Hidden Surface Elimination on Reduced Terrain Data using Quad Trees

Patrick J. Angeles
pangeles@cs.oberlin.edu

Oberlin College Computer Science Program

Abstract

Oct-trees are commonly used in solid modelling to accelerate the rendering process. This paper presents a hidden surface algorithm similar to the basic oct-tree algorithm, but tailored for specific use with terrain data. The algorithm aims to reduce computation by taking advantage of a terrain's two-dimensionality. The term *quad-tree* is introduced to denote an hierarchichal spatial structure in two dimensions whose properties resemble those of a three-dimensional oct-tree.

1 Introduction

A basic problem associated with terrain rendering is the immensity of the data set. Compared to the whole set, the portion of the terrain that is actually seen by the viewer is relatively small. To illustrate, imagine a viewer sitting in the middle of a vast square field. Regardless of the direction of the viewer's gaze, more than one half of the field is always unseen. In most cases, the field of vision is restricted such that there is a constant area which falls under the viewer's gaze. If we had a quick method of isolating this visible area from the rest of the terrain, we can speed up the rendering process by a large factor.

This paper introduces an efficient method for eliminating portions of the terrain that are out of view. The algorithm is designed for objects that are spread out across the xy plane and whose minimum and maximum z coordinates are relatively close. Although unsuitable for general graphics applications such as solid modelling, it works well with terrain rendering.

2 Background and Related Work

2.1 Terrain Data

Terrain data in its raw form comes as a two-dimensional grid with a height field. In coded form, it would appear as `float grid[ MAXX][ MAXY]`. A point p would appear as an ordered triplet (x, y, z) where $z = grid(x, y)$. Before rendering, we construct triangles out of the grid, essentially splitting each unit "square" into two triangles. To render, we simply push each triangle through the pipeline.

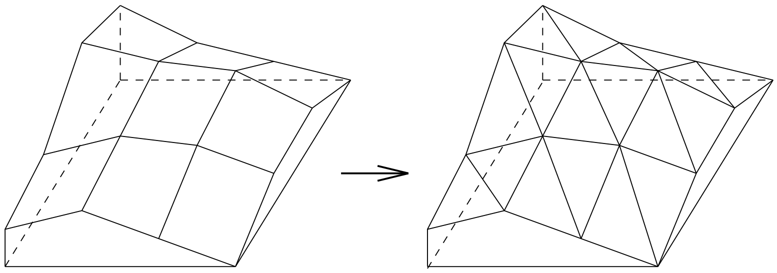

Figure 1: Basic and triangularized grid.

Representing the terrain object in terms of the grid has the advantage of being compact. What's more, the indexing scheme makes it easy to restrict the rendering process to any subgrid. Li and Curkendall describe a *ray-identification* algorithm which takes advantage of the grid format to restrict processing within a bounded region in xy [1].

An alternative representation to the grid format is the pointer-to-vertex format. In this scheme, we keep two lists—one for vertex coordinates and another for triangle definitions. An entry in the vertex list would is structure containing that vertex' x, y and z coordinates. An entry on the triangle list would contain three pointers, each one refering to that triangle's vertices in the vertex list:

```
typedef struct { float x,y,z; } vertex;
typedef struct { vertex *verts[ 3]; } triangle;
vertex vertexlist[ MAX_VERTS];
triangle trilist[ MAX_POLYS];
```

This format is more flexible, as it places no restrictions on vertex positioning. It can therefore represent such things as reduced terrain meshes. However, there is a compromise involved–the price of flexibility is the loss of spatial coherence that is otherwise implicit in the grid format. The pointer-to-vertex representation prohibits us from limiting the rendering operations to a subgrid. The problem becomes acute in the case of terrain rendering. Terrain rendering usually involves the use of a large data set, which makes the ability to bound the domain extremely necessary.

We can restate the content of this paper as an attempt to restore spatial coherence on a data set that uses pointer-to-vertex representation. The motivation is to allow for fast hidden surface elimination on a large, reduced data set.

2.2 Mesh Reduction

An object's surface is represented, in the pointer-to-vertex convention, as a mesh defined by two things: a list of points and another list of triangle definitions. Given these, one might want to reduce the mesh if **a)** the mesh is dense (i.e., the surface area of the average triangle is very small) or **b)** there are points on the mesh which do not contribute to the overall appearance (e.g., flat areas).

There are a variety of approaches to simplifying a mesh. Hoppe, et al., [2] describe a method called *mesh optimization* where the reduction problem is solved by minimizing an *energy function* that "captures the competing desires of tight geometric fit and compact representation." Turk [3] introduces a surface re-tiling method that finds a new mesh through point repulsion. The surface is sampled randomly, and the resulting points are repulsed against each other to distribute them along the surface. Then a process referred to as *mutual-tesselation* is used to create a mesh from the new points. For this project, a third method was implemented, based on the *decimation* algorithm described by Schroeder, et al. [4].

The basic *decimation* algorithm goes through each vertex in the data set and performs the following:

- Construct an average plane using the surrounding vertices.
- Measure the distance between the given vertex and the average plane.
- If the distance is less than a certain value, discard the vertex.
- If the vertex is discarded, triangulate the resulting "hole".

From this, a multiple-pass version can be implemented which automatically determines the optimal distance value for each pass until a specified number of vertices are eliminated.

The *decimation* algorithm works well with terrain data, which tend to have large flat areas such as lakes and planes. After applying the *decimation* operation, the resultant data is characterized by high-point density in detail rich areas, and sparse representation in flat expanses of terrain. By contrast, the *optimization* and *tesselation* algorithms tend to produce points that are roughly equi-distant from each other, which result in simple flat surfaces with excess vertices.

2.3 Hidden Surface Elimination and Hierarchical Spatial Trees

Jackins and Tanimoto [5] and later on Meagher [6] are generally credited for introducing the use of hierarchichal structures for use in three dimensional rendering. In these papers, they describe the oct-tree structure. Several improvements and modifications on the general oct-tree structure have since been proposed.

Glassner [7] and several others, for example, propose the use of oct-trees and other hierarchical structures to speed up ray-tracing. The premise for using an oct-tree structure is to maintain object-space coherence, thus making hidden surface elimination a faster process.

Greene, et al. [8] take it further by proposing a second structure, "z pyramid" to preserve image-space coherence for z-buffer algorithms. For the interested reader, Yamaguchi, et al. [9], provide a good entry point to literature regarding oct-trees and related data structures.

3 The Quad-Tree Algorithm

The quad-tree method is derived from traditional visibility algorithms but specialized for work with terrain data. The idea is to project the view volume onto the xy-plane and perform visibility operations based on the 2D projection. A quad-tree structure, not unlike the traditional oct-tree, is used to speed up the visibility determination.

3.1 The Quad-Tree Structure

A quad-tree is an hierarchical spatial tree in two dimensions. Each node in the tree contains boundary information of the form of (x_{min}, y_{min}) and (x_{max}, y_{max}) to describe the rectangular area it covers. In addition to this, each node can either have four children that are themselves nodes, or describe a list of polygons that are classified under that node.

A polygon is said to be classified under a certain node if any of its vertices lie within the rectangular bounds covered by the node. This is done by dropping the z coordinate of the vertex. Thus, given triangle T and node N, we say that T is classified under N if $(x_{min} \leq v_x \leq x_{max}) \bigwedge (y_{min} \leq v_y \leq y_{max})$ where $v \in T_{vertexlist}$ and $x_{min}, y_{min}, x_{max}, y_{max}$ are the boundary definitions for N.

In creating a quad-tree, we start with a root node whose boundaries cover the entire data set. We divide this area along the x and the y axes to produce four rectangular quarters which become the children of the current node. We then recurse on each of the four nodes until a specified criterion is met. In this implementation, the children are spawned until a specified depth is reached. Other criteria are available. One can, for example, end the recursion upon spawning a node whose area (or the number of triangles it covers) reaches a certain minimum.

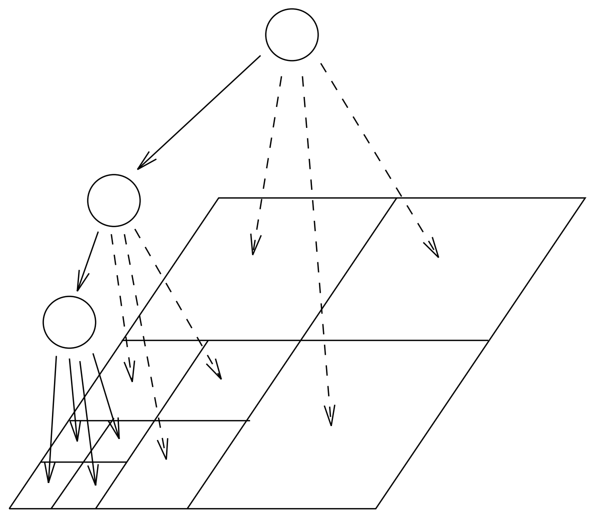

Figure 2: Spatial hierarchy of quad-trees.

With the termination of each recursion, triangles are assigned to the terminating node (or leaf). Note that we only require triangle assignment to leaves. This is possible because the area covered by any node is identical to the aggregate area covered by its children. Therefore, the aggregate area of all the leaves should be identical to the area covered by the root node (i.e., the entire data set).

3.2 Redundancies

There are cases in which a triangle might be classified under more than one node. This happens when a vertex belonging to that triangle lies on a split line, or when one vertex lies in the area covered by one node while a second vertex lies in an area covered by a different node.

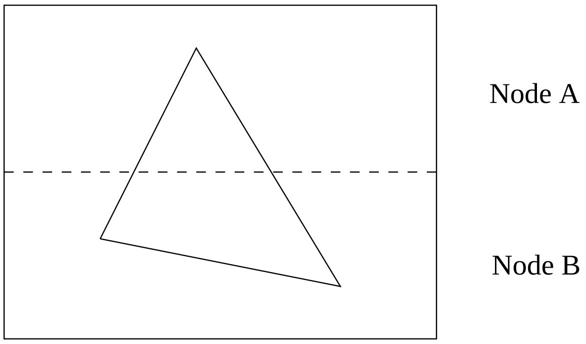

Figure 3: Triangle in split line between nodes A and B.

One way to deal with this problem is to split the triangle along the boundary and reassign the parts to their respective nodes. A simpler solution would be to redundantly classify the triangle under all nodes that could possibly contain it. Of course, we run into the problem of processing the redundant triangle more than once. To minimize the cost, a flag is added to the triangle structure which indicates whether a triangle has been previously rendered.

3.3 Visibility Determination

To use the quad-tree structure for determining visible surfaces, we traverse through the quad-tree starting from the root. For each node, we check its boundary values to see if it is visible. If the node is not visible, the algorithm terminates. Otherwise, we do either of two things, depending on whether or not the node is a leaf. If the node is a leaf, we rasterize all the polygons classified under that leaf. If not, we recurse on all four children. The following is a pseudo-code version of the algorithm:

```
procedure displaytree( Q)
begin
    if (visible( Q))
        if (isleaf( Q))
            < display associated polygons >
        else
```

```
            displaytree( Q->child[ 0])
            displaytree( Q->child[ 1])
            displaytree( Q->child[ 2])
            displaytree( Q->child[ 3])
        endif
    endif
end
```

A quad-tree node is said to be visible if any of its corners lie within the *visible region*. The visible region is determined by basically projecting the view volume onto the plane P_{min}, defined as the plane originating from $(0, 0, z_{min})$ perpendicular to the vector $(0, 0, 1)$. The term z_{min} refers to the lowest z coordinate in the entire data set (i.e., $min(v_z)$ for all vertices v in the set).

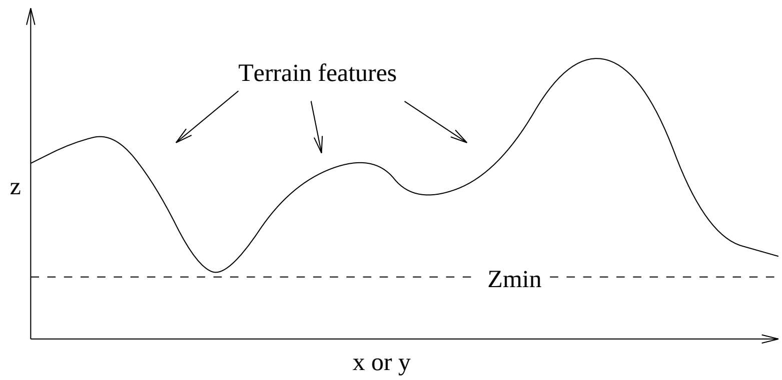

Figure 4: z_{min} as a function of x and y.

3.4 Projecting the View Volume

The common way to specify a view is to have a view-reference-point $v\vec{r}p$, a view-plane-normal $v\vec{p}n$, an up-vector $v\vec{u}p$, the depth-of-projection D and the half-width and half-height of the view plane (u and v, respectively). The $v\vec{r}p$ is the viewer's position in world coordinates. The $v\vec{p}n$ and $v\vec{u}p$ respectively define the viewer's gaze direction and vertical orientation. The depth-of-projection determines the perspective vanishing point, and u and v define the dimensions of the projection plane. (For more on this and other standard graphics terminology, refer to Foley, et al. [10]) From these, we can derive the view volume and its projection onto P_{min}.

We begin by finding the vectors $N\vec{W}$, $N\vec{E}$, $S\vec{E}$ and $S\vec{W}$, which are the four rays originating from vrp and going through the four corners of the view plane. Assuming that $v\vec{r}p$ and $v\vec{u}p$ are of unit length, we can derive the corner vectors as follows:

$$N\vec{W} = \vec{V_d} - \vec{V_u} + \vec{V_v}$$
$$N\vec{E} = \vec{V_d} + \vec{V_u} + \vec{V_v}$$

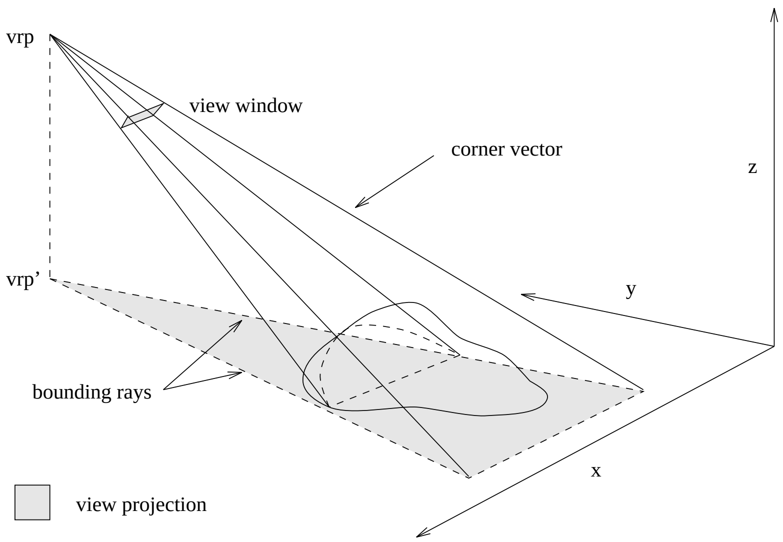

Figure 5: View volume and its projection onto z_{min}.

$$\vec{SE} = \vec{V_d} + \vec{V_u} - \vec{V_v}$$
$$\vec{SW} = \vec{V_d} - \vec{V_u} - \vec{V_v}$$

Where $\vec{V_u} = (v\vec{p}n \times v\vec{u}p) \times u$, $\vec{V_v} = v\vec{u}p \times v$ and $\vec{V_d} = v\vec{p}n \times D$.

By ignoring the z component of these vectors, we arrive at their projections onto P_{min}. (Note that P_{min} is parallel to the xy-plane.) Among these four projections, we find the two vectors which have the greatest angle between them. (Label them u and v. These two vectors can be used to define two sides of the bounding area. We can derive a 2-dimensional halfspace using a point (in this case, the projection of $v\vec{r}p$ onto $P_m in$) as a source of origin, and a vector to indicate direction. In our case, the directional vectors are the vectors perpendicular to u and v.

Given a halfspace defined by $\vec{vrp}'$ (the projection of $v\vec{r}p$ onto $P_m in$) and a vector $\vec{v}$, we can determine whether a given point lies on one side of the halfspace or the other. The formula is given by $r = \frac{x - \vec{vrp}'}{||x - \vec{vrp}'||} \cdot \vec{v}$. If r has a value greater than or equal to zero, then x lies on the visible side of the halfspace.

With two halfspaces, we can eliminate a substantial portion of the unseen area. For most cases, however, we might want to use three to further limit the view area. This can be done simply by finding the intersection of $\vec{NW}$, $\vec{NE}$, $\vec{SE}$ and $\vec{SW}$ against P_{min}. The point of intersection farthest away from $\vec{vrp}'$ can be used as the origin of the third halfspace. The directional vector for this halfspace can be specified as $\frac{o - \vec{vrp}'}{||o - \vec{vrp}'||}$, where o is the point of origin. Note that we can only derive a third halfspace if all the corners intersect P_{min}. We can improve this even further by restricting the view projection with regard to the near and far clipping plates. However, the core algorithm remains essentially the same.

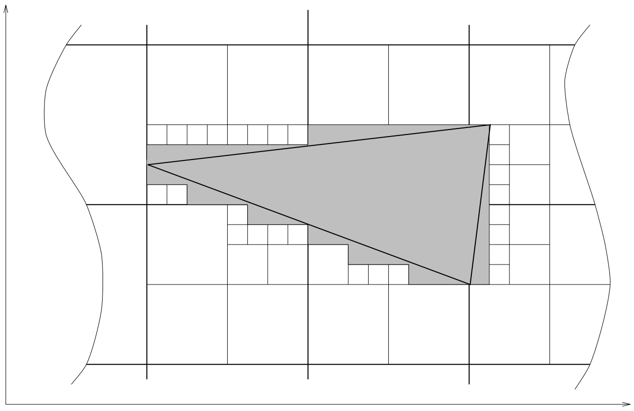

Figure 6: Diagram showing visible nodes and hierarchic arrangement.

3.5 Implementation

A rudimentary terrain renderer was implemented using C and the X11 library. Among its features were Gouraud shading, backface culling, and homogenous clipping. Quad-tree elimination was added to the front of the rendering pipeline to cull away hidden areas prior to vertex transformation. The classification of triangles to quad-trees and mesh *decimation* are both implemented as pre-processing steps.

4 Conclusion

There has been an increase in demand for terrain renderers of late. The primary applications are realistic flight-simulation, terrain visualization, and Geographic Information Systems (GIS). But as terrain rendering is a computationally intensive task, we must seek ways to improve on performance. This paper has provided a survey of methods which can contribute greatly to terrain rendering. The use of view projections and quad-trees for hidden surface elimination merits discussion. To my knowledge, this method has not been proposed in past papers.

5 Acknowledgements

Thanks to my advisor Bob Geitz and the rest of the Oberlin CS staff. Thanks is also due to Paul Coddington and NPAC for expressing interest in this project and providing test data. Finally, thanks to family and friends for their support and helping me with little things like doing laundry.

References

[1] P. Li and D. W. Curkendall. "Parallel Three Dimensional Perspective Rendering." *Parallel Computing: From Theory to Sound Practice*. Proceedings of the EWPC 1992.

[2] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. "Mesh Optimization." *Computer Graphics (SIGGRAPH '93 Proceedings)*. 27(2):19-26, Aug. 1993.

[3] G. Turk. "Re-Tiling Polygonal Surfaces." *Computer Graphics (SIGGRAPH '92 Proceedings)*. 26(2):55-64, Jul. 1992.

[4] W. Schroeder, J. Zarge and W. Lorensen. "Decimation of Triangular Meshes." *Computer Graphics (SIGGRAPH '92 Proceedings)*. 26(2):65-70, Jul. 1992.

[5] C. L. Jackins and S. L. Tanimoto. "Oct-trees and Their Use in Representing Three-Dimensional Objects." *Computer Graphics and Image Processing*. 14:249-70, 1980.

[6] D. Meagher. "Geometric Modelling Using Octree Encoding." *Computer Graphics and Image Processing*. 19:129-47, 1982.

[7] A. Glassner. "Space subdivision for fast ray tracing." *IEEE Computer Graphics and Applications*. 4(10):15-22, Oct. 1984.

[8] N. Greene, M. Kass and G. Miller. "Hierarchichal Z-Buffer Visibility." *Computer Graphics (SIGGRAPH '93 Proceedings)*. 27(2):231-8, Aug. 1993.

[9] K. Yamaguchi, T. L. Kunii and K. Fujimura. "Octree-Related Data Structures and Algorithms." *IEEE Computer Graphics and Applications*. 4(1):53-9, Jan. 1984.

[10] J. Foley, A. van Dam, S. Feiner and J. Hughes. *Computer Graphics Principles and Practice*. 2nd. ed. Addison-Wesley Publishing, 1990.