

# FRUSTUM AND OCCLUSION CULLING IN A QUADTREE-BASED TERRAIN RENDERER

Nathaniel Reed  
Extended Essay — Computer Science  
February 2, 2004

## ABSTRACT

In computer graphics, two important methods of hidden surface removal are frustum culling, which removes objects outside the camera's field of view, and occlusion culling, which removes objects that lie behind other objects. This essay presents research into the effectiveness of frustum and occlusion culling algorithms in increasing the framerate of a simple terrain rendering engine.

In the program constructed for this research, terrain is modeled by a height field and organized into a quadtree by recursive subdivision. Axis-aligned bounding boxes are computed for each node and compared against viewing and occlusion frustums to determine visibility. The terrain does not occlude itself directly, but hidden convex polyhedron "occluders" are placed under large terrain features, such as mountains, and occlusion frustums are formed from these.

Data is collected with no culling, frustum culling only, occlusion culling only, and with both by averaging the framerate over half-second intervals as the camera traverses a predesignated path over the terrain that provides a variety of different vantage points and types of views. Then this data is statistically analyzed to determine the effectiveness of the two culling algorithms.

Frustum culling resulted in a 44% average increase in framerate, while occlusion culling resulted in an average 5% increase. Thus, it was concluded that while both methods produce statistically significant framerate gains, frustum culling is more effective than occlusion culling for this type of graphics system.

## ACKNOWLEDGEMENTS

I would like to thank the following people for providing me with assistance and support during the undertaking of this project: my supervisor, Julie Nygaard; my school's IB coordinator, Heidi Lohnes; my parents, Jim Reed and Sally Garrigues; my uncle, Robert Reed, for getting me interested in computer graphics in the first place; and all my friends and classmates, for providing shoulders to lean on when the going got hard.

# TABLE OF CONTENTS

|                                     |    |
|-------------------------------------|----|
| ABSTRACT.....                       | 2  |
| ACKNOWLEDGEMENTS.....               | 2  |
| TABLE OF CONTENTS .....             | 3  |
| INTRODUCTION.....                   | 4  |
| Factors Affecting Framerate .....   | 4  |
| Frustum and Occlusion Culling ..... | 5  |
| APPLICATION .....                   | 6  |
| Subdivision and the Quadtree .....  | 6  |
| Implementation .....                | 7  |
| Hypothesis.....                     | 9  |
| Procedure .....                     | 10 |
| Analysis of Results .....           | 10 |
| Conclusion .....                    | 12 |
| BIBLIOGRAPHY.....                   | 13 |

# INTRODUCTION

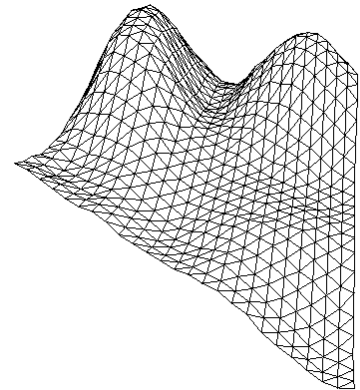
In the modern world, computer graphics is a ubiquitous technology. It encompasses a range of techniques, from the simple display of an application's user-interface to the creation of "photorealistic" images that look like the real, three-dimensional world surrounding us. Motion pictures, video games, scientific simulations, and industrial design and manufacturing processes are all examples of applications of computer graphics.<sup>2</sup>

Herein, we are concerned with interactive graphics. Interactive graphics do not follow a pre-planned sequence of events, as do the graphics in a film; instead, the graphics system must respond dynamically to user actions. Thus, while a single frame from a film may take hours to render, a single frame from a video game (for example) is allotted no more than a split second. An interactive graphics application, then, must produce a reasonably photographic image of a complex virtual scene, and it must do so in 1/30 of a second or less. Even for modern computers that can perform many billions of calculations per second, this is not an easy task.

This fact explains why speed, measured by *framerate* (the number of video frames displayed per second), is of paramount importance in interactive graphics. An interactive graphics programmer must be constantly aware of how much time the code he or she writes will take to execute. He or she must ask, Is there a faster way to perform this task? Could data be organized differently, making retrieval more efficient? Can the results of a calculation be stored and reused later? How can unnecessary computations be eliminated? This last question is the subject of the research presented herein.

## Factors Affecting Framerate

Modern interactive graphics systems almost all operate from the same paradigm. The virtual scene that is to be visualized is described in terms of a three-dimensional space with a Cartesian coordinate system. Objects in this scene are modeled by using triangles. Triangles alone are used because they are one of the simplest geometric entities to *rasterize* (display as a set of pixels on a computer screen).<sup>2</sup> Although objects may be modeled at a high level as quadric surfaces, bicubic patches, or other complex primitives, they are always decomposed into an approximating triangle mesh before being rendered. Figure 1 shows an example of a triangle mesh that approximates a smoothly curving surface.



**Figure 1: A triangle mesh**

The virtual scene also includes a *camera*, which defines the viewpoint from which the scene is to be rendered. The camera is localized at a specific point in space and is pointed in a specific direction.

The process of rendering an image, then, consists of two major steps. First, the triangles (or rather, their vertices) must be taken through a series of transformations that brings them from the three-dimensional *world space* in which they are defined to the two-dimensional *screen space*. The second step is to rasterize the triangles, which can include calculation of lighting and

texturing. In thinking about the factors affecting the speed of a graphics system, it becomes apparent that either of these two steps can limit the framerate.<sup>2</sup>

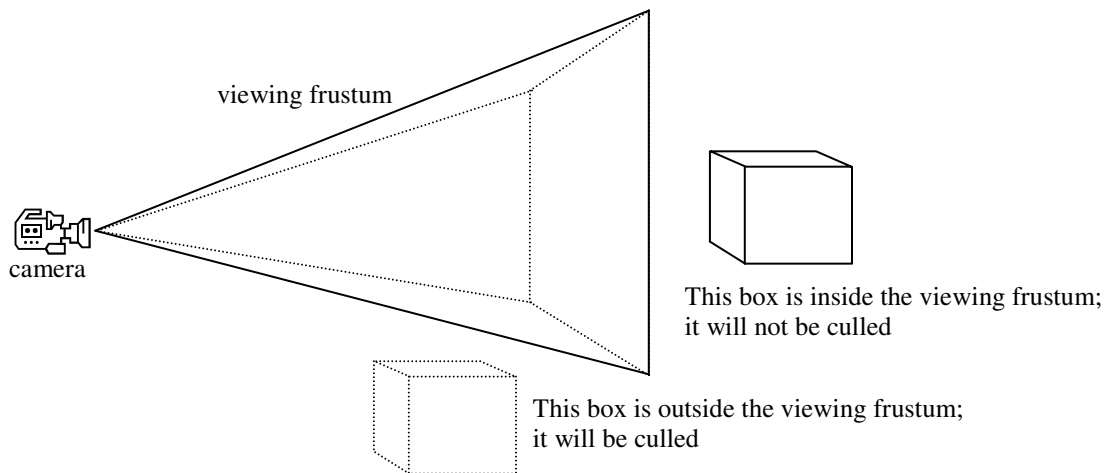
If the speed of a graphics system is limited by its *vertex throughput* (the number of vertices that can be transformed or “put through” per second), then it is called *vertex-limited*. This means that the vertex processing and transformation step is the slowest in the graphics pipeline. On the other hand, the system is *fillrate-limited* if the rasterization step is the slowest. Then, the determining factor for the framerate is the number of pixels that can be rasterized per second.<sup>2</sup> Herein, we are concerned with vertex-limited systems only.

The most obvious way to speed up a system limited by vertex throughput is to decrease the number of vertices. One way to accomplish this is to eliminate those vertices that one can determine, *a priori*, will not be needed in the final image. For example, a triangle may fall outside the camera's field of view, or it may lie behind some object that occludes it; then the triangle cannot be seen, and its vertices need not be transformed.

The problem of identifying objects that cannot be seen and removing them from the rendering pipeline is called *hidden surface removal* (HSR). In this essay we investigate two methods of HSR that are commonly used in contemporary interactive graphics. These are called *frustum culling* and *occlusion culling*.

## Frustum and Occlusion Culling

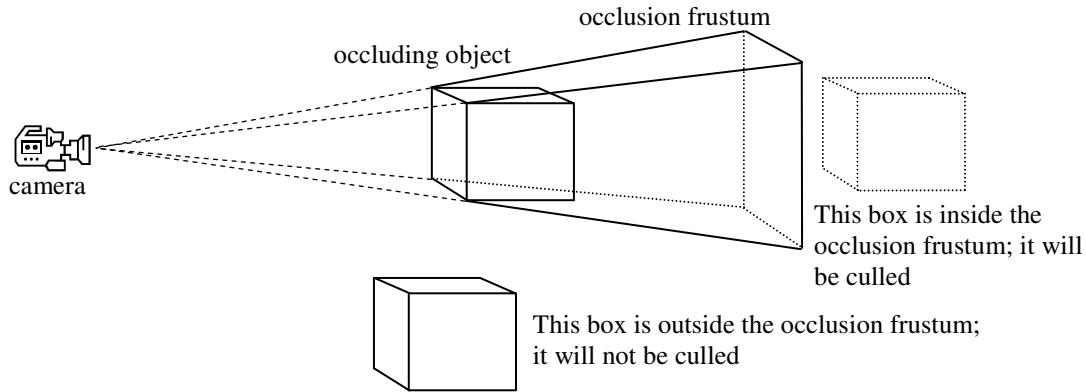
Frustum culling is so named because it identifies objects that lie outside of the camera's *viewing frustum*. A frustum is an unbounded pyramidal volume formed by the intersection of several planes sharing a common point. The viewing frustum consists of planes formed from the camera location and the four edges of the user's screen. Thus, the viewing frustum represents exactly the volume of space visible to the camera. An object outside this frustum is either behind the camera or so far to one side that it is off-screen, as shown in Figure 2.



**Figure 2: Frustum culling**

The aim of occlusion culling is to identify objects that are invisible because they are behind, or occluded by, another object. It operates by forming the *occlusion frustum* of an object, which

consists of planes drawn from the camera location through the silhouette edges of an object, as shown in Figure 3. An object's occlusion frustum represents exactly the volume of space that falls behind that object from the camera's point of view. Objects completely inside the occlusion frustum cannot be therefore be seen, and may be culled (removed from the graphics pipeline).<sup>1</sup>



**Figure 3: Occlusion culling**

Depending on the type of graphics system and the type of objects being displayed, frustum culling or occlusion culling or the combination of the two may be able to eliminate many invisible objects each frame. This can reduce the number of vertices that must be transformed. In a vertex-limited system, frustum and occlusion culling have the potential to significantly improve the framerate.

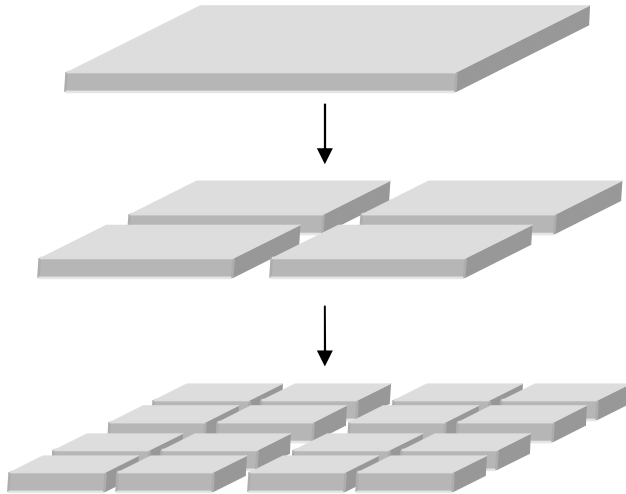
## APPLICATION

Interactive graphics systems are often called upon to simulate the natural world, and landscapes and terrain are an important feature of the natural world. Thus, the efficient rendering of terrain is a problem of some importance in contemporary interactive graphics. In the research that follows, the effects of frustum and occlusion culling on a simple terrain renderer are investigated.

Terrain, like everything else, is modeled by using a set of triangles. Because Earthlike terrain is flat and does not ordinarily overlap itself, we can represent a patch of it by a square or rectangular mesh of triangles arranged on a regular grid in the  $xy$  plane, but having varying heights in the  $z$  axis.

### Subdivision and the Quadtree

In order to efficiently apply frustum culling and occlusion culling to a terrain patch, the rectangular patch is recursively subdivided into smaller and smaller rectangles. The amount of subdivision is controlled to achieve a balance wherein the granularity is small enough for the culling algorithms to be effective, but not so small that the number of rectangles becomes unmanageably large. The process of recursive subdivision is illustrated in Figure 4.



**Figure 4: Recursive subdivision of a terrain patch**

To organize the pieces of the terrain patch, a *quadtree* is used. This is simply a tree in which each node has four children. The root node of the quadtree contains the entire terrain patch. Successively lower levels of the quadtree contain smaller portions of the terrain patch. The leaves of the tree then contain the individual rectangles produced by the last level of subdivision.

Organizing terrain into a quadtree is useful because it allows efficient application of frustum and occlusion culling. To apply frustum culling to a quadtree, for example, one first tests the root node against the viewing frustum. If the root node is

completely within the frustum it can be accepted and drawn immediately; if it is completely outside the frustum, it is rejected. If the node is partially inside and partially outside the frustum, as is evidently the more common case, it is subdivided into its four children and the process is repeated with each. The following pseudocode illustrates this algorithm:

```

add root_node to queue;
while (queue is not empty) {
    pop node off queue;
    if (node is completely inside frustum)
        accept node;
    else if (node is completely outside frustum)
        reject node;
    else // Node must be partially inside, partially outside
        for each child of node
            add child to queue;
}

```

Occlusion culling operates in a similar manner, but nodes are rejected if they are inside the frustum and accepted if they are outside it. Herein, we do not use the terrain blocks themselves to create the occlusion frustum, but instead define special objects, called *occluders*, for this purpose. Occluders are discussed in more detail later.

## Implementation

In order to study the effect of frustum culling and occlusion culling on framerate, we construct a program that will put these algorithms to practical use. This program was written from scratch (that is, with all original source code) in C++ for a Microsoft® Windows® platform, and makes use of the OpenGL library for its graphics interface. It implements the quadtree, frustum culling, and occlusion culling algorithms discussed above, and allows for the culling algorithms to be toggled on and off. It also includes features for the controlled collection of framerate data.

The program uses a heightfield-based terrain patch 128×128 grid squares in size, for a total of 32,768 triangles and 16,641 vertices. The number 128 is chosen because it is a power of 2, and

therefore very easy for the quadtree algorithm to recursively subdivide. It constructs a quadtree five levels deep (including the root), so the deepest level contains 256 nodes arranged in a 16×16 grid. Each leaf node will then encompass an 8×8 subset of the terrain, with 128 triangles and 81 vertices. No lighting and only simple texture mapping is used, to ensure that the program does not become fillrate-limited.

For the purposes of testing against a viewing or occlusion frustum, the quadtree nodes are represented by axis-aligned bounding boxes (AABBs). During construction of the quadtree, an AABB is constructed for each leaf node by looping through its vertices and recording the minimum and maximum extents in the  $x$ ,  $y$ , and  $z$  directions. The AABBs of the parent nodes are then constructed in an analogous manner, resulting in the smallest AABB that encloses all of the child AABBs.<sup>3</sup>

To determine the intersection of a frustum (composed of a set of planes with outward-facing normals) and an AABB, the following algorithm, inspired by the Cohen-Sutherland method of line clipping, is used:<sup>4</sup>

```
a = 0;
for each plane in frustum {
    b = 0;
    for each vertex in aabb {
        if vertex is outside plane
            ++b;
        else
            ++a;
    }
    if (b == number of vertices in aabb)
        return completely_outside;
}
if (a == number of vertices in aabb * number of planes in frustum)
    return completely_inside;
else
    return partially_inside;
```

This is not an exact test. It is exact if the AABB is completely or partially inside the frustum; however, situations occur in which the AABB is completely outside the frustum, yet is classified as partially inside.<sup>4</sup> This presents only a minor issue and does not affect the functioning of the frustum and occlusion culling algorithms.

Visibility determination is implemented in the following way. Each node in the quadtree possesses a visibility flag, which can be set to invisible, completely visible, or partially visible. The invisible and completely visible states are inherited by descendants of the node, while the partially-visible state indicates that some child nodes are visible and others are not.

To determine the visibility state each frame, the subdividing frustum culling algorithm described earlier is first applied to the quadtree. This algorithm stores its results by setting the visibility flags in the quadtree structure. Then, occlusion culling is applied. The nodes already marked as invisible are ignored, but visible (completely or partially) nodes are tested against the occlusion frustum and, if they fall inside it, are marked invisible:

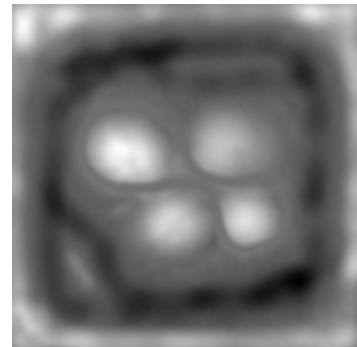
```

for each occluder {
    create occlusion frustum for occluder;
    add root_node to queue;
    while (queue is not empty) {
        pop node off queue;
        if (node is partially visible) {
            if (node is completely inside frustum)
                mark node invisible;
            else
                add children of node to queue;
        }
        else if (node is completely visible) {
            if (node is completely inside frustum)
                mark node invisible;
            else if (node is partially inside frustum) {
                mark node partially visible;
                mark children of node completely visible;
                add children of node to queue;
            }
        }
    }
}

```

Finally, the quadtree is traversed again to build a list of all visible nodes, which are then rendered using an OpenGL vertex array. Each quadtree node stores a pointer to a list of indices into the master vertex array, which can be easily passed to the OpenGL `glDrawElements()` function.

A grayscale height-map of the terrain used by the program appears in Figure 5. Dark areas are low, light areas are high. This terrain, as well as the texture map applied to it, was generated by the Terragen shareware terrain generation software. The landscape consists of a plateau or island with four mountains, surrounded by a moat. The mountains were chosen because they are effective occluders. When the camera gets close to a mountain, a large amount of the terrain may be occluded by it.



**Figure 5: Terrain height-map**

Each of the four mountains hosts an *occluder*. This is a convex polyhedron, not ordinarily displayed, that lies inside the mountain and is constructed to include as much of the mountain's volume as possible. When occlusion culling is active, the program forms occlusion frustums from these polyhedra and tests the terrain against them using the recursive algorithm described earlier.

A side effect of using occluders in this way is that only nodes occluded by the mountains can be culled. If a node is obscured by some other terrain feature, the occlusion culling algorithm will not be able to determine this. However, it is unlikely that any other feature of this terrain will cause significant occlusion. Only the mountains are significant visibility blockers.

## Hypothesis

It is hypothesized that frustum culling will have a greater impact on the framerate of the system than occlusion culling. This is because frustum culling will typically be able to cull many more

triangles each frame than occlusion culling.<sup>1</sup> A large portion of the landscape may be outside the camera's field of view in a typical shot, but unless the camera comes close to the mountains, the landscape will not significantly occlude itself. For the purposes of statistical analysis, the null hypothesis will be that neither frustum culling nor occlusion culling will have a significant effect upon the framerate of the system.

The experimental design therefore consists of the following. The program will be tested in each of the following situations:

- (a) neither frustum nor occlusion culling (as a control);
- (b) frustum culling only;
- (c) occlusion culling only; and
- (d) both frustum and occlusion culling.

In each case, framerate data will be collected as the camera traverses a path that takes it around the landscape. Then the framerate data may be statistically analyzed to estimate the effect of the algorithms.

## Procedure

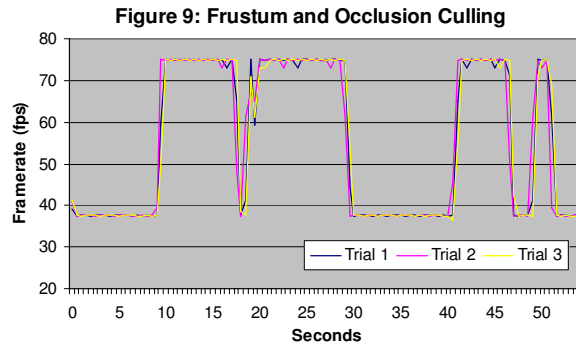
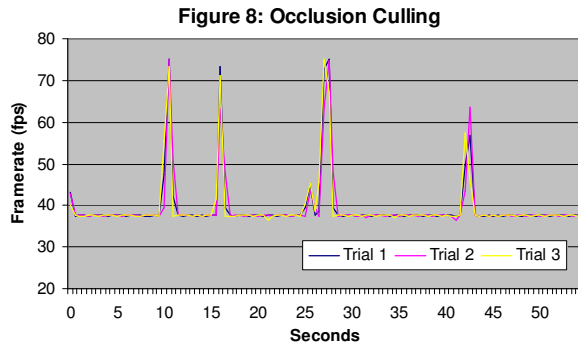
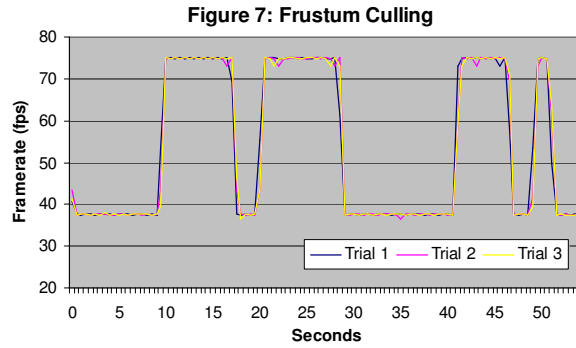
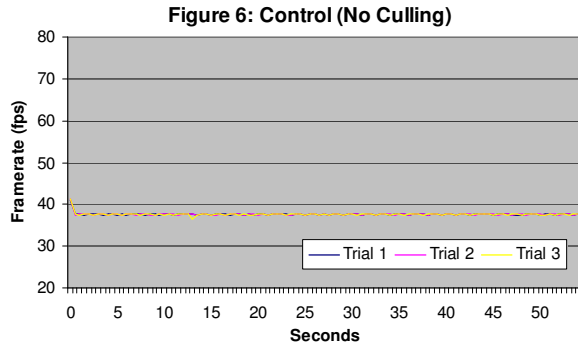
In order to control variables as much as possible, the program will be executed starting from a clean boot with no other applications running. However, because the program is executing in a multitasking environment, random fluctuations in framerate are inevitable due to variations in CPU time allotted to the program. To further control variables, three trials will be conducted for each of the four experimental groups listed above, and averaged.

During each trial, the camera is controlled by the program and follows a preset path through the landscape. This path is approximately one minute long. Some sections of the path move up into the air and allow large portions of the terrain to come into view, while others stay close to the ground and move in and around the mountains. Thus, the path encompasses a wide variety of views of the landscape, allowing the frustum and occlusion culling algorithms to be tested in a variety of situations.

The program automatically collects framerate data during the trial. The framerate is calculated, averaged over half-second intervals, and logged to a text file. Once all trials are complete, the data can be imported into a spreadsheet application such as Microsoft® Excel® for analysis.

## Analysis of Results

The following graphs show the framerate, plotted against time during the one-minute preset path, for each of the four experimental groups.



In Figure 6, neither of the culling algorithms is active. The framerate holds steady throughout the camera path at about 37.5 frames per second. This is expected, since in this group the number of vertices to be transformed does not change from frame to frame. Thus, if the system is vertex-limited, we would expect a constant framerate. The lack of variation in framerate in Figure 6 is evidence that the program is indeed vertex-limited.

Figure 7 shows what happens to the framerate when frustum culling is applied. The framerate initially remains at approximately 37 fps, just as in the control group. However, at certain time intervals the framerate jumps up dramatically, to about 75 fps. The intervals in which the framerate increases correspond to those portions of the camera path in which the camera is pointed away from the center of the terrain patch. Within these intervals, much more of the terrain is outside the viewing frustum than inside it. Thus, the number of vertices transformed decreases dramatically, causing the observed jump in framerate. In other intervals of the camera path, the framerate remains at the control baseline of 37 fps. These intervals correspond to those portions of the camera path in which most or all of the terrain patch is visible, and few nodes can be culled.

In Figure 8 the effects of occlusion culling on the framerate can be seen. Like the frustum culling group, there is a framerate baseline of 37 fps, and sporadic jumps up to the 75 fps plateau can be seen. However, these jumps are less frequent and of shorter duration than in the frustum culling group. They correspond to points on the camera path where the four mountains, which are the only occluders, block visibility to a significant portion of the landscape. It is clear that occlusion culling can improve the framerate, but not as much as frustum culling in this system.

Figure 9 shows frustum and occlusion culling combined. It displays the same behavior as the other groups.

The standard deviation between each corresponding data points in the three trials, averaged across all groups, is 0.62 fps. This value is a gauge of the degree of random error in the framerate measurements, and can be used as an indicator of statistical significance. If activating a culling algorithm causes a change in the framerate greater than 0.62 fps, it would be considered statistically significant; if not, the change cannot be distinguished from random error.

The average framerate over the whole time interval is 37.5 fps for the control group and 53.9 fps for the frustum culling group. Thus, frustum culling produces a framerate increase of 16.4 fps (44%), which is far more than 0.62 fps. Thus, frustum culling produces a quite significant increase in framerate.

For the occlusion culling group, the average framerate is 39.4 fps, which differs from the control baseline by 1.9 fps (5%). While this is a much less dramatic result than that of the frustum culling group, it is still statistically significant.

When frustum culling and occlusion culling are combined, the average framerate is 55.5 fps, an increase of 18.0 fps (48%) over the baseline and 1.6 fps (3%) over frustum culling alone. Thus, occlusion culling still produces a significant framerate increase when combined with frustum culling.

## Conclusion

The original hypothesis in conducting this experiment was that both frustum culling and occlusion culling would produce significant increases in the framerate of the graphics system. Analysis of the data bore this out. Frustum culling produced a 44% increase in the framerate on average, and occlusion culling produced a 5% increase on average. It was also hypothesized that frustum culling would have a greater effect on the framerate than occlusion culling. This hypothesis was also supported by the data. Frustum culling, at least, is certainly worth implementing in a terrain renderer. The effectiveness of occlusion culling for this type of graphics system is less clear. Occlusion culling is significantly more difficult to implement than frustum culling, and for a 5% increase in frame rate, it may not be worth the effort.

One curious feature of the data is that the framerate appears to oscillate between two fixed values, 37 fps and 75 fps. The system remains at either the low or high value for a certain time and then jumps abruptly to the other, rather than varying smoothly. The fact that the framerate has a plateau at 75 fps may indicate that when the system reaches that speed, some other part of the graphics pipeline becomes the speed-limiting step, so that the system ceases to be vertex-limited. Indeed, the oscillatory behavior may indicate that some other speed-limiting factor, which has not been accounted for, is affecting the system. This behavior remains unexplained, but one possible clue to an explanation may be the mathematical relationship that 75 is almost exactly double 37.

A possible limitation of this experiment is the number of vertices in the terrain patch. Only about 17,000 vertices were used, a rather small number for the status of modern graphics hardware. This size was chosen for reasons of technical convenience: on this researcher's graphics hardware, a single vertex buffer can hold a maximum of about 65,000 vertices, and the next higher possible terrain size (256×256) exceeds this vertex limit. However, to ensure that the

system is completely vertex-limited at all times, it may be necessary to increase the number of vertices. This would necessitate multiple vertex buffers, which would increase the complexity of the program.

In a “real” graphics system, a landscape is rarely encountered by itself. Ordinarily, there are a variety of objects placed upon it that are separate from the heightfield. For example, in a video game, a piece of terrain could have characters and vehicles moving around on it and structures built on it. None of these aspects were investigated in this experiment. Theoretically, the frustum culling and occlusion culling methods should be as applicable to objects on the landscape as they are to the landscape itself, but it is possible that the presence of objects and structures would affect the results in unforeseen ways. This is one possible opportunity for further research in this area.

Other opportunities for further research lie in more advanced methods for reducing the number of vertices. Hidden surface removal is addressed by frustum culling and occlusion culling, but another way to improve vertex efficiency is to “smooth out” or decrease the detail level of surfaces that are far away (since this detail cannot be seen from great distances, it makes sense to eliminate the extra triangles). A method of accomplishing this that works particularly well with quadtrees is called *geomipmapping*. Other algorithms include ROAM (Real-time Optimally Adapting Mesh), which combines and splits triangles dynamically as the camera moves,<sup>5</sup> and CLOD (Continuous Level of Detail), which is typically used to adaptively triangulate landscapes modeled by bicubic patches. However, algorithms like these are far outside the scope of this paper.

## BIBLIOGRAPHY

1. Bacik, Michal. “Fast Occlusion Culling for Outdoor Environments.” *Game Developer Magazine*, April 2002. (Also available online at: [http://www.gamasutra.com/features/20020717/bacik\\_01.htm](http://www.gamasutra.com/features/20020717/bacik_01.htm); last accessed 2 Feb 2004.)
2. Foley, van Dam, et al. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company: New York, 1996.
3. Lander, Jeff. “When Two Hearts Collide: Axis-Aligned Bounding Boxes.” *Game Developer Magazine*, February 1999. (Also available online at: [http://www.gamasutra.com/features/20000203/lander\\_01.htm](http://www.gamasutra.com/features/20000203/lander_01.htm); last accessed 2 Feb 2004.)
4. Magarshak, Greg. “Theory and Practice: Collision Detection, Part 2.” *Flipcode* online Web site, May 2000. <http://www.flipcode.com/tpractice/issue02.shtml>; last accessed 2 Feb 2004.
5. Turner, Bryan. “Real-Time Dynamic Level of Detail Rendering with ROAM.” *Gamasutra* online Web site, April 2000. [http://www.gamasutra.com/features/20000403/turner\\_01.htm](http://www.gamasutra.com/features/20000403/turner_01.htm); last accessed 2 Feb 2004.