Voxel Column Culling:
Occlusion Culling for Large Terrain Models

Brian Zaugg Parris K. Egbert

Computer Science Department, Brigham Young University

Abstract. We present Voxel Column Culling, an occlusion culling algorithm for real−time rendering of large terrain models. This technique can greatly reduce the number of polygons that must be rendered every frame, allowing a larger piece of terrain to be rendered at an interactive frame−rate. This is accomplished by using a form of 3D cell−based occlusion culling. A visibility table for the terrain model is precomputed and stored on disk. This table is then used to quickly bypass portions of the model that are not visible due to self−occlusion of the terrain model. This technique improves performance of real−time terrain simulations by reducing the number of polygons to be rendered.

1 Background

Large terrain models are difficult to render in real−time because so many polygons are needed to accurately represent them. Most such models are based on a digital elevation model (DEM), which is simply a rectangular array of elevation values. The most obvious triangulation of a DEM yields $O(nm)$ triangles, where n and m are the grid dimensions. Although the number of polygons can be greatly reduced using terrain decimation and mesh simplification techniques, the distance to the far clipping plane must be fairly short in order to keep the polygon count low enough to maintain interactive frame rates on typical hardware.

This paper presents Voxel Column Culling, an occlusion culling technique for large terrain models. This technique uses a visibility scheme to cull large portions of the database, thus allowing larger pieces of terrain to be rendered in real−time.

2 Related Work

Algorithms which accelerate real−time terrain rendering fall into two categories, terrain decimation and culling. Terrain decimation algorithms produce a terrain model with fewer polygons than is produced by simple triangulation of a DEM. Culling algorithms quickly identify invisible portions of a scene and skip over them, effectively reducing the scene's polygon count.

2.1 Terrain Decimation Algorithms

Terrain decimation algorithms produce a terrain model that has far fewer polygons than a full−resolution triangulation, but which looks very much like the full−resolution

mesh. Several different terrain decimation methods have been used successfully. An overview of these techniques is presented here.

General mesh simplification techniques can be applied to triangulated DEM data. Schroeder *et al.* [15] describe a decimation algorithm that removes a vertex, then re−triangulates the resulting hole. Hierarchical dynamic simplification [12] regenerates the model for every frame, providing a view−dependent simplification. Vertices are clustered together hierarchically, then the hierarchy is queried to supply only the polygons that are important from the current viewpoint. Hoppe [8] introduced progressive meshes, which represent a model as a low−resolution base mesh and a set of vertex split transformations. This method has been extended [9] to provide view−dependent models.

Many terrain decimation algorithms produce Triangulated Irregular Networks, or TINs. These methods minimize the number of triangles created based on some error metric. Examples of TIN methods are given in [5] and [14]. These methods are generally not fast enough to generate a new model each frame, and are not usually view−dependent. They also suffer from popping artifacts if multiple TIN models are used to provide discrete levels of detail.

Semi−regular subdivision methods restrict the triangles in the resulting model to be 45−45−90 triangles. This restriction causes the resulting model to be less optimal, but allows the algorithm to run faster, providing for view−dependent simplification of the model. Examples of terrain decimation algorithms based on semi−regular subdivision include [6] and [11]. Quadtree morphing [4], which also uses semi−regular subdivision, is fast enough to create a view−dependent model in real−time, and morphs between models based solely on the viewpoint.

2.2 Culling Algorithms

Culling algorithms accelerate rendering by quickly removing geometry that will not contribute to the final image. Back−face culling and view frustum culling are well known and widely used techniques [3]. Clustered culling algorithms decide with a single test whether larger sets of polygons should be rendered [10].

Cell−based culling, introduced by Airey *et al.* [1], divides a scene into 2D cells. For each cell, a preprocessing step determines a potentially visible set (PVS) containing all geometry which may be visible from some location within that cell. Luebke and Georges [13] define portals as portions of cell boundaries which don't obstruct a line−of−sight (such as doors and windows), and use these to determine the PVS.

Greene *et al.* [7] introduced an occlusion culling technique known as hierarchical z−buffers. This algorithm uses an object−space octree together with an image−space z−pyramid to determine visible portions of the scene. Zhang *et al.* [16] proposed hierarchical occlusion maps, an image−space culling algorithm. Both hierarchical z−buffers and hierarchical occlusion maps are slow enough that they must be able to cull

a large percentage of the model to achieve speedup, and so work best for models with very high depth complexity.

3 Voxel Column Culling

This section gives an overview of Voxel Column Culling, the culling technique described in this paper. This technique determines terrain visibility in a viewpoint dependent fashion. Significant reduction in rendering time is achieved through the use of this algorithm.

3.1 Partitioning the Space

Voxel Column Culling is a cell−based object−space algorithm. The object−space surrounding the terrain model is partitioned into tiles, columns, and cells. This is illustrated in figure 1.

The terrain model itself is divided into a uniform grid of square *tiles*. The length of a tile side must be a power of two. Varying the tile size provides a mechanism for adjusting a performance trade−off. Selecting a larger tile size means that the offline visibility computations take less time, but it also means that less of the terrain can be culled away at runtime.

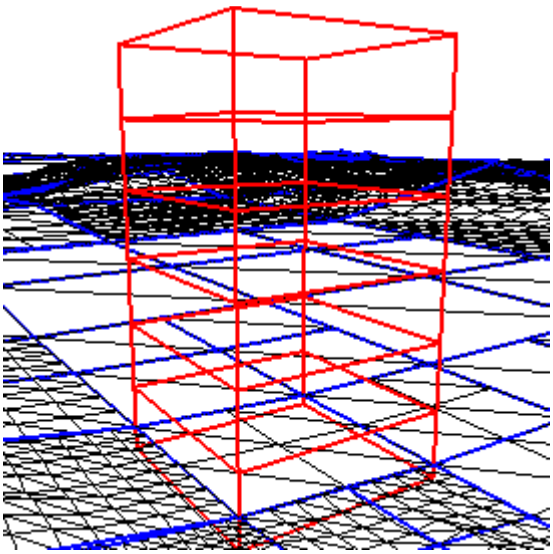


Fig. 1. Partitioned object−space. Terrain tiles are displayed on the grid, and one column of voxels is shown. One such column exists above every tile.

Extending up vertically from each tile is a viewpoint *column*. Visibility from each viewpoint column is computed and stored separately. Each column is sliced into *voxels* at a user−selected number of altitudes. The same trade−off is involved here as was described for the selection of tile size. Visibility is computed from each viewpoint voxel in object−space to every terrain tile in the model. All viewpoints within a given voxel are considered the same viewpoint by the culling algorithm.

3.2 Reducing the Problem Size

General 3D visibility computation for large models is not feasible. Each potential viewpoint must be tested for visibility with every point of geometry in the scene, and there is no good way of knowing what objects in the scene might cause an occlusion. Visibility for terrain models can be computed much faster than visibility can be computed for arbitrary models for two reasons. First, heightgrid−based terrain models exist at only one altitude z for each (x, y) point. This makes it much easier to determine whether the model is self−occluding from a given viewpoint. Second, a significant amount of computation is only required for one viewpoint voxel in each column. For each viewpoint column and terrain tile, there is a special voxel which we will call the *horizon* voxel from which the tile is invisible. The horizon voxel is the voxel in that column below which the tile is always invisible and above which that tile is always visible. Computation of visibility for voxels above the horizon voxel is very fast, usually requiring only one point−point test (See section 4). Visibility determination for each column is done in top−to−bottom order. Since this calculation is very fast for the voxels above the horizon, and the tile is known to be occluded from all voxels below the horizon, only one voxel−tile visibility calculation takes a significant amount of time for each column−tile pair.

4 Computing Visibility

Determining visibility from any viewpoint around the terrain model to any piece of terrain is slow enough that it must be precomputed. Visibility is tested from each viewpoint voxel to every terrain tile in the model. As explained in section 3.2, many of these tests are very fast. Each voxel−tile visibility test is accomplished by testing some of the points in the voxel against some of the points in the tile.

4.1 Voxel−Tile Visibility Determination

Voxel−tile visibility tries to find an unobstructed line−of−sight (LOS) from some point in the voxel to some point on the tile. If one such unobstructed LOS can be found, the entire tile is considered to be visible from anywhere in the voxel. If no unobstructed LOS can be found, the tile is not visible from the voxel. This is shown in figure 2.

If a viewpoint from which a tile is visible is moved down towards the terrain, at some point that tile will become occluded by another piece of terrain. However, if a viewpoint from which a tile is not visible is moved down towards the terrain, the tile will never become visible. Thus, we need test only the four edges of the top face of

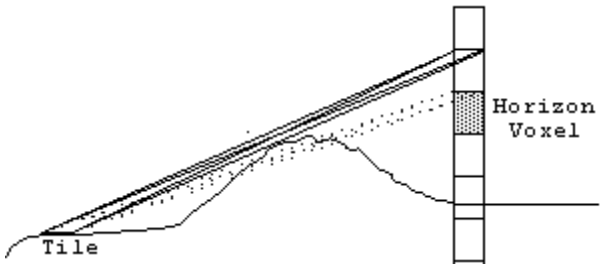


Fig. 2. The horizon voxel is the highest voxel in the current column from which the current tile is occluded.

the voxel. If the tile is not visible from those edges, it will not be visible from anywhere in the voxel.

In order to find an unobstructed LOS as quickly as possible, point−point visibility tests are done repeatedly on successively finer sub−samplings of the edges of the top of the voxel and the terrain. On the first iteration, every 32nd point along the top voxel edges will be tested against every 32nd point in the tile. If no unobstructed LOS is found, this process is repeated for every 16th point in the edges and tile. This is repeated until an unobstructed LOS is found or every edge point has been tested against every point in the tile.

Voxel−tile visibility tests are done in top−to−bottom order for each column and tile. For most voxels above the horizon, this test will be very fast because the first point−point test will find an unobstructed LOS. After the horizon voxel is found, the tile is marked as being occluded for viewpoints in what remains of the column below the horizon. This means that only the horizon voxel and sometimes the voxel above it take much time to compute.

4.2 Point−Point Visibility Determination

The point−point visibility test is fairly simple. The heightgrid data that was used to create the terrain model is treated as a 2D image, across which a line is drawn from one point to the other. The altitude of the start and end points is known, and altitudes are interpolated along the line at each point. We use a modified Bresenham line algorithm [2] to do the interpolation using integer arithmetic. This provides for a very fast 3D line. At each point along the line, the current elevation of the line is compared with the elevation of the terrain at that point. If the line ever drops below the terrain elevation, the test fails; otherwise the points are mutually visible.

4.3 Storage Format

The visibility information computed during this preprocessing stage is stored for each column. Only one bit is stored per voxel−tile pair, so the total space required to store the visibility information is minimal.

5 Real−Time Culling

Culling at runtime is simple and fast. As each tile is considered for rendering, a table lookup is done to determine whether that tile is visible from the current viewpoint. If so, it is added to a set of visible tiles which are then rendered. For subsequent frames, a quick check is done to determine whether the new viewpoint is still within the same voxel. If it is, the same set of visible tiles is used to render the new frame. Otherwise, a new set must be created by performing the table lookup for each tile.

Because of the speed of this method, the simulation is not noticeably slowed even if all of the terrain is visible and there is no reduction in polygon count.

6 Results

We tested the Voxel Column Culling algorithm using terrain data covering the Wasatch Front in northern Utah, including most of the sites that will be used for the 2002 Winter Olympics. For view−dependent terrain decimation, we used the Quadtree Morphing algorithm developed by Cline and Egbert [4]. The Wasatch Front DEM used contains 2070x2637 samples and covers about 5500 square kilometers. A tile size of 32x32 samples (about one square kilometer) was used, and the columns were divided vertically into 7 voxels at 500, 1000, 1500, 2000, 2500, and 3000 meters above the lowest point in the DEM. All data was collected with the far clipping plane beyond the far end of the terrain model. Figure 5 shows the large portion of the model that is culled. Figure 6 shows a frame rendered in three modes, textured, wireframe without Voxel Column Culling, and wireframe with culling enabled. Note the distant terrain that is rendered when culling is disabled, but disappears when culling is enabled.

6.1 Polygon Count

The results discussed in this section are from a flyby of the Wasatch Front dataset including both high− and low−altitude flight over both flat and mountainous terrain. As can be seen in figure 3, the number of polygons sent to the rendering pipeline was significantly decreased when the Voxel Column Culling algorithm was used. As was expected, the biggest improvement was observed for low−altitude flight over mountainous terrain. These portions of the flyby can be identified in the graph as the areas where there is a large difference between the two polygon counts. However, as can be seen in the graph, high−altitude and flat−terrain portions of the flyby have very low polygon counts to start with.

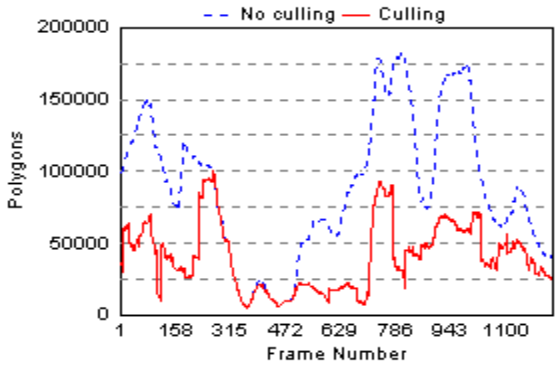


Fig. 3. Polygon count for flyby of Wasatch Front dataset at 1600x1200 resolution with and without occlusion culling enabled.

6.2 Framerate

Although the framerate is mostly a function of polygon count, we wanted to verify that the culling algorithm wasn't slowing the framerate significantly in portions of the flyby where little or no culling was possible. Figure 4 shows the framerate attained on an SGI 320 Visual Workstation with a Pentium II 350 MHz processor. Also note that the framerate isn't noticeably decreased by the culling algorithm even in high−altitude, flat−terrain portions of the flyby.

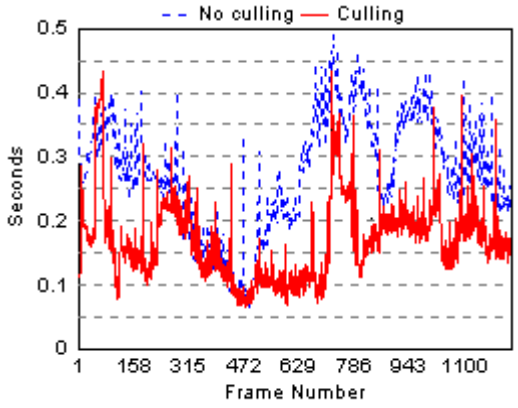


Fig. 4. Rendering time required per frame for Wasatch Front dataset flyby.

7 Future Work and Conclusion

In this paper we have presented an occlusion culling algorithm which achieves significant speedup of interactive terrain simulation systems. The space surrounding the terrain is partitioned into voxels, and visibility from each of these voxels to each tile of terrain is precomputed and stored on disk. This visibility information is then used during simulation to cull occluded portions of the terrain model.

This research can be extended in several directions. These include consideration of buildings and other objects on the terrain and improving pre−processing speed using coherence. Buildings and other objects on the terrain are currently ignored by the algorithm. This poses a problem if objects on the terrain are culled along with the terrain when the tops of those objects should actually be visible. One possible solution to this problem is to add the heights of the objects to the values of the heightgrid at the appropriate locations.

Computation of visibility from voxels to tiles is currently performed without taking previous visibility results into account. The preprocessing step which computes visibility could probably be sped up by exploiting coherence between neighboring voxels and tiles.

Acknowledgments

Thanks to David Cline and Kirk Duffin for building the terrain−rendering infrastructure which was used to test this algorithm. This work was funded in part by a grant from the Utah State Center of Excellence.

References

1. Airey, John M., John H. Rohlf, and Frederick P. Brooks Jr.: Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics*), vol. 24, no. 2, pp. 41−50, March 1990.

2. Bresenham, J. E.: Run−length slice algorithm for incremental lines. *Fundamental Algorithms for Computer Graphics*, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, pp. 59−105, 1985.

3. Clark, James H.: Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, vol. 19, no. 10, pp. 547−554, October 1976.

4. Cline, David and Parris K. Egbert.: Terrain Decimation through Quadtree Morphing. To be published in *IEEE Transcations on Visualization and Computer Graphics.*

5. DeFloriani, L. and E. Puppo.: Hierarchical Triangulation for Multiresolution Surface Descriptions. *ACM Transactions on Graphics*, vol. 14, October, 1995.

6. Falby, John S. *et al.* NPSNET: Hierarchical Data Structures for Real−Time Three−Dimensional Visual Simulation. *Computers and Graphics*. vol. 17, pp. 65−69, 1993.

7. Greene, N., M. Kass, and G. Miller.: Hierarchical Z−Buffer Visibility. *Computer Graphics (SIGGRAPH '93 Proceedings)*, vol. 27, pp. 231−238, 1993.

8. Hoppe, Hugues.: Progressive Meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)*, vol. 30, pp. 99−108, 1996.

9. Hoppe, Hugues.: View−Dependent Refinement of Progressive Meshes. *Computer Graphics (SIGGRAPH '97 Proceedings)*, vol. 31, pp. 189−198, 1997.

10. Kumar, S., D. Manocha, B. Garrett, and M. Lin.: Hierarchical Back−Face Computation. *Proceedings of Eurographics Rendering Workshop 1996*, pp. 235−244, June 1996.

11. Lindstrom, Peter *et al*:. Real−Time Continuous Level of Detail Rendering of Height Fields. *Computer Graphics (SIGGRAPH '96 Conference Proceedings)*, vol. 30, pp. 109−118, 1996.

12. Luebke, David, and Carl Erikson.: View−Dependent Simplification of Arbitrary Polygonal Environments. *Computer Graphics (SIGGRAPH '97 Proceedings)*, vol. 31, pp. 199−208, 1997.

13. Luebke, David P. and Chris Georges.: Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. *Proceedings 1995 Symposium on Interactive 3D Graphics*, pp. 105−106, April 1995.

14. Schroder, F. and P. RossBach.: Managing the Complexity of Digital Terrain Models. *Computers and Graphics*, vol. 18, pp. 65−70, 1994.

15. Schroeder, William J., Jonathan A. Zarge, and William E. Lorensen.: Decimation of Triangle Meshes. *Computer Graphics (SIGGRAPH '92 Proceedings)*, vol. 26, pp. 65−70, July 1992.

16. Zhang, Hansong, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III.: Visibility Culling using Hierarchical Occlusion Maps. *Computer Graphics (SIGGRAPH '97 Proceedings)*, vol. 31, pp. 77−88, 1997.

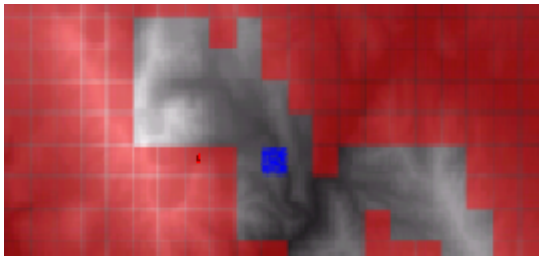
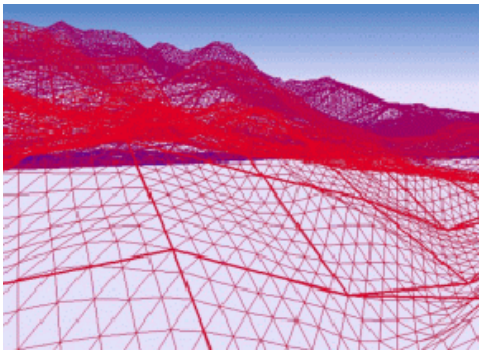
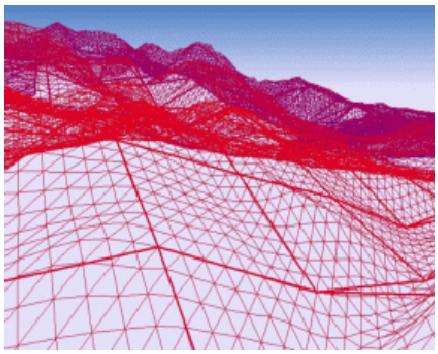


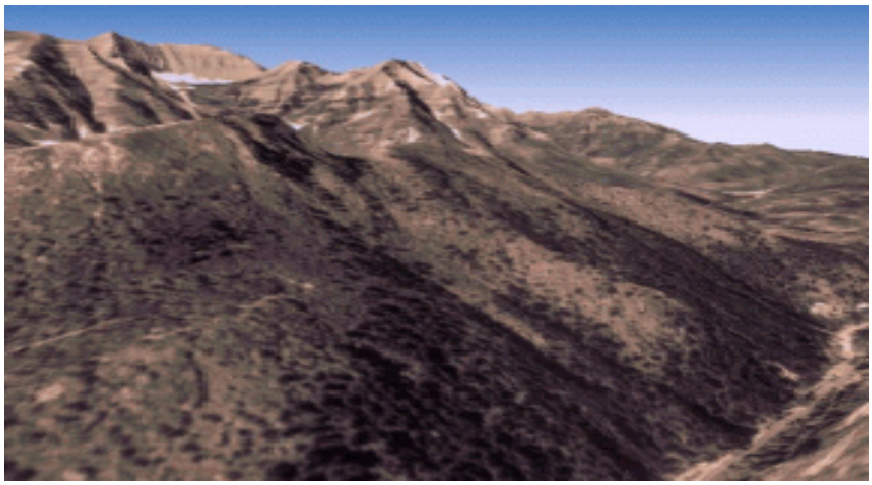
Fig. 5. Visualization of visibility. The viewpoint voxel is colored blue, and occluded tiles are shown in red.



a)



b)



c)

Fig. 6. A frame rendered from the viewpoint shown in blue in figure 6. a) wireframe with culling disabled, b) wireframe with culling enabled, c) textured view.