable-size host memory cache and use a least-recently-used cache replacement policy.

Our image-warping algorithm is based on that of [McMillan95] and [Mark97]. To render, we use either a variable size reconstruction kernel or 3 by 3 fixed-size splats as in [Rafferty98].

5. IMPLEMENTATION

We implemented our preprocessing algorithm and run-time system in C++, on a SGI Onyx^2, with Infinite Reality Graphics, 4 R10000's @ 195 MHz and 1024 MB of main memory. The run-time system uses OpenGL and GLUT for all its graphical rendering.

Our preprocessing algorithm uses multiple processors to simultaneously compute b-node solutions. The algorithm takes as input:

- the octree of the 3D model
- the geometry budget
- the FOV and destination framebuffer size
- resolution of the initial grid of b-nodes
- image quality threshold
- maximum tree depth to use for the DCE
- tree depth at which to start replicating b-node subsets for efficiency

We used the follow cost-benefit constants:

$$B_1 = 0.2,\ B_2 = 0.8,\ B = 0.6,\ C = 0.4.$$

The run-time system uses a multi-processor pipelined implementation to simultaneously perform image warping, view-

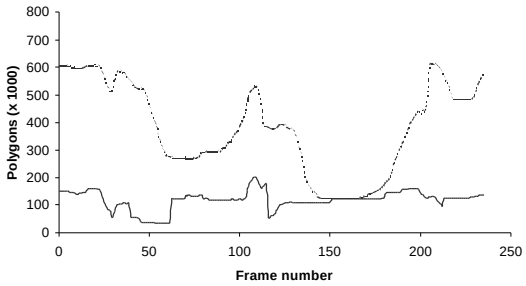


Figure 5: Number of primitives rendered per frame during a path through the torpedo-room model, with (solid) and without (dashed) bounded geometry. The bound was 200,000 primitives.

frustum culling and geometry rendering. We typically render 256x256 or 512x512 reference images. Since we simultaneously image warp and render primitives, we cannot warp directly to the framebuffer. Hence, we warp to a 640x480 common warp buffer. In a single operation, we copy and scale the warp buffer to the framebuffer.

We implemented the DCE data structure that accounts for yaw-rotation (disk). We have found that this yields us an accurate enough culling representation for our typical walkthrough scenarios.

6. RESULTS

We have applied our algorithm to three large models: (a) a submarine torpedo room model consisting of 836,815 primitives, (b) an auxiliary machine room model containing 502,476 primitives, and (c) a radiosity-illuminated model of a house with 528,744 primitives.

For each of the models, we selected a geometry budget of one-fourth the original model size. We then applied our preprocessing algorithm. The algorithm took approximately 4, 2 and 8 hours for the torpedo room, machine room, and house models respectively. For the torpedo and machine room models, we computed 4096 b-nodes before replication. For the house model, we generated a higher quality solution with 22,702 b-nodes before replication.

We recorded a path through the torpedo room model using a 60 degree FOV. Figure 5 compares the number of primitives rendered per frame using conventional culling to the number of primitives displayed after employing our algorithm to bound the geometric complexity to 200,000 primitives. Note we never exceed our geometry budget. The quality of the solution does not affect the rendering time but can slightly change the amount of geometry to render. This is because the additional b-nodes potentially have image replacements that cull a slightly different number of primitives.

Figure 6a compares the overall frame times for the same path. We image warp 512x512-pixel reference images. This graph does not exactly match the rendered primitives graph, since we incur a constant overhead for image warping. We have decomposed the frame time into *render* time and *cull-warp* time. Culling and image warping for the next frame occur simultaneously with geometry rendering for the current frame. The larger time determines the overall frame time.

As one would expect, the render-time graph (Figure 6b) is very similar to the number-of-primitives-rendered graph. Rendering time is directly proportional to the number of primitives.

On the other hand, Figure 6c shows a constant image warping overhead of 100ms during our sample path. We have timed the run-time performance of our image warper. On average, we can warp and display 256x256-pixel reference images at 28 frames/second and 512x512-pixel reference images at 9.6 frames/second. This is consistent with our measured cull-warp time.

In our implementation, culling occurs in the same process as warping. For this path, this increases the cull-warp time by approximately 50ms. We could move the culling and warping to two different processors to increase our overall frame times. Furthermore, since culling time grows logarithmically with model size, we would probably be warping bound. As long as we render no more primitives than it takes us to warp, we can achieve a very constant frame rate.

We generated solutions of three levels of quality for the machine room model (Color Plate 6). The maximum tear size is measured as a percentage of the screen width in pixels. The lower bound on quality for each of the solutions is 46%, 15% and 7%. In addition to further improving our solutions, we look to higher quality image warping, such as using layered depth images.

7. FUTURE WORK

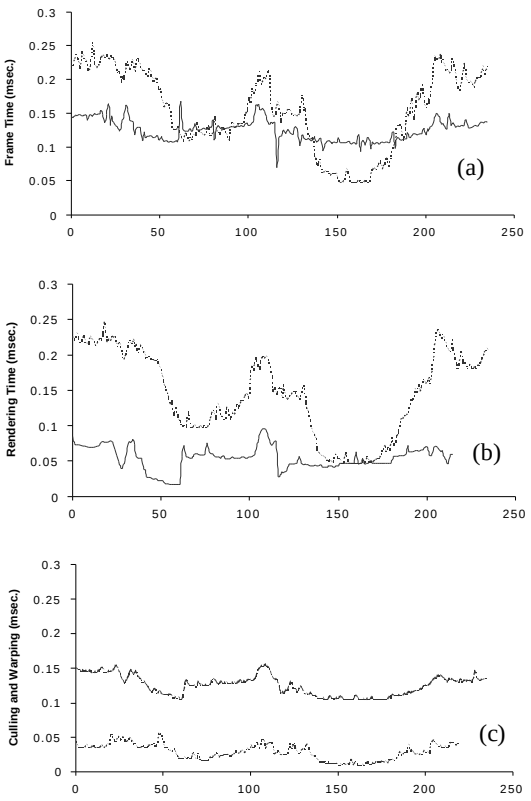


Figure 6: Various processing times for the same path shown in Figure 5. The solid line is with bounded geometry, the dashed line without. (a) Overall frame times. These are a combination of the following two times. (b) Geometry rendering time (proportional to the number of primitives). (c) Total of culling and warping time.

Our first task is to use pre-computed layered depth images [Gortler97] instead of conventional depth images. Since layered depth images include data from multiple views (rendered during the preprocess construction phase) they will essentially eliminate the tears and reduce the number of b-nodes required. Also, creating the images during preprocessing will make the run-time frame rates smoother.

We plan to refine our run-time system in order to make sure that we maintain a constant frame rate. This includes dividing the culling and warping tasks across different processors in order to eliminate the cost of culling.

8. CONCLUSIONS

We have presented an algorithm for bounding geometric complexity by automatically computing subsets of the model to replace with images. At run-time, we warp these images to reduce error. We have coupled a quality metric with our optimization process to control the overall image quality that we obtain. Additionally, we present a data structure to take advantage of the discrete nature of culling a model represented with a hierarchical spatial data structure.

We also define a cost-benefit function to quantify the *best* subsets of the model for replacing geometry with 3D image warping. We search for the smallest and farthest (in screen-space) subset. Our framework can be made slightly more general. For example, we can define a similar cost-benefit function to select a subset of the model on which we wish to concentrate level-of-detail resources. If we specify to the preprocessing algorithm a rendering budget of 70,000 primitives, but our actual budget is 100,000 primitives, then we must simplify the subset for the current image node from its original complexity to at most 30,000 primitives.

Finally, we have presented a run-time system which when coupled with the preprocessing algorithm provides us with an automatic way of selecting the best subset of the model to render using 3D image warping.

References

[Airey90] John Airey, *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision*, Ph.D. Dissertation, University of North Carolina (also UNC-CS Tech. Report TR90-027), 1990.

[Aliaga96] Daniel G. Aliaga. "Visualization of Complex Models Using Dynamic Texture-Based Simplification", *IEEE Visualization '96*, 101-106, 1996.

[Darsa97] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney, "Navigating Static Environments Using Image-Space Simplification and Morphing", Proceedings of 1997 Symposium on Interactive 3D Graphics (Providence, RI), April 27-30, 1997, 25-34.

[Funkhouser93] Thomas A. Funkhouser and Carlo Sèquin, "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments", *SIGGRAPH 93*, 247-254, August 1993.

[Hoppe97] Hughes Hoppe, "View-Dependent Refinement of Progressive Meshes", *SIGGRAPH 97*, 189-198, August 1997.

[Gortler97] Steven J. Gortler, Li-wei He, Michael F. Cohen, *"Rendering Layered Depth Images"*, Technical Report, MSTR-TR-97-09.

[Luebke95] David Luebke and Chris Georges, "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets", *Proc. Symp. on Interactive 3D Graphics*, 105-106, 1995.

[Luebke97] David Luebke and Carl Erikson, "View-Dependent Simplification of Arbitrary Polygonal Environments", *SIGGRAPH 97*, 199-208, August 1997.

[McMillan95] Leonard McMillan and Gary Bishop, "Plenoptic Modeling: An Image-Based Rendering System", *SIGGRAPH 95*, 39-46, August 1995.

[Maciel95] Paulo Maciel and Peter Shirley, "Visual Navigation of Large Environments Using Textured Clusters", *Symp. on Interactive 3D Graphics*, pp. 95-102, 1995.

[Mark97] William R. Mark, Leonard McMillan and Gary Bishop, "Post-Rendering 3D Warping", *Proc. Symp. on Interactive 3D Graphics*, 7-16, 1997.

[Mueller95] Mueller, Carl, "Architectures of Image Generators for Flight Simulators", University of North Carolina Computer Science Technical Report TR95-015, 1995.

[Rafferty98] Matthew M. Rafferty, Daniel G. Aliaga, Anselmo A. Lastra, *"3D Image Warping in Architectural Walkthroughs", to*

appear, VRAIS '98, March 14-18 1998.

[Schaufler96] Gernot Schaufler and Wolfgang Stuerzlinger, "A Three Dimensional Image Cache for Virtual Reality", *Eurographics '96*, 227-235, 1996.

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

[Shade96] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose and John Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments", *SIGGRAPH 96*, 75-82, 1996.

[Sillion97] Francois Sillion, George Drettakis and Benoit Bodelet. "Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery", Eurographics 97, Aug. 1997.

[Teller91] Seth Teller and Carlo H. Séquin, "Visibility Preprocessing For Interactive Walkthroughs", *SIGGRAPH 91*, 61-69, 1991.

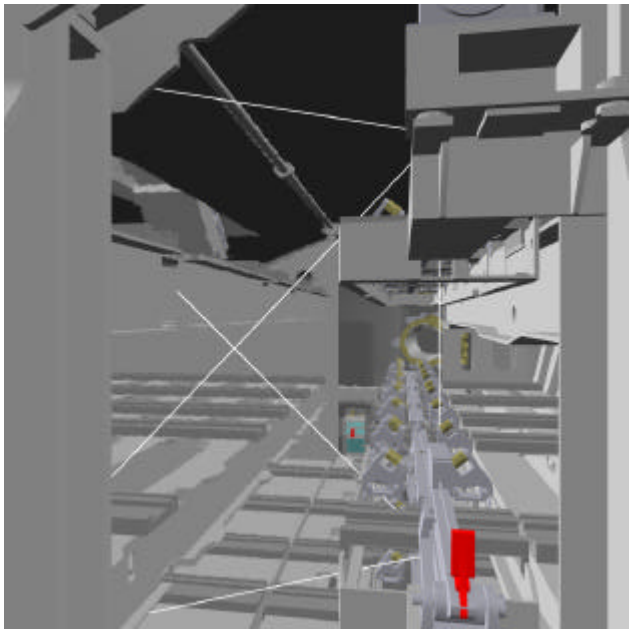


Plate 1: View of a torpedo room. We have rendered the area highlighted in white using image warping. Our algorithm automatically selected this subset of the model for warping in order to meet our geometry budget.

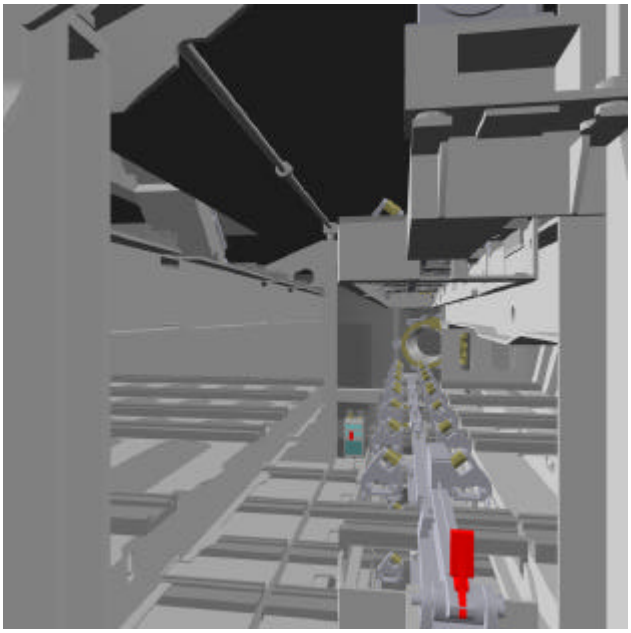


Plate 2: View of a torpedo room. Rendered normally for comparison.

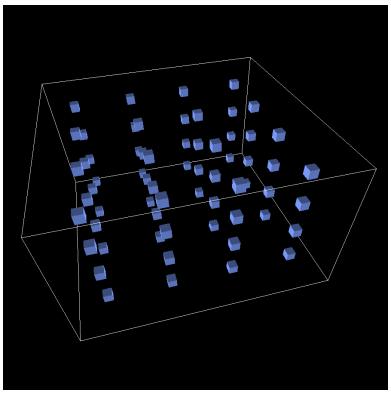


Plate 3: The model space is divided into an initial grid of nodes.

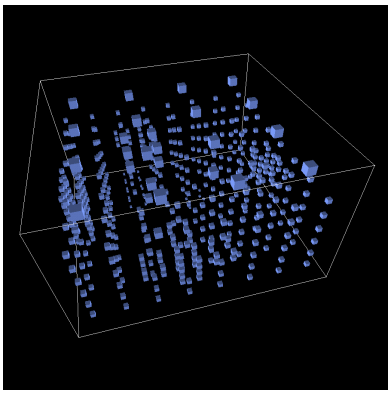


Plate 4: Additional nodes are created to meet a quality threshold throughout the model.

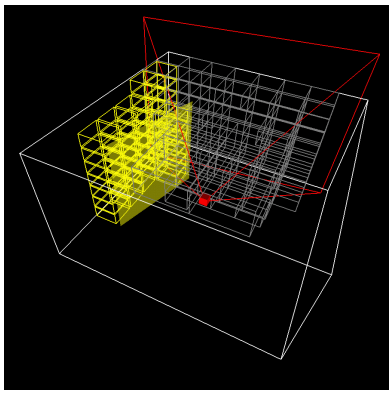
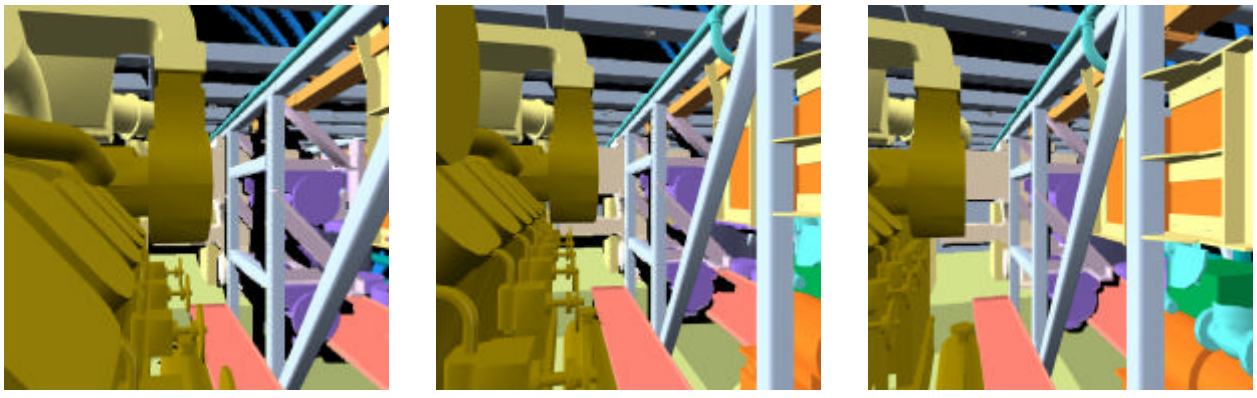


Plate 5: The subset of the model to image warp for the red viewing frustum is highlighted in yellow.



(a) (b) (c)

Plate 6: Images from automatically generated solutions for three quality levels: low (a), medium (b), and high (c). The viewpoints illustrate some of the largest errors that we encounter in this model. They are not rendered from the same location because the worst errors occur at different places in the three solutions.