# Preprocessing Occlusion For Real-Time Selective Refinement

Fei-Ah Law* Tiow-Seng Tan†

National University of Singapore‡

## Abstract

Visibility computation and level of detail modeling are two important components of efficient scene rendering algorithms. Both aim to lessen the graphics load by lowering polygon count. This paper presents a novel framework that integrates the two techniques to optimize rendering. To improve the efficiency of occlusion computation for densely tessellated models, we introduce the use of simplification to automatically deduce virtual occluders. This paper illustrates the use of the technique to efficiently preprocess occlusion for outdoor scenes. Using virtual occluders to pre-compute visibility, only visible surfaces may be refined for real-time selective refinement. Compared to selective refinement without incorporating occlusion culling, our implementation of the framework demonstrates significant polygon count reduction and speedup in frame rate.

**CR Categories and Subject Descriptors:** I.3.5 [Computer Graphics]: Computer Geometry and Object Modeling - surfaces and object representations, object hierarchies.

**Additional Keywords:** occlusion culling, level of detail modeling, occlusion preserving simplification, cell, selective refinement.

## 1 INTRODUCTION

Visibility computation and level of detail (LOD) modeling are two important graphics acceleration techniques used to reduce the computational requirement of rendering complex 3D scenes. There are three kinds of visibility culling: view frustum culling, back-face culling and occlusion culling. These techniques avoid processing invisible portions of a scene by discarding polygons that are off-screen, oriented away from the viewer or occluded respectively. LOD modeling lowers polygon count by replacing distant models with appropriate approximations without significant loss in image quality.

Visibility culling and LOD modeling complement each other to achieve the common goal of speeding up visualization through polygon count reduction. For objects that are visible but far away from the viewpoint, simplification can be used to generate substituting models with low LODs. However, objects near the viewpoint but are invisible will still be represented with high LODs. In this case, visibility culling is helpful in lowering the polygon count. Some recent work on selective refinement algorithms takes into consideration view frustum culling and back-face culling [8, 10]. Yet, none has incorporated LOD modeling with occlusion culling.

This paper proposes a framework that accelerates interactive visualization through both occlusion culling and LOD modeling. The contributions of this paper are:

- It introduces the use of simplification to automatically deduce big virtual occluders from densely tessellated models to have more efficient and effective occlusion culling.
- It presents a framework to pre-compute occlusion information (with no assumption on the convexity of occluders) to guide real-time selective refinement in achieving significant polygon count reduction.
- It offers a uniform treatment on the cell-based visibility preprocessing for architectural models as well as outdoor environments.

The rest of the paper is organized as follows. Section 2 reviews some related work and Section 3 gives an overview of our framework. Section 4 explains how simplification helps in yielding virtual occluders. Next, Section 5 describes the occlusion preprocessing algorithm and Section 6 discusses the steps to incorporate selective refinement with occlusion culling. Section 7 presents our experimental results. Lastly, Section 8 concludes the paper.

## 2 RELATED WORK

### 2.1 Occlusion Culling

We are interested in practical means of speeding up rendering by performing efficient software checking to overestimate the set of visible polygons. The general approach of these conservative visibility algorithms is to select some polygons to act as occluders and check if they occlude any objects as seen from the viewer. To reduce the cost of checking, occludees (culled objects) are usually approximated by bounding volumes.

Teller and Sequin [11] present an object space algorithm for fast architectural walkthrough system which divides a database into cells, roughly corresponding to rooms in a building. Cell-to-cell visibility can be computed in a preprocessing phase.

Conservative visibility testing for general polygonal models in [3] performs object space culling by introducing the notion of separating and supporting planes. Visibility testing is performed between a pair of occluder and occludee by determining the regions of partial and full occlusion. A kD-tree is used to organize a scene. During run-time, the algorithm dynamically selects a subset of convex objects near the viewpoint as occluders and recursively applies

*Email: lawfeiah@alum.comp.nus.edu.sg

†Email: tants@comp.nus.edu.sg

‡School of Computing, National University of Singapore, Lower Kent Ridge Road, Singapore 119260. This work is supported by the National University of Singapore under grant RP960618.

the conservative visibility test to the nodes of kD-tree (which are treated as occludees).

Cohen-Or *et al.* [2] outline a conservative visibility preprocessing method for outdoor environment by partitioning the viewspace into cells. A conservative superset of visible objects is computed for each cell by searching for a strong occluder for each object such that it cannot be seen from any point in the cell.

In the case of image space algorithms, the fundamental idea is to perform visibility computation for each frame by scan converting some potential occluders and checking if the projections of the bounding volumes of occludees fall entirely within the image area covered by the occluders. Hierarchical Z-buffer algorithm [5], hierarchical tiling algorithm [6] and hierarchical occlusion maps [14] are examples of such recent developments.

## 2.2 LOD Modeling

Many recent work on simplification emphasizes on achieving continuous LOD with view-dependent selective refinement for real-time applications. LOD hierarchies are used to organize models with various LODs in a hierarchical manner to achieve selective refinement. The real-time continuous levels of detail simplification algorithm in [9] uses quadtree data structure to structure a uniformly-grided height field into blocks. During visualization, discrete LOD is first determined for each block and vertices within each block are further considered for removal to obtain continuous resolution. Grouping data into blocks also facilitates efficient view frustum culling.

An algorithm for performing simplification dependent on viewing direction, lighting and surface orientation is proposed by Xia and Varshney [12]. A merge tree is first constructed off-line and used at run-time to guide the selection of appropriate triangles for display.

Hierarchical dynamic simplification works by clustering vertices together in a hierarchical fashion [10]. Nodes of the vertex tree to be collapsed or expanded are continuously chosen based on their screen projected size. Thus, the entire system operates dynamically, retessellating the scene continuously as the view position shifts.

Hoppe presents the idea of achieving selective refinement for progressive meshes by constructing vertex hierarchies based on edge collapse information [7, 8]. Selective refinement according to changing view parameters is done by moving the vertex fronts up and down through the hierarchies. A bounding volume and a normal cone are associated with each vertex for view frustum and back-face culling respectively.

## 3 PROPOSED FRAMEWORK

This section gives an overview of the proposed framework which assists real-time selective refinement with a cell-base visibility preprocessing to accelerate real-time display. The steps of the framework is outlined in Figure 1.

Given a scene, our framework constructs a cell hierarchy which divides the viewspace into regions. Simplification is applied to models of the scene to generate LOD hierarchies. As each node of the LOD hierarchies corresponds to some part of the scene, it can be considered as occludee for occlusion culling. Thus, visibility for each cell is computed by finding those nodes of the LOD hierarchies that are occluded for all the viewpoints in the cell.

As considerable amount of time is required to process many small polygons for occlusion culling of densely tessellated models, we alleviate the problem by introducing the use of simplification to generate approximated polygons. These polygons are ensured to be occlusion preserving to act as virtual occluders through our proposed edge error correction method. By having fewer but larger

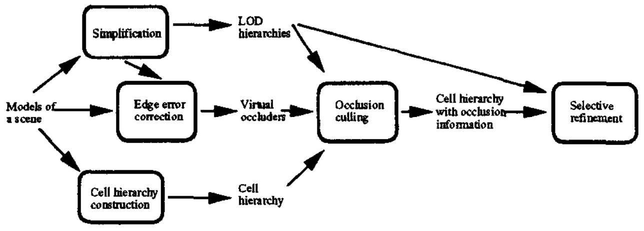


Figure 1: Steps of our framework.

occluders, computational efficiency and effectiveness can be improved for any occlusion culling algorithm.

The pre-computed visibility information is used to guide selective refinement during run-time. In general, selective refinement algorithms maintain a list of current active nodes of the LOD hierarchies to exploit view coherence. For each frame, the active list is traversed to evaluate the refinement criteria for each active node. To include occlusion as one of the refinement criteria, we first obtain a list of occluded nodes by locating the viewpoint in the cell hierarchy. The list of occluded nodes is then used to check against the active list of the LOD hierarchies to determine the visibility of each active node. Only those unoccluded nodes are considered for refinement.

## 4 VIRTUAL OCCLUDERS

The demand for accurate graphical representation leads to an increase in the use of densely tessellated models. However, processing many small polygons for occlusion checking is time consuming and ineffective for both object space and image space algorithms. Potential occlusion may not even be identified as there is no big occluder. As a result, it is of utmost importance to be able to replace each occlusive polygon set with a large, simplified virtual occluder.

## 4.1 Occlusion Preserving Simplification

Simplification provides an efficient automatic means of obtaining large polygons which closely resemble the original model. Current polygonal simplification algorithms can be used to reduce polygon count of complex models with tight error bounds. However, none has occlusion preserving as one of the criteria in simplification.

A simplified polygon is *occlusion preserving* and thus can act as a virtual occluder if it gives correct visibility information. For any viewing position, let the set of objects occluded by the simplified polygon be $A$ and the set of occludees found using the corresponding original occlusive polygon set be $B$. The condition $A \subseteq B$ should always hold.

A polygon generated by a simplification algorithm cannot act as a virtual occluder if certain part of the polygon is larger in extent than the original surface and gives more occlusion than the original. Therefore, to guarantee conservative approximation of occlusion, the basic idea is to modify the simplification algorithm or approximated output such that the screen projection of the final polygons obtained are within that of the original model as seen from the viewer. Note that to satisfy the condition, the simplification process must not fuse holes of the original model.

## 4.2 Possible Attempts

An intuitive solution to obtain occlusion preserving simplification is to make sure that the final approximation obtained is within the original. That is, the simplified model is entirely enclosed by the original mesh. For simplification adopting decimation approach, we can enforce the constraint at each step of the algorithm. Each transformation (e.g. vertex removal or edge collapse) is legal only if the newly formed polygons are still within the original surface. However, considerable amount of checking may be required and restricting the possible choice of transformation may adversely affect the fidelity of the simplified model.

A more efficient method is to first simplify a surface within a specified error which bounds the deviation of an approximation from the original. Then, the resultant model is shrunk by the amount of the specified error. Another equivalent alternative is to shrink the original surface first before performing simplification. For example, with simplification envelopes technique, occlusion preserving simplification can be achieved by forcing the outer envelope to be the original mesh [13]. In either way, the final result may be unnecessarily small as the specified error only indicate the overall maximum error of the simplified model and thus cannot distinguish between positive and negative errors nor identify the parts that are responsible for the error.

Ideally, instead of adjusting the whole simplified model, only those parts that violate the occlusion preserving constraint should be corrected. One implementation is to compare each simplified polygon against the original such that only polygons having some parts outside the original mesh are identified and adjusted. However, inspecting every polygon against the original model for error checking is computationally expensive. Therefore, next section presents a general approach which adjusts only edges of a simplified polygon and does not restrict the whole polygon to be within the original model.

## 4.3 Edge Error Correction

We propose to only correct error of edges of simplified polygons to yield valid virtual occluders. *Error of an edge* of a simplified polygon is the maximum positive deviation of the edge from the original mesh in the direction of the edge's normal. Normal of an edge is the average of the normals of two or less adjacent polygons sharing the edge. To guarantee occlusion preserving, an edge is shifted in the direction opposite to its normal by the amount of error. This approach is based on the observation that only edges are sufficient to determine visibility. This is because edges define the extent of a polygon which in turns defines how much the polygon can occlude.

For a simplified polygon to be occlusion preserving, its screen space projection, $S$, must be a subset of the screen space projection, $M$, of the original model for any given viewpoint. This condition, $S \subseteq M$, holds if the silhouette of $S$ is within $M$ and $S$ does not intersect with any hole inside $M$.

To obtain simplified polygons that are occlusion preserving, edge error correction can be applied to polygons generated from existing simplification algorithms. This is because majority of the existing simplification algorithms generate approximations with small deviations from original models, usually through local modifications. Thus, we can visualize a positive and a negative offset surfaces from an original model such that they enclose both the original and simplified model. There may be variation of distance between the two offset surfaces due to varying deviation error of the simplified surface from the original at different parts. For a simplified polygon, by shifting edges which are between the original and positive offset surfaces to correct the errors, it is guaranteed that the silhouette of $S$ is within $M$. In addition, given that the edges of the simplified polygon are within the original, it is impossible for $S$ to cover any hole of $M$ as this will violate the fact that the simplified polygon falls completely within the volume enclosed by the two offset surfaces. Hence, occlusion preserving constraint is enforced.

After edge error correction, an occluder may no longer be a valid polygon as moving edges may change the connectivity and the edges may not be coplanar. However, this does not affect the

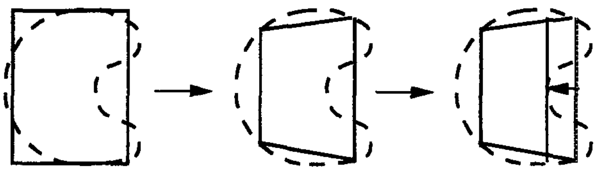


Figure 2: An example of edge error correction. A cross section view of a 3D original object and its simplified model with the viewpoint directed into the paper. Silhouette of original object is drawn in dashed line and that of the simplified one in solid line.

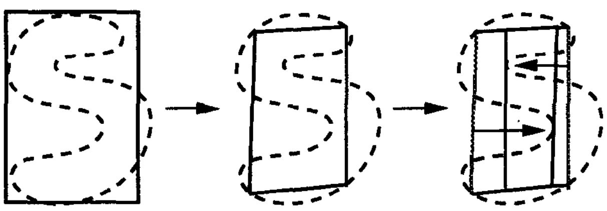


Figure 3: A case where a simplified polygon is "flipped" after edge error correction.

correctness of visibility results. This is because each edge contributes to defining the extend of occlusion independent of other edges. Thus, occlusion culling performs computation for each edge separately without requiring other knowledge such as whether the edges are coplanar or connected.

Our approach has several advantages. It works with many simplification algorithms which produce simplified surfaces that can be enveloped by positive and negative offset surfaces of the original models. It gives largest possible occluders for visibility checking as compared to the other alternatives mentioned in Section 4.2. This is because it does not restrict simplified polygons to be completely inside the original surface and keeps those simplified polygons which are already occlusion preserving intact. More importantly, by replacing many small polygons with a few large virtual occluders, it can greatly reduce the amount of computation for both image space and object space occlusion culling algorithms.

**Implementation.** The steps of edge error correction are as follows. Vertices of the simplified model are shifted in the reverse direction of their normals such that all the vertices are on or within the original model. We are done if the original model is convex. Otherwise, error of each edge is then obtained by finding the maximum distance the edge is outside the original surface on the plane containing the edge and its normal. Finally, each edge is corrected by shifting it along the reverse direction of its normal by the amount of error. Figure 2 shows a cross section view of a simple 3D example to illustrate the steps of edge error correction. Note that in general, simplified polygons after correction are not necessary completely within the original models. In addition, not all simplified polygons after edge error correction are good occluders. Figure 3 gives a case (in cross section view of 3D) where a simplified polygon is "flipped" after adjusting the edges and thus cannot occlude any objects — it is still considered as a valid occluder and does not give wrong result.

In using edge collapsing simplification, error of an edge can be decided by tracking the set of related original edges. Set of original edges for an original vertex are its outgoing edges. When two vertices are collapsed into one, their set of original edges are combined as well. The set of original edges for an edge of the simplified mesh is thus the union of the sets of its two endpoints. These original edges are used to check against the simplified edge to determine the error. As maximum deviation of a simplified edge from the original surface can only occur at the edges, using original edges for edge error checking is sufficient.

## 5 VISIBILITY PREPROCESSING

To minimize the computation needed during run-time, obtaining visibility information as much as possible with a preprocessing stage is justifiable. Cell division for architectural walkthrough to pre-compute cell-to-cell visibility is an excellent example [11]. Our framework applies the same concept to outdoor environments with large occluding objects. An outdoor scene is partitioned into cells and visibility of each cell is calculated off-line. Locating the cell containing the viewpoint during visualization gives the details about parts of the scene that are visible or occluded.

Cohen-Or *et al.* present a similar idea of viewspace partitioning to pre-compute conservative visibility for outdoor scenes [2]. Our framework differs by incorporating simplification to accelerate visibility preprocessing and achieve selective refinement based on occlusion information.

### 5.1 Cell Hierarchy

For visibility checking of building interiors, it is reasonable to treat each room as a single cell with walls as natural occluders. However, due to the lack of apparent cell structure of outdoor environments, defining rules for cell identification is not as straight-forward. For city walkthrough with architectural models appear to "randomly" sitting on a terrain, there is no distinct boundary for cell identification.

Ideally, an outdoor scene should also be subdivided into cells according to occlusion information. Nearby regions with same visibility sets are grouped together to form one cell. However, it is not practical to have such partitioning based on precise visibility information. This is because results of cell division is highly scene-dependent. As such, there is no control over the shape or size of a cell. Locating a viewpoint in cells with irregular shape may require substantial checking (e.g. a cell may be represented as a many-sided non-convex polygon). Hence, controlled viewspace partitioning is preferred. That is, cell identification is determined before performing visibility computation rather than using visibility information to help the partitioning.

With controlled viewspace partitioning, although visibility information can only be encoded conservatively, the main advantage is simplicity. It gives structured organization of information which assists data enquiry and viewpoint locating. Besides, hierarchical data structure for cell organization can be conveniently employed to exploit spatial coherence.

Controlled viewspace partitioning can be performed by employing hierarchical spatial data structure such that each node represents a cell. The choice of the data structure for organizing the cells is application-dependent. For architectural walkthrough, it is best to have a cell hierarchy that divides a building into cells according to levels and then rooms for each level. For walkthrough of outdoor environment, controlled subdivision with data structure like BSP tree or quadtree is more appropriate. If an urban walkthrough restricts the viewer to move along the roads, then cells should be associated with the roads only. 3D spatial partitioning data structure such as octree or kD-tree can be used for general flythrough application.

With a cell hierarchy, next section discusses the occlusion preprocessing phase to determine the visibility for each cell. To capture the computed visibility information, each cell is associated with an occludee set. The occludee set of a cell specifies the occludees if the viewpoint is located in the cell.

### 5.2 Occlusion Culling

Existing occlusion culling algorithms are usually targeted for dynamic checking during run-time. Occluders are selected based on the viewer's location and then occludees are identified. To allow visibility preprocessing without any specific viewpoint in mind, there is a need of systematically choosing occluder and occludee pairs.

Our occlusion preprocessing uses virtual occluders obtained from occlusion preserving simplification as described in Section 4. For LOD hierarchies which are used to represent objects for selective refinement algorithms, a bounding volume is usually associated with each node of the hierarchies for view frustum culling (e.g. quadtree for terrain [9] and vertex hierarchies for progressive mesh [8]). Since bounding volume of a node bounds part of the original object covered by the node and its descendants, it can be treated as an occludee. As we go down the levels of an LOD hierarchy, the bounding volumes of nodes correspond to smaller and smaller parts of an object. In this way, hierarchical culling of objects is possible.

A polygon can act as an occluder of an object if the object is completely inside the negative half-space of the plane of the polygon, that is, the polygon is oriented away from the object. Given an occluder $O$ and an occludee $X$, the viewpoint is in the full occlusion region if $X$ is completely occluded by $O$, as illustrated in Figure 4 (a top view of 3D) where region 1 is the full occlusion region.

**Implementation.** The list of simplified polygons from a low LOD of each object are used as virtual occluders. Edge error correction is performed for these virtual occluders to enforce occlusion preserving constraint. To identify the occluders for each node of the LOD hierarchies (which are treated as occludees), each node is associated with a set of occluders. This is done by traversing the LOD hierarchies top-down for each occluder. For each path of the LOD hierarchies, the occluder is added to the occluder set of the highest level node if the occluder is oriented away from the bounding volume of the node. In this way, all the occluders of a node is the union of its occluder set with all the occluder sets of its ancestors. During occlusion culling, a list of occluders is accumulated while we traverse down each path of the LOD hierarchies. As occluders may share common edges, adjacent occluders are joined to form larger occluders to take advantage of the combined occlusion effect.

To compute the visibility information for the cell hierarchy, occlusion testing is recursively applied to nodes of each LOD hierarchy through a pre-order traversal. For each node of a LOD hierarchy, full occlusion region is determined for each of its virtual occluder. Cells in a full occlusion region are found by traversing the cell hierarchy top-down such that only highest possible cells in the cell hierarchy are identified with their occludee sets updated accordingly. To allow efficient searching and avoid duplication of cells found when we go from a node of an LOD hierarchy to its descendants, we keep track of the cells which have already been identified. For the example in Figure 4, cells in region 1 are identified with their occludee sets updated. When children of $X$ are recursively processed, only cells in region that has not been considered are searched. For instance, only occludee sets of cells in region 2 are updated for the lower left child of $X$.

### 5.3 Non-convex Occluders

Full occlusion region can be determined by object space occlusion culling algorithm using supporting planes [3]. A supporting plane is constructed for each edge of an occluder and a vertex of an occludee. It is oriented in a way such that both the occluder and occludee lie on positive half-space of the plane (e.g. In Figure 4,

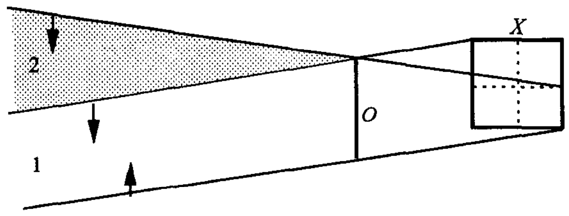


Figure 4: Occlusion culling of an occluder $O$ and an occludee (quadnode) $X$.

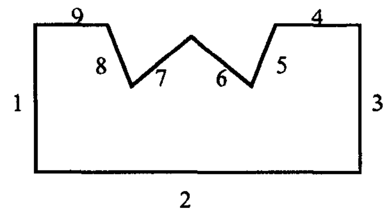


Figure 5: A non-convex occluder.

the two lines that define region 1 correspond to two 3D supporting planes). A viewpoint *satisfies* a supporting plane iff it is inside the plane's positive half-space. For any convex occluder, $O$ with $n$ edges, full occlusion region is the region where all the supporting planes are satisfied. Let $p_e$ be the supporting plane of edge $e$, where $1 \le e \le n$. A viewpoint $v$ is in the full occlusion region iff $\mathcal{F}(p_1, v) \wedge \mathcal{F}(p_2, v) \wedge \cdots \wedge \mathcal{F}(p_n, v)$, where $\mathcal{F}(p_e, v)$ returns *TRUE* if $v$ satisfies $p_e$.

The use of supporting planes to enclose the region that corresponds to full occlusion was originally designed for convex occluders only. We found that the algorithm can be extended to handle non-convex occluders as well. Assume edge $x$ of $O$ is replaced with edge $a$ and edge $b$, and the internal angle formed by $a$ and $b$ is greater than $\pi$. In this case, the occludee can only be seen if the viewpoint is in negative half-spaces of both $a$ and $b$'s supporting planes. That is, occlusion occurs when the viewpoint satisfies either the supporting plane of $a$ or $b$. Thus, the expression for full occlusion checking becomes $\mathcal{F}(p_1, v) \wedge \cdots \wedge \mathcal{F}(p_{x-1}, v) \wedge (\mathcal{F}(p_a, v) \vee \mathcal{F}(p_b, v)) \wedge \mathcal{F}(p_{x+1}, v) \wedge \cdots \wedge \mathcal{F}(p_n, v)$. The is done by substituting $\mathcal{F}(p_x, v)$ with $(\mathcal{F}(p_a, v) \vee \mathcal{F}(p_b, v))$.

In general, given a planar occluder $O$ with $n$ vertices ordered in counterclockwise direction, we can obtain the expression which determines a full occlusion region with respect to $O$ as follows.

**Step 1:** Construct polygon $A_1$, with $m_1 \le n$ vertices, which equals to the convex hull of the $n$ vertices. The expression $E_1$ for full occlusion region with respect to $A_1$ is defined by the conjunction of terms $\mathcal{F}(p_e, v)$ for all the edges $1 \le e \le m_1$. In each of the subsequent step $l$, $A_{l-1}$ and $E_{l-1}$ from the previous step is modified to form $A_l$ and $E_l$.

**Step k** : Assume we have a correct expression $E_k$ to determine the full occlusion region for $A_k$, and $m_k < n$ (otherwise done).

**Step k+1:** **Case 1:** *Edge $x$ of $A_k$ with endpoints vertex $i$ and vertex $j$ where $i$ precedes $j$ in counterclockwise direction, and vertices in between $i$ and $j$ are on the left side of edge $x$.*

Construct a convex hull $C$ with all the vertices from $i$ to $j$. $A_{k+1}$ is obtained by removing edge $x$ from $A_k$ and adding all the edges of $C$ except edge $x$ to $A_k$. As all the newly added edges of $A_{k+1}$ form internal angles greater than $\pi$, $E_{k+1}$ is obtained by substituting the term $\mathcal{F}(p_x, v)$ in $E_k$ with $(\mathcal{F}(p_s, v) \vee \cdots \vee \mathcal{F}(p_t, v))$, where edges $s$ to $t$ are the newly added edges.

**Case 2:** *Edge $x$ of $A_k$ with endpoints vertex $i$ and vertex $j$ where $i$ precedes $j$ in counterclockwise direction, and vertices in between $i$ and $j$ are on the right side of edge $x$.*

Construct a convex hull $C$ with all the vertices from $i$ to $j$. $A_{k+1}$ is obtained by removing edge $x$ from $A_k$ and adding all the edges of $C$ except edge $x$ to $A_k$. As all the newly added edges of $A_{k+1}$ form internal angles smaller than $\pi$, $E_{k+1}$ is obtained by substituting the term $\mathcal{F}(p_x, v)$ in $E_k$ with $(\mathcal{F}(p_s, v) \wedge \cdots \wedge \mathcal{F}(p_t, v))$, where edges $s$ to $t$ are the newly added edges.

Since at least one vertex is added to $A_l$ in each step, the construction terminates in less than $n$ steps. It is not difficult to see that a proof by induction follows naturally from the construction (with careful consideration of the combined regions covered by supporting planes of newly added edges and that of their neighbors from one step to another).

Figure 5 gives an example of non-convex occluder, the expression for full occlusion testing is $\mathcal{F}(p_1, v) \wedge \mathcal{F}(p_2, v) \wedge \mathcal{F}(p_3, v) \wedge \mathcal{F}(p_4, v) \wedge (\mathcal{F}(p_5, v) \vee (\mathcal{F}(p_6, v) \wedge \mathcal{F}(p_7, v)) \vee \mathcal{F}(p_8, v)) \wedge \mathcal{F}(p_9, v)$.

In practice, an occluder may be formed from joining polygons which are not coplanar. In this case, after computing the supporting planes for all the silhouette edges, we can take a cross section of the supporting planes to give a planar occluder for deriving the expression for full occlusion region.

## 6 VISUALIZATION

After the occlusion preprocessing phase, visibility of each cell is determined. To allow efficient selective refinement to take into account the pre-computed visibility information, we assign to each node of the LOD hierarchies an ID. The IDs are numbered through pre-order traversal of the LOD hierarchies. Each node also stores the maximum ID among its descendants. In this way, it is easy for any node to recognize if another node is one of its descendants, as all its descendants' ID must be between the node's ID and the maximum ID.

Given a viewpoint, the cell hierarchy is traversed to find the cells from the root to the leaf that contain the viewpoint. Occludee sets of these cells are merged to give a final list of occludees, $F$. Thus, $F$ contains the information about the nodes of the LOD hierarchies that are being occluded for the current viewpoint. By maintaining both $F$ and the active list of the LOD hierarchies to be sorted in increasing order of IDs, we can traverse the two lists together for selective refinement. Active nodes which themselves or some of their ancestors appear in $F$ are classified as invisible and should not be refined. In this way, selective refinement can be guided by $F$ such that only visible parts of a scene may be refined while invisible portions are coarsen. Figure 6 gives an illustration of the idea. Without lose of generality, a binary tree is used as an LOD hierarchy for simplicity. Assume that the active list is $\{4, 5, 6, 11, 12, 13\}$. Given a viewpoint, $F$ is found to be $\{3, 8, 12\}$. Thus, the list of visible active nodes is $\{6, 11, 13\}$.

Our approach can be incorporated with any selective refinement algorithms utilizing some form of LOD hierarchies. For each frame, each node of the active list is checked if it is being occluded by simple ID comparisons in addition to view-frustum culling and back-face culling. Only visible nodes are further considered for refinement based on some screen-space geometric error metrics. By having an occlusion preprocessing phase and performing culling in a hierarchical fashion, minimum effort is required to find all the highest level nodes that are being occluded during run-time. As a result, selective refinement can easily take into consideration visibility information to achieve high polygon count reduction.

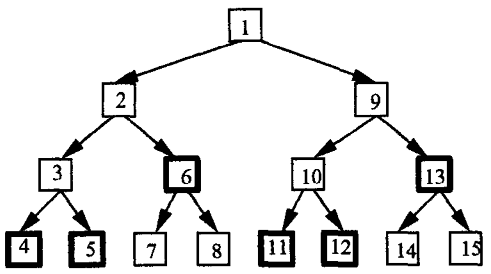


Figure 6: An LOD hierarchy with nodes numbered through a pre-order traversal. Thick-lined boxes represent active nodes.

## 7 EXPERIMENTAL RESULTS

We have experienced with our framework for terrain walkthrough application on an SGI Indigo$^2$ workstation with a 250 MHz MIPS R4400 CPU/R4000 FPU and 128 MB of main memory. Our testing used an edge collapsing simplification [4]. Vertex hierarchies were built for selective refinement as described by [8]. Our preliminary experiments used five datasets, each consists of a terrain model and some buildings sitting on it as occluding objects. Details of each dataset such as the number of polygons of the models, original occluders and virtual occluders obtained from simplification are summarized in Table 1. The total time used to simplify the terrain and occluding objects are also shown. A quadtree was used as cell hierarchy. Table 2 lists the number of levels of the cell hierarchy and statistics for visibility preprocessing.

For each dataset, sample running time was measured for a walkthrough of 1000 frames without and with occlusion culling (Table 2). In both cases, view frustum culling was enabled. For all the experiments, there are significant speedup in running time for walkthrough with occlusion culling. Dataset 4 and 5 have the same terrain model with dataset 5 having a denser set of occluding objects. We experimented the two datasets with different number of levels of cell hierarchy for the same walkthrough path.

The number of polygons rendered and frame time in the walkthrough of datasets 1 and 3 are plotted in Figure 7 and 8. The graphs show that both the polygon count and frame time are greatly reduced with occlusion culling (lower curves) as compared to without occlusion culling (upper curves). Figure 9 and 10 each shows top and 3D views of an adaptively refined terrain without and with occlusion culling for an instantaneous viewpoint. For each top view, the terrain mesh is drawn in red, buildings are in blue and two side planes of the view frustum are in yellow. With occlusion culling, the terrain meshes are much coarser for parts that are occluded though inside the view frustum.

## 8 CONCLUSION

This paper presents a framework that incorporates LOD modeling with occlusion culling. We have shown the technique of using simplification with edge error correction in deducing big virtual occluders for efficient occlusion culling. With controlled cell subdivision for outdoor scenes, off-line occlusion culling is performed using virtual occluders. Through pre-computed visibility and the use of IDs, real-time selective refinement taking into account visibility can be achieved efficiently to attain significant polygon count and frame time reduction.

One area of possible future work is intelligent occluder selection for occlusion preprocessing. Possible strategies include selecting potential occluder based on the size of occluder and occludee and the distance between them, and detecting redundant occluder where one is closely behind another. Beside walkthrough, another area of possible future work is to explore the use of our framework in other rendering applications.

## Acknowledgment

We would like to thank Kelvin Sung for his valuable comments to the paper.

## References

[1] Cohen, J., Varshney, J., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, F. and Wright, W. Simplification envelopes. *Proc. of ACM Siggraph*, pp 119–128, 1996.

[2] Cohen-Or, D., Fibich, G., Halperin, D. and Zadicario, E. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. *Proc. of Eurographics*, 1998.

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

[4] Garland, M. and Heckbert, P. Surface Simplification Using Quadric Error Metrics. *Proc. of ACM Siggraph*, pp 209–214, 1997.

[5] Greene, N., Kass, M. and Miller, G. Hierarchical Z-Buffer Visibility. *Proc. of ACM Siggraph*, pp 231–240, 1993.

[6] Greene, N. Hierarchical polygon tiling with coverage masks. *Proc. of ACM Siggraph*, pp 65–74, 1996.

[7] Hoppe, H. Progressive Meshes. *Proc. of ACM Siggraph*, pp 99–108, 1996.

[8] Hoppe, H. View-Dependent Refinement of Progressive Meshes. *Proc. of ACM Siggraph*, pp 189–198, 1997.

[9] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L., Faust, N. and Turner, G. Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proc. of ACM Siggraph*, pp 109–118, 1996.

[10] Luebke, D. and Erikson, C. View-Dependent Simplification Of Arbitrary Polygonal Environments. *Proc. of ACM Siggraph*, pp 199–208, 1997.

[11] Teller, S. and Séquin, C. Visibility Preprocessing For Interactive Walkthroughs. *Proc. of ACM Siggraph*, pp 61–69, 1991.

[12] Xia, J. and Varshney, A. Dynamic View-Dependent Simplification for Polygonal Models. *Proc. of the IEEE Visualization*, pp 327–334, 1996.

[13] Zhang, H. Effective Occlusion Culling for the Interactive Display of Arbitrary Models. *PhD Dissertation*, University of North Carolina at Chapel Hill, Department of Computer Science, 1998.

[14] Zhang, H., Manocha, D., Hudson, T. and Hoff, K. Visibility Culling using Hierarchical Occlusion Maps. *Proc. of ACM Siggraph*, pp 77–88, 1997.

| *Dataset* | *Terrain* | | *Occluders* | *Virtual occluders* | *Simplification (sec)* |
|---|---|---|---|---|---|
| 1 | Crater Lake | 8142 | 744 | 30 | 5.3 |
| 2 | Ashby | 19602 | 1236 | 50 | 13.5 |
| 3 | WestUS | 33282 | 1980 | 80 | 23.5 |
| 4 | Crater Lake | 44402 | 2220 | 90 | 31.3 |
| 5 | Crater Lake | 44402 | 4932 | 200 | 34.0 |

Table 1: Number of polygons and simplification timing for various datasets.

| *Dataset* | *Cell levels* | *Preprocessing* | | *Walkthrough of 1000 frames* | | |
|---|---|---|---|---|---|---|
| | | *Edge error correction (sec)* | *Occlusion culling (sec)* | *Without occlusion culling (sec)* | *With occlusion culling (sec)* | *Speedup (%)* |
| 1 | 6 | 0.2 | 89.6 | 22.4 | 16.0 | 29 |
| 2 | 6 | 0.3 | 373.3 | 41.6 | 33.2 | 20 |
| 3 | 6 | 0.4 | 867.5 | 48.5 | 32.0 | 34 |
| 4 | 5 | 0.5 | 673.4 | 75.8 | 64.0 | 16 |
| 4 | 6 | 0.5 | 1399.3 | 76.3 | 58.3 | 24 |
| 5 | 5 | 1.0 | 1257.2 | 93.2 | 77.0 | 17 |
| 5 | 6 | 1.0 | 2462.4 | 93.6 | 70.3 | 25 |

Table 2: Preprocessing and visualization timing.

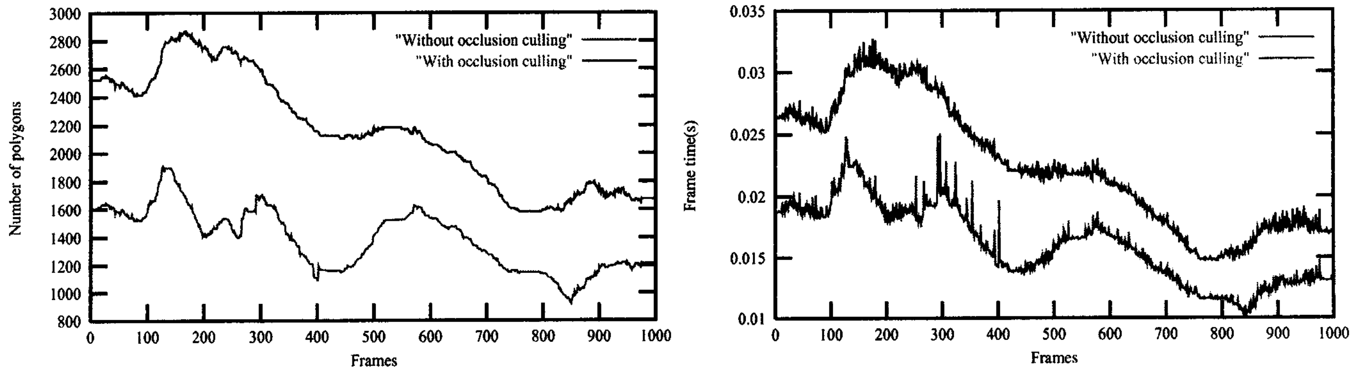


Figure 7: Number of polygons and frame time in the 1000-frame wakthrough of dataset 1.

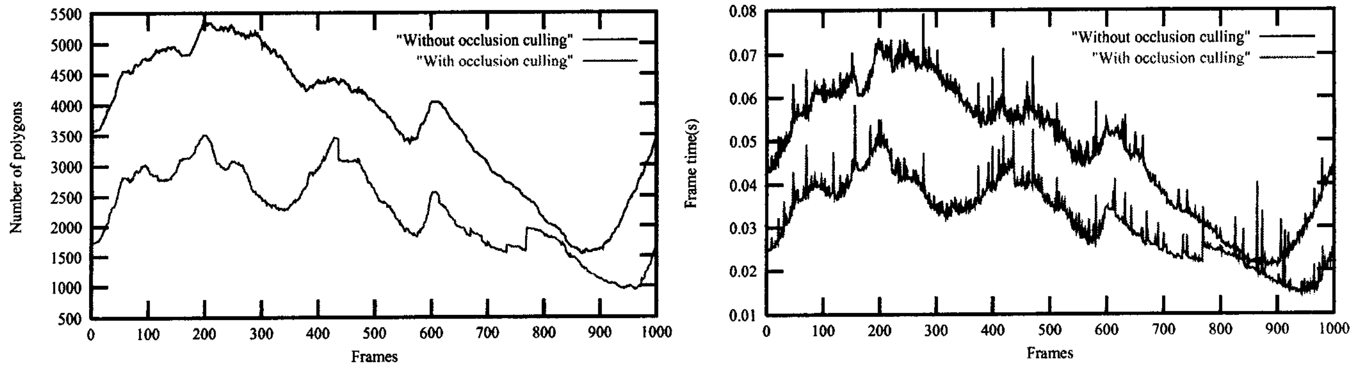
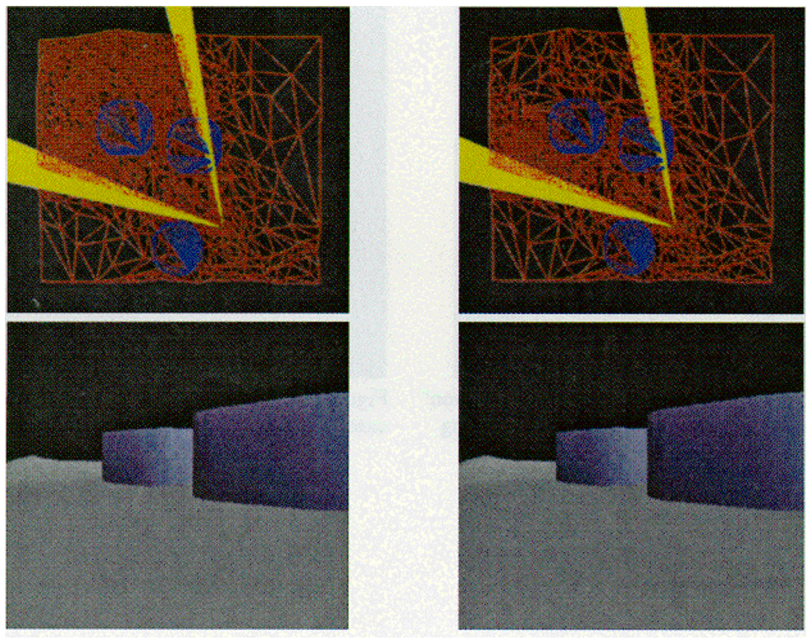
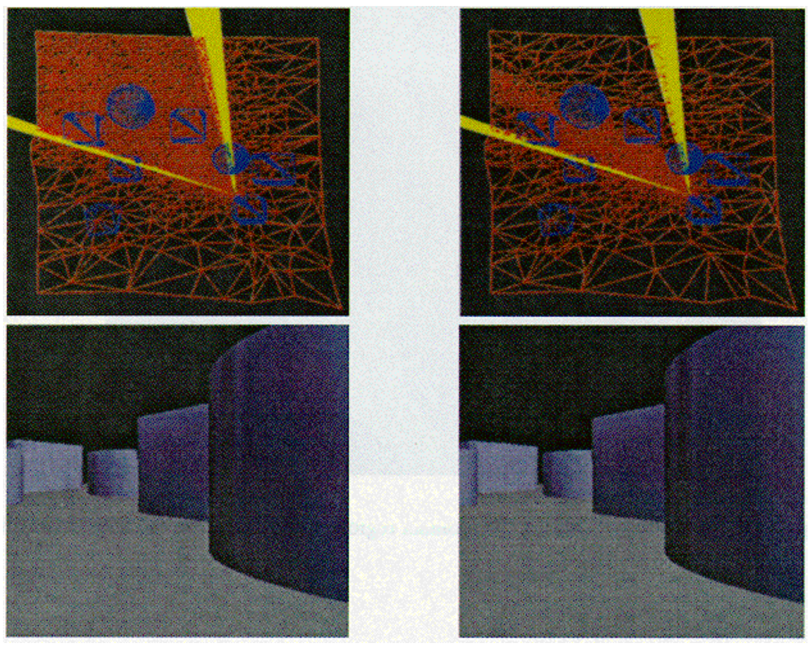


Figure 8: Number of polygons and frame time in the 1000-frame wakthrough of dataset 3.



(a) (b)

Figure 9: Top and 3D views of dataset 1 (a) without occlusion culling (2866 polygons), and (b) with occlusion culling (1851 polygons).



(a) (b)

Figure 10: Top and 3D views of dataset 3 (a) without occlusion culling (6957 polygons), and (b) with occlusion culling (3059 polygons).