EUROGRAPHICS 2004 /
Oliver Deussen, Charles Hansen, Daniel A. Keim and Dietmar Saupe
(Editors)

Volume 22 (2003), Number 3

Out-of-core Interactive Display of Large Meshes Using an Oriented Bounding Box-based Hardware Depth Query

Haeyoung Ha,[1†] Benjamin F. Gregorski[1] and Kenneth I. Joy[1]

[1] Visualization and Computer Graphics Group, Center for Image Processing and Integrated Computing(CIPIC),
Department of Computer Science, University of California Davis, Davis, CA, USA

Abstract

In this paper we present an occlusion culling method that uses hardware-based depth queries on oriented bounding boxes to cull unseen geometric primitives efficiently. Using an out-of-core design approach enables this method to interactively display data sets that are too large to fit into main memory. During a preprocessing phase, a spatial subdivision (such as an octree or BSP tree) of a given data set is constructed where, for each node, an oriented bounding box containing mesh primitives is computed using principal component analysis (PCA). At runtime, the tree indicated by the spatial subdivision is traversed in front-to-back order, and only nodes that are determined to be visible, based on a hardware accelerated depth query, are rendered.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation I.3.3 [Computer Graphics]: Viewing Algorithms

1. Introduction

Large data sets consisting of several million polygons are becoming commonplace, and more efficient means to visualize them are necessary. For example, satellite imagery and laser scanning applications produce large triangle meshes that need to be visualized efficiently. It is still difficult to interactively display these large meshes despite recent advances in graphics hardware technology. Multiresolution and data simplification methods are often used to reduce the number of primitives sent to the graphics hardware allowing the display of large data sets at interactive rates. Another approach to reducing graphics primitive count is occlusion culling. Occlusion culling reduces the number of primitives sent to the graphics hardware by identifying and ignoring parts of the mesh being visualized that are not visible from a current view point. This approach is very effective for models with high depth complexity.

The occlusion culling method presented in the paper uses the hardware depth query avaiblable on modern graphics hardware and oriented bounding boxes to cull unseen geometric primitives that are within the viewing frustum. An out-of-core design enables data sets that are too large to fit into main memory to be displayed at interactive rates. In a preprocessing phase, we build a hierarchical spatial subdivision of a given dataset. Each node of the hierarchy is associated with a tight-fitting oriented bounding box around the mesh primitives that intersect the node. This bounding box is computed using principal component analysis (PCA). The preprocessing creates two files –the subdivision tree and mesh data files– that are used to drive the occlusion queries and the asynchronous rendering algorithm. At runtime, the subdivision tree is traversed in front-to-back order, and a hardware depth query is performed using the oriented bounding box at each node to determine visibility for the geometry contained inside the node. If the node is not visible, it and its children are not considered for rendering. Only nodes that are determined to be visible are traversed and rendered. We use oriented bounding boxes since they yield tighter fitting bounding boxes with a smaller image space projection than the conventional “min-max” axis-aligned bounding box.

Contributions The main contributions of our paper are the

[†] hha,bfgregorski,kijoy@ucdavis.edu

spatial subdivision of a data set using tight-fitting oriented bounding boxes to perform occlusion culling on large polygonal models. Additionally we present an out-of-core design which allows models that are too large to fit into main memory to be viewed at interactive rates. Lastly, we leverage the capabilities of modern graphics hardware through the use of hardware-based occlusion queries and accelerated graphics port (AGP) memory to perform fast occlusion tests using the bounding boxes and asynchronous rendering on mesh primitives.

2. Previous Work

Much research has been done in occlusion culling and visibility computation. Many culling algorithms have been designed for specialized environments [ARB90] including architectural models and urban data sets composed of large occluders. Occluders are objects in the data set known to cover or occlude parts of the data set (such as walls in architectural visualization). These specialized occluder-based approaches work well for the intended environments but perform less optimally in general settings.

Hillesland *et al.* [HSLM02] use hardware-based depth query for occlusion culling. They precompute a spatial subdivision of a mesh and render geometric primitives in front-to-back order, using hardware depth queries to determine visibility. For fast front-to-back traversal of a subdivision hierarchy, they use a user-programmable vertex shader. Their method was tested on a power-plant model having significant depth complexity. They obtained a speed-up factor of four, on average, over view-frustum culling alone and up to a speed-up factor of ten on constrained cases. The advantages of their method are that it requires no explicit occluders, reduces bandwidth to the graphics card, and performs conservative occlusion culling so that all visible parts are rendered. Yoon *et al.* [YSM03] describe a system for view-dependent rendering from continuous level-of-detail models with conservative occlusion culling. A vertex hierarchy based on edge collapes is built for level-of-detail rendering, and a cluster hierarchy based on vertex clustering is layered on top of the vertex hierarchy for occlusion. For occlusion, frame-to-frame coherence is exploited by reusing the visible set from the previous frame to approximate the new visible set.

El-Sana *et al.* [ESSS01] integrate occlusion culling with view-dependent rendering. View-dependent rendering alone can allow occluded areas to be rendered in high detail. While occlusion culling removes occluded triangles, it may still render many small triangles that cover only a few pixels. However, geometric simplification, dependent upon the view point, reduces the number of triangles that occupy small regions, but still renders occluded triangles. They combine geometric simplification and occlusion culling to achieve a greater reduction in rendered triangles. They do this by adding “visibility” as another parameter in selection criteria of the appropriate LOD. Andújar *et al.* [ASVNB00] also show an approach that integrates occlusion culling with view-dependent rendering using hardly visible sets (HVS). These are subsets of the potentially visible cells that contribute only a small number pixels to the overall image. A framework using a user-specified error bound selects a mesh from a fixed set of LOD representations based on the HVS error estimates. Greene *et al.* [GKM93] describe a visibility culling method that uses two hierarchical data structures, an object-space octree and an image-space Z-pyramid, to rapidly cull hidden geometry. They divide the mesh geometry into octree nodes, and scan convert the faces of each node to determine visibility. Thus, if the node is “hidden” none of its children are considered. A Z-pyramid hierarchy is employed to reduce the cost of computing node visibility. Also, to exploit temporal coherence, they maintain a list of visible nodes from the previous frame and render those first. Zhang *et al.* [ZMHI97] uses a hierarchical occlusion map (HOM) for software visibility culling. For each frame, occluders are selected from an occluder database and rendered, forming the occlusion map hierarchy. Occluders are chosen from the set of occluders intersecting the viewing frustum (with temporal coherence in mind). Occluders are rendered as white polygons on a black background and a depth estimation buffer is constructed to record depth values. Next, the method recursively builds coarser level images by averaging blocks of pixels using texture mapping. Then, the bounding volume hierarchy of the model database is traversed to determine visibility culling. For each potential occludee, the HOM is traversed to determine if the occludee’s screen-space projection lies completely within the opaque area of the maps and whether or not it is behind the occluders.

Our algorithm is similar to the work described in [HSLM02, YSM03] in that a hardware depth query is used for occlusion culling. Our algorithm differs from [HSLM02] by using hardware depth query on tight-fitting oriented bounding boxes to cull more occluded geometric primitives. In addition, our method’s out-of-core approach (integrated with fast AGP memory) allows data sets that are too large to fit in main memory to be interactively displayed. Similar to Yoon et al. [YSM03] we perform clustering splitting using principle component analysis; however, we utilize space partitioning structures (octree and BSP tree) to divide the vertices into clusters. Additionaly, our preprocessing clustering algorithm operates out-of-core, and our runtime viewing algorithm allows models to be viewed at interactive rates in full resolution.

3. Principal Component Analysis

Principal component analysis (PCA) is a versatile multivariate technique that has many scientific applications, ranging from model-based approaches to algorithmic ideas from neural networks [Jia96, YR00]. Given a set of scattered data points, PCA can be used to find the point set's primary axis. PCA first computes a covariance (or correlation) matrix and then performs an eigen decomposition of this matrix. In two dimensions, the eigenvectors define a local orthogonal coordinate system of an ellipse (an ellipsoid in 3D) induced by the point set being evaluated. Figure 1 shows a 2D point set and the orthogonal axes as determined using PCA.

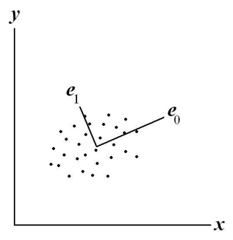


Figure 1: *Primary and secondary axes $\mathbf{e}_0$ and $\mathbf{e}_1$ as determined by PCA applied to a point set $\mathcal{P}$. Axes have their origin at the centroid of $\mathcal{P}$.*

A detailed description of PCA is provided in [Jol86], however, since this analysis is significant to the work described here, a brief description is included. Given a set $\mathcal{P}$ of n points in 3D space, defined as $\mathcal{P} = \{\mathbf{p}_i\}, 1 \le i \le n$, where $\mathbf{p}_i = (x_i, y_i, z_i)$, the covariance (or correlation) matrix $\mathbf{S}$ of $\mathcal{P}$ is defined as

$$\mathbf{S} = \frac{1}{n-1}\left(\mathbf{D}^T\mathbf{D}\right), \tag{1}$$

where $\mathbf{D}$ is given as

$$\mathbf{D} = \begin{pmatrix} x_1-\bar{x} & y_1-\bar{y} & z_1-\bar{z} \\ \vdots & \vdots & \vdots \\ x_n-\bar{x} & y_n-\bar{y} & z_n-\bar{z} \end{pmatrix} \tag{2}$$

and

$$\begin{aligned} \bar{x} &= \frac{1}{n}\sum_{i=1}^{n} x_i, \\ \bar{y} &= \frac{1}{n}\sum_{i=1}^{n} y_i, \text{ and} \\ \bar{z} &= \frac{1}{n}\sum_{i=1}^{n} z_i. \end{aligned} \tag{3}$$

The matrix $\mathbf{S}$ can be factored as $\mathbf{U}^T\mathbf{L}\mathbf{U}$, where $\mathbf{L}$ is diagonal and $\mathbf{U}$ is an orthonormal matrix. The diagonal elements of $\mathbf{L}$ are eigenvalues λ_{max}, λ_{mid}, and λ_{min} of $\mathbf{S}$ (ordered by decreasing absolute values). The columns of $\mathbf{U}$ correspond to the normalized eigenvectors $\mathbf{e}_{max}, \mathbf{e}_{mid}$, and $\mathbf{e}_{min}$. These mutually orthogonal vectors define the three axes of a local coordinate frame positioned at the centroid $\mathbf{c} = (\bar{x}, \bar{y}, \bar{z})$ of $\mathcal{P}$.

PCA can be applied to the set (or subset) of points defining a triangle mesh, yielding axes that define an oriented bounding box for that set of points. This method almost always yields a "tighter" bounding box (one with a smaller volume) when compared to the standard min-max box. Figure 2 compares a point set bounded by both a standard and an oriented bounding box.

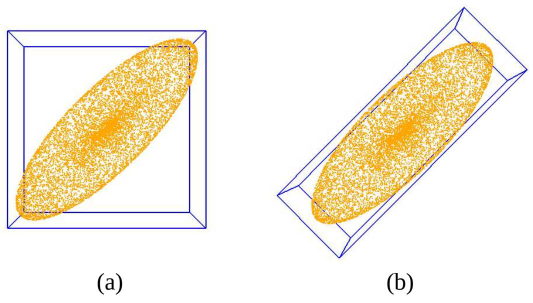


Figure 2: *Comparison of standard and oriented bounding box. Image (a) shows a standard bounding box for a point set $\mathcal{P}$ and image (b) shows an oriented bounding box for $\mathcal{P}$.*

4. Preprocessing Phase

The preprocessing phase begins by computing a spatial subdivision of the mesh to be visualized. In this paper, the mesh consists of vertices and triangles defining the connectivity of the vertices. However, this method can be extended to handle other types of meshes (or volumes) for different types of visualization such as volume rendering. The mesh is recursively partitioned into smaller volumes and an oriented bounding box that closely fits the geometry contained within each volume is computed using PCA. The tree defined by the spatial subdivision and the geometry contained inside each node are stored in separate files for use during runtime. The first file, the subdivision file, stores the spatial subdivision and the oriented bounding box corresponding to each node in the tree. The second file, the data file, stores the mesh geometry, grouped according to subdivision tree node, in the same order as the nodes appear in the subdivision file.

The subdivision file consists of an array of tree nodes. Each node contains the following:

- **Child Identifier**. The index of the first child node.
- **Offset**. The offset into the data file.
- **U, V, W and Origin**. Three orthogonal axes and origin representing the local frame of the oriented bounding box.
- **Width, Height, and Depth**. Scalar dimensions of the oriented bounding box.

Only leaf nodes have associated geometry so the offset is only used if the node is a leaf node. Similarly, the child index is only used if the node has child nodes. The U, V, and W basis vectors and the origin together represent the local frame that describes the oriented bounding box for a node. The node sequence in the file is depth first and each non-leaf node always has the maximum number of children allowed. Thus, at runtime, this file can be accessed quickly by using the appropriate offsets. The data file contains mesh geometry and topology for each leaf node in the subdivision tree. Each leaf node has its own local set of vertices and triangles that index into those vertices. The following information is stored for each leaf node in the subdivision tree:

- **Number of vertices**.
- **Number of triangles**.
- **List of vertices**. A triple of floating-point values.
- **List of triangles**. A triple of indices into the list of vertices.

The subdivision tree is constructed according to a set of user-specified parameters. These parameters are: available memory, maximum tree depth, and maximum number of triangles per leaf node. The maximum tree depth and the maximum number of triangles serve as termination criteria for the tree construction process. The general tree construction algorithm performs two steps:

1. Create a root node having an oriented bounding box that contains the entire mesh.
2. Recursively subdivide the root node and its children until leaf nodes that contain the user-specified maximum number of triangles or the maximum depth are achieved.

The subdivision tree is implemented with an octree and a binary space partition (BSP) tree.

Each octree node is represented physically by an oriented bounding box and is subdivided at a split point (contained inside the box) into eight parallelpiped boxes of varying dimensions. Choosing a good split point location is important since it determines how the triangles are distributed among each of the eight children. Even distribution is desired so that the resulting tree is balanced. The average of the vertices contained within a node determines the split point used to compute the child nodes, see Figure 3.

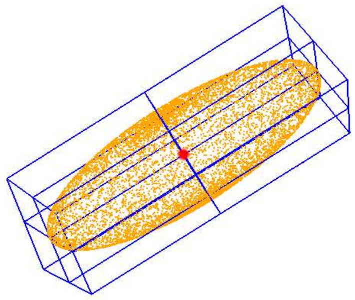


Figure 3: *Oriented bounding box and associated split point in red.*

A binary space partition is obtained by recursively bisecting bounding boxes with a plane. Each BSP tree node stores its split plane and has two children. Each child represents the space “in front of” or “behind” the split plane. The split plane is chosen to be the plane with the least dominant vector of the three principal directions (as determined using PCA) as its normal (i.e., the plane is orthogonal to the longest direction of the oriented bounding box).

It is possible that there may not be enough available memory to hold the entire data set during the tree construction. To handle this situation, a simple memory management scheme is used. A portion of the available memory is used for a least-recently used (LRU) cache that contains vertices. The remaining memory is used for loading triangles for each node. Each node in the subdivision tree has corresponding triangles in the source data file which are loaded as needed.

5. Runtime Phase

At runtime, the subdivision file is used to recreate the subdivision tree, which is traversed in front-to-back order based upon the current view point. For each node visited, its oriented bounding box is tested for visibility using a hardware depth query. If the node is not visible, traversal is terminated and none of the leaf nodes are rendered, thus, saving the time required to render these elements.

For a given view point, beginning at the root node, we determine in which octant (or half space) the view point lies. The dot product between a plane normal and the vector $\mathbf{r} = \mathbf{v} - \mathbf{c}$, where $\mathbf{v}$ is the view point and $\mathbf{c}$ is the split point of a box, is used to determine the side of a plane the view point lies. For a BSP tree, the plane used is the split plane and front-to-back traversal is simply a variant of an in-order binary tree traversal. The child representing the side of the split plane that contains the view point is visited first. For an octree, three planes are tested to determine the traversal order of the children. The local frame for the oriented bounding box implies the three planes that are used. A recursive algorithm using bit-toggling (assuming a lexicographic ordering of the children) is used to specify the traversal sequence.

For large data sets, there may not be enough main memory to load the geometry for every node. A memory management scheme is used to manage the available memory for loading node geometry into main memory. The AGP supports fast, asynchronous access to memory for transfering data to the graphics card independent of the CPU. Rendering performance is enhanced by allocating a chunk of AGP memory and dividing this into smaller fixed-sized chunks (large enough to hold a vertex array for any node). Additional main memory is allocated for holding the vertex and triangle arrays that cannot fit into AGP memory. Two AGP chunks are reserved for an arbitrary vertex array and a triangle array. As nodes are tested for visibility and rendered, two memory chunks are assigned per node, one for the vertices and the other for triangles. AGP chunks are used first, if available, otherwise main memory chunks are used. If there are no memory chunks available (either AGP or main), the two reserve chunks are used for loading and rendering but are not assigned to the node. To take advantage of temporal coherence, memory chunks are replaced using a Least Recently Used (LRU) replacement scheme.

Vertex arrays are used for efficient rendering and reduced function call overhead. Our current implementation uses the NVIDIA extension `glDrawRangeElements` to place vertex arrays in AGP memory and the `glSetFenceNV` and `glFinishFenceNV` extensions for synchronization between the GPU and the CPU. However, we note that future implementations should probably be based upon the recently released `ARB_vertex_buffer_object` extension.

5.1. Hardware Occlusion Query

The hardware occlusion query scan converts a set of graphics primitives (but does not render them to the screen) and determines whether or not any pixels in the frame buffer would be affected if the primitives were actually rendered to the screen. Modern graphics hardware allows multiple occlusion queries to be sent at once and permits other processing to continue while waiting for the results. The query returns the number of pixels that would be affected so that a user can determine whether or not to render based on some defined threshold. An adaptive rendering technique could also be developed that selects a mesh from a hierarchy of meshes based on the number of pixels affected. Our algorithm uses NVIDIA's occlusion query extension and renders all of the contained primitives if one or more pixels are affected and ignores the geometry when no pixel is affected. The occlusion query process has three steps:

1. Disable updates to color and depth buffers
2. Specify the query geometry
3. Obtain and process query results

The result of an occlusion query is not available until the query geometry has finished rasterization which creates a potential for pipeline stalls. Keeping the pipeline as full as possible by submitting multiple queries at a time helps alleviate stalls. It should be noted that it is possible to lose some amount of culling by submitting multiple queries at once if some of the queries are dependent on each other. Submitting queries for geometry that overlaps other geometry within a set of queries can potentially lead to a false invisibility test. (False invisibility tests can cause more than the necessary number of triangles to be rendered, but this does not generate cracks or holes in the rendered image.)

6. Results

Both an octree and a BSP tree spatial partitions were tested for several large meshes. All data sets were obtained from the Stanford 3D Scanning Repository *(http://www-graphics.stanford.edu/data/3Dscanrep/)*. All tests were performed on a Pentium 4 3.2 GHz system with 2 GB of main memory and NVIDIA's GeForce FX 5200. Meshes tested contained from one to 28 million triangles and frame rates were roughly 66 fps to 3 fps, respectively. Table 1 shows the data sets used during the tests. For consistency, a viewing path having 70 to 300 frames was saved for each data set and used to reproduce viewing positions for the different tree implementations and traversal methods. Table 2 shows the average percentages of triangles culled and average frame rates for each data set over the course of this test. Preprocessing requires a few minutes for the smaller data sets and 30 minutes for the Lucy data set (the largest mesh tested).

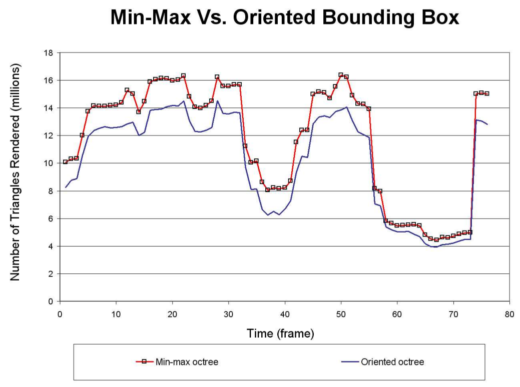


Figure 7: *Performance of occlusion culling using an octree constructed with conventional min-max bounding boxes and another octree constructed with oriented bounding boxes. Graphs indicate the number of triangles rendered per frame. Octrees were constructed on the Lucy data set having a maximum of 5000 triangles per leaf node.*

Figure 4 shows a rendering of the Lucy data set. The left image shows the rendering from the user's view point. At the right, the model is rotated and zoomed to show the back side and the culled regions of the mesh. Frustum culling is automatically included in this method as indicated by the culling of the tip of the torch and the bottom of the torso. Using the octree construction, about 5.66 million triangles (with 80% culling) were rendered in about 0.33 seconds. Figures 5 and 6 show the same for the Buddha and the Dragon data sets.

The main feature of this method is the ability to interactively display meshes that are too large to fit into memory. The raw geometry of the Lucy data set requires over 500 MB of memory. This method can interactively display this data set at up to 5 frames per second using an artificial main memory limit of 270 MB. Another feature is the use of tight-fitting oriented bounding boxes for improved culling. For the Lucy data set, two octrees were constructed and compared: one using conventional min-max bounding boxes and the other using oriented bounding boxes. Both were constructed using the same parameters and each were tested using the same viewing path. The octree having oriented bounding

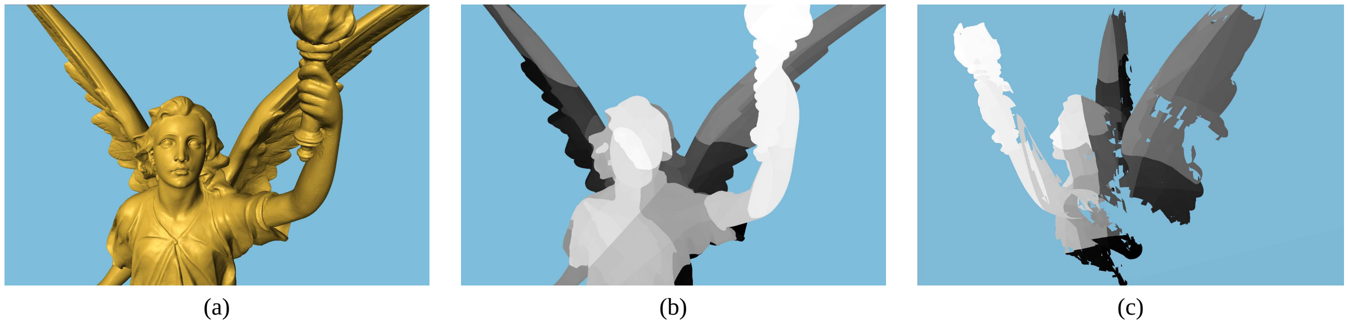


Figure 4: *Rendering of Lucy model using an octree for the spatial partioning. Image (a) shows a lit, shaded rendering of the Lucy model. Image (b) shows the nodes colored in gray scale based upon tree traversal (white being the closest to the viewer). Image (c) shows a side view of the nodes used to render images (a) and (b).*

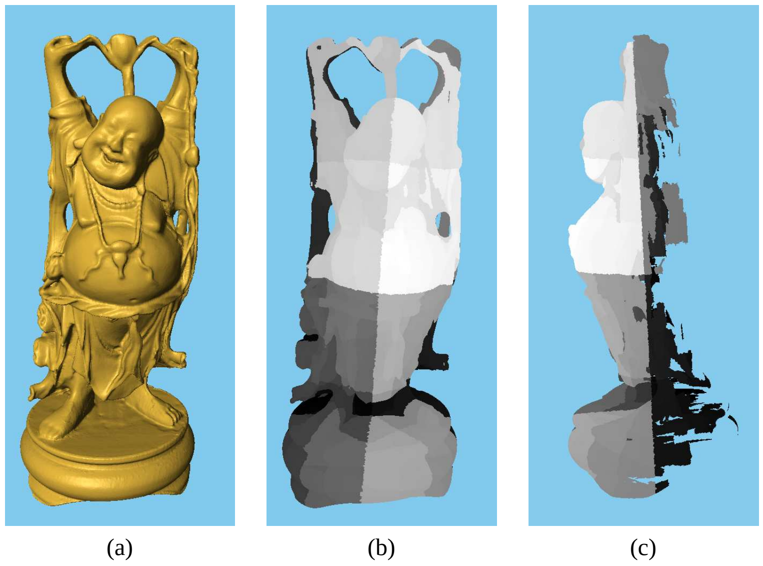


Figure 5: *Buddha model and rendered nodes using an octree for the spatial partioning. Image (a) shows a lit, shaded rendering of the Buddha model. Image (b) shows the nodes colored in gray scale based upon tree traversal (white being the closest to the viewer). Image (c) shows a side view of the nodes used to render images (a) and (b).*

boxes, on average, culled 6% additional triangles and up to 7.5% additional triangles than the conventional octree, see Table 3. Figure 7 shows a graph of the rendered triangle count obtained using both trees.

The traversal method was tested to see how pipeline stalls affect rendering and culling results. Two variations on traversal were implemented for comparison. Variation (a) – denoted in the graphs with asterisk– performs one hardware depth query at a time, for each node, as the tree is traversed. Variation (b) traverses the tree, keeps track of the sequence of nodes and issues the depth queries in groups of nodes. Figures 8 and 9 show the results for both tree types. Variation (a) culls more triangles but variation (b) achieves faster rendering time. This happens because variation (b) issues multiple queries at once. (Culling rate can be affected by overlapping oriented bounding boxes within a set of queries as previously discussed in Chapter 5.) Variation (b) achieves faster rendering time due to reduced pipeline stalls since less time is spent on queries.

When using the same parameters for tree construction, an octree has more leaf nodes than a BSP tree due to the fact that an octree split produces four times as many children. Generally, the leaf nodes (in an octree) store fewer triangles, which leads to smaller bounding boxes, ideal for culling. This is indicated by the graphs shown in Figures 8 and 9 since octrees have better culling results. However, the efficiency of the memory management scheme is adversely affected by the greater number of leaf nodes since

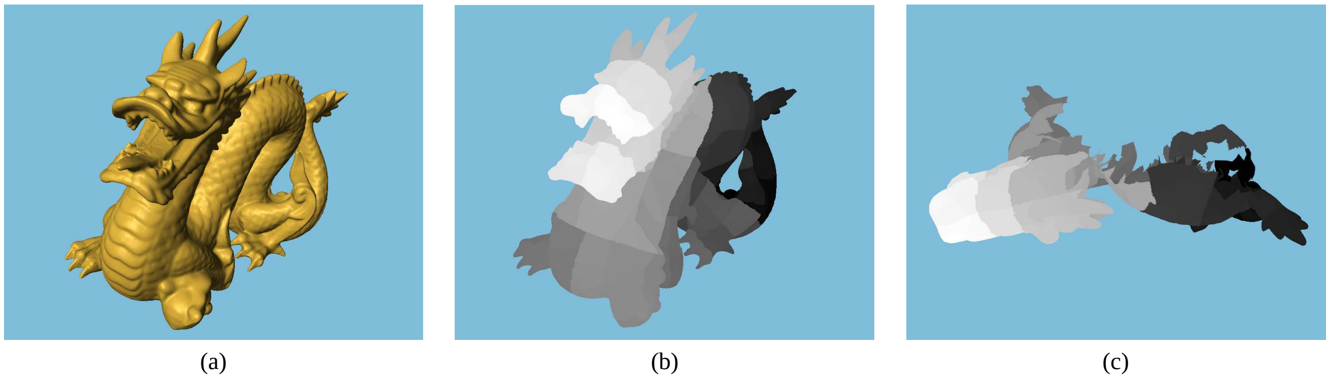


Figure 6: *Dragon model and rendered nodes using an octree for the spatial partioning. Image (a) shows a lit, shaded rendering of the Dragon model. Image (b) shows the nodes colored in gray scale based upon tree traversal (white being the closest to the viewer). Image (c) shows a top view of the nodes used to render images (a) and (b).*

Data set	No. of vertices	No. of triangles	Max triangles per leaf
Happy Buddha	543,652	1,087,716	1000
Dragon	566,098	1,132,830	1000
Lucy	14,027,872	28,055,742	5000

Table 1: *Data set statistics.*

this number is proportional to the number of memory chunks needed. Since fixed-sized memory chunks (as determined by the largest leaf node) are used and there are more nodes with fewer triangles, less memory is actually utilized. This causes more disk access and increases the rendering time. This explains why the BSP tree culls less geometry than the octree but renders faster. A histogram showing the number of nodes having a certain number of triangles, for both octree and BSP tree constructions for the Lucy data set, is shown in Figure 10. The BSP tree has a better distribution since there is a high concentration of leaf nodes containing a high number of triangles. The octree distribution shows most of the leaf nodes containing a few number of triangles. (The remaining data sets had similar results.)

7. Conclusions and Future Work

We have presented a new occlusion culling algorithm for rendering large polygonal models at their full resolution. A spatial subdivision of a given data set is computed and traversed in front-to-back order to perform visibility culling. Our approach integrates tight-fitting oriented bounding boxes (computed by PCA) with an out-of-core rendering approach (using fast AGP memory) for a display system that culls a large number of occluded triangles efficiently. We have tested our system on several large models consisting of several million triangles. The Buddha and Dragon data sets, each containing approximately a million triangles, are rendered at 30-66 frames per second, and the Lucy data set, with 28 million triangles, is rendered at 2-3 frames per second. In addition, with an out-of-core design approach, our algorithm is able to interactively display data sets that are too large to fit into main memory.

Our future work is focused on improving the performance and scalability of both phases of our algorithm. The preprocessing phase can take a long time for very large data sets. The current implementation is file I/O intensive due to memory constraints. A more efficient file I/O management scheme will help reduce required preprocessing time. Additionally we would like to investigate bottom-up tree building algorithms based on local clustering and integrated our preprocessing with streaming mesh formats such as those used in [ILGS03].

Although a large number of geometric primitives can be culled using this method, in some circumstances, there may still be too many visible regions that need to be rendered. A data simplification method or multiresolution analysis is needed to further reduce the primitive count to maintain interactivity. As hinted at Chapter 5, an adaptive rendering technique could be developed to select a mesh from a hierarchy of meshes depending on the number of pixels affected or the distance from a view point.

Data set	Octree		BSP tree	
	Avg. % culled	Avg. frame rate	Avg. % culled	Avg. frame rate
Happy Buddha	49.8%	62.5	41.0%	66.7
Dragon	46.9%	30.1	37.4%	30.8
Lucy	63.9%	1.9	55.6%	2.3

Table 2: *Average percentages of triangles culled and average frame rates for each data set.*

Octree type	Min number rendered	Max number rendered	Avg. rendered	Avg. % culled
Oriented	3,585,050	13,896,857	9,471,981	67%
Standard	3,995,501	15,753,649	10,967,944	61%

Table 3: *Comparison of two different octree constructions for Lucy data set: one with conventional and the other with oriented bounding boxes. The octree with oriented bounding boxes culls, on average, 1.5 million (6%) additional triangles.*

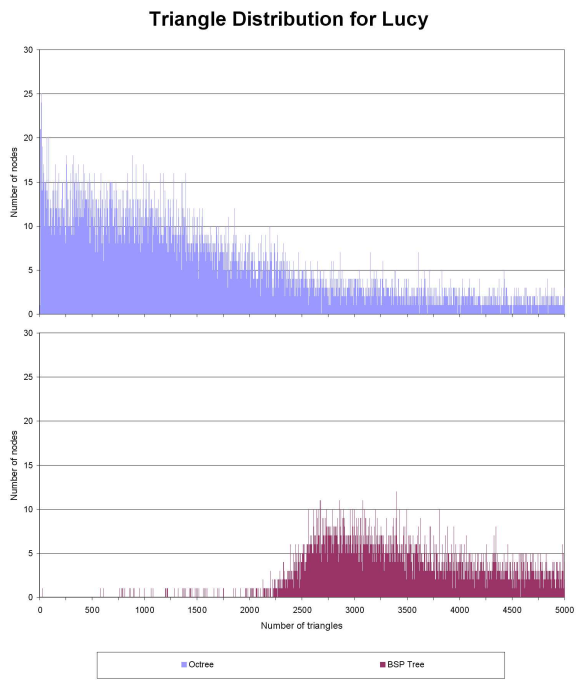


Figure 10: *Triangle distribution for Lucy data set. The top image shows the distribution for the octree and the bottom for the BSP tree. This graphs plot the triangle distributions of the leaf nodes for each tree.*

Acknowledgements

This work was supported by the National Science Foundation under contract ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the National Institute of Mental Health and the National Science Foundation under contract NIMH 2 P20 MH60975-06A2; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; and the Lawrence Berkeley National Laboratory. We thank the members of the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

References

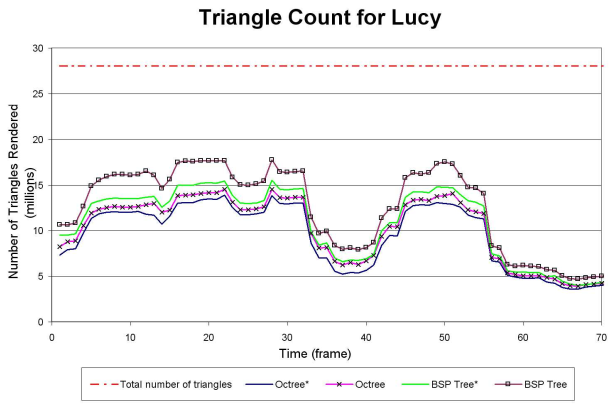
[ARB90] AIREY J., ROHLF J., BROOKS F.: Towards image realism with interactive update rates in complex virtual building environments. *Symposium on Interactive 3D Graphics* (1990), 41–50.

[ASVNB00] ANDÚJAR C., SAONA-VÁZQUEZ C., NAVAZO I., BRUNET P.: Integrating occlusion culling and levels of detail through hardly-visible sets. *Computer Graphics Forum 19*, 3 (August 2000), 499–506.

[ESSS01] EL-SANA J., SOKOLOVSKY N., SILVA C.: Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization* (2001).

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical z-buffer visibility. *Computer Graphics Proceedings, Annual Conference Series* (1993), 231–240.

[HSLM02] HILLESLAND K., SALOMON B., LASTRA



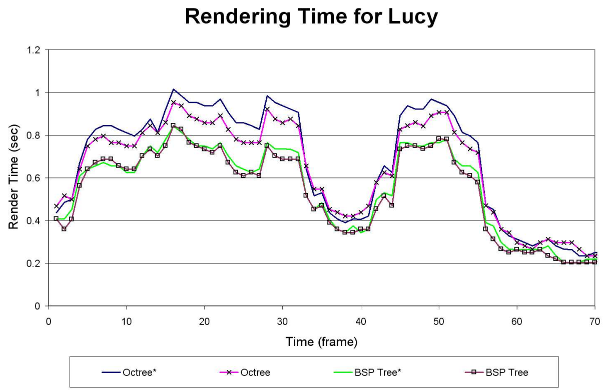
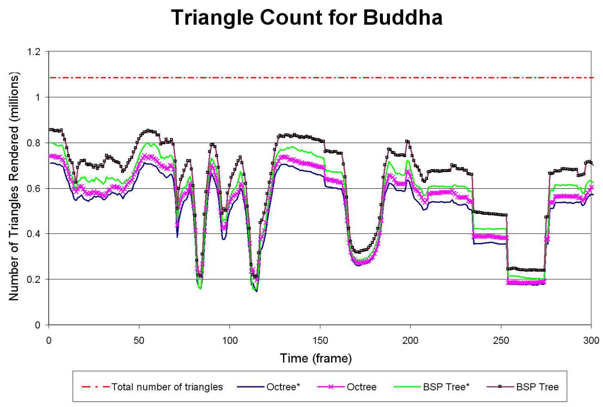


Figure 8: *Left: The number of triangles rendered at each frame for the Lucy data set. The octree culls more triangles than the BSP tree. Right: Comparison of the rendering time per frame for the octree and the BSP tree for Lucy data set. Asterisk denotes variation (a) traversal method.*



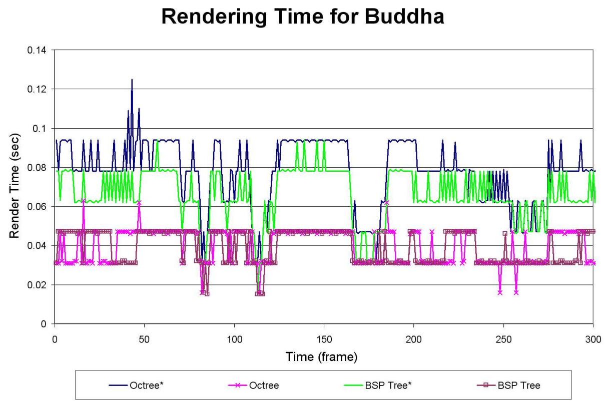


Figure 9: *Left: Comparison of the number of triangles rendered for each frame for Buddha data set. Right: Comparison of the rendering time per frame of the octree and BSP tree for Buddha data set. Asterisk denotes variation (a) traversal method.*

A., MANOCHA D.: *Fast and Simple Occlusion Culling using Hardware-Based Depth Queries*. Tech. Rep. UNC-CH-TR02-039, Computer Science Department, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 2002.

[ILGS03] ISENBURG M., LINDSTROM P., GUMHOLD S., SNOEYINK J.: Large mesh simplification using processing sequences. *Proc. of IEEE Visualization* (2003).

[Jia96] JIANG Q.: *Principal Component Analysis and Neural Network Based Face Recognition*. Master's thesis, University of Chicago, 1996.

[Jol86] JOLLIFFE I.: *Principle Component Analysis*. Springer-Verlag, New York, NY, 1986.

[YR00] YEUNG K., RUZZO W.: *An empirical study of Principal Component Analysis for clustering gene expression data*. Tech. Rep. UW-CSE-2000-11-03, University of Washington, Seattle, Washington, 2000.

[YSM03] YOON S.-E., SALOMON B., MANOCHA D.: Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *Proceedings of IEEE Visualization 2003* (2003).

[ZMHI97] ZHANG H., MANOCHA D., HUDSON T., III K. H.: Visibility culling using hierarchical occlusion maps. *Proceedings of SIGGRAPH* (August 1997), 77–88.