Crossroads
The ACM Student Magazine

Faster 3D Game Graphics by Not Drawing What Is Not Seen

By ***Kenneth E. Hoff III***

Abstract

The increasing demands of 3D game realism - in terms of both scene complexity and speed of animation - are placing excessive strain on the current low-level, computationally expensive graphics drawing operations. Despite these routines being highly optimized, specialized, and often being implemented in assembly language or even in hardware, the ever-increasing number of drawing requests for a single frame of animation causes even these systems to become overloaded, degrading the overall performance. To offset these demands and dramatically reduce the load on the graphics subsystem, we present a system that quickly and efficiently finds a large portion of the game world that is not visible to the viewer for each frame of animation, and simply prevents it from being sent to the graphics system. We build this searching mechanism for unseen parts from common and easily implemented graphics algorithms.

Introduction

Over the past several years, the majority of the effort in writing 3D polygon-based games on low-end machines involved implementing a very specialized scene drawing, or rendering, pipeline, typically consisting of the following stages, roughly in this order:

- modeling and viewing matrix transformations to properly orient the world with respect to the viewer;

- perspective depth transformation and projection;
- polygon clipping (cutting to fit) to the screen window; and per-pixel scan conversion or rasterization, which includes polygon filling, shading, and hidden surface removal.

Each stage provides a very basic, low-level routine that can potentially be very computationally expensive. The viewing, perspective, and clipping stages require a large number of arithmetic operations for the vertices and edges of each polygon received; typically, the most expensive stage involves filling and shading the polygons, and resolving visibility relationships on a per pixel basis. The basic pipeline operations proceed as follows: each polygon in the world is

- transformed into a position convenient for viewing,
- clipped to fit within the viewer's field-of-view (or possibly rejected if invisible),
- projected onto the screen using proper perspectives,
- and then drawn, shaded, and tested for visibility *one pixel at a time* for all of the pixels it covers on the screen.

This is a very expensive pipeline that often requires implementation in assembly language or hardware in order to achieve the desired performance level.

These prohibitively expensive pixel (or raster) level routines have imposed severe restrictions on the 3D world complexity and on the generality of the overall graphics system. Often, in order to conform to these restrictions, the game worlds had to be so simple that the displayed scene quality was severely impaired; those of higher image quality often lacked smoothness of animation or sometimes could not even be animated (consider the case of *Myst*). More sophisticated real-time graphics engines (such as those used in games like *Doom*, *Descent*, and *Quake*) that are capable of higher quality while still achieving very fast frame rates have extremely specialized graphics pipelines; these often differ dramatically from the one previously described, and restrict the generality of the models used by imposing a more complex structure. Clearly, for games, this seems to be a very ingenious strategy that has worked amazingly well. However, these types of graphics engines can become quite difficult to implement, are often not easily ported to different computer graphics systems, and only benefit from hardware acceleration after some significant modification.

More recently, with an improved base-level of performance, a larger selection of fairly

standardized graphics libraries (OpenGL [see **http://www.sgi.com/Technology/openGL/**] and Direct3D [see **http://www.microsoft.com/mediadev/graphics/D3Dintro.html**]), and the coming of 3D graphics acceleration boards, the efforts of writing 3D game graphics engines can be somewhat more abstract, greatly simplified, and more easily generalized. Within this newly established framework, the average game programmer is allowed to assume that such a low-level graphics pipeline already exists on all systems. This allows them to focus on the higher-level aspects of game design without being overburdened by the low-level details. These basic operations will be provided with a fairly common programming interface, and the programmer can have the added confidence that their game engines will achieve greater performance not only from CPU upgrades, but also significantly from hardware acceleration of this common set of basic graphics routines.

Nevertheless, these low-level routines - whether they be highly specialized assembly routines or in hardware - are still not fast enough when given a large complex scene to draw. Very soon, it is quite likely that there will be graphics boards capable of rendering up to 100,000 polygons in a single second; however, for animation this rate is still not high enough. For smooth animation, we need to achieve approximately 30 frames per second. By our previous performance estimate, this results in only a little over 3,000 polygons capable of being rendered per frame. Polygon counts for newer game worlds push way past these limits, and future demands are sure to skyrocket. The problem becomes quite obvious: in a naive scene drawing strategy, all of the polygons in the scene are sent to these low-level drawing routines regardless of the current viewpoint. Inevitably, with even fairly small scenes we quickly become limited by the graphics pipeline's rate of processing these polygons. The solution is to simply avoid sending to the pipeline those polygons that we cannot see.

The idea of *not* drawing things we cannot see seems so obvious. So why even talk about it? The reason is that this idea brings up a slightly different concept called **conservative visibility** where we conservatively estimate the set of visible polygons. The goal here is to ultimately avoid sending polygons into a potentially overloaded graphics pipeline. Since it may be too expensive to determine the exact set of polygons that we can see, we can slightly overestimate to greatly reduce the complexity of the problem and provide a much faster solution. The simplest and most common technique, called **view-frustum culling**, involves discarding polygons that are outside of the viewer's current field-of-view.

View-Frustum Culling

In view-frustum culling, we wish to form a **viewing volume** (or viewing frustum) in the shape of an infinitely extending four-sided pyramid formed from the viewer's position at the peak, with a rectangular window into the world (representing the display screen window) at the base (this is not really a volume or a frustum in this context since it is unbounded, but the terminology is still used since other variations used bounded volumes). Before drawing the entire scene for the current frame, we can test each polygon to see if it is inside the viewer's field-of-view and simply discard those that are not. If polygons lie outside of the viewing frustum, we can remove, or **cull** them from the list of polygons to be drawn. Polygons that are inside or partially overlapping the frustum must be drawn since they can potentially be seen by the viewer. This explains why we call this a conservative visibility algorithm: polygons that are sent to the graphics subsystem may still be invisible to the viewer since they could be behind other polygons (for example, another room or the rest of a building behind a wall). Clearly, we can avoid sending many of the polygons to the more expensive low-level drawing routines, but at what point can this overlap test become so expensive so as to not gain anything? Obviously if all of the polygons are in the viewer's current field-of-view, they will have been tested for nothing. However, in large immersive scenes that are typical of 3D game worlds, we maintain a high average rate of culling. Nevertheless, this overlap test is not trivial in terms of computational complexity, and testing each polygon individually often leads to diminishing returns, especially since each polygon sent to the graphics pipeline that is outside the frustum will typically only undergo a matrix transformation for each vertex and a clipping operation (the most expensive stage is polygon filling, hidden surface removal, and shading, which is only done for polygons in the frustum). So what can we do as an alternative?

Bounding-Volumes

Often in 3D games and CAD programs, we have some structure to the world that can be exploited by the view-frustum culling system. If we can form clusters of polygons that can be culled away together, we can create a much more efficient overlap test. Imagine placing a simple bounding-volume (a sphere or cube) around a collection of polygons forming a single object in the scene. If we simply test the bounding-volume for overlap with the viewing-frustum, we can potentially trivially cull away or accept the entire group of polygons with a single overlap test! If (in the unfortunate case), we determine that the volume partially overlaps, we must then revert back to our previous individual testing of each of the polygons in the bounding volume. Nevertheless, this

provides a powerful method for rapid testing that can easily be extended in a recursive or hierarchical fashion.

Bounding-Volume Hierarchies

Previously, we took collections of polygons and surrounded them by bounding volumes. View-frustum culling begins by first testing the higher-level, simple bounding-volumes and then testing the individual polygons contained in the volumes that were determined to be partially overlapping. This results in dramatic savings, but if there is a large number of objects in the scene, it can still be prohibitively expensive. Imagine now recursively forming higher-level groupings formed by clustering neighboring bounding-volumes into larger volumes until the entire scene is contained in a tree-structure, with the root bounding-volume containing the entire scene. View-frustum culling can now proceed by first testing the root of the tree. If the root is completely inside or completely outside, we can trivially accept or reject respectively all of the children. In the case of a partially overlapping root node, we must traverse down to the children and test their subtrees. This process is repeated recursively down towards the leaves until the overlap test is resolved for all polygons in the scene. At first this seems like a lot more work, but the potential savings is now much higher since many polygons can be culled with only a few overlap tests. In fact, upon inspection only the bounding volumes crossing the view-frustum boundaries will be traversed to the lower depths of the tree; in practice, this dramatically improves the culling performance, especially for very large scenes with high complexity. It is very common to obtain 60% or higher rates of culling on average. Nevertheless, this still does nothing for all of the polygons in the viewing frustum that we cannot see; these are sent to the graphics pipeline only to get covered up by those occluding polygons. Culling these polygons requires a somewhat deeper conceptual understanding.

Backface Culling

If the scene is structured in such a way that the viewer never sees the backside of a polygon, we can safely cull away all polygons whose frontsides face away from the viewer. At first, this type of structure seems rather limiting, but it is quite common to have scenes composed of opaque, closed, solid models as those formed by rooms, hallways, and doorways. These examples can easily be modeled so that the user can only view the frontside of a polygon, and are quite typical in most games and

CAD designs. As a simple example, imagine an opaque sphere in front of the viewer. All of the polygons making up the sphere are clearly in the viewing frustum so they would not be culled with the previous method alone; however, all of the polygons completely in the hemisphere facing away from the viewer are effectively invisible and can be culled. This idea actually extends to all solid, closed models regardless of whether they are convex or not (assuming they are not self-intersecting). How do we know which way a polygon is facing? Each polygon has a **surface normal** (a vector perpendicular to the plane of the polygon) that points in the front-facing direction. If the angle between the surface normal and the vector formed from the viewer to the polygon is less than 90 degrees, then the polygon must be back-facing. This is a very simple test that can cull away a large portion of the scene lying in the viewing frustum. Using view-frustum and backface culling together, we seemingly have a very powerful and potentially efficient culling pipeline that will greatly reduce the load on the lower-level graphics system. In fact, this forms the basis of the classic culling system that has been used for many years; however, for game graphics, this is still not enough.

Cells and Portals

In the current system that combines view-frustum and backface culling, we still often perform a large number of relatively expensive bounding-volume and polygon overlap tests for view-frustum culling, and the amount of time performing the per polygon backfacing test could still be quite high because of the potentially high number of polygons that still lie in the frustum. Is there a better way? Yes, if we can restrict our models to fairly typical immersive 3D game worlds or large-scale architectural CAD designs that have rooms or open spaces separated by doorways or other connecting structures. Previously, we made view-frustum culling more efficient by clustering groups of polygons; we can now group by rooms, or **cells**. Each cell is separated by some doorway, or **portal**, and from any particular cell, other cells are only visible through a sequence of portals. While this may seem overly restrictive, imagine the scenes in any sophisticated game engine or architectural design; they often fall very nicely into the framework of cells and portals. This structure allows for an incredibly efficient culling strategy that can be used with or without the previously described methods.

The basic approach is very simple. First, find out which cell the viewer is in. Then, find out which portals leading out of that cell are in the viewer's field-of-view (use VFC strategy). Recursively apply the previous step for each visible cell, but restrict the viewer's field-of-view to only perform the portal overlap test against the frustum

formed from the viewer through the portals. Finally, render all of the cells visited. The recursion is necessary since it is possible to see a portal leading out of an adjacent room. View-frustum culling tests can be performed between the viewer/portal frusta entering a particular cell and the objects within that cell. Each cell can have a hierarchical bounding-volume representation of the objects contained. Also, backface-culling tests can be applied to the visible polygons through the viewer/portal frusta.

Conclusion

Meeting the increasing performance requirements for today's 3D games requires sophisticated techniques for reducing the load on current low-level graphics pipelines; naively sending all of the polygons to these potentially expensive routines simply does not suffice. In response, we have presented a system that not only meets these demands, but also provides a simple, easily expandable, portable, and general framework that can greatly reduce the load on any graphics pipeline. By combining these relatively simple algorithms, we can achieve a high average rate of culling for each frame of animation for scenes with high complexity. We have exploited the simplicity of conservatively finding polygons that we cannot see versus finding all those we can see, and have created a system that dramatically reduces the load on the graphics subsystem. We have shown this system's applicability to polygon-based rendering systems, but it is also generally applicable to scenes containing arbitrary primitives such as curved surfaces, conics, etc. The system can even be used as a culling stage before applying high-quality, still-image, realistic rendering schemes such as ray-tracing or radiosity based methods.

Suggested Readings

1. Foley J., A. Van Dam, J. Hughes, and S. Feiner. ***Computer Graphics: Principles and Practice***. Addison Wesley, Reading, Mass, 1990.
2. Zhangh, Hansong and Kenneth E. Hoff III. *Fast Backface Culling Using Normal Masks*. To appear in ACM Interactive 3D Graphics Conference, 1997.
3. Airey, John. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. Ph. D. thesis, UNC-CH CS Department TR #90-027 (July 1990).
4.

Luebke, David and Chris Georges. *Portals and Mirrors: Simple, Fast Evaluation of Potentitally Visible Sets*. ACM Interactive 3D Graphics Conference, Monterey, Ca, 1995.

Teller, Seth. *Visibility Computation in Densely Occluded Polyhedral Environments*. Ph.D. thesis, UC Berkeley CS Department, TR #92/708 (1992).

Kenneth E. Hoff III is currently a graduate student at the University of North Carolina at Chapel Hill working with the geometric modeling and virtual environment walkthrough groups. His research interests include real-time rendering techniques of large geometric databases consisting of polygon-based or NURBS-based primitives; realistic rendering such as ray-tracing, radiosity, and hybrid Monte Carlo methods; free-form surface modeling; physically-based modeling and animation; and 2D and 3D interaction techniques for navigating and manipulating a virtual environment. Many of his past and current projects are available at **http://www.cs.unc.edu/~hoff/**.