HIERARCHICAL OCCLUSION CULLING FOR ARBITRARILY-MESHED HEIGHT FIELDS

A Thesis

by

PAUL MICHAEL EDMONDSON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2004

Major Subject: Computer Science

HIERARCHICAL OCCLUSION CULLING FOR

ARBITRARILY-MESHED HEIGHT FIELDS

A Thesis

by

PAUL MICHAEL EDMONDSON

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

John Keyser (Chair of Committee)	Donald H. House (Member)
Nancy Amato (Member)	Valerie Taylor (Head of Department)

May 2004

Major Subject: Computer Science

ABSTRACT

Hierarchical Occlusion Culling for Arbitrarily-Meshed Height Fields. (May 2004)
Paul Michael Edmondson, B.S., Texas A&M University
Chair of Advisory Committee: Dr. John Keyser

Many graphics applications today have need for high-speed 3-D visualization of height fields. Most of these applications deal with the display of digital terrain models characterized by a simple, but vast, non-overlapping mesh of triangles. A great deal of research has been done to find methods of optimizing such systems.

The goal of this work is to establish an algorithm to efficiently preprocess a hierarchical height field model that enables the real-time culling of occluded geometry while still allowing for classic terrain-rendering frameworks. By exploiting the planar-monotone characteristics of height fields, it is possible to create a unique and efficient occlusion culling method that is optimized for terrain rendering and similar applications.

Previous work has shown that culling is possible with certain regularly-gridded height field models, but not until now has a system been shown to work with all height fields, regardless of how their meshes are constructed. By freeing the system of meshing restrictions, it is possible to incorporate a number of broader height field algorithms with widely-used applications such as flight simulators, GIS systems, and computer games.

DEDICATION

to Gimp, king of the newts

ACKNOWLEDGMENTS

A number of people have provided immeasurable help in this undertaking. First and foremost, my profoundest thanks to John Keyser, whose countless hours of assistance and guidance have helped see this work through to completion. My fullest appreciation also to Don House, who has always given help wherever possible and was always available to lend an ear to my questions. This work was also tremendously helped by the tutelage of Nancy Amato, who helped me to understand enough of the field of computational geometry so that I could grasp the previous work on the subject.

Several others in the field also deserve mention: without the previous work of James Stewart, Hughes Hoppe, Leila De Floriani, and Paola Magillo, my task would have been tremendously more daunting. They were tackling the issues of terrain geometry long before I even understood many of the issues at hand, and have indirectly contributed much to my own work and understanding of the topic.

Of course, there are many others whose help, albeit not in the academic arena, has made it all possible. Thanks to my parents and friends who have supported me through all of the rough times and long nights, and to those that were always willing to help me out at a moment's notice during the crunch times. Of course, special thanks to David Eberle and Nathan Jones who stayed on my case and made sure that I finished this. But perhaps the greatest appreciation of all goes to Retno Sulistijo, without whom I never could have survived the final year of my graduate work. That this thesis was finished at all is a testament to her love, devotion, and understanding.

Last, but certainly not least, words cannot express my gratitude to God for giving me the gifts that got me here in the first place.

Thank you all.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

1 INTRODUCTION

In modern day computer graphics, one of the more commonly-faced problems is how to efficiently visualize a graphical environment in real time. Despite nearly exponential growth in the performance of graphics hardware, the complexity of the environments continues to limit performance because environment size tends to expand at the speed with which graphics hardware can operate.

There is a problem with this progression, however — not all aspects of graphics grow linearly. For example, the use of vaster landscapes or more detailed scenes cause quadratic or cubic growth in the complexity of their environments with respect to their dimensions, which translates to greater computation loads on the graphics hardware. This often happens regardless of whether the on-screen complexity changes, because the two most common ways of managing visible-surface determination through hardware these days are the post-computational methods of *z-buffering* and the *painter's algorithm* [16].

As it is wasteful to process geometry that will never be shown, computational geometers have looked at ways of reducing the problem complexity to an *output-sensitive* solution, where the complexity is proportional to the number of polygons actually being displayed on the screen [7].

It is a classic computational geometry problem to, given a set of line segments in 2-D space, find all segments visible from a current viewpoint. It is a rather simple task to formulate an algorithm that performs this task in $O(n \log n)$ time, using a rotating sweepline [8]. However, when one tries to generalize the 2-D visibility algorithm into three dimensions with polygons instead of line segments, it becomes a vastly more complex problem. Add the issue of output-sensitivity into the system, and the problem

---

This thesis follows the style and format of *IEEE Computer Graphics and Applications Proceedings*.

becomes harder still. In fact, the problem of 3-D visibility is quite non-trivial indeed [38].

Many graphics applications today have need for high-speed 3-D visualization of height fields. Most of these applications deal with the display of digital terrain models characterized by a simple but vast, non-overlapping mesh of triangles. A great deal of research has been done to find methods of optimizing such systems.

Previous work has shown that culling is possible with certain regularly-gridded height field models, but not until now has a system been shown to work with all height fields, regardless of how their meshes are constructed. By loosening the meshing restrictions, it is possible to incorporate a number of broader height field algorithms into widely-used applications such as flight simulators, GIS systems, and computer games.

The goal of this paper is to establish an algorithm to determine a hierarchical visibility system that makes it possible to cull occluded geometry during the rendering of height fields, while still allowing for classic terrain frameworks. By exploiting the specific nature of height fields, it is possible to create a unique and efficient occlusion culling method that is optimized for terrain rendering and similar applications.

2 CLASSIFICATION

Before further discussing the various aspects of this work, it is important to classify what is actually involved in the primary elements of this work. Both this and previous papers rely on at least a basic agreement on what constitutes a height field, occlusion culling, and level of detail rendering. This section puts forth a basic classification of these systems.

2.1 Height Fields

2.1.1 Definition

A height field, most simply defined, is a function of two variables $z = f(x, y)$, where the variables x and y are defined over a continuous 2-D domain D. The fact that the most commonly-cited application of height field displaying is terrain rendering has led to an almost synonymous definition of a terrain as the above function f, to the point that some literally define a *terrain* as a simple variant of the above definition [10].

It should be noted that according to this definition, only the two variables x and y are necessary to reference a particular point in the height field. It has also been pointed out that a characteristic of height fields is that the upper boundary of their projection on the $z - y$ plane is monotone with respect to increasing x values [31].

These characteristics of height fields are what allow us to simplify the standard 3-D occlusion culling algorithms. Indeed, height fields are sometimes referred to as "2.5-D", since they are a bit more than 2-D, but are still not entirely 3-D.

2.1.2 Variations

The fact that a height field is defined over a *continuous* domain suggests that we would not be able to merely store a discrete sequence of data points that fully define the

function. This is why most height fields dealt with in computer graphics are in fact only piecewise planar approximations of a continuous function or real-world domain. Such approximations come from a sampling function, whose responsibility is to sample points from the continuous function, and then build a polygonal approximation of the height field. As triangles tend to be the polygon of choice for computer graphics, the approximation usually amounts to a 3-D triangular mesh whose collective domains together cover the entire domain *D*.

The primary distinction between variations of height fields comes from how this sampling function operates. In many cases, the height field is sampled to build a mesh at equal intervals along the *x* and *y* axes, in a *regular* distribution. This sampling is often favored due to its geometric simplicity and the fact that field measurements are often taken at regular intervals. It also allows for the storage of samples with only one value, that of the height function. However, this method also has potential to produce aliasing artifacts. To combat such a problem, we can perform context-sensitive *irregular* sampling of data points, but then we must store the triangulation of the points in addition to the coordinates of each sampling point. De Floriani and Magillo note that irregular sampling usually provides a better approximation of the original surface than regular sampling [10]. It should also be noted that many height field applications such as those used in GIS and ocean-floor mapping systems must often use irregularly-sampled height fields due to the nature of their sampling methods, as the conversion from an irregular mesh to a regular one typically causes a severe loss of quality.

2.2 Occlusion Culling

2.2.1 Definition

When drawing polygonal scene elements, the complexity of scenes can very quickly reduce processing time to a crawl. In high-speed real-time environments, this prospect is to be avoided at all costs. One of the more obvious ways to speed up the drawing of elements is through *culling*. Culling seeks to eliminate geometry from the display

pipeline before it is rasterized, or drawn. There are three types of culling — backface culling, which eliminates polygons that are facing away from the viewpoint and therefore cannot be seen; frustum culling, which eliminates polygons that are outside of the view frustum, in other words, beyond the edge of the screen; and lastly, occlusion culling, which eliminates mass groups of polygons that are blocked by other screen objects. While backface and frustum culling can be performed in hardware, occlusion culling can not be done with current commercial hardware.

Often, occlusion culling is grouped with descriptions of *hidden surface removal*, or HSR, algorithms. HSR algorithms, which often work at the single image or polygon level, are more precise than occlusion culling, which is typically conservative and group-oriented in nature. A typical HSR algorithm is z-buffering, which operates in image space as practically the last step in the rendering pipeline. Unlike occlusion culling, z-buffering is performed on most commercial graphics hardware. Other HSR algorithms, like the Painter's Algorithm, operate in object space, but can be very complex. It is also not uncommon for an HSR algorithm to include an occlusion culling step [5].

2.2.2 Taxonomy

Occlusion culling has several different variations. In [5], Cohen *et al.* classify the techniques according to the following taxonomy:

Point vs. Region A primary distinction between occlusion culling algorithms is whether they operate in real-time from one viewpoint, or preprocess a method of handling visibility computation over a larger region of space. The region methods are more attractive due to their real-time speed prospects, but also can require long preprocess steps and may not handle dynamic environments as well. Region methods also tend to be more difficult to formulate, and are as a result less common.

Object vs. Image Precision A somewhat lesser-made distinction is whether an occlusion culling algorithm works in object space or image space. This is because the conservative nature of occlusion culling makes it rather difficult to imagine an algorithm that would operate in image space, which by its very nature is discrete.

HSR algorithms tend to more commonly work in the image domain than occlusion culling algorithms.

Specific vs. Generic Scenes A rather important distinction to make is what assumptions an algorithm makes about the scene that it is operating upon. A common application of occlusion culling is in architectural visualization. In such an environment, culling algorithms can take advantage of the partitioned nature of the environment. Such algorithms do not generalize well to the height field domain, however.

2.3 Level of Detail Rendering

2.3.1 Definition

Due to the nature of height fields, any given rendering of a height field will typically show certain portions of the field at very close range (and hence, high detail), and some portions at a very great distance (losing detail simply to the resolution of the rendering). Level of detail (LOD) rendering can take advantage of that characteristic by rendering close objects at high detail and distant objects at low detail. Other less obvious optimizations are also possible — for example, surfaces that are more or less parallel to the view plane do not need to be as high detail as surfaces whose edges make up the horizon.

2.3.2 Methods

LOD terrain simplification operates on a local, typically view-dependent scale. As a result, LOD methods are a bit more complex than typical mesh simplification methods, as they must transition the model between various levels of detail in addition to actually performing the simplification processes. On the other hand, since only small areas of a height field are being dealt with at a single point in time, it is possible to avoid preprocessing steps and actually calculate the terrain simplification in real-time. Assuming that the simplification can be done in real-time, the door is opened for

handling deformable terrain (for deformations that fall within the "local" framework) in real-time.

There are a number of extant LOD taxonomies, including the basic overview of Erikson [15] and the more specific classification presented by Mortenson [29]. For the purposes of this work, the most important elements are as follows:

Continuous vs. Discrete Perhaps the biggest distinction between various LOD algorithms is the mechanism and quality of transformation between varying levels of detail. Some algorithms operate by storing several distinct levels of detail, and switch between them as various metrics dictate (the geometric analog of mipmapping). Such a setup allows for customized tweaking of the model at each detail, but often lends itself to distinct "popping" artifacts where the LOD changes abruptly. The response to this problem is traditionally either to use some method to blend between levels, or to use a more continuous LOD model, where the changes to the detail are made constantly, often only a number of vertices at a time.

Hierarchical vs. Linear The method of modification to the polygonal mesh is also important to the overall algorithm. While some algorithms tend to operate in a linearly-updated fashion, most LOD algorithms are based on a hierarchical framework, as it allows selective refinement to the mesh without having to deal with the effects of such refinement on the entire system. Terrain LOD methods almost universally utilize mesh or refinement hierarchies to streamline updates.

3 RELATED WORK

3.1 Mesh Simplification

Traditionally, the field of real-time height field display has focused not around real-time optimization, but rather around preprocessed simplification. This is mainly due to the fact that most of the data normally stored in a height field is superfluous — it makes sense to eliminate unnecessary data points before attempting to evaluate or display the data. As height fields typically exhibit a high degree of area coherency, many methods of simplification attempt to exploit this coherency to eliminate unnecessary polygons. Much of this work has been collected and summarized by Garland and Heckbert [20], who have surveyed most extant methods of pre-rendered terrain simplification, as well as introducing several methods of their own [17] [18].

For mesh simplification to be useful, the algorithm must be free of the restrictions of fixed-density regular meshing. Fixed-density meshes are by definition non-adaptive, and will be resistant to alteration in a way that allows high detail to be preserved while simplifying low detail. Brown notes that the simple subsampling of regularly-spaced meshes can lead to the loss of significant topology features [3]. Variable-density quadtree or bintree meshing alleviates some of the shortcomings of fixed-density meshes, but can still lead to greater aliasing artifacts than the results of irregular mesh simplification. Some systems propose a hybridization of regular meshes with smaller irregular “microstructures. to attain a compromise between the simplicity of regularity and the accuracy of irregularity, such as Baumann in [2].

Often, pre-simplification methods can eliminate as much as 95% of the input data without the resulting height field differing significantly from the original [18]. The cost is a rather hefty preprocess step, and of course, less detail in the final models. Most methods rely on geometry for their reductions, using constructs such as a Delaunay triangulation, as in [17] [18] [24]. However, several take a more mathematical approach, such as the partitioning approach used by Agarwal and Desikan [1]. For a more complete

survey of the above methods, the aforementioned work by Garland and Heckbert [20] compares a majority of the extant methods.

Preprocessed simplification methods rely heavily on the assumption that a great amount of time can be made available prior to visualization to simplify the terrain. However, this assumption may not hold for certain applications. For example, certain systems may have the need for *deformable terrain*, where the height field values are dynamic and may change after all preprocessing steps have completed. Furthermore, once a terrain has been simplified, the detail eliminated by the simplification is forever lost, an effect that may be undesirable if the terrain will be viewed at very close range.

3.2 Multiresolution Modeling

The typical response to the limitations of static mesh simplification has been to use *multiresolution modeling*, a method of structuring data to enable switching (often seamlessly and continuously) between multiple hierarchical simplifications of a mesh, depending on various refinement metrics. Much of the formative work on multiresolution rendering has been by De Floriani *et al*, culminating in their Multi-Triangulation system [12], and with the Progressive Mesh system described by Hoppe [22] (later optimized in [21]). Early applications of the concept to height fields were summarized and expanded on by de Berg and Dobrindt [6].

3.3 Height Field Systems

The simplest and most obvious height field system is quite possibly still the most-used, where a regular height field is stored as a matrix of z values, and then rendered as a sequence of triangle strips. Of course, the sheer wastefulness and awkwardness of such a solution has prompted a number of variations and optimizations of the typical height field model. Almost all incorporate some implementation of mesh simplification under a multiresolution framework, rendered in a view-dependent LOD fashion.

3.3.1 Bintree and Quadtree Algorithms

Since the most common implementation of a height field is regularly sampled, a number of systems have been implemented specifically with the structure of a regular height field in mind. LOD schemes for such height fields typically use bintrees, a hierarchical data structure where each triangle is recursively divided in each lower level, or quadtrees, where each quad (formed by four adjacent height field samples) is recursively divided into four smaller quads. A potential issue with such algorithms is the possibility of *T-junctions*, discontinuities in the mesh where an edge is bisected only on one side by a subdivision, causing a gap. While some implementations choose to ignore or hide the gaps with filler polygons (such as Ulrich's ribbon and skirt methods [39]), the typical solution to T-junctions is to cascade the edge splits across the edge by subdividing the geometry on the other side. Subdivisions of bintrees can create fewer T-junctions per subdivision, but the results typically look similar regardless of the method of subdivision used.

The simplest form of LOD rendering is that of frustum-sensitive, distance-based LOD. Since geometry further away from the camera will be smaller on the screen, it is wasteful to render the distant geometry at the same resolution as the nearer mesh elements. Furthermore, elements outside the view frustum can be reduced to minimal detail to reduce the geometry traveling through the pipeline. In [36] Stewart used a simple quadtree subdivision hierarchy based solely on distance and frustum angle to simplify occlusion culling computations.

The problem with static levels of detail is that when the camera moves, the LOD rendering changes, causing features to potentially "pop" into view. The typical solution to such a problem is *geomorphing*, where elements near to the boundaries of an LOD level gradually morph from one level to the next. This is especially obvious in areas of greater unevenness in the surface. Lindstrom *et al.* and Röttger *et al.* proposed solutions to such a problem, allowing for a continuous level of detail with weighting toward areas of greater surface roughness [27] [32]. The former also gave a framework for more complex LOD determination metrics to be used if desired, and later was refined to

handle data more efficiently and accommodate features such as triangle stripping [28]. Szoka also described an approach to quadtree-based refinement that was favorable to geomorphing and rendering contiguous areas of like detail with triangle-strips [37].

The primary work with bintrees was established by Duchaineau *et al.* with a method called ROAM, short for Real-time Optimally Adapting Meshes [14]. The ROAM framework allowed for a number of features provided by other extant systems, with the noteworthy inclusion of support for dynamic terrains. The framework is simple and relatively easy to implement, but comes with the heftier memory footprint of a bintree hierarchy.

3.3.2 Irregular Height Field Refinement

Of course, by loosing the restrictions of regular sampling, one is free to become a bit more resourceful in the creation of an LOD framework. Formative work in this category was done by de Burg and Dobrindt, who created a hierarchy of geometry subdivisions based on Delaunay triangulation that could seamlessly fit multiple levels of detail together [6]. De Floriani *et al.* also provided their own method of irregular height field LOD with the VARIANT system proposed in [13], based off of earlier multiresolution height field work.

Hoppe in [23] applied the broader concepts of generic multiresolution modeling to a terrain-specific solution, continuously refining the geometry of a height field through a hierarchical split-merge framework. Under the progressive mesh framework, as the camera moves vertices are split into edges in areas of increasing detail, and edges collapse back into vertices in areas of decreasing detail, with no visible popping artifacts. Like other algorithms, Hoppe also used frustum movement as a cue for LOD coarsening and refinement. Brown provides yet another alternative with geomorphing based on Delaunay triangulation in [3]. An interesting variant on the irregular height field concept was a hybrid framework presented by Baumann *et al*, which consisted of a quadtree hierarchy in which the nodes each contained “microstructures. or small chunks of irregular data.

3.4 Occlusion Culling

Since occlusion culling in terrains is merely a simplification of the problem of 3-D visibility, much of the work with occlusion culling in terrains deals with specializations of the 3-D visibility problem. By taking advantage of the "2.5-D" principle, it is often possible to greatly simplify previously intractable or expensive algorithms.

3.4.1 Hierarchical Subdivision

When desiring an output-sensitive result to the occlusion culling problem, one of the first things desired is a way to remove large amounts of non-visible geometry. Hierarchical subdivision of the domain is a simple and easy way to do this faster with the help of a preprocessing step — in fact, it is one of the most widely used techniques for visibility culling [19].

3.4.1.1 Octrees

A common method of hierarchical subdivision is that of octrees. Octrees recursively subdivide space by splitting each cube of volume (starting with the entire domain) into eight separate subcubes. The recursion ends when a minimum cube size is reached, or the cube is entirely filled by the same object. Culling is performed top-down, by intersecting the view frustum with the octree. If a suboctree is occluded by an entire cube in front of it, then it may be culled.

It is not difficult to see that using a full octree for a height field is overkill. While octrees provide for shapes of arbitrary orientation and topology, we need not worry about such things for a height field. To figure out how one could simplify the octree, we need to first make the observation that the "solid" object defined by the height field is functional, and has no "bottom" that we care about. In essence, we can eliminate the third dimension for our calculations (to make a *quadtree*), and merely remember the value of f, the height value at each point. When performing visibility computations, we can eliminate any quadtree node (representing a square in the xy domain) by minimax-ing — if (and only if) the node in front of it has a minimum f' greater than the maximum

f' of the node behind it (where f' is f scaled to reflect the current perspective of the viewpoint), then it can be culled.

The above algorithm was implemented by Stewart in [36] with some rather impressive results, observing a performance increase of more than 50% in some cases. He continues to refine his horizon-based visibility determination in [34]. However, an important restriction to note is that the quadtree structure is based on a regular height-field — to use an irregular height field, one would have to regularly sample the irregular height field using an interpolation function or overlay a quadtree structure or the irregular height field. Thus, using an irregular height-field with this approach would almost assuredly introduce aliasing artifacts into the geometry or at very least, inefficient groupings of geometry.

3.4.1.2 Binary Space Partitioning

Another method of hierarchical subdivision is that of binary space partitioning, or BSP. This method uses a binary tree system to subdivide space along arbitrarily oriented planes. The remarkable and versatile nature of the BSP tree allows one to simply organize a front-to-back ordering of a scene from a particular viewpoint.

Hadwiger and Varga propose that an auxiliary bounding box system could be attached to the BSP, to determine if an entire subtree could be culled before it was drawn [19]. However, this system, like the octree system, would be overkill for a height field display system. In fact, since BSP trees actually split edges to subdivide space, irregularly-sampled height fields would take a severe performance hit, because of the density of edges in a typical height field. It is conceivable however to implement a BSP tree that incorporates a regularly-spaced height field inside, as Wiley *et al.* did in [40]. However, such a setup allows for little optimization for the “2.5-D” case, and little work has been done to present an efficient occlusion culling system for height fields using it.

3.4.2 Potentially Visible Sets

Apart from hierarchical subdivision, many occlusion culling techniques make use of the concept of potentially visible sets, or PVSs. A PVS is a preprocessed collection of scene items that share similar viewability characteristics. PVSs may be nested hierarchically, or be based upon a hierarchical subdivision. Visibility occlusion is *conservative* with PVSs, *i.e.*, if any element of one PVS can see an element of another PVS, they are said to be visible to each other.

The most common implementation of PVS uses portals. A portal is a closed cell-cell boundary that signifies the only location that one cell could potentially see through to another. This method is usually used for indoor environments, because of the easy partitioning of rooms as PVSs. The problem with using the portal implementation with height fields is that any visibility partition of a height field would have *no* closed boundaries, because it is defined by a closed, continuous *function*.

In [26], Law and Tan provide a mechanism for implementing a PVS system based on hierarchical subdivision techniques, including a height field system using quadtrees or BSP. Therefore, one could implement a PVS system based off any working hierarchical subdivision method, like those presented in the previous section.

3.4.3 Image-Based

An interesting algorithm that deserves mention is presented by Stamminger and Drettakis in [33]. They abandon altogether the concept of polygonal rendering for a point-based approach that operates in image space, and define an adaptive sampling scheme to render procedural and displacement-mapped objects. One of the advantages of their parameterization method of rendering height fields is that occlusion culling comes as a virtual byproduct of their rendering process. Furthermore, the adaptive sampling scheme has almost no overhead to perform dynamic LOD rendering. Of course, the benefits come at the greater cost of scene render-time, which is directly proportional to the pixel density of the screen.

3.4.4 Viewshed Model

De Floriani and Magillo in [10] offer an extensively detailed outline for an algorithmic model that, given a viewpoint over or even outside of a multiresolution irregular height field, will produce a set of visible polygons that they call a *viewshed*. Their algorithm centers on a robust hierarchical modeling extension of a previous algorithm presented in [9]. The paper also provides an exhaustive analysis of alternative visibility algorithms extant at the time it was written.

In the incremental algorithm, they take advantage of the simple topology of a height field. The data structure that they create for the mesh storage, called a *Hierarchical Triangle Irregular Network*, or HTIN, may be traversed from any triangle in the network to any of the adjoining triangles, and so on. This property holds true even when adjoining triangles are at different depths of subdivision. They slowly grow the viewshed from one triangle (at any level of the hierarchy), continuously maintaining the property that no triangle inside the viewshed is visually behind a triangle not in the viewshed. The polygon describing the viewshed grows in a *star-shaped* fashion, since it is concave but radially monotone. Occlusion culling is performed because all inner subdetail children of an occluded triangle can then be eliminated.

The primary downside to their algorithm is that it relies on the use of their own restrictive hierarchical triangle system, which, even though it is irregular, allows for no multiresolution point perturbations over the *xy* domain. That is, any subdetail triangle sets must be perfectly bounded by a larger triangle, an unlikely prospect in any system that defines a bottom-up hierarchy (*e.g.*, those that use Delaunay triangulations as their basis). With that crucial restriction taken into account, there is little that their viewshed model adds over and above the HSR systems that their paper references, like those of Reif and Sen [31] and Katz *et al.* [25]. There are still some cases where an HTIN is a viable option, however — for instance, networks that involve patches of regularly-sampled grid at different orientations are quite common in topographical surveys. The regularly-sampled regions will compress quite well into a multiresolution network as suggested.

4 VISIBILITY DETERMINATION

4.1 Overview

At the core of the previously discussed optimization algorithms lies a simple goal — to reduce the amount of geometry processed by the rendering pipeline. This work seeks to maximize such reductions by enabling the culling of non-visible mesh regions before they even enter the pipeline, while preserving the characteristics of the mesh that allow other optimizations, such as LOD rendering. The determination of what is visible and what is not visible is not an easy task, and is best suited to building renderer-specific "metadata" through a pre-processing step. By spending the bulk of our processing time offline, we can streamline the data flow into the online system.

4.2 Visibility in Height Field Applications

Before addressing the problem of how to efficiently determine visibility, it is important to look at what applications would benefit from height field geometry culling. By tailoring our solution to such frameworks, we can glean the best benefit possible for each situation.

4.2.1 GIS Systems

Often requiring visualization of large datasets spanning a sizeable region of terrain, GIS systems stand to gain much from occlusion culling. More computing power is usually at stake with GIS dataset rendering, as multiple datasets are often layered onto the same terrain to visualize not just simple terrain data, but also watershed analysis, political maps, development planning, forestry data, industrial drilling sites, etc. Limiting the extent to which such data needs to be processed and rendered can have sweeping benefits, not necessarily always in the rendering pipeline. Furthermore, it

cannot always be assumed that the processing for GIS systems will be operating solely in main memory.

4.2.2 Flight Simulators

Flight simulators often need to display much larger amounts of geometry than typical land-based terrain renderers. To cope with such limitations, flight simulators often must employ large-scale LOD algorithms and distance fading or fogging. A visibility solution for a flight simulator must be able to efficiently cull geometry in coarse LOD models as well as the refined ones. It should also be noted that the higher elevations utilized in commercial simulations will benefit much less from occlusion culling, as terrain becomes drastically less occluded the higher the viewpoint is. The overhead for a visibility solution in flight simulators must then be able to be scaled back so that it does not require a disproportionate amount of resources relative to the minimal benefits gained in such circumstances.

4.2.3 Computer Games

Since most computer games using terrain systems (with the exception of the previously-noted flight simulators) are typically viewed from vantage points close to the ground, they have much to gain from an efficient visibility-based culling algorithm. Since in-game operations include not just terrain rendering needs but object rendering, physically-based collision detection, and dynamic lighting scenarios, visibility-based culling can help to reduce the region of active objects that must be processed. Since computer games often are at the forefront of rendering technology, the solutions must also address the abilities and shortcomings of the latest and most advanced graphics hardware.

4.2.4 Dataset Visualization

The traditional applications of dataset visualization also occasionally have need for height fields and processing of height field data. Such visualization is typically needed

for the rendering of a set of experimentally-defined data with two input variables and one output variable, like a “surface” graph in Microsoft Excel. Unfortunately, such data systems tend to not be tied so directly to the class of terrain rendering applications as above. They rarely have need of real-time walkthroughs or flythroughs of precompiled data, instead needing frequent updates of one-time-viewable data. When used as a source for computation, typically the entire dataset must be processed, precluding the need for any view-dependent optimization. As a result, these systems would benefit little from visibility-based culling, or even culling in general (an HSR algorithm will suffice for visualization). For that reason we will not address this particular application in our solution.

4.3 Visibility with Terrain

Given the above applications, we can start to see a framework emerging from which to build a visibility model. Most algorithms operate on a medium-to-large scale, need to be scalable to reflect the degree to which visibility-based culling can benefit them, and utilize height fields as a method to visualize terrain. Such concepts define the specifications for our system, but it now becomes necessary to see how that can be used to apply a solution.

4.3.1 General Height Field Characteristics

There are several general characteristics of height fields that enable us to make important steps toward achieving a visibility model:

Continuity All proper height fields are continuous over the planar domain for which they are defined. The lack of discontinuities and holes ensure that we will not have to account for “windows” or gaps in the terrain that affect the visibility.

Planar Monotonicity By definition, a height field is a function of two variables, and as such, will never loop back on itself over its planar domain. Put another way, there are no loops or caves to look through. This is significant because it ensures that as

long as we are on or above the height field, the only way we can see past another point on the height field is if we can see *over* it.

Vertex Extrema Since the typical implementation of a height field is a simple polygonal mesh, we are guaranteed that the extrema of the height field are vertices of the mesh. We can therefore conceivably reduce our height field computations to deal with only the vertices that define the mesh.

4.3.2 Terrain Visibility

When viewing a particular terrain, certain additional characteristics become immediately obvious. There is typically a high degree of higher-order continuity — that is, the data rarely fluctuates rapidly from one sample to the next. When the continuity is broken, it is usually a sharp break. Simply put, flat areas tend to remain flat, and if the terrain is not flat, there are usually well-defined valleys, cliffs, and ridges. Valleys typically merge in the downhill direction, forming troughs, rivulets, or gorges. Ridges typically converge to form peaks. Such ridges are usually visible from large regions of the terrain, and consequently, those ridges also form the horizons for vantage points from most of the terrain. When one is on or above a ridge, he can conceivably see all of the terrain around him, and occlusion culling would not do much good. But when one is in a valley, objects in the distance are occluded by nearby ridges. It would follow then, that such ridges, from the vantage point of a valley, would be a good place to start for determining what is visible from a certain point.

4.4 Single Point Visibility

4.4.1 Visible Terrain from a Given Point

Let us consider a position in a valley as our starting point. Figure 1 shows a cross-section of the terrain in the neighborhood of the valley.

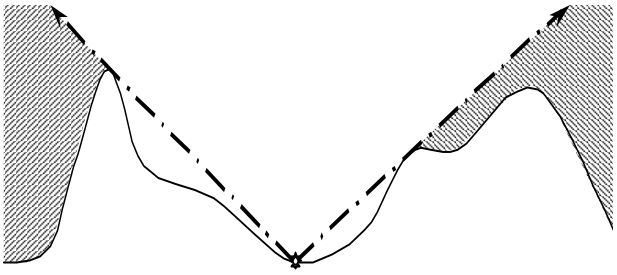


Figure 1. Occluded space from a sample vantage point.

In the figure, we notice that the visible horizon[1] (in this case represented by the two rays extending from the point) defines the border between visible terrain and occluded terrain. We can also make the observation that since the rays have a high slope outward from the vantage point, any terrain beyond the edges of the figure will have to be increasingly tall to be visible from our vantage point. After a certain distance we can safely conclude that nothing beyond the local ridges is visible.

But is space only occluded by the horizon? As Figure 2 shows, it also possible that a small patch of local space can be occluded with only a slight modification of the previous figure.

---

[1] It is important to make a distinction between the *visible* horizon and the *rational* horizon. The former marks the visual boundary between the earth and the sky. The latter refers to the flat, circular horizon at an infinite distance, often used in artistic rendering. In this paper, we are referring to the former.

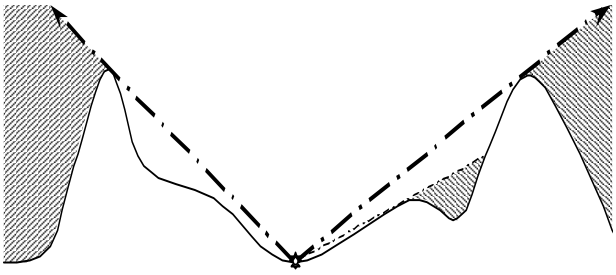


Figure 2. Multiple occlusion regions in terrain.

Going back to our potential algorithm applications, we remember that real-time visualization of terrains can sometimes occur from a higher elevation than merely on the ground. Figure 3 shows the effect of such a vantage point, and the resulting occlusion regions. Note that while Figure 3 shows a vantage point from “thin air. placing the vantage point on the side of one of the mountains would have a similar effect.

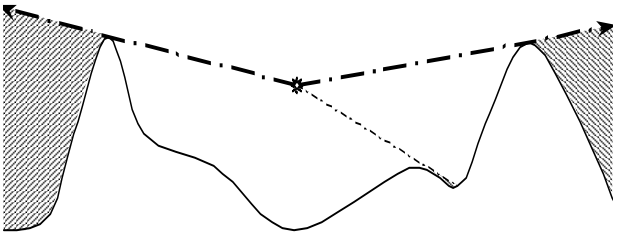


Figure 3. Multiple occlusion regions from a higher vantage point.

The fact that higher vantage points virtually eliminate local occlusion while maintaining nearly the same horizon suggests that the most effective model to base occlusion culling on would be that of horizon-based visibility. In such a system, occlusion would be performed in such a way that all points under (and therefore beyond) the horizon in any given direction are occluded. Occlusion would be conservative, since

no point that is visible would be culled by the algorithm, even though some points that are not visible would not be culled. Implementation would be simple and require minimal processing power and storage to cull large sections of terrain.

4.4.2 The Horizon for a Single Point

Probably the simplest and most straightforward algorithm for determining a horizon would be a naïve algorithm that radially sorts the points about a certain vertex, and then incrementally builds a horizon using a rotating sweepline. Such an algorithm is not unlike the 2-D point visibility algorithm assigned as an exercise in [8], and would run in $O(n \log n)$ time for each point (where n is the number of vertices in the terrain). Any system creating a global visibility algorithm based off of such an algorithm would then run in $O(n^2 \log n)$ time, since the horizon must be computed for every vertex in the terrain. Stewart suggests a faster approximation-based variant of the naïve algorithm in [35] where the horizon is broken into radial sectors to which the points are incrementally compared. Such an algorithm would run in $O(n^2)$ time. Cabral, Max and Springmeyer propose an algorithm to trade a bit of accuracy to further reduce the running time to $O(n^{1.5})$ time, by sampling the horizon only along certain lines [4].

4.5 Finding the Horizon for All Points Simultaneously

An algorithm that runs in $O(n^{1.5})$ may be relatively fast compared to others, but given the large size of terrains (sometimes in excess of a million points) it would be beneficial to further optimize the algorithm. Stewart in [35] makes the observation that in a given sector direction, it is possible (by sorting the points and using some clever data structures) to incrementally build the horizon in that sector for all points simultaneously in less time than his $O(n^2)$ algorithm. The suggested algorithm runs in worst case $O(sn(\log^2 n + s))$ time and $O(n \log n)$ space, where s is the number of sectors being used. Despite having the inaccuracies of the $O(n^2)$ algorithm, it approximates better

than the $O(n^{1.5})$ algorithm and runs faster for large datasets. Below we will introduce a slight variation of this algorithm that runs in worst case $O(sn\log^2 n)$ time, of which the worst-case memory usage is $O(n\log n)$.

4.5.1 Definitions

Assume that we split the horizon from each point (mesh vertex) into *s* piecewise-constant sectors, where the elevation in each sector is the maximum apparent elevation for that sector. By "apparent. we mean that we take into account the effects of perspective transformation — that is, elevations further away appear to be lower. By assuming a constant horizon elevation across the sector, we introduce the potential for non-conservative error, a problem that we will address later in our algorithm.

Let us name each sector σ_i, where *i* is the integer index of the sector, such that $0 \le i < s$. A given sector *i* is therefore defined to be the closed angular range $\left[\frac{2\pi}{s} i, \frac{2\pi}{s}(i+1)\right]$. Let $\vec{r}_i$ be the horizontal vector in the direction of the center of the sector, in the direction of the angle $\frac{2\pi}{s}\left(i+\frac{1}{2}\right)$. Now let π_i be a vertical plane coincident with the vector $\vec{r}_i$.

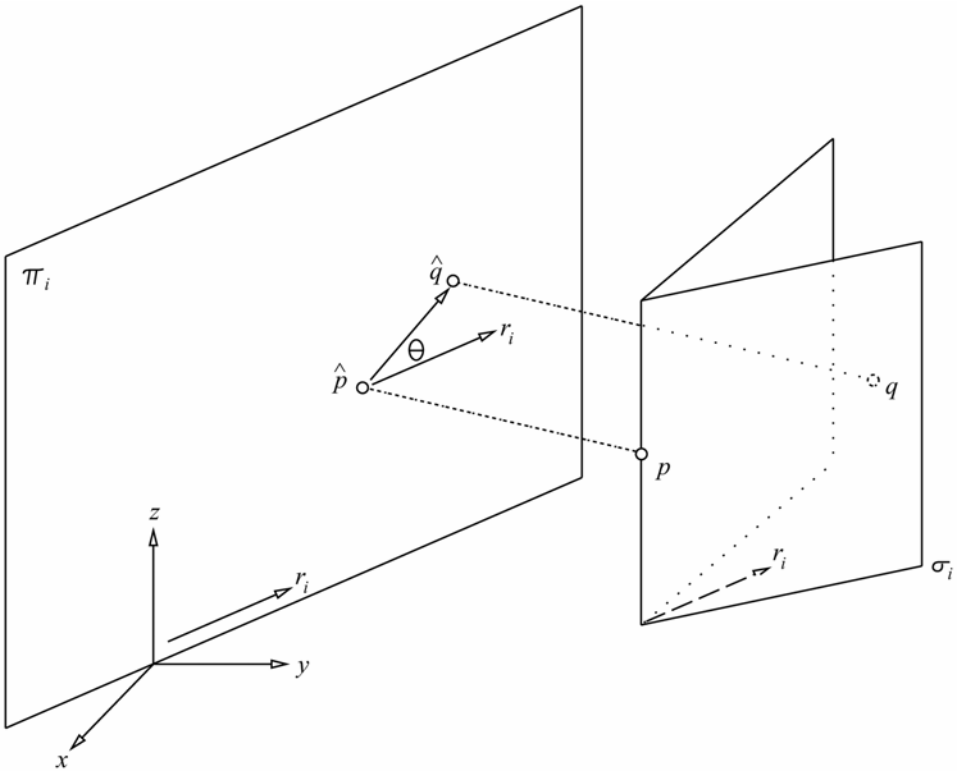


Figure 4. The sector σ_i of point *p*, and the resulting projection of *p* and *q* onto π_i.
Figure taken from Stewart [35].

Consider now the projection of points *p* and *q* onto π_i, which we will call $\hat{p}$ and $\hat{q}$. Let $\theta_{\hat{q}}$ denote the angle between the vector projection of $\vec{r}_i$ onto π_i and the vector from $\hat{p}$ to $\hat{q}$. We care about the projection onto π_i because we are looking for a piecewise linear approximation of the horizon, which would suggest that we treat all points in the sector as if they were in the center[2].

[2] Technically, the projection should take into account the curvature of the sector, but for the author's implementation a linear projection was used for simplicity and speed. The discrepancy should not be noticeable except in cases where very large sectors are used.

4.5.2 Recursive Definition of the Horizon

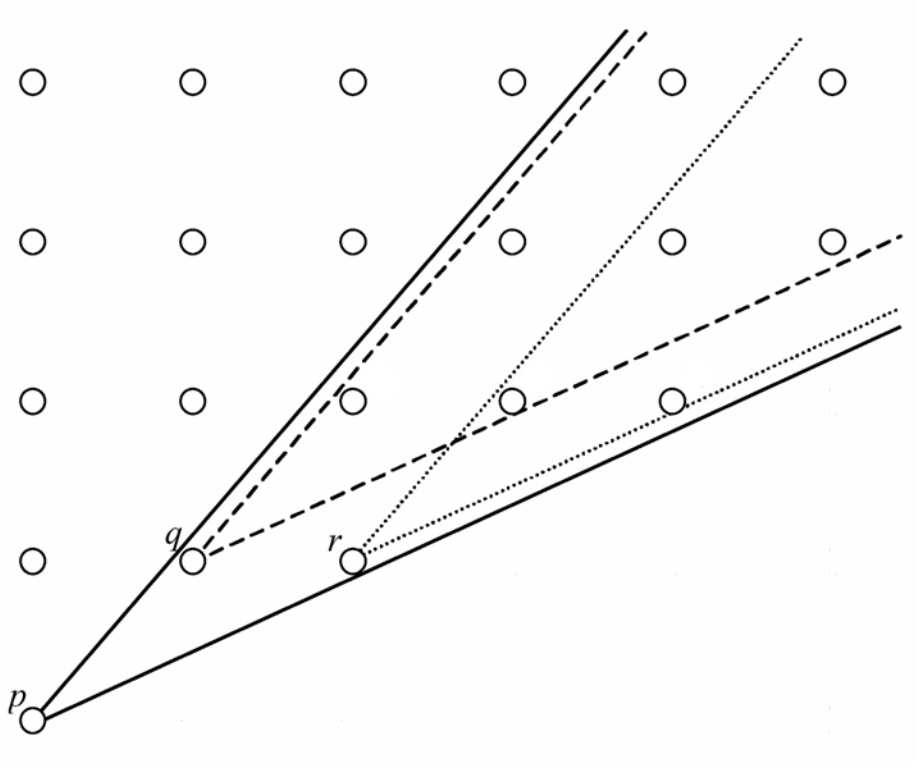


Figure 5. The sector σ_i of point *p*, from above, and its relationship to points *q* and *r*. A regular height field is used for simplicity. Based on a similar figure in [29].

In Figure 5 we view sector σ_i of point *p* (which we will call σ_i^p) from above (along the $-z$ axis); we see that there is a relationship between it and the sector σ_i of each point contained within that of point *p*, in this case, points *q* and *r*. In fact, we may recursively define the set of all of the points in sector σ_i^p to be the union of the points in sector σ_i^q and the points in sector σ_i^r. Thus, the horizon point for point *p* (denoted h_p) in the direction of $\vec{r}_i$ must either be *q*, *r*, or contained in σ_i^q or σ_i^r.

Now let us consider the projection of all points in σ_i^p onto π_i. By definition, point h_p is the point whose projection $\hat{h}_p$ has the maximum angle θ from point $\hat{p}$. Since $\hat{h}_p$ is an

extreme point of the set of all points in the projection of σ_i^p, it follows that $\hat{h}_p$ is on the 2-D upper convex hull of the projected points. Let such a convex hull be called the *covering set* C_i^p.

Since σ_i^p contains σ_i^q and σ_i^r, and the convex hull of two sets of points is the same as the convex hull of the points on the convex hulls of the respective sets, we can define a simple means for recursively finding the horizon point. The covering set C_i^p is the upper convex hull of the union of $\hat{q}$, $\hat{r}$, and the points in C_i^q and C_i^r.

4.5.3 Visibility for a Sector

Given a recursive definition for the covering sets C_i^*, we can start to see a course of action for finding the horizon in a given sector for all points without having to resort to the naïve $O(n^2)$ method of comparing all points with each other. If we can build covering sets in an order that is based on the sector direction, we can find all horizon points for the sector with only one pass.

This is a slightly more difficult task than it seems. The difficulty arises from the fact that a sector is not defined by a single direction vector, but rather an angular swath whose origin differs depending on what point we are finding the covering set for. Consider Figure 6, where we have a point *q* that is "in front" of a point *p* with respect to the projection vector $\vec{r}_i$ even though *q* is not a member of the covering set C_i^p.

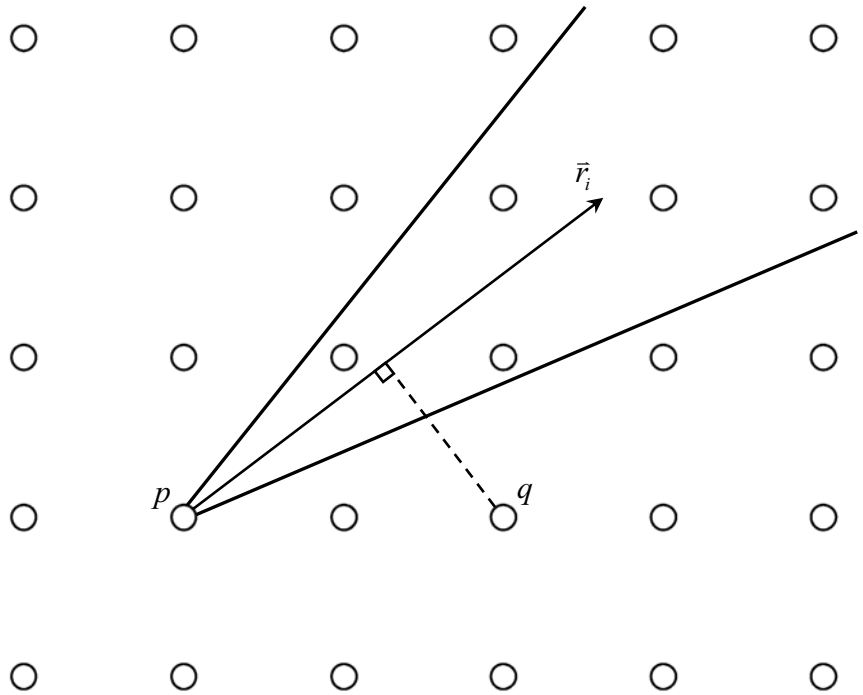


Figure 6. A case where sector σ_i of point p does not contain point q in the direction $\vec{r}_i$.

In this case, however, we can avoid the problem by considering two vectors instead of one. Rather than using the vector $\vec{r}_i$ (remember that this is defined as the direction of the angle $\frac{2\pi}{s}\left(i+\frac{1}{2}\right)$), we can use the vectors defining the directions of the sector edges, $\frac{2\pi}{s}i$ and $\frac{2\pi}{s}\left(i+1\right)$. For simplicity's sake, let us call these vectors $\vec{a}$ and $\vec{b}$, respectively.

Now consider the vectors perpendicular to $\vec{a}$ and $\vec{b}$, denoted $\vec{a}^{\perp}$ and $\vec{b}^{\perp}$. Let us generate two orderings of the points based on their scalar projections onto $\vec{a}^{\perp}$ and $\vec{b}^{\perp}$ respectively, as in Figure 7.

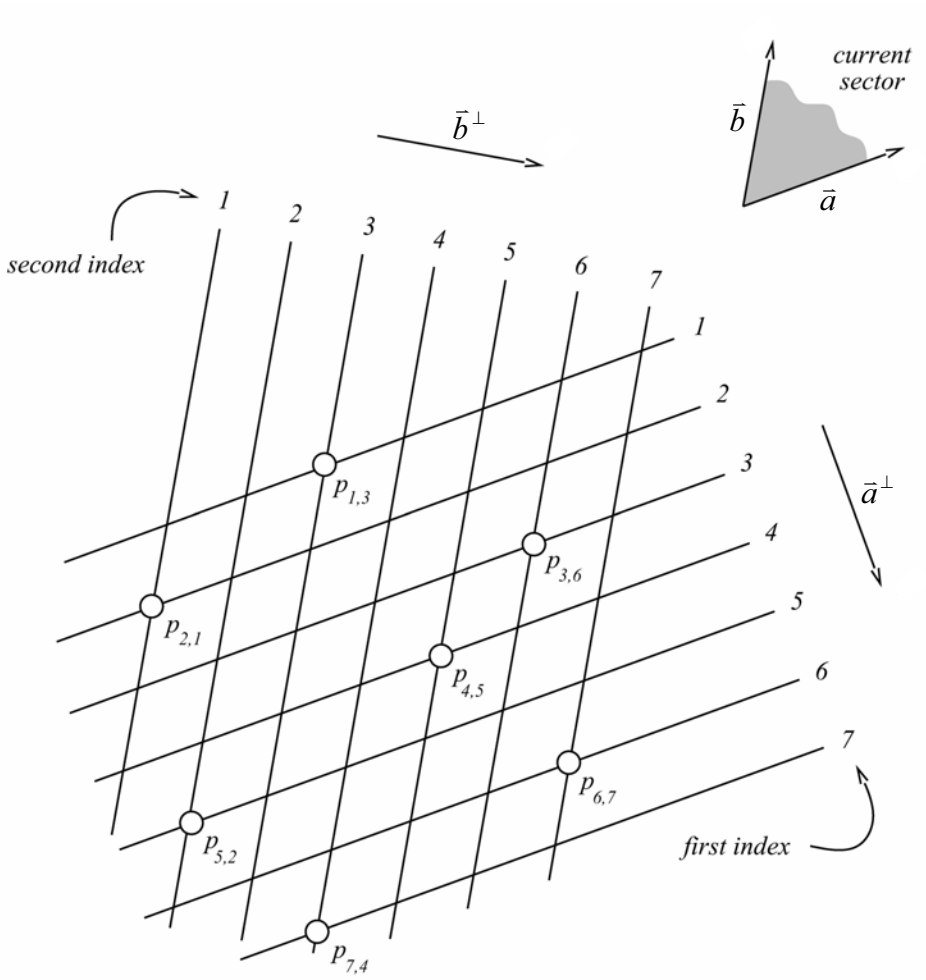


Figure 7. Ordering of the points along $\vec{a}^{\perp}$ and $\vec{b}^{\perp}$.
Based on a similar figure in [34].

Given such orderings, we can see that a point is in σ_i^p if and only if its $\vec{a}^{\perp}$ index is less than or equal to that of p and its $\vec{b}^{\perp}$ index is greater than or equal to that of p. We

can exploit this bidirectionality to build the covering sets. Consider the following algorithm:

For each sector i :

Initialize C_i^* to be empty

Calculate ordering of points along $\vec{a}^{\perp}$ vector

Calculate ordering of points along $\vec{b}^{\perp}$ vector

For each point p (in order of increasing $\vec{a}^{\perp}$ index):

Find max tangent extreme point from $\hat{p}$ to C_i^p to find the horizon point for p

Insert $\hat{p}$ into the covering sets for all points whose $\vec{b}^{\perp}$ index is less than that of p

Since we are calculating the covering sets in order of increasing $\vec{a}^{\perp}$ index, we ensure that no point would insert itself into C_i^p unless its $\vec{a}^{\perp}$ index is less than or equal to that of p. We also stipulate that we only insert $\hat{p}$ into covering sets whose $\vec{b}^{\perp}$ index is less than that of p, therefore ensuring that $\hat{p}$ will be in the covering set of all later-processed points for which p would be a horizon point candidate. As a result, we are sure that when we process each point p we are considering all (and only) points in σ_i^p as possible horizon points. Figure 8 shows the algorithm steps on a small set of points.

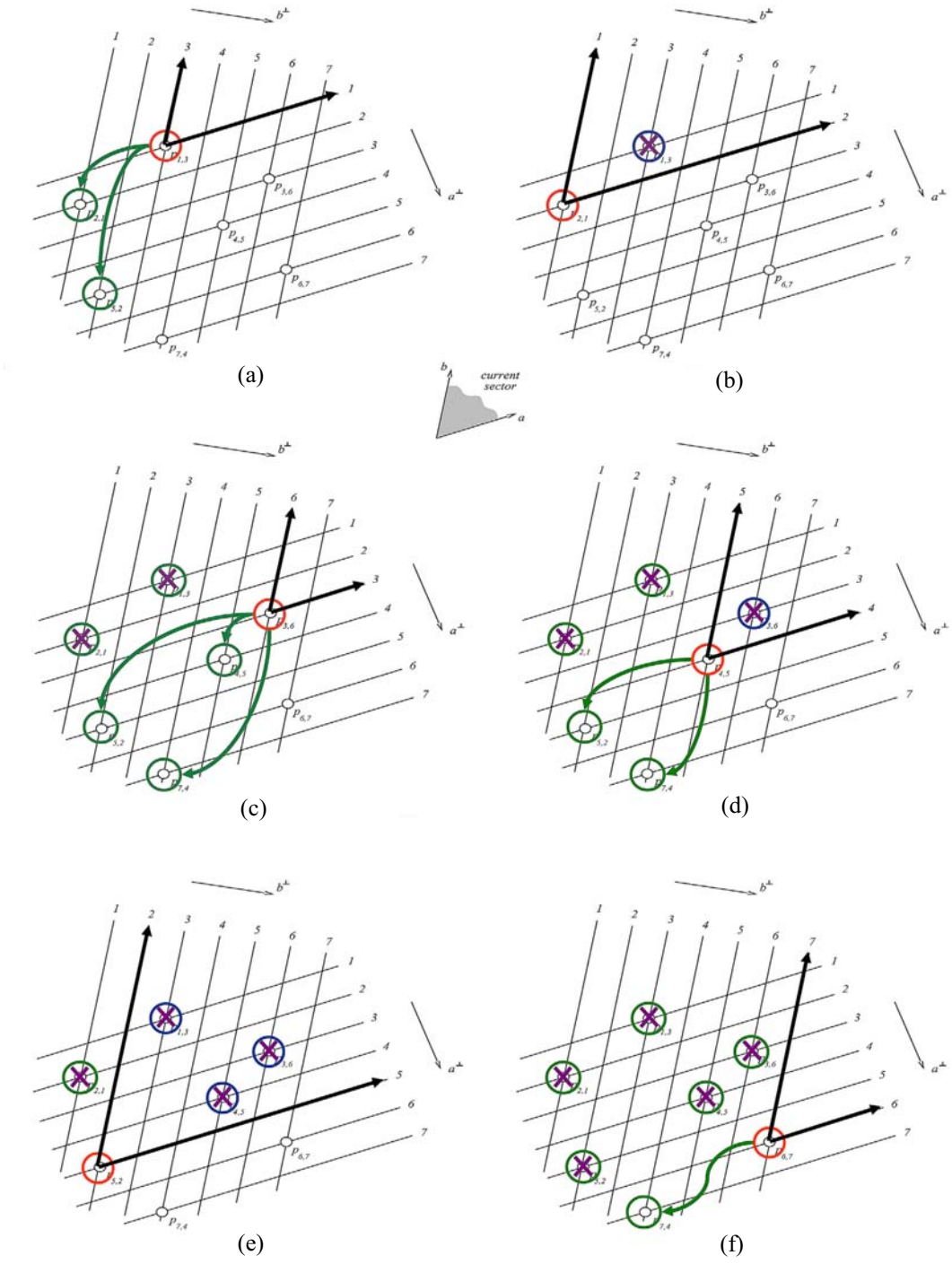


Figure 8. Finding the horizon in a sector for a set of points.
The red circle denotes the current vertex being evaluated. Blue circles indicate possible horizon candidates. Purple "X"s mark the vertices that have already been processed.
Green circles mark vertices with lesser $\vec{b}^{\perp}$ indices, with the arrows showing the insertion of the current vertex into their covering sets. The processing of the last vertex is not shown.

While this finds the horizon for all of the points, we are not entirely done. The insertion of a point into a covering set is effectively the insertion of a point against a convex hull, an operation which runs in $O(\log n)$ time [30]. Since there are potentially $n-1$ points whose $\vec{b}^{\perp}$ index is less than that of p, that step runs in worst-case time $O(n \log n)$. When calculated for n points this yields a general runtime no better than the non-approximated runtime of $O(n^2 \log n)$ and actually worse than the naïve approximation runtime of $O(n^2)$. Clearly, this is not acceptable.

We can circumvent this problem by altering the above algorithm to take advantage of the recursive definition of a covering set. Remember that since a covering set is an upper convex hull, all of the properties of convex hulls apply; most importantly the distributive nature of the convex hull operation. We know that finding the extreme point of a number of convex hulls will yield the same result as finding the extreme point of the convex hull of all of the points in the convex hulls [30]. This suggests that perhaps we can apply a divide-and-conquer approach to the covering set operations.

Consider if, instead of maintaining a covering set for each point, we maintain a static balanced binary tree with $2n-1$ nodes (n leaves) where each node is a covering set. We reference the leaves from left to right in the $\vec{b}^{\perp}$ ordering, such that the leftmost leaf represents the point with a $\vec{b}^{\perp}$ index of 0, and the rightmost leaf represents the point with a $\vec{b}^{\perp}$ index of $n-1$. Instead of inserting each point p into the covering sets of all points whose $\vec{b}^{\perp}$ index is less than that of p, we insert it only into the covering sets of the nodes that are left *children* of the leaf-to-root path of p. When calculating the horizon point of p, we find the maximum tangent from $\hat{p}$ to each of the covering sets of nodes on the leaf-to-root path for p. Figure 9 shows the tree and which nodes are affected by a particular iteration of the loop.

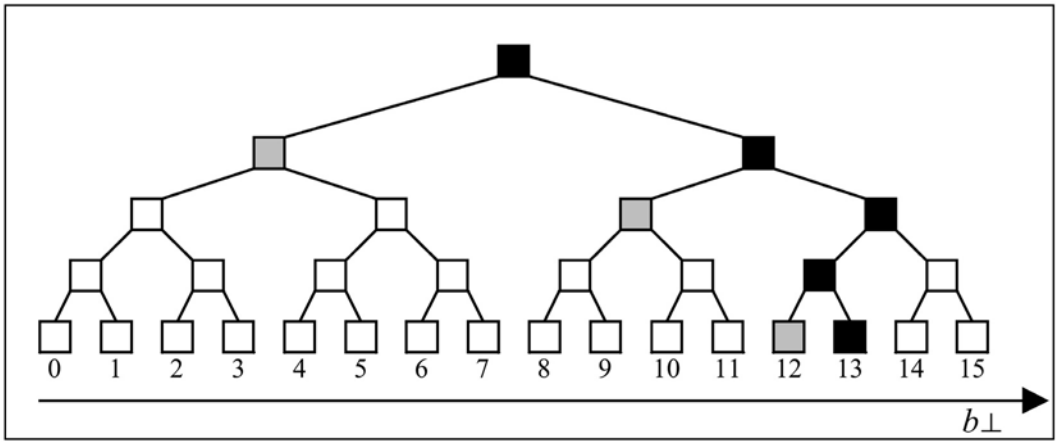


Figure 9. Querying the tree for p where $\vec{b}^{\perp} = 13$.
Black nodes represent the nodes whose covering sets will be queried, gray nodes the nodes that $\hat{p}$ will be added to. Figure taken from Stewart [35].

Upon close examination, it is evident that every node (and only nodes) to the left of the node for p will come across a covering set containing $\hat{p}$ in the leaf-to-root path. This ensures that all nodes will consider p as being a candidate iff the $\vec{b}^{\perp}$ index of p is greater than or equal to theirs. Since the leaf-to-root traversal runs in $O(\log n)$ time and each covering set query or insertion runs in worst-case $O(\log n)$ time, the total runtime for each point is $O(\log^2 n)$, for a grand total runtime of $O(sn \log^2 n)$, where s is the number of sectors. (The resorting of the points for each sector only trivially affects the runtime, since such a sort runs in $O(n \log n)$ time.) The memory requirements are $O(n \log n)$, since each point can lie in $O(\log n)$ covering sets. Note that the runtime memory is not proportional to the number of sectors, since the covering sets only need to be maintained for the duration of each sector calculation (one can infer from this that the sector calculations can be performed in parallel). The memory required for storage of the horizon data is simply $O(sn)$, s horizon points for each point.

4.5.4 Potential Issues

There are several limitations to the algorithm presented above. Most importantly, it is not entirely conservative — that is, there are points below the approximated horizon that may not actually be occluded. This is because the algorithm finds the *maximum* horizon point for a sector. When approximating the horizon for occlusion purposes, we really want the *minimum* horizon point for a sector.

For a large sector, the potential for non-conservative visibility error is relatively high, especially for rugged terrain. The natural solution is to reduce the width of the sectors (*i.e.*, increasing the number of them), but by increasing the number of sectors, we increase the runtime and the amount of data to be stored with each point, something that may not be desired. A compromise can be achieved, however, by computing for each sector a set of subsectors, from which the minimum horizon point is determined to be the horizon point for the entire sector. That way, greater accuracy can be achieved in the horizon calculation while still keeping the final storage as low as possible. Since we are working with the assumption that precomputation runtime is less important than the visualization runtime, this is an acceptable compromise.

Of course even with such a solution, non-conservative error is still possible — just on a much smaller scale. This error will be unavoidable in the end since we are using an approximation algorithm. Fortunately as we will see later, the grouping algorithms used to implement occlusion culling will minimize and almost eliminate such problems, so long as the subsectors remain relatively small.

With the reduction in the size of the sectors (or use of subsectors), the possibility of undersampling increases. If any edges completely transect the sector, the visibility algorithm will not evaluate them, as it only evaluates the elevations of vertices; that is to say, an up-close ridge represented with sparse geometry may be entirely missed.

In the end, this error is conservative because the missed geometry only causes the algorithm to mark more of the terrain as visible. At the same time, this can greatly reduce the effectiveness of the occlusion culling by missing what are potentially the strongest occluders at a given point. Stewart suggests in [34] that for regular height

fields, one can sample the points on either side of the sector until the width of the sector is greater than the spacing between the height field samples, and add any points that would affect the horizon. This is simply an application of the Cabral, Max and Springmeyer visibility algorithm for a small, bounded area around the sample point. This is not applicable for irregular height fields, however. A possible solution would be to keep a k-d tree of points with edge information (creation time $O(n \log n)$), and for each point p evaluate whether any of the edges of any point within a certain distance cross one of its sectors. If an edge crosses a sector without either vertex being inside the sector, then the edge is checked to see if it alters the horizon for p. So long as the distance checked is relatively small, the extra checks will not significantly affect the overall algorithm runtime.

Undersampling also has the potential to cause a different kind of error in the unlikely situation that an edge *behind* the horizon point transects the entire sector and is higher (visually) than the horizon point. While this causes an inaccurate horizon, this error is non-conservative and does not adversely affect the occlusion culling process. This is because the distant ridge lies outside of the occlusion region defined by the horizon point — an almost desirable effect, as it almost certainly increases the amount of potentially-occluded geometry.

4.6 Mesh Element Visibility

Remember that the whole point of finding the horizon-based visibility for the height field was to determine the regions of occluded space from any point. This region was shown to be best represented by the space under a ray (in the 2-D case) from the horizon to infinity. But in the end we are concerned about the regions of space from which each *piece of mesh geometry* is occluded, since the terrain is rendered as a set of distinct mesh elements.

This turns out to be rather simple to compute, since the points that we tested are the vertices of the mesh geometry (we can assume triangles for simplicity). Just as in a Venn diagram, the region of occluded space common to all points in each mesh element is

merely the intersection of the individual points' occluded spaces. Since the intersection operation is entirely associative, one may take any number of points in any order, and find their combined visibility. Figure 10 shows the effect of the computation for a triangle in a sector.

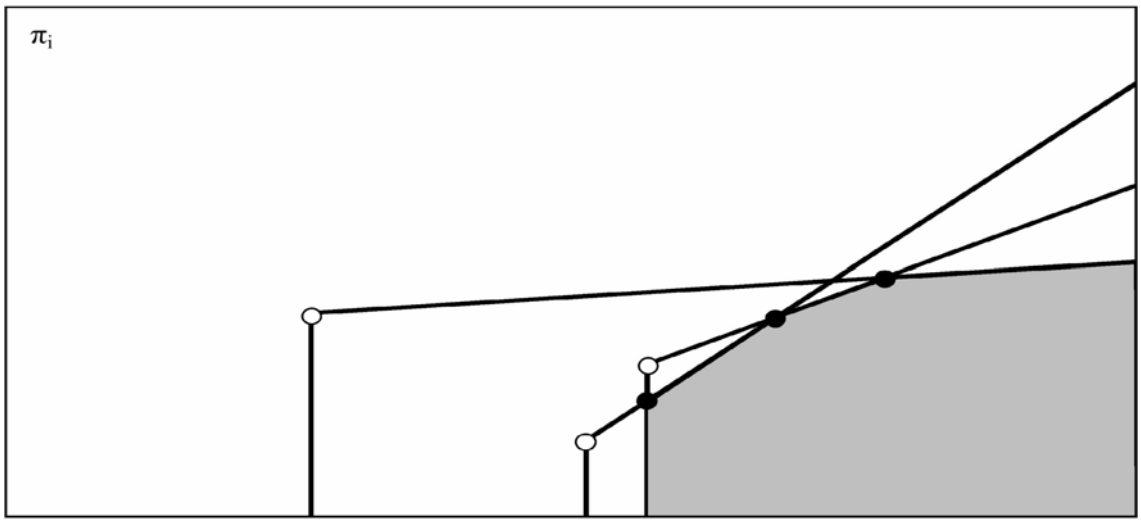


Figure 10. The intersection of occlusion regions for a mesh triangle in a given sector. Figure taken from [29].

We can see that the closed boundaries of the occluded space are convex. As the intersections of convex spaces are themselves convex, the new common occlusion region is itself convex. Given the occlusion regions of two mesh elements, one can find their combined occlusion region by determining only the intersection of the two hulls.

While individual point-visibility information is not very useful, the ability to find the combined visibility of mesh elements or groups of mesh elements is incredibly powerful. Given such information, we can formulate a method to hierarchically store visibility to perform efficient, real-time occlusion culling.

5 OCCLUSION CULLING

5.1 Occlusion Culling from Visibility Information

5.1.1 Occlusion Culling of Simple Mesh Elements

In the previous section, we established an algorithm to preprocess the horizon-based visibility of any mesh element in a terrain mesh. From such visibility information we know the occlusion regions from any mesh element, a crucial step in fast, real-time occlusion culling. But how do we get from knowing the occlusion regions to real-time culling?

Let us consider the simplest runtime algorithm, where we choose to consider whether or not each triangle in a terrain mesh should be culled as we render. Assume that we have already precomputed the visibility determination, and have stored the occlusion regions for each sector, for each triangle in the mesh. When we are ready to visualize our terrain in real-time, we load the mesh and its occlusion region data.

When rendering our terrain, at any point in time our camera will have a location c in space. Consider the following algorithm:

For each triangle t*:*
    For each sector s *relative to* t *that* c *lies in:*
        Project c *into* π_s *to find* $\hat{c}$
        Compare $\hat{c}$ *against the occlusion region for* σ_s^t
        If $\hat{c}$ *is outside the occlusion region:*
            Render t
        Otherwise:
            Do not render t

To compare $\hat{c}$ against the occlusion region for σ_s^t, we merely find the two points of the occlusion region that $\hat{c}$ lies between, and see if it is above or below the two. If $\hat{c}$ lies before any point in the occlusion region, we know that it is outside. If it lies after any point, we compare it to the ray extending to infinity.

An interesting benefit of this method is that we can perform frustum culling with little effort. Consider our camera with view frustum *F*. (As an optimization, we need only consider the 2-D projection of the view frustum, so that we can define *F* as a closed angular range between the angles defining the left and right frustum boundaries.) Simple geometry shows that the angular range of *F* is merely 180 degrees opposite the range defining the possible sectors that *c* could lie in (*i.e.*, if *F* is the closed angular range $[0^\circ, 45^\circ]$, then the angular range we want to consider is $[180^\circ, 225^\circ]$). Since we already have to find what sector *c* lies in, we can restrict what sectors we even check against in a modified version of the algorithm:

Determine the set Δ *of sectors whose angular ranges contain the angular range directly opposite* F
For each triangle t*:*
    If c *lies in* Δ *relative to* t*:*
        For each sector s *relative to* t *that* c *lies in:*
            Project c *into* π_s *to find* $\hat{c}$
            Compare $\hat{c}$ *against the occlusion region for* σ_s^t
            If $\hat{c}$ *is outside the occlusion region:*
                Render t
            Otherwise:
                Do not render t
    Otherwise:
        Do not render t

There is of course no need to test whether an object outside the view frustum is occluded, so our modification merely makes the rendering of the mesh more efficient. With this simple modification we significantly reduce the amount of geometry sent through the rendering pipeline without adding significant overhead to the rendering process.

5.1.2 Hardware Complications

Even though the above algorithm achieves our goal of performing occlusion culling, it does so at the expense of a great number of CPU cycles to process the visibility for each triangle. Luckily (for the sake of overall performance), today's graphics hardware is capable of handling triangles much faster than a CPU, so much that the average graphics hardware would be idle for most of the above loop. Graphics processors (GPUs) are best-equipped to handle *batches*, or groups, of triangles.

Figure 11 shows the effect of changing the number of triangles per batch[3] for a number of commercially-available NVIDIA graphics cards. In the figure, we see that with less than 130 simple triangles per batch, no GPU performs any better or worse than any other. This means that the load on the CPU is the bottleneck for such cases — as it is effectively spending its cycles batching the triangles together.

---

[3] This test was performed by NVIDIA and does not use the terrain algorithm developed in this paper.

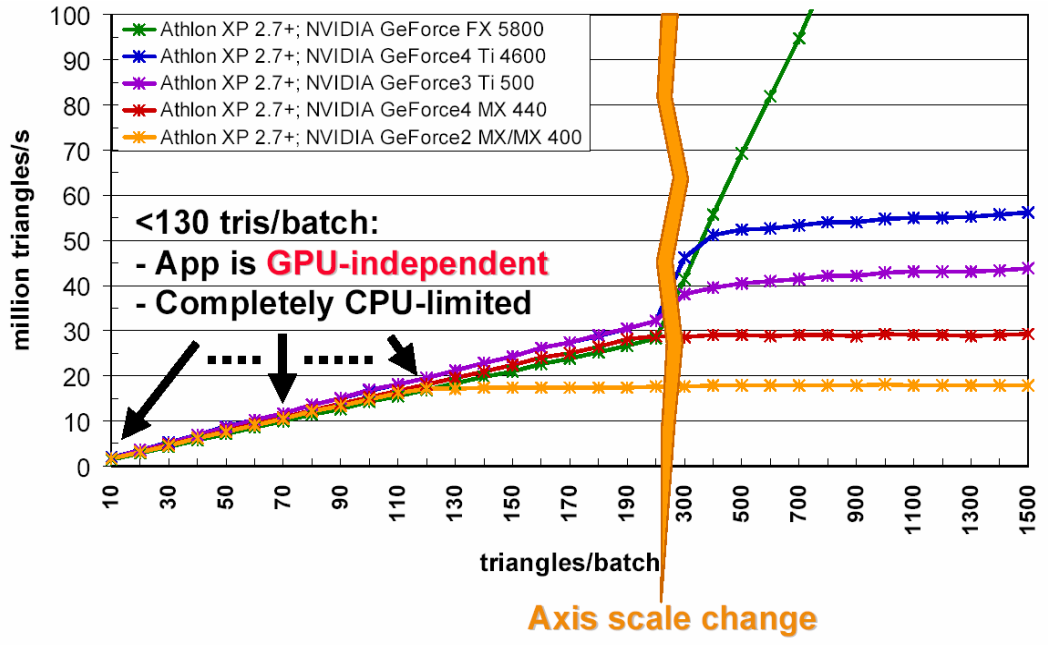


Figure 11. Geometry output as a function of triangles per batch.
Figure taken from [41].

We can verify the CPU limitation by looking at Figure 12. In the figure, we can see that even though the number of triangles per batch changes, the number of batches that can be rendered per second stays relatively constant unless we change the CPU speed. In fact, the 2.7x performance gain directly matches the 2.7x change in the processor speed. Even as the number of triangles per batch increases, there is no significant degradation in performance — meaning that one can add more triangles per batch at virtually no cost.

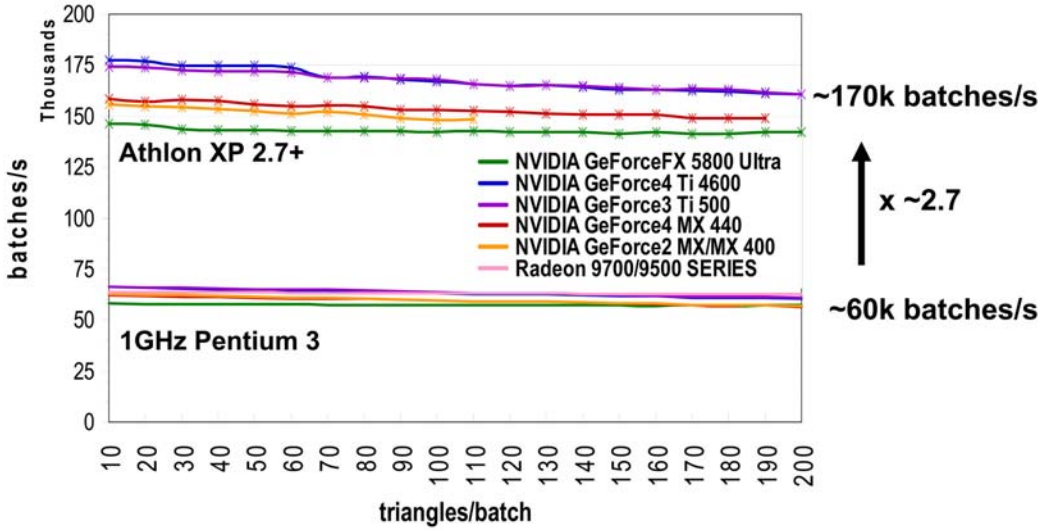


Figure 12. CPU-limited output in terms of batch performance.
The amount of batches per second changes with the speed of the processor.
Figure taken from [41].

The conclusion that we can draw from looking at these graphs is that it is important to minimize the number of batches being sent to the processor by maximizing the number of triangles sent in each batch. We can also extrapolate that the computation overhead introduced by an occlusion culling algorithm will *increase* the minimum number of triangles that need to be batched to be maximizing the rendering pipeline. It is important to note that aspects of the individual terrain use cases (the use of multi-pass shaders, for example) can further change the critical number, and for any case this number must be empirically determined for the particular rendering engine implementation. But clearly, we need to take the minimum batch size restrictions into consideration for our occlusion-culling algorithm.

5.2 Building a Hierarchical Framework

5.2.1 The Purpose of a Hierarchy

As we discussed in the last section, a valuable characteristic of our visibility data was the ability to merge the visibility information of multiple points, with no restriction on whether the terrain is regularly or irregularly sampled. This allows us to see if a group of mesh elements are occluded, rather than having to test each triangle individually. The first step that we can take to reduce the CPU bottleneck is to create groups of mesh elements that number no less than the empirically-determined minimum batch size as discussed above. Instead of testing the visibility of a single triangle, we ignore the individual triangle data and test only the visibility of the entire group, thereby reducing the amount of CPU work. If the entire group cannot be culled, then we render all of the elements of the group. It is important to realize that, as the previous figure shows, there is no performance difference between rendering one triangle or rendering 100 triangles if they are batched together — both operations will take the same amount of GPU time. Therefore, it is in our interest to batch geometry together as much as possible.

It also follows that once we have simple groups of geometry, some benefit may be had by eliminating multiple groups at once — the fewer occlusion tests that we have to perform, the better. By creating a hierarchy of mesh element groups, we can efficiently cull multiple groups at the same time. At the same time, we should also recognize that a *full* hierarchy is wasteful in this case. For example, there is no point in storing and querying the visibility information of all mesh elements grouped together — it's pretty obvious that the root node of our hierarchy would never be occluded in any normal situation. Therefore when we build our hierarchy, we should truncate it in such a way that the top handful of nodes are not queried (unless we find that they do offer a potential occlusion benefit, as in out-of-core systems).

5.2.2 Grouping Methods

So how do we build our groups of mesh elements? Since in the end we can group together any elements we choose, the simplest method would be to use groupings that already exist. For example, since many LOD implementations already use a hierarchy, it would make sense to utilize the existing hierarchical groupings as the occlusion groups. This minimizes storage, and simplifies the amount of work needed to compute occlusion. This also would simplify LOD calculations — if a group is occluded, the LOD calculations need not be performed for the group. Stewart establishes in [36] a simple quadtree-based algorithm for marrying occlusion culling with hierarchical LOD. His method relies upon a regularly-sampled mesh, however, which is not acceptable in our implementation. And while it would be possible to adapt his quadtree algorithm to grouping of irregular meshes, it would not be able to take advantage of some of the more important characteristics of irregular meshes, such as variations in mesh density. Furthermore, a quadtree-based algorithm imposes much more structure over the grouping of an irregular mesh than we would want or need.

Another potential method for determining our groupings would be to use texture-mapping cues. Since the changing of textures ends the current rendering batch on a GPU, it would be good to take texture-mapping into account when we form our groups. Regions of terrain with similar appearance often have similar visibility as well (think of a meadow or a hillside), so the resulting groupings will probably be more efficient.

5.2.3 The Crystal Method

Of course, the most efficient method of grouping will be one based directly on the visibility information. For example, it would not help us to group geometry from one side of a ridge with geometry from the other side of the ridge — no matter which side of the ridge you were on, you would see part of the group. When we create the groups, we want to ensure that the triangles in each group all have similar visibility characteristics with more or less the same horizon.

Suppose that we distributed a number of “seeds” randomly across the mesh, and from the seeds incrementally grew our groups one triangle at a time, using a breadth-first traversal of adjoining faces. Like crystals, the groups would grow until they collided with each other and had no other place to expand to. Since they grow outward from a single point, we guarantee spatial locality, which will typically yield regions of similar visibility. For further visibility-based constraint of the growth, let us also keep track of the “seed point” and use that in determining our growth — the crystal will not be allowed to grow out of sight (beyond the horizon) of the seed point. This helps to ensure that no crystal will grow over a ridge.

To assist in the creation of a hierarchy, we also maintain for each group a memory of crystal “collisions” during the growth process. Whenever a crystal tries to grow into another, we increment the collision count between the two crystals. In this way we keep a record of spatial convergence between crystals that we can use as a metric to guide the merging of groups to create a hierarchy. Note that this is not a record of spatial adjacency — crystals that are adjacent may not ever collide, unable to grow into each other because of the visibility limitations we introduced into the growth.

The random element of group creation acts as a hedge against the pitfalls of a bottom-up deterministic algorithm, and the method of crystal growth allows us to more easily distinguish visibility-contingent spatial locality from simple adjacency. It would be a much more complex and difficult task to formulate a bottom-up, divide-and-conquer deterministic algorithm that properly hedges the stricter visibility limitations of group growth in favor of the requirements for larger groups. The randomized process allows larger groups to grow more freely while keeping the overall viewshed of the group in check. As we will see later in this section however, randomization is used only in initial group creation — the group merging process is in fact deterministic.

The process of group creation invariably introduces the possibility that a large number of mesh elements may be missed, because they lie outside of the sight of all the seed points. This is in fact precisely what we want. These “orphan” elements are orphaned because they are distinctly different in terms of visibility from the crystallized groups.

Since these are likely to be grouped together, we can run another pass where we scatter a proportionally smaller number of seeds over the orphans. This process can be iterated until no large groups of orphans remain (which can be determined by looking at the size of the largest new groups being created in a given pass). The remaining orphans at this point are orphaned because of their wildly differing visibilities, and can be lumped into a group of "always visible" mesh elements.

At this point, we have associated every mesh element with a group, but we may have groups that are too small to be efficient, given the restrictions on standard graphics hardware as discussed before. So a final step would be to merge groups that are too small to be useful to us either into a similar small local group, or free them into the "always visible" set. The determination in this matter can be based on the collision record. If a trivially-sized group has collided with another trivially-sized group, we can merge the two groups into one larger group (merging their collision records and visibility information as well). Otherwise, we free the group into the "always visible" set. A more complicated metric could also be developed for merging a trivially-sized group with a larger, non-trivial group as well, if desired.

We have now created the bottom level of our hierarchy. To create the other levels of the hierarchy, we iterate through the groups, finding for each group the group that has the largest number of collisions with it. If that group has not already been merged with another group, those two groups are then merged for next level of the hierarchy. For the merged group, we store their combined collision record and their merged visibility data. The combined collision record will be smaller than the two separate collision records since the collisions between the two groups no longer need be recorded.

As we move up the hierarchy, continuing to merge groups, the collision records will slowly get smaller and smaller until, for some, no collisions remain. At this point, the group is considered top-level, and will no longer be considered for merging. This is part of the truncation of the hierarchy discussed earlier. The other groups, however, may continue to be merged until they reach a common node, which may span most of the height field.

If we go back to our occlusion-test algorithm, we remember that in the render loop one must test for occlusion in *all* sectors that our geometry lies in, relative to the viewpoint. If our groups span too much of the map, that could mean that for much of the map, every sector would have to be tested for the higher-level hierarchy nodes. This is less than optimal. It is therefore suggested that an empirically-determined cutoff point be established where a group is deemed too expansive to be merged any more. At such a point, the group would be considered top-level. For the author's implementation, this cutoff point was established to be when the circle circumscribing the merged group occupied more than 25% of the height field area. Note that the area of the geometry elements themselves does not determine the cutoff, but rather the expansiveness of the geometry.

5.3 Using the Hierarchy

Now that we have built a hierarchy of geometry groupings, we can generalize our occlusion-testing algorithm to accommodate the hierarchy with a recursive function:

```
TestOcclusionForGroup( HierarchyNodeGroup g )
    If c lies in Δ relative to g :
        For each sector s relative to g that c lies in:
            Project c into π_s to find ĉ
            Compare ĉ against the occlusion region for σ_s^g
            If ĉ is outside the occlusion region:
                If g is a leaf node:
                    Render g
                Otherwise:
                    For each child i of g:
                        TestOcclusionForGroup( i )
            Otherwise:
                Do not render g
```

Otherwise:

Do not render g

When we want to render the terrain, we call this recursive procedure for each top-level group in the hierarchy, and let it recursively handle the individual groups. With the top-down approach, we can cull large amounts of geometry without even having to test the visibility of the smaller groups contained in the hierarchy. In doing so we have achieved our goal of minimizing the amount of runtime computation required to perform the occlusion culling.

6 RESULTS

6.1 Algorithm Implementation

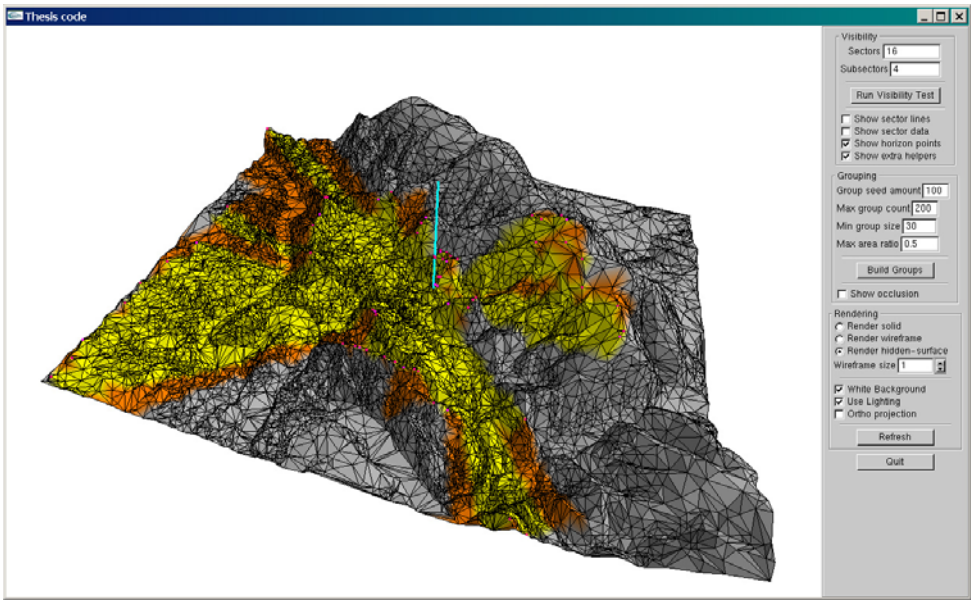


Figure 13. A screenshot of the implementation system.

6.1.1 System Description

The implementation of this algorithm was written with Microsoft Visual Studio 7.0 in Windows 2000, using OpenGL and GLUT for the graphics. The graphical user interface for the system was made using Paul Rademacher's GLUI toolset. All statistics are taken from runs of the program on a dual-processor 2.4 Ghz Dell desktop machine with 1 GB of RAM. A screenshot of the interface can be seen in Figure 13.

6.1.2 Terrain Dataset Creation

When faced with the task of identifying an appropriate data set to test the visibility and occlusion algorithms, there were a number of requirements identified.

Actual terrain data First and foremost, since the primary uses for this work involve terrain rendering, it was important to choose a data set that represented real and natural terrain. Artist-modeled and procedural terrains do not accurately mimic real-world landforms.

Suitability for occlusion culling Of course, it goes without saying that the dataset must stand to actually benefit from occlusion culling. If too flat of a dataset is chosen, the occlusion culling algorithm cannot adequately be tested.

Irregularly-sampled height fields One of the important contributions of this work is the ability to work with any height fields, regardless of its sampling method. Therefore, it is important to use irregularly-sampled height fields to ensure that the algorithm is not dependent on regular meshes.

Application-specific details A number of other application-specific matters need to be addressed for an adequate sample set. First of all, with irregular meshes, large variations in triangle density can be expected, where some areas of the height field have far more triangles than other areas. Secondly, as most renderers do not handle long “sliver” triangles, datasets with such geometry should be avoided.

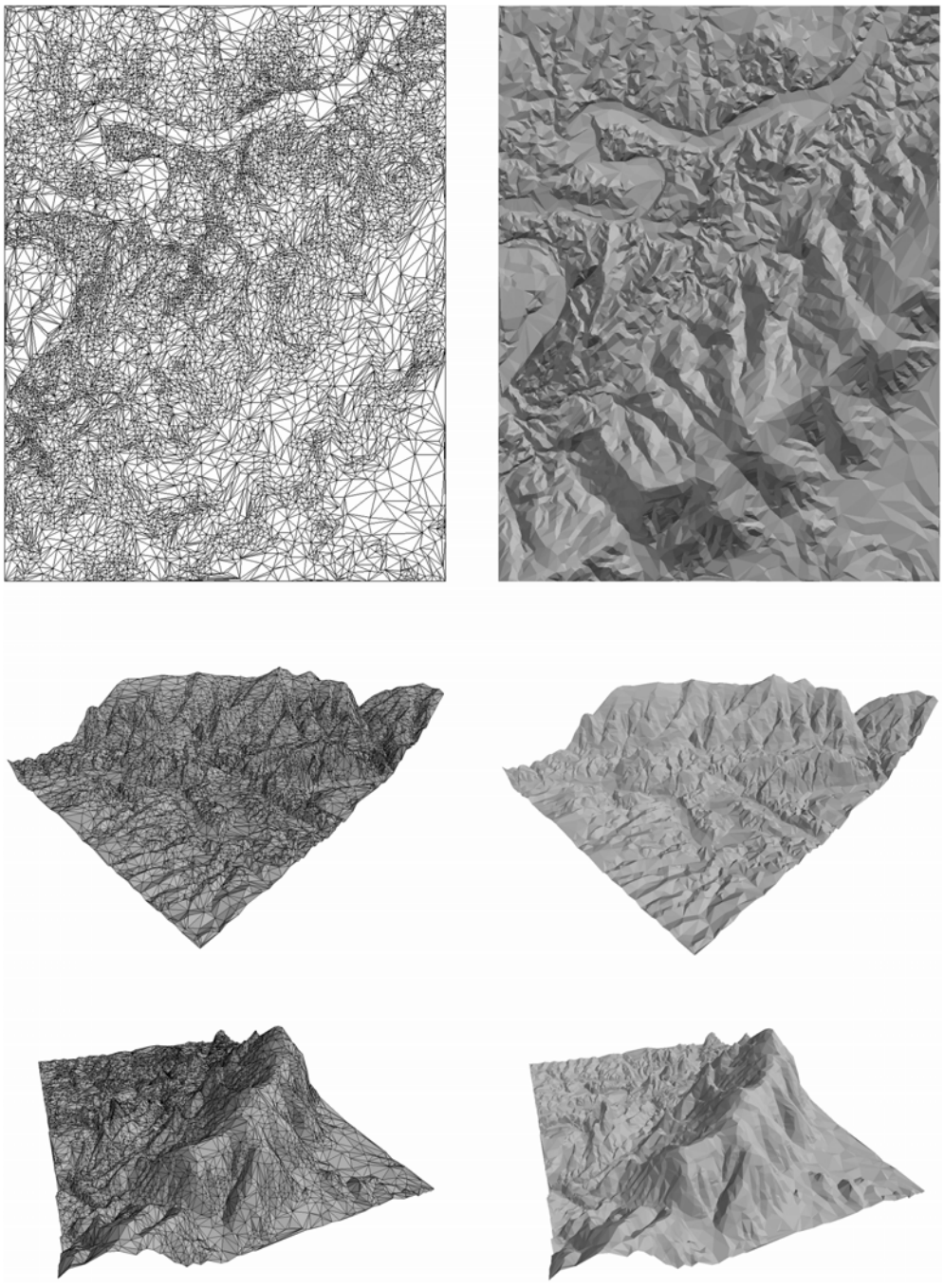


Figure 14. The Ashby Gap dataset.
Mesh density is shown on the left, and the traditional render method on the right.

Since most real terrain datasets tend to start with classic regular height-field datasets, the simplest way of obtaining an irregular dataset would be to take a traditional regular height field (such as those available from the U.S. Geological Survey), and reduce the number of polygons through mesh simplification. The “Scape” program outlined by Garland and Heckbert in [17] was used to drastically simplify the mesh geometry while still more or less preserved the original topology. The Scape program does not preserve the regular grid data, instead creating an irregular approximation of the original data set. As a result, areas of the terrain data set with smoother features will contain significantly fewer triangles in the simplification, while areas with much rougher features or potential silhouette edges will more closely match the original height field data. Furthermore, the Scape algorithm works to guarantee that as few sliver triangles as possible are created. It should be noted that given the bounded nature of finite data sets, some sliver triangles are unavoidable.

The final two datasets (shown in Figure 14 and Figure 15) settled upon came from meshes of different size and topography. Vertical distances in the datasets have been visually exaggerated to make the topographic features more clear in the visuals; such vertical exaggerations do not affect the visibility, grouping, or occlusion calculations. Table 1 shows the source information for each dataset.

Source DEM	Original Mesh Vertex Count	Simplified Mesh Vertex Count
Ashby Gap, Virginia	156,392	10,000
Mt. Tiefort region, California	1,048,576	100,000

Table 1. Terrain sample datasets.

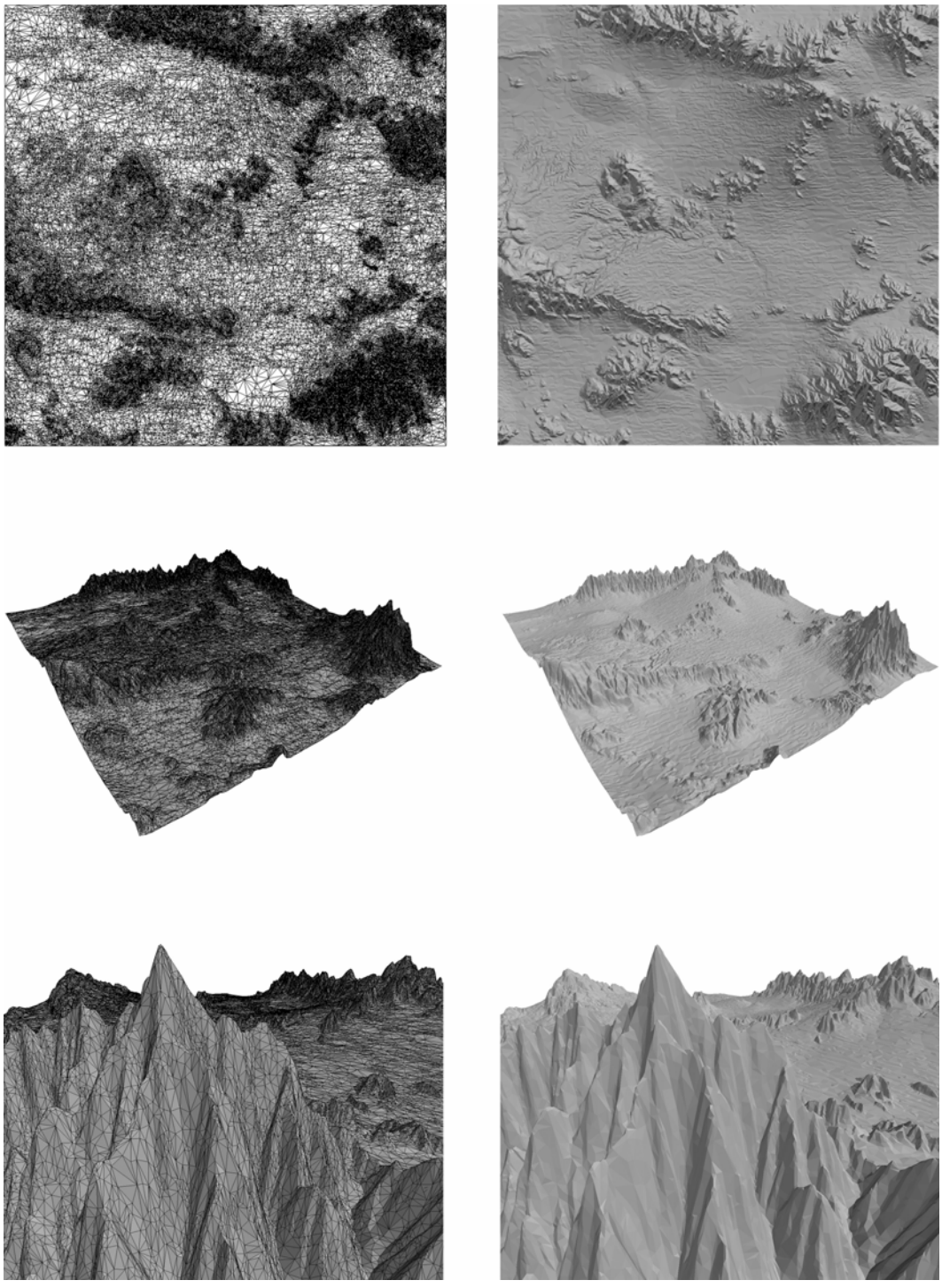


Figure 15. The Mt. Tiefort region dataset.
Mesh density is shown on the left, and the traditional render method on the right.

The two source meshes were simplified to less than one tenth of their original vertex counts to convert them into proper irregular meshes for testing. Other than the difference in size between the two datasets, their topographies also differed significantly — the Ashby dataset is dominated by a single mountain ridge and has relatively smooth topography variation, while the Tiefort dataset has very rugged terrain separated by large expanses of relatively smooth terrain.

In the end, the choice of terrain datasets worked very well, as the results differed significantly based on the type of terrain.

6.2 Visibility Determination

6.2.1 Implementation and Difficulties

For implementation of the visibility determination algorithm, the problem of non-conservative undersampling was not addressed in the end, primarily because such a solution is not the goal of this work. However, as it was important to establish the relative speed of the algorithm, all parts were optimized to run with the fastest possible runtime. This caused certain problems, however, as it turns out that the convex hull insertion and query algorithms that run in $O(\log n)$ time are rather involved and difficult to debug. For future implementations, it might be worthwhile to determine first if such an optimization is truly necessary — the complex hulls are rather small and the algorithms that run in $O(n)$ time are appreciably simpler with less overhead.

The static binary tree proved much easier to optimize, and half of the space required by the tree was saved by a simple optimization where only left child nodes were stored (since the right child nodes never get queried). The sorting was simple to speed up as well, since each particular ordering of the vertices gets used four times (as $\vec{a}^{\perp}$, $\vec{b}^{\perp}$, $-\vec{a}^{\perp}$, and $-\vec{b}^{\perp}$). In the end, the only significantly difficult part was the handling of the convex hulls.

6.2.2 Results

Dataset (sector count / subsectors per sector)	Average subsector calculation time (sec)	Total visibility calculation time (sec)
Ashby (8 / 1)	0.035	0.344
Ashby (8 / 2)	0.030	0.484
Ashby (8 / 4)	0.026	0.859
Ashby (8 / 8)	0.024	1.531
Ashby (16 / 4)	0.024	1.547
Ashby (32 / 2)	0.024	1.547
Ashby (64 / 1)	0.024	1.547
Tiefort (8 / 1)	0.873	7.063
Tiefort (8 / 2)	0.709	11.422
Tiefort (8 / 4)	0.607	19.516
Tiefort (8 / 8)	0.535	34.359
Tiefort (16 / 4)	0.532	34.141
Tiefort (32 / 2)	0.533	34.266
Tiefort (64 / 1)	0.542	34.985

Table 2. Visibility determination.

For the visibility determination process, each dataset was run with seven different sector-subsector input combinations to show the effects of changing the parameters relative to each other. Table 2 shows the running times of the algorithm with different parameters.

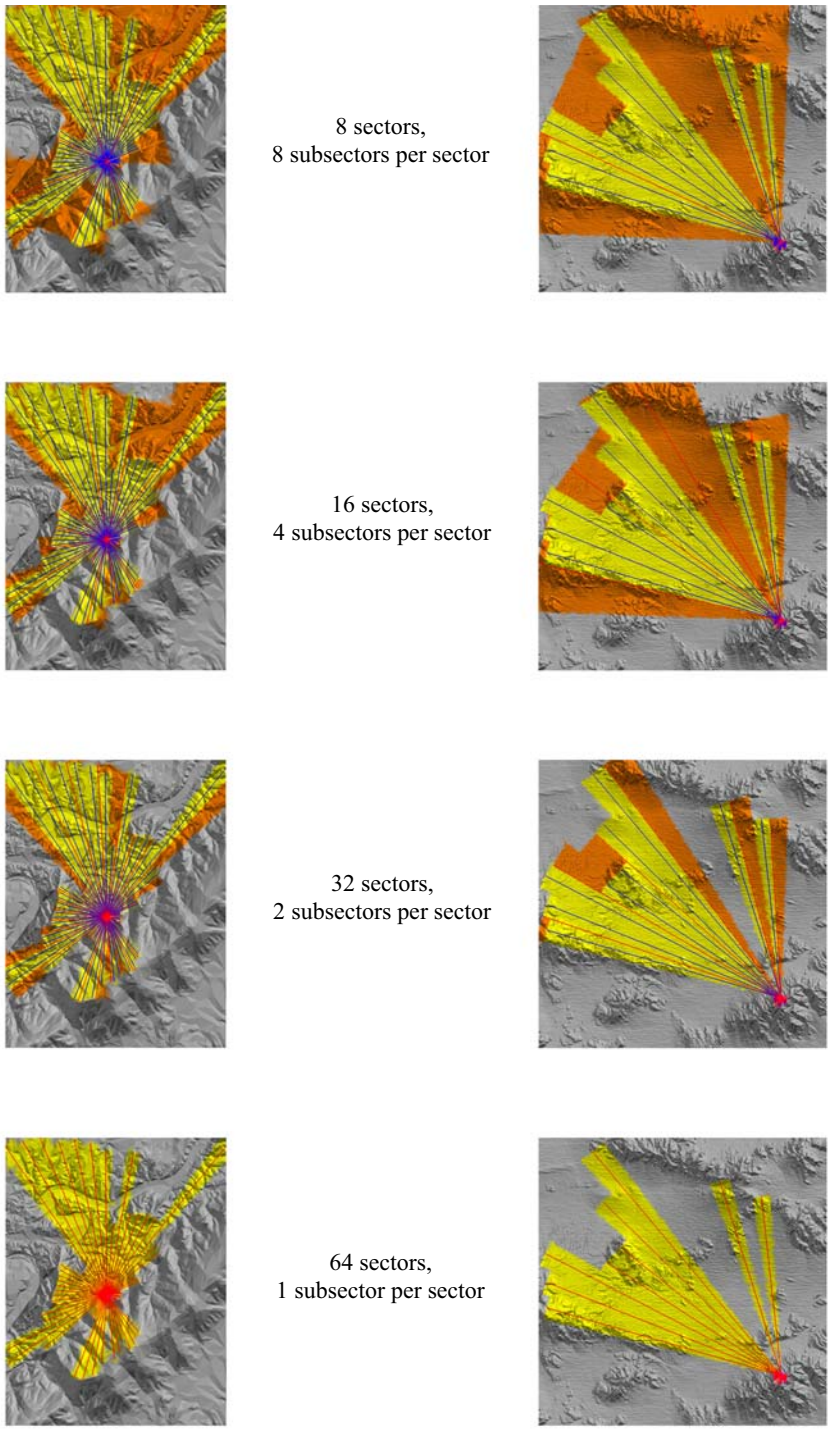


Figure 16. Visibility results vs. the sector-to-subsector ratio for a sample vertex.
The yellow areas are considered visible by the subsector visibility algorithm. The orange areas are conservatively considered visible because only sector data is stored.

It comes as no surprise that the overall running time is directly proportional to the total number of sectors and subsectors, as that is the main iteration loop. Similarly, it can be expected that the subsector calculation time will go down as more sectors are used, since fewer points will fall in a given sector for a given point.

The differences between the various subsector ratios tested become more evident when the results are visualized. In Figure 16 we can see the effects of changing the number of stored sectors while keeping the total number of sectors (or the number of sectors times the number of subsectors) constant. While the amount of non-conservative error remains the same, the areas of conservative error caused by sectoring (shown in orange) gradually diminish to zero. Of course, storing 64 sets of horizon information for every mesh element group is probably less than desirable from a storage perspective, so a happy medium would need to be found for the visibility results to be truly useful.

In the figure, it is important to note that the sector swaths should be round on the outside, but for the sake of a speedy implementation, the maximum extent of the sectors have been extended perpendicular to the central angle of the sector, since distances can then be calculated with a single dot product. Such an implementation leads to slightly more conservative error and causes the sectors and subsectors to not overlap entirely, but the speed benefits are desirable.

Figure 17 shows the effect of changing the number of subsectors while keeping the number of sectors constant. Note that when the total number of sectors remains low, non-conservative errors cause the viewshed to be smaller than it should be. When the total number of sectors gets larger, the small number of stored sectors leads to more conservative errors occurring.

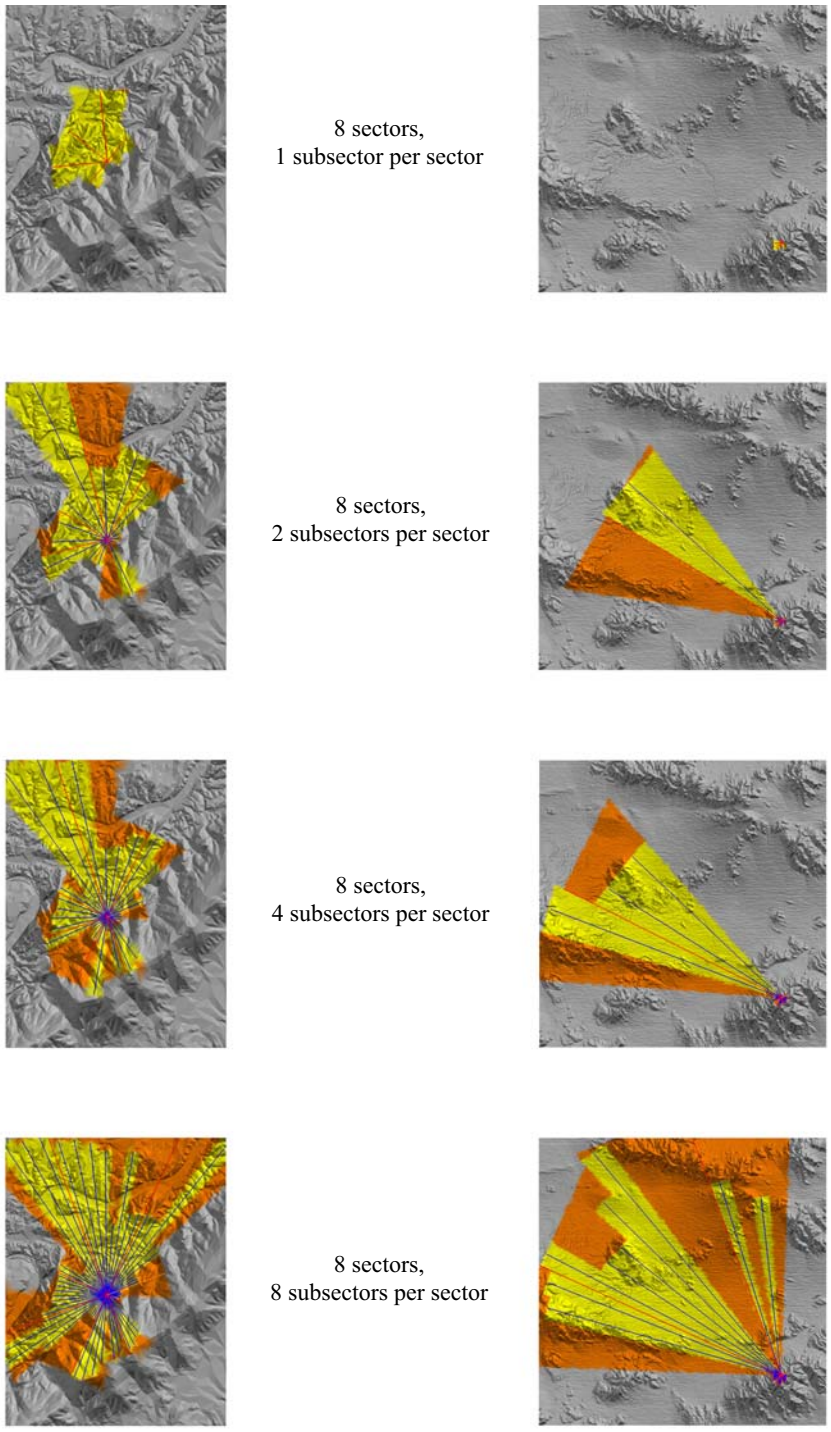


Figure 17. Visibility results vs. the number of subsectors for a sample vertex.
The yellow areas are considered visible by the subsector visibility algorithm. The orange areas are conservatively considered visible because only sector data is stored.

Since this particular implementation did not take conservative undersampling into account, the narrowness of the sectors can potentially cause problems. Figure 18 shows the result of such an undersampling error. Note that this will not cause any visible error in the occlusion culling, but will hinder the efficient grouping of mesh elements.

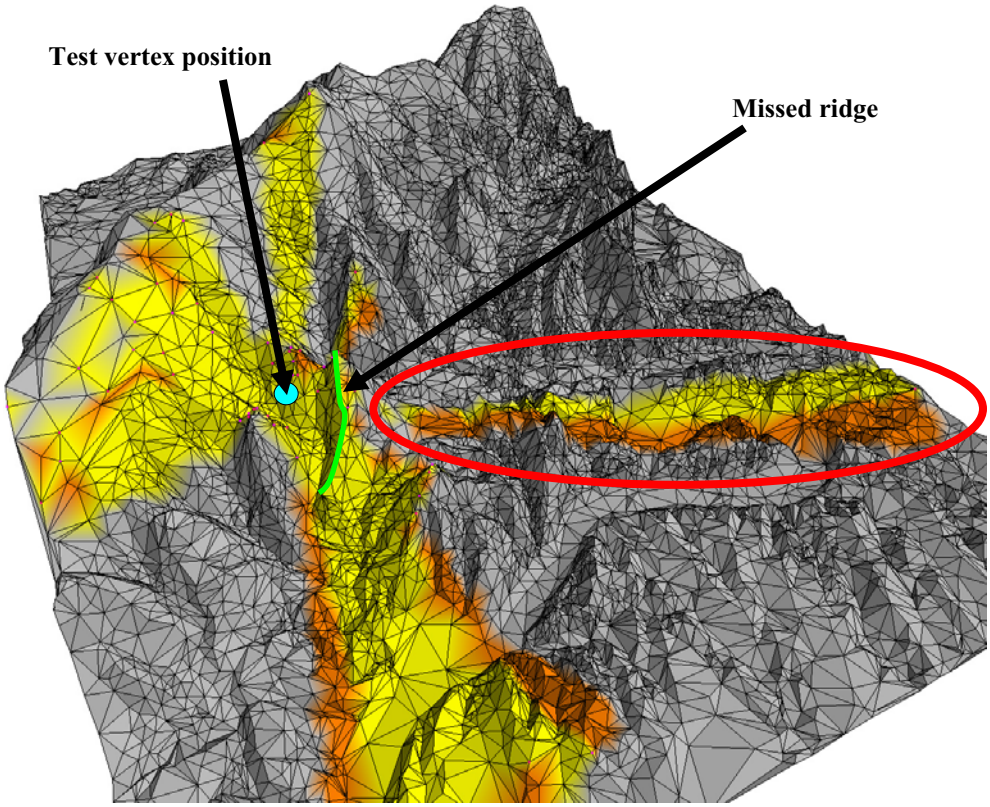


Figure 18. Undersampling error with thin sectors.

6.3 Mesh Element Grouping

6.3.1 Implementation and Difficulties

In the implementation of the "crystal method" grouping algorithm, it was found that the speed of the element grouping was not a bottleneck, so to simplify the visibility

merging process (and circumvent more $O(\log n)$ convex hull debugging) it was decided to implement the more simple linear convex hull merging algorithm.

In implementation, it was also noticed that when a seed started at the top of a mountain or ridge, it would spread out in all directions, creating a useless grouping that would always be visible. To combat this problem, a slight modification to the grouping algorithm was made where the horizon checks (to limit growth over a ridge) would always be made from the lowest vertex point in the group. This ensured that crystals would grow away from ridges instead of expanding across the top.

As a side effect of this, groups of orphans would tend to form at the top of ridges and peaks. This was actually desirable; as such geometry would always be visible from many viewpoints, anyway.

6.3.2 Results

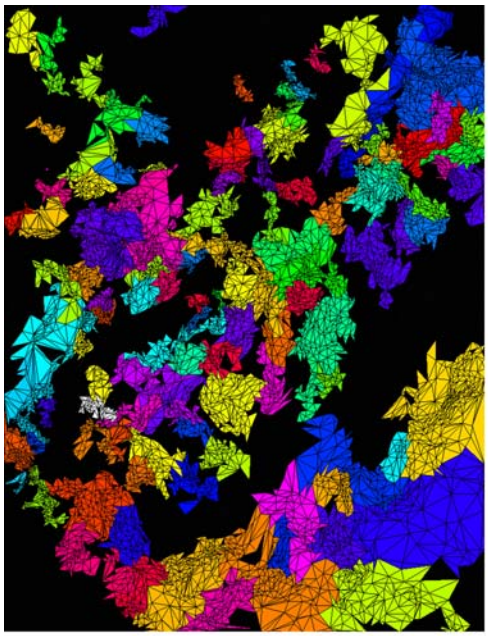
The goal of the grouping process is to create the largest possible groups of mesh elements sharing similar viewsheds. Therefore, it follows that one wants to create as few groups as necessary while still minimizing the number of always-visible orphan polygons (some of which will always exist). It was expected from the visibility data that when the number of sectors is too small, the grouping would not properly function. This is primarily because the non-conservative error artificially shrinks the viewshed, in effect causing the grouping algorithm to become "nearsighted." This is the reason that the grouping process ended for those datasets before the prescribed maximum number of orphan passes. This was in fact observed in the results.

Further analysis of the grouping results yielded another interesting factor. When the sectors themselves became too small, the conservative sector errors hindered the grouping process as well. This is because conservative undersampling errors can cause two adjacent triangles with very similar true viewsheds (*i.e.*, actual viewsheds as opposed to the approximated ones) to have completely different visibility results. If two triangles have wildly differing visibility results, they will not be grouped together.

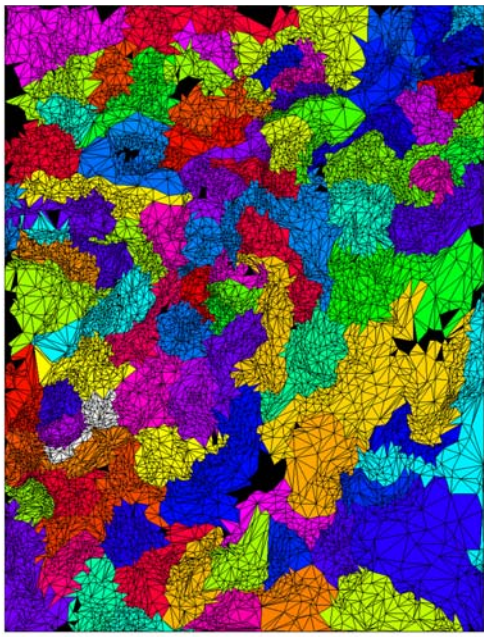
In Table 3, we can see the results of the crystal method on the test cases. For this test, all groups with less than 30 elements were orphaned — in many applications, one would probably want to increase that minimum size. Note also that due to the different terrain sizes, more crystal "seeds" were cast for the Tiefort dataset, 500 instead of 100.

Dataset (sector count / subsectors per sector)	Orphan passes	Final orphan count	Final group count	Total grouping calculation time (sec)
Ashby (8 / 1)	3	8,890	200*	0.047
Ashby (8 / 2)	5	2,878	200*	0.062
Ashby (8 / 4)	10*	766	161	0.062
Ashby (8 / 8)	10*	357	113	0.063
Ashby (16 / 4)	10*	697	144	0.094
Ashby (32 / 2)	10*	1,488	200*	0.140
Ashby (64 / 1)	3	5,659	200*	0.188
Tiefort (8 / 1)	6	22,036	1,000*	1.203
Tiefort (8 / 2)	10*	9,032	839	1.406
Tiefort (8 / 4)	10*	3,966	622	1.343
Tiefort (8 / 8)	10*	1,508	555	1.375
Tiefort (16 / 4)	10*	3,796	633	1.391
Tiefort (32 / 2)	10*	8,525	842	2.469
Tiefort (64 / 1)	5	27,961	1,000*	3.187

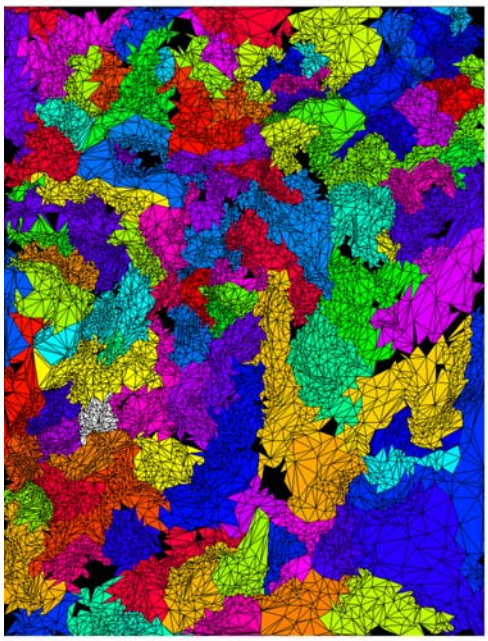
Table 3. Mesh element grouping results.
An asterisk denotes maximum allowable value (user-defined as a limiting parameter).



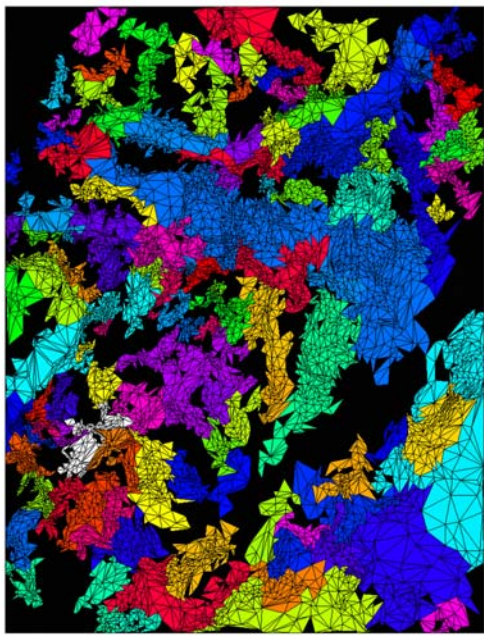
8 sectors,
1 subsector per sector



8 sectors,
8 subsectors per sector



16 sectors,
4 subsector per sector



64 sectors,
1 subsector per sector

Figure 19. Grouping results for the Ashby dataset.
Groups are displayed in randomized colors, orphans in black. Note that two adjacent groups may have similar randomized hues, almost appearing as one large group.

In Figure 19, we can see the several grouping results for the Ashby dataset. As can be seen from the distribution of orphans in the images, the best parameter combinations are where there is a relative balance between the number of sectors and subsectors. The case with 8 sectors and 8 subsectors yields the fewest number of groups as well as the fewest number of orphans.

Upon closer examination, however, it should be noted that the fewest number of groups and orphans does not necessarily yield the most optimal grouping. The case with 16 sectors and 4 subsectors performs quite well comparably, and the group boundaries tend to follow the ridges more closely — the extra orphans seem to come mostly from ridges or local peaks. Such orphans are actually desired — such geometry will most likely be rendered so often that it is a waste of processor cycles to test for their occlusion. When there are only 8 sectors, such a wide swath of the horizon is approximated by one value that the conservative error will mount.

It is likely that incorporating a fix to the undersampling conservative error problem will help the grouping perform much better for smaller-size subsectors, eventually yielding the exact horizon with enough sectors. However, in the interests of keeping storage amounts minimal, the grouping seems to (at least for the datasets tested) perform best with 16 sectors, and if 16 is too many, 8. Of course, intermediate values may be used in lieu of using powers of two to count the sectors.

By switching to a 3-D view of the grouping output, we can see in Figure 20 the distribution of the groups in relation to topography. The groups in the Tiefort dataset vary wildly according to the flatness of the terrain. This is primarily a byproduct of the mesh simplification algorithm used on the terrain, since the seeds were scattered randomly among the faces rather than over the height field domain. Over 90% of the groups in the latter example have more than 130 faces, with minimal orphans relative to the overall geometry size.

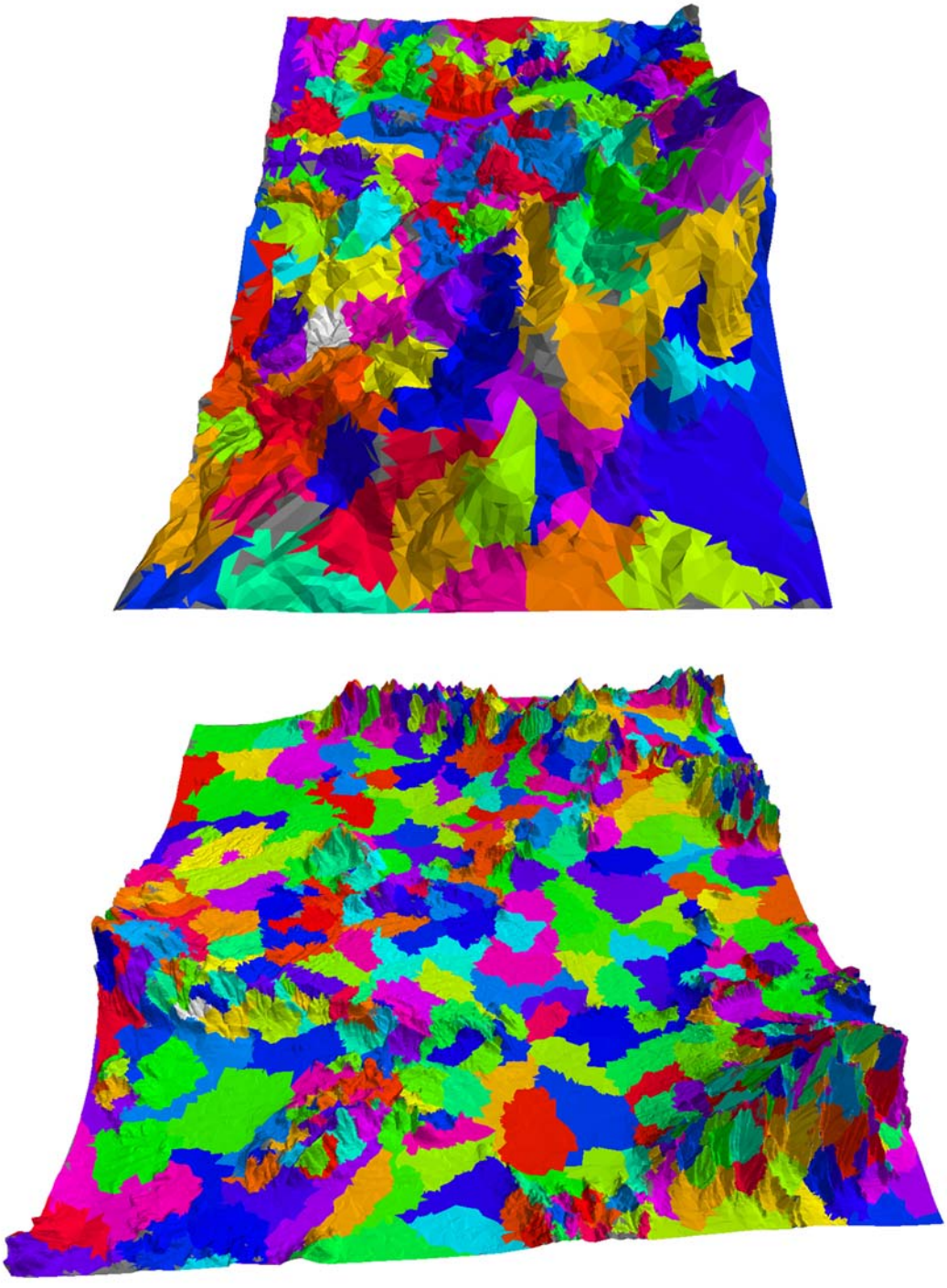


Figure 20. 3-D view of grouping for the Ashby and Tiefort datasets.
Grouping generated with 16 sectors containing 4 subsectors apiece. Orphans are shown in gray.
Note that the flatter areas have larger groups due to lower detail and mesh density.

6.4 Occlusion Culling

6.4.1 Implementation and Difficulties

The actual occlusion culling process was by far the easiest part of the application. After all of the occlusion regions and groups have been processed, it is simply a matter of traversing the hierarchical group nodes. There was virtually no problem in implementation.

6.4.2 Results

The simplicity and efficiency of the occlusion culling algorithm allowed for easy real-time updates to the active viewpoint from which to source the viewpoint. Figure 21 and Figure 22 show two sample viewpoint positions, for each respective dataset. The groupings that were shown in Figure 20 are used for the run. A red target marks the location of the viewpoint, and the occluded terrain is rendered in wireframe. In order to show as much of the culling as possible, the FOV of the view frustum was set to 360°, or omnidirectional. None of the viewpoints imaged below had any amount of non-conservative error, but all of them resulted in at least some conservative error. (This is to be expected, due to the nature of the algorithm.)

In running a number of more occlusion culling tests, it was verified that when the number of sectors got too low, non-conservative errors started occurring more frequently. This was primarily seen in situations with occlusion occurring over large distances in the Tiefort dataset, where the visibility sectors got too large to be accurate (sometimes even in the case with 16 sectors and 4 subsectors per sector). The solution would be of course to use more subsectors, with a fix for the undersampling issues. Since the visibility determination is quite fast, even a doubling of the runtime would not be a particularly large change.

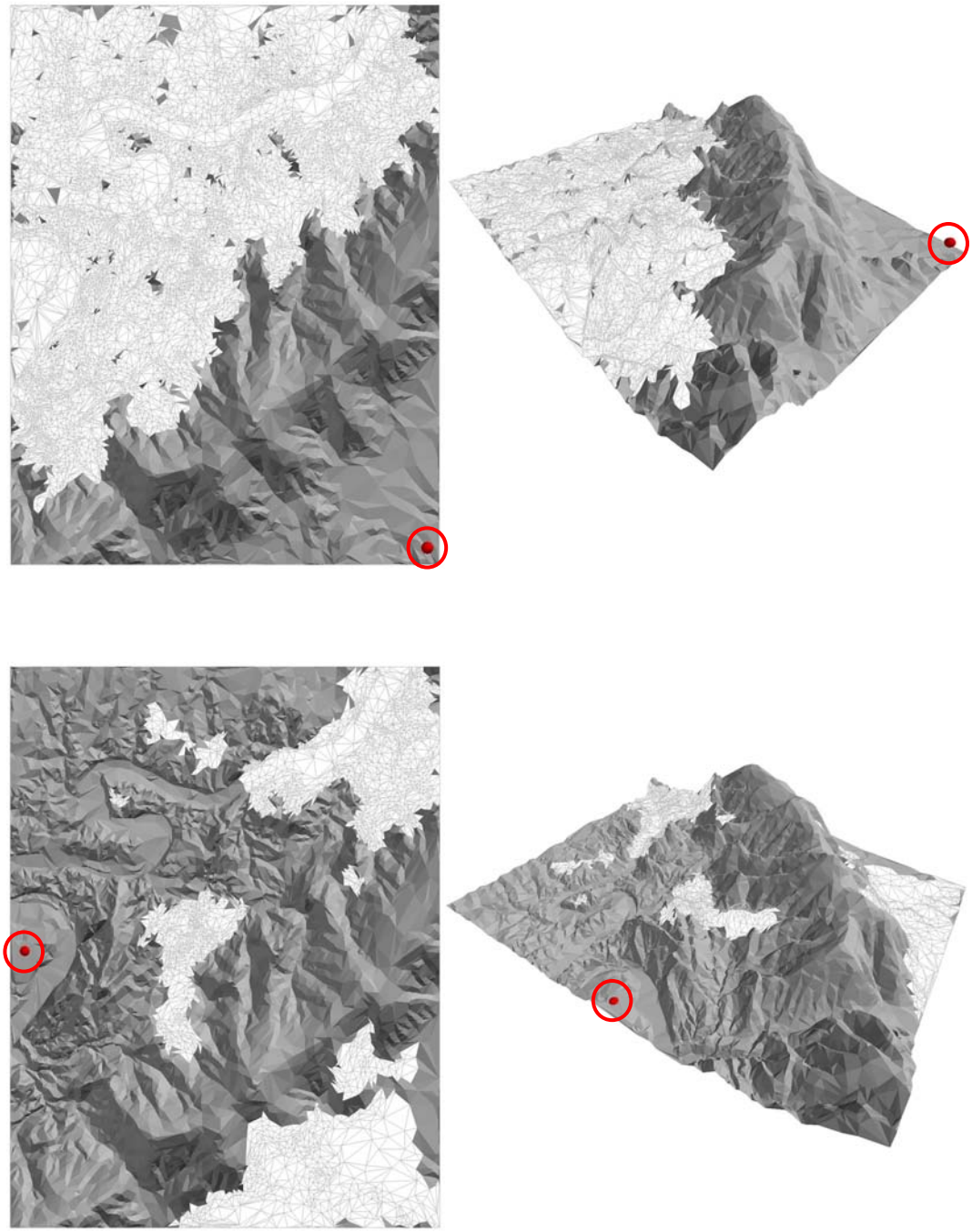


Figure 21. Occlusion culling with the Ashby dataset.
Occluded regions are rendered as a light-colored wireframe.

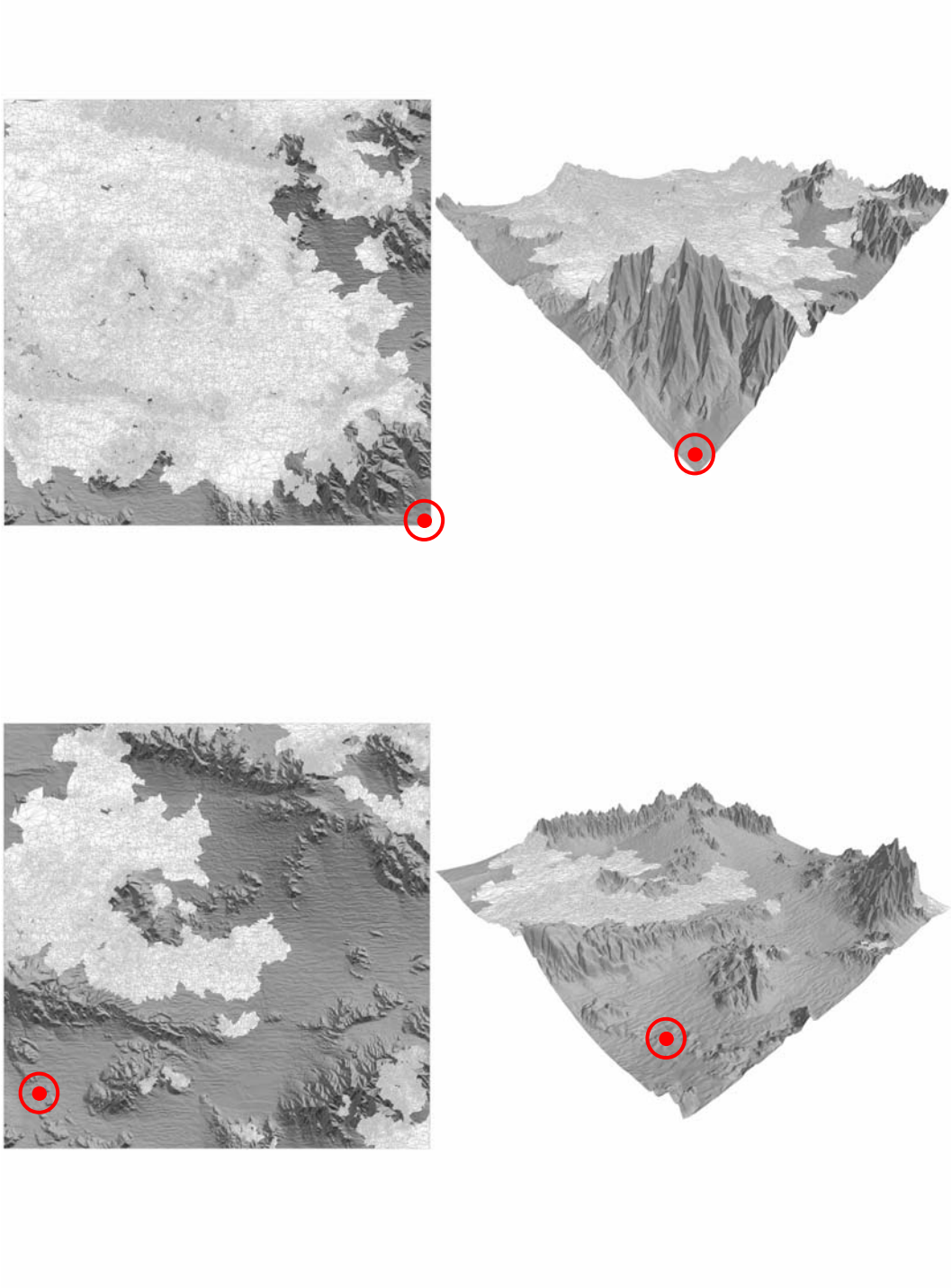


Figure 22. Occlusion culling with the Tiefort dataset.
Occluded regions are rendered as a light-colored wireframe.

7 FUTURE WORK

While useful in and of itself, there are a number of potential extensions to the above method of occlusion culling for height fields. In this section we discuss possible future work that could continue from the theory discussed in this paper.

7.1 Out-of-Core Mesh Handling

While finite height fields are very useful in most current applications, there are many current and future applications where it is conceivable that all of the height field data may not be able to fit in core memory. In such situations, it would be necessary to deal with visibility data in novel ways to circumvent the storage limitations.

There are several possible ways that one could approach out-of-core visibility determination for occlusion culling. First, it is conceivable that the height field could be partitioned into logical, smaller divisions that would fit into core memory, and then each partition could be considered a group at some point in a greater hierarchy for the purposes of occlusion culling. Of course, this may not yield the best grouping, as simple partitioning schemes may group highly visible mountains with terrain that would never be seen outside the partition. Alternatively, one might be able to propagate visibility information from one partition to the next.

If a suitable out-of-core solution could be found, this would be of great benefit to a number of height field applications. Since an out-of-core simulation is inherently unable to process the entire height field at once, occlusion culling would be critical to quickly rendering a scene without visual artifacts.

7.2 Terrains with Objects

While we have dealt with the occlusion of terrain geometry, we have not addressed the fact that rendering systems often are drawing more than just a height field for terrain.

Objects like trees, rocks, and buildings can all affect the rendering of a scene. Culling objects that are occluded by the terrain could greatly affect the rendering time. Similarly, culling the terrain occluded by larger objects could also optimize applications that merge indoor and outdoor spaces. A useful extension of the work in this paper would be to establish a framework for determining terrain-object occlusion in an efficient manner.

7.3 Simulation Bounding

In computer games and other real-time situations, there is often a need to bound various computationally intensive behaviors. Sometimes there are physics simulations that take a sizable amount of CPU time, or maybe a large number of artificial intelligence entities to be simulated. Without some method of bounding the computations, the processing load could quickly reduce the system to a crawl. By taking advantage of visibility knowledge, one could limit simulations to only occur in visible space. This could potentially be an extension of the occlusion and object culling frameworks.

8 CONCLUSION

In this paper, we have achieved a method for performing efficient runtime occlusion culling for height field rendering through an extensive preprocessing step. By generalizing the extant work in visibility-based terrain rendering, we find that it is possible to establish a much broader framework than has been discussed in previous works. Of these the most notable are the ability to drop the restrictions of using only regularly-sampled height fields, and allowing multiple metrics for the development of a visibility-determination hierarchy.

We also introduced a new method for visibility-sensitive hierarchy formation. With the “crystal” method, we can build groups that are formed and merged with visibility-aware metrics. This allows the occlusion culling to be much more efficient than with the use of traditional hierarchy-building algorithms such as quadtrees. Furthermore, the method is customizable, leaving space for empirical determinations that allow fine-tuning of multiple parameters that will behave differently in the varied applications of the algorithm.

The potential applications for such a system are widespread, as is the potential for customization and extension of the ideas presented. It would also be beneficial to see further development of the ideas in the future work section, especially in the use of out-of-core meshes where the expansive nature of the data sets would find the most use of runtime occlusion culling.

REFERENCES

[1] P. Agarwal and P. Desikan. An efficient algorithm for terrain simplification. In *8th ACM-SIMA Symp. on Discrete Algorithms*, 1997. Accessed on February 8, 2003, http://www.cs.duke.edu/~pankaj/papers/terrain.ps.gz.

[2] K. Baumann, J. Döllner, K. Hinrichs, and O. Kersting. A hybrid, hierarchical data structure for real-time terrain visualization. In *IEEE Proc. Computer Graphics International '99*, pages 85–92, 1999.

[3] P. Brown. Selective mesh refinement for interactive terrain rendering. Technical report, University of Cambridge Computer Laboratory, Cambridge, England, Tech. Rep. 417, February 1997.

[4] B. Cabral, N. Max, and R. Springmeyer. Bidirectional reflection functions from surface bump maps. *Computer Graphics (SIGGRAPH '87 Proc.)*, vol. 21, pages 273–281, July 1987.

[5] D. Cohen-Or, Y. Chrysanthou, and C. T. Silva. A survey of visibility for walkthrough applications. *SIGGRAPH 2001 Course Notes*, vol. 4, 2001.

[6] M. De Berg and K.T.G. Dobrindt. On the levels of detail in terrains. In *Proc. 11th ACM Symp. on Computational Geometry*, Vancouver, BC, pages c26–c27, 1995.

[7] M. De Berg, M. Overmars, and O. Schwarzkopf. Computing and Verifying Depth Orders. Technical report, Utrecht University, Utrecht, Netherlands, Tech. Rep. RUU-CS-91-41, November 1991.

[8] M. De Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, Germany, 1997.

[9] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. Polyhedral terrain description using visibility criteria. Technical report, Institute for Applied Mathematics, National Research Council, Genova, Italy, Tech. Rep. 17, 1989.

[10] L. De Floriani, P. Magillo. Visibility computations on hierarchical triangulated terrain models. In *Geoinformatica*, Vol.1, No.3, pages 219–250, 1997.

[11] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *IEEE Visualization '97 Proc.*, 1997, pages 103–110.

[12] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *IEEE Visualization ’98 Proc.*, pages 43–50, October 1998.

[13] L. De Floriani, P. Magillo, and E. Puppo. VARIANT: A system for terrain modeling at variable Resolution. In *Geoinformatica*, Vol.4, N.3, September 14, pages 287–315, 2000.

[14] M. Duchaineau, *et al*. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization '97 Proc.*, pages 81–88, 1997.

[15] C. Erikson. Polygonal simplification: an overview. Technical report, Dept. of Comp. Sci., University of North Carolina at Chapell Hill, NC, UNC Tech. Rep. TR96-016, 1996.

[16] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, Second Edition in C*. Addison-Wesley, Reading, MA, USA, 1996.

[17] M. Garland and P. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, CS Department, Carnegie Mellon University, Rep. CMU-CS, pages 95–181, 1995.

[18] M. Garland and P. Heckbert. Fast triangular approximation of terrains and height fields. *submitted for publication in SIGGRAPH '97 Course Notes*. Accessed on January 15, 2003, ftp://ftp.cs.cmu.edu/afs/cs/project/anim/ph/paper/multi97/release/heckbert/terrain.pdf.

[19] M. Hadwiger and A. Varga. Visibility culling. For seminar, Inst. of Comp. Graphics and Algorithms, Vienna University of Technology, 1998. Accessed on February 2, 2003, http://www.cg.tuwien.ac.at/~msh/viscull.pdf.

[20] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. In *Multiresolution surface modeling (SIGGRAPH '97 Course Notes #25)*, ACM SIGGRAPH, pages 1–19, 1997.

[21] H. Hoppe. Efficient implementation of progressive meshes. In *Computer Graphics*, Vol. 22, No. 1, pages 27–36, 1998.

[22] H. Hoppe. Progressive meshes. In *Computer Graphics (SIGGRAPH '96 Proc.)*, pages 99–108, 1996.

[23] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98 Proc.*, pages 35–42, 1998.

[24] B. Jünger. Terrain modeling as an application of a fast reconstruction algorithm for Delaunay triangulations. Technical report, CS Department, UBC, Tech. Rep. 978, 1997.

[25] M. J. Katz, M. H. Overmars, M. Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry: Theory and Applications 2*, pages 223–234, 1992.

[26] F. Law and T. Tan. Preprocessing occlusion for real-time selective refinement. In *Proc. of the 1999 symposium on Interactive 3D graphics*, pages 47–53, 1999.

[27] P. Lindstrom, *et al*. Real-time, continuous level of detail rendering of height fields. In *Computer Graphics (SIGGRAPH '96 Proc.)*, pages 109–118, 1996.

[28] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proceedings of IEEE Visualization 2001*, pages 363–370, 2001.

[29] J. Mortenson. Real-time rendering of height fields using LOD and occlusion culling. Master's thesis, Dept. Comp. Sci., Univ. College London, London, England, 2000.

[30] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[31] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm for polyhedral terrains. In *Mathematical Computing Modeling*, 21(5), pages 89–104, 1995.

[32] S. Röttger, W. Heidrich, P. Slussallek, and H-P. Seidel. Real-time generation of continuous levels of detail for height fields. In *Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualization*, pages 315–322, 1998.

[33] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Rendering Techniques 2001 (Proc. EG Workshop on Rendering)*, Springer Verlag, pages 151–162, 2001.

[34] A. Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), pages 82–93, March 1998.

[35] A. Stewart. Fast horizon computation for accurate terrain rendering. Technical report, CS Department, Univ. Toronto, Canada, Tech. Rep. 349, June 1996.

[36] A. Stewart. Hierarchical visibility in terrains. *presented at Eurographics Rendering Workshop*, June 1997.

[37] E. Szoka. Triangle strip preserving LOD. Accessed on February 5, 2003, http://chat.carleton.ca/~eszoka/tstriplod/tstrip.htm.

[38] S. Teller. Visibility computations in densely occluded polyhedral environments. Ph.D. thesis, University of California at Berkeley, CA, 1992.

[39] T. Ulrich. Rendering massive terrains using chunked level of detail control. *presented at SIGGRAPH* 2002. Accessed on February 3, 2003, http://tulrich.com/geekstuff/sig-notes.pdf.

[40] Wiley, *et al*. Multiresolution BSP trees applied to terrain, transparency, and general objects. In *Graphics Interface*, pages 88–96, May 1997.

[41] M. Wloka. “Batch, batch, batch:” what does it really mean?. *presented at Game Developers Conference 2003*, March 2003. Accessed on May 10, 2003, http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.ppt.

[42] J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96 Proc.*, pages 327–334, 1996.

VITA

Paul Michael Edmondson
280 Merrydale Rd., Apt. #7
San Rafael, CA 94903-3945

Paul Michael Edmondson started attending Texas A&M University in 1997, after graduating from Jersey Village High School, outside of Houston, Texas. In 2001, he earned a Bachelor of Science Degree in Computer Science, and continued with his studies, finishing the coursework for a Master of Science in 2003.

During his graduate studies, Paul spent a near-majority of his time working with students and faculty at the Texas A&M Visualization Laboratory, studying computer graphics and simulation. He started studying terrain rendering as a result of his forays into the world of game development. Driven by a desire to create realistic, real-time landscape renderings for use in computer games and other similar applications, he started studying the topic as an undergraduate, and continued the studies into graduate years. His work was pursued as a self-motivated topic, independent of the research of any other student or professor.

Upon finishing graduate studies in 2003, Paul moved to California to work at the game development company, LucasArts, where he has been working since. In his spare time he teaches at the Art Institute of California – San Francisco.