A Survey of Real-Time Rendering Algorithms

J. Scott Hofmann
CSci 361 Fall 2000

Abstract

This paper presents a survey of algorithms to aid in maintaining a real-time frame rate in computer-graphics applications. “Real-time” here refers to frame rates around 60 Hz, and the techniques are organized into two broad categories: Designing and implementing an efficient graphics pipeline and reducing the number of polygons displayed through level-of-detail algorithms, view-frustum and occlusion culling, and the use of image-based rendering to replace large amounts of complex geometry with simple polygons containing an image of the complex geometry.
1. Introduction .............................................................................................................................................. 3

2. Implementing the graphics pipeline ........................................................................................................ 3
   2.1 Immediate-mode Interfaces .................................................................................................................. 4
   2.2 Display Lists ......................................................................................................................................... 5
   2.3 Scene-graph Libraries .......................................................................................................................... 5

3. Geometry Processing ................................................................................................................................ 6
   3.1 Triangle and Quadrangle Strips ........................................................................................................... 6
   3.2 Level-of-Detail Processing ................................................................................................................... 7
      3.2.1 Height Fields, ................................................................................................................................. 9
         3.2.1.1 Scape ......................................................................................................................................... 10
         3.2.1.2 Virtual Geographic Information System (VGIS) ................................................................. 11
         3.2.1.3 Real-Time Optimally Adapting Meshes (ROAM) ............................................................. 12
      3.2.2 Static LOD Algorithms .................................................................................................................. 13
         3.2.3 Continuous LOD Algorithms ...................................................................................................... 14

4. Culling ..................................................................................................................................................... 15
   4.1 Bounding Volumes .............................................................................................................................. 16
      4.1.1 Oriented Bounding Boxes ............................................................................................................ 16
      4.1.2 Bounding Spheres ......................................................................................................................... 16
      4.1.3 Capsules ....................................................................................................................................... 17
   4.2 Frustum Culling .................................................................................................................................. 17
   4.3 Backface Culling .................................................................................................................................. 18
   4.4 Occlusion Culling ................................................................................................................................ 19
      4.4.1 The Hierarchical Z-Buffer ........................................................................................................... 19
      4.4.2 Hierarchical Occlusion Masks ..................................................................................................... 19
      4.4.3 Shadow Volume Culling .............................................................................................................. 22
   4.5 Portal Culling ..................................................................................................................................... 22

5. Image-Based Rendering ............................................................................................................................. 24
   5.1 Billboards .......................................................................................................................................... 24
   5.2 Impostors .......................................................................................................................................... 24
   5.3 Hierarchical Image Caches .................................................................................................................. 25

6. Conclusion ............................................................................................................................................... 26

7. References ............................................................................................................................................... 27
1. Introduction

A traditional focus of computer graphics research has been to provide interaction with 3D geometry displayed on screen. Interaction is usually only possible if the computer can provide a frame rate that is at least 15 frames per second, with higher frame rates providing a more useful or enjoyable experience. Many projects attempt to guarantee a frame rate which is equal to the monitor refresh rate, which is somewhat arbitrarily defined to be 60 Hz. This means that the computer has to refresh the frame buffer 60 times per second, which only gives 16 2/3 milliseconds of computation time per frame. Below that (also somewhat arbitrary) threshold of 15 Hz, or 66 2/3 milliseconds per frame, the display updates so slowly that the user finds it difficult to interact with the system.

Methods to provide or accelerate the frame rate can be categorized into hardware-based and software-based techniques. Hardware-based techniques center either around drawing more polygons in less time [Ake93, Mont97], or adding features such as sophisticated blending and compositing [Moln92, Torb96]. Economic or design constraints may prevent the use of graphics hardware, so in those cases a software solution is required.

Software-based solutions to the problem of providing a real-time frame rate can be organized into two broad categories: Designing an efficient graphics pipeline and reducing the number of polygons displayed, either through a polygon decimation algorithm or by culling polygons which are outside of the view frustum or otherwise occluded. These techniques are not mutually exclusive, and indeed multiple algorithms are usually used together to provide the highest possible frame rate.

2. Implementing the graphics pipeline

In order to refresh the frame buffer contents 60 times a second, the computer can only spend 16 2/3 milliseconds per frame. In these 16 2/3 milliseconds, the computer needs to project, clip, scan convert, and shade every polygon in the display [Watt92]. This is accomplished through the use of a graphics library. This library can either provide an interface to specialized graphics hardware if it exists, or it can implement the graphics pipeline entirely in software.
Graphics libraries can be sorted into three categories: immediate-mode libraries which retain only minimal state between polygon specifications, libraries built around the notion of a display list, and systems built around the concept of the scene graph. Many of the packages discussed here span categories, as they provide both immediate-mode and display-list-based interfaces, and some scene-graph libraries are built on top of an immediate-mode library.

### 2.1 Immediate-mode Interfaces

Immediate-mode libraries are called ‘immediate’ in that they retain only minimal state between function calls; retained state is typically limited to current color, current transformation matrix, and similar attributes. They retain no geometry between function calls, and frequently function as a kind of ‘assembly language’ for computer graphics. Due to their relative simplicity, most of the earliest graphics libraries were immediate-mode systems, and a lot of modern systems include an immediate-mode interface to better support applications which don’t need scene-database management or other sophisticated support in the graphics library. Some of the more advanced immediate-mode libraries are Starbase [HP91], IrisGL [SGI91] and its derivative OpenGL [ARB93], and Direct3D’s immediate mode interface [Micr00].

The function calls implemented by these libraries execute quickly because they aren’t doing much, and rarely need to allocate memory or other permanent or semi-permanent resources. Due to this lack of resource management, they necessarily foist off scene-database management and render-state management responsibility onto the user, so system performance is almost entirely based upon the quality of the application’s design and data structures.

Immediate-mode libraries include none to minimal support for interaction, as the library contains no information about the scene being rendered. Another weakness of the immediate-mode library is that it typically is coupled very tightly to a specific piece of hardware- IRIS GL does not exist outside of the Silicon Graphics hardware, and Starbase does not exist off of the HP hardware. This weakness is not an attribute of all immediate-mode libraries, as OpenGL has been ported to more platforms than any other graphics library in existence.
2.2 Display Lists
The next generation of graphics libraries included rudimentary support for retaining geometry, through a
data structure called a “display list”. A display list is a static list of graphics commands that have been
organized in the most efficient fashion possible. This is why the list is static – any changes may result in the
list being suboptimal, so the library does not allow the user to change the list in situ, if a change is desired
the old list must be destroyed and a new list created.

Two of the earliest toolkits that included display list support were GKS [ISO88] and PHIGS+ [vanD88].
Many modern libraries include support for display lists, most notably OpenGL.

Use of display lists can improve frame rates, as the graphics library is able to bundle rendering commands
together and transmit them to the rendering pipeline in bursts, therefore getting better use of the system bus
and other graphics hardware. For example, on PC hardware supporting the Accelerated Graphics Bus
(AGP) feature [Inte98], the NVidia OpenGL driver allocates AGP memory for display list storage. AGP
memory is a part of the machine’s main memory which has a separate high-bandwidth connection to the
video card.

Display lists are still merely an optimization of an immediate-mode library, and contain no support for
scene database management, or interaction with the rendered scene.

2.3 Scene-graph Libraries
The most sophisticated libraries that have been created to date are those libraries which include scene
database management facilities, typically through a data structure known as the scene graph. A scene graph
is a graph structure (usually a directed acyclic graph) of objects called nodes which are connected by edges.
The node objects contain the scene-database information, including geometry, materials, textures, lights,
cameras, and anything else necessary to implement the display.

The scene graph was first popularized in Silicon Graphics’ IRIS Inventor system [Stra92] (later known as
OpenInventor [Wern94]), and then refined in their IRIS Performer product [Rohl94]. Many other systems
implement the scene graph data structure in some form, including VRML97 by the VRML Consortium
[ISO97], NetImmerse from Numerical Design Labs [Bish98], and WildMagic from Magic Software
[Eber00].
Scene-graph libraries provide detailed support for scene-database management and operations upon that database, such as interaction, culling, and rendering. Graph traversal becomes paramount in producing a system which can provide a real-time frame rate, and traversal strategies are many and varied, from a simple tree walk (VRML, NetImmerse) to an “action” framework (OpenInventor, IRIS Performer).

3. Geometry Processing

The limiting factor in the display pipeline is the fact that each vertex of each polygon needs to be processed. Most vertices in a model are shared among several polygons, however, so the workload can be reduced by only processing shared vertices once. Also, as the polygon count increases, or as the camera moves further away from the polygons being displayed, several polygons may project onto a single pixel or small group of pixels. If we can remove these redundant polygons from the display, we can speed up the display process.

3.1 Triangle and Quadrangle Strips

An optimization implemented by every graphics library is to process shared vertices only once. To accomplish this goal effectively, the scene database needs to be organized so that the graphics library can readily identify shared vertices. Processing each vertex only once is also crucial to maximize bandwidth across the system or AGP bus. One algorithm to accomplish this goal is through the use of triangle strips and quadrangle strips (Figures 1a and b).

![Figure 1a, A Triangle Strip, and Figure 1b, A Quadrangle Strip](image)

Figure 1a shows a triangle strip containing 6 triangles. In order to specify each triangle individually, 18 vertices would have to be fed into the graphics library. The triangle strip primitive reduces that vertex count down to just 8 vertices, as each vertex is only processed once. The quadrangle strip containing 4 quads in figure 1b would take 12 vertices if each quadrangle was specified independently, but by organizing the
quadrangles into a quadrangle strip that vertex count can be reduced to 8 vertices. This reduction in the number of vertices that need to be fed into the graphics pipeline results in better utilization of the available bus bandwidth and fewer calculations necessary to project these polygons.

Generating these triangle strips in general is an NP-complete problem [Evan96], but some simple heuristics can yield some very useful (if sub-optimal) strips [Evan96b] (see figure 2). (Quadrangulation is also possible [Bose95], but as every quadrangle strip can be reduced to an equivalent triangle strip by the addition of a diagonal edge connecting the first vertex to the fourth vertex this problem hasn’t received nearly the attention that the triangle-strip-generation problem has received.)

Figure 2. A tri-striped 3D model. Each triangle strip is displayed in a different color from its neighbors.

One thing to notice about the object in Figure 2 is that while the vertices of adjoining triangle strips are shared, they are still fed to the graphics pipeline twice, once for each strip.

**3.2 Level-of-Detail Processing**

Another approach to the problem of reducing the number of polygons used to generate the display of a polygon mesh is through a family of algorithms known as Level-of-Detail (LOD) algorithms. What a LOD algorithm does is reduce the number of polygons in a polygon mesh by removing, combining, or
restructuring the mesh so that the mesh still looks the same or similar but with a reduced polygon count (see figure 3).

Figure 3. A polygon mesh and two level-of-detail approximations.
As Figure 3 shows, a significant number of polygons can be removed from a mesh without affecting the display fidelity much. Given the simplified mesh, storage, transmission, computation, and display of the mesh becomes faster and more efficient. Mesh simplification has applications that range widely beyond the problem of real-time rendering; LOD algorithms have been used in cartography, computer vision, finite element analysis, computational geometry, and approximation theory. A good survey of the applications of mesh simplification is available in [Garl97]. LOD algorithms are a subset of a more general category called multiresolution methods, which study the properties of functions evaluated at several different resolutions.

In our classification of the various level-of-detail algorithms, we’re going to sort among several criteria. The first criteria is the input mesh. Algorithms which process height fields, and algorithms which process polygon meshes. Since height fields have no polygonal representation, it is desirable to use level-of-detail criteria during the polygonalization step, rather than polygonalizing using some static criteria (e.g. building two triangles between four adjacent height-field values) and then simplifying that mesh. A second criteria is view-dependent vs. view-independent. View-dependent LOD algorithms simplify based on metrics derived from the viewing parameters, such as distance from the eye or number of pixels covered by the mesh’s projection onto the screen. Level-of-detail algorithms which process polygon meshes can further be classified as static (or discrete) vs. continuous. Static LOD algorithms generate a fixed number of reduced-complexity polygon meshes from an input mesh, while continuous LOD algorithms generate the simplified
mesh from an input mesh (usually also a simplified version of the original mesh) based on some criteria, such as distance from the eye.

### 3.2.1. Height Fields

Height fields are unique in that they contain no polygons – a height field is defined as an equation of the form $z = f(x, y)$ where $x$ and $y$ range over some subset of a Cartesian plane. The polygons are then derived from the $(x, y, z)$ values generated by evaluating $f$ for points on that plane, so that rather than simplifying a pre-existing polygon mesh we are creating a new polygon mesh. Therefore, it behooves us to take great care during the creation of the polygon mesh in order to avoid having to simplify the mesh in a post-processing step. Creation of an optimal polygon mesh (where optimality is defined as a minimization of the distance between the height field value $z$ and the generated polygon vertices) is a NP-hard problem [Agar94], which means that an optimal approximation cannot be computed in less than exponential time, so there is a tradeoff between the quality of the approximation and the real-time requirement to spend no more than 16 2/3 milliseconds per frame to generate and render the polygons.

Several algorithms have been developed to generate polygon meshes from height field data at real-time rates; they can be classified based on two attributes: Do they assume that the viewer is close to the height field (e.g. a character walking across a terrain), or do they assume that the viewer maintains a significant separation from the height field (e.g. an airplane flying over the terrain). Dave Eberly calls this the close observer vs. the distant observer assumption [Eber00]. The location of the viewer has significant impact on how we sample the height field and what error metric we use to determine the size and position of the generated polygons, and an algorithm which is built around the distant viewer approximation (e.g. [Lind96]) does not work well when the viewer is close to the terrain [Eber00].

Like other kinds of LOD algorithms, height-field triangulation algorithms can be sorted into view-dependent and view-independent algorithms. However, most near-real-time systems use view-dependent algorithms, since if the entire height field were triangulated ahead of time the resulting mesh would have too many polygons to display all at once and the mesh would be far too large to contain in memory, or even
in some cases in secondary storage. In addition, a view-dependent algorithm can avoid generating lots of polygons outside of the viewing frustum (see figure 4).

Figure 4. View-dependent triangulation of a height field, with the eye looking right. The dark region is outside of the view frustum, the light region is inside, and the grey region contains the frustum boundary. This particular triangulation was generated using ROAM [Duch97].

Michael Garland surveys techniques used to triangulate height fields in his survey of polygonal surface simplification algorithms [Garl97], and again in his survey of multiresolution modeling [Garl99]. I am going to highlight several algorithms used in production systems which offer real-time frame rates on consumer hardware (i.e. a machine with an Intel Pentium III chip and a NVIDIA GeForce2-based graphics card).

3.2.1.1 Scape
Michael Garland developed a view-independent system built around a triangulated irregular network (TIN) (see figure 5). In [Garl95] he presents a greedy insertion algorithms which takes a simple triangulation of a height field and on each pass, finds the data point with the greatest error and inserts a vertex to reduce that error. His algorithm requires that the height field reside in memory and also does not take the viewing parameters into account, so it will generate lots of triangles outside of the viewing frustum. It also does not handle a dynamically changing height field, such as the terrain in a game or military simulation which is changed by mudslides, explosions, and other events during the simulation. Changes to the height field require a re-triangulation pass, which cannot be accomplished in the span of time between one frame and the next.
Figure 5. A Triangulated Irregular Network produced by the Scape system [Garl95].

3.2.1.2 Virtual Geographic Information System (VGIS)
Peter Lindstrom et. al. produced a view-dependent continuous level-of-detail system which generates a polygon mesh from a binary triangle tree (also known as a triangle bintree) generated from a height field (see figure 6). Their system takes advantage of frame coherence to reduce the amount of work done on a per-frame basis by updating a set of visible vertices on each frame to remove vertices now outside of the viewing frustum and add vertices now within the frustum. It can also handle deformable height fields in that it measures the polygon mesh error with respect to the height field on every frame as well. VGIS builds its mesh bottom-up, using a vertex-reduction methodology which looks to remove vertices from a regular mesh whose error with respect to the height field is below a specified threshold. This bottom-up approach is also its biggest problem, in that its runtime complexity is proportional to the number of vertices in the polygon mesh. They also cannot support triangle-count limits because the algorithm works solely at the vertex level.
3.2.1.3 Real-Time Optimally Adapting Meshes (ROAM)

Mark Duchaineu et. al. developed an algorithm called ROAM which attempts to solve the problems of the VGIS system by taking a top-down approach based on the bounding boxes of height-field segments. It is also built around a triangle bintree, and it adds two priority queues, one containing vertices which require splits because their error has grown too large, and one containing vertices which require merges because their error has become too small (i.e. the vertices should be merged to reduce the vertex and polygon counts and therefore improve renderer performance). These operations are displayed in figure 7. ROAM can also handle deformable terrain, and since it takes a top-down approach it can support rendering primitives other than polygon soup such as Bezier patches. The only restriction is that the rendering primitive needs to be linked to the triangle bintree somehow. ROAM has problems at high levels of detail because it only considers how large the bounding volumes are when projected onto the screen; it does not consider the orientation of the bounding volume, so a lot of polygons are generated which alias together or project to small slivers on the display [Blow00].
3.2.2. Static LOD Algorithms

Static (or discrete) LOD algorithms pre-generate a set of reduced-complexity meshes from an input mesh (see figure 3). When an object is rendered, one of these meshes are drawn based on some criterion such as screen-space coverage or number of polygons. Static LOD algorithms have been used to great effect in architectural walkthrough systems [Funk93], and in computer games [Eber00], among other applications.

Static LOD algorithms are view-independent, as the levels of detail are generated in a pre-processing step prior to the execution of the program which will display the polygon meshes. The algorithms can be sorted based on how they generate the reduced-detail mesh, whether they preserve the topology of the input mesh, and what kind of error metric (local vs. global) is used when deciding how to reduce the input mesh.

Michael Garland has exhaustively surveyed the large number of static LOD algorithms in his surface simplification survey [Garl97] and his multiresolution modeling survey [Garl99], and the interested reader is referred to those survey papers for algorithm details.

Switching between different levels-of-detail at runtime introduces a problem known as “popping”. When switching between meshes, if the first mesh is just replaced by the second mesh a perceptible switch can be seen by the user. This transition can be smoothed by morphing from the first mesh to the second. [Turk92] describes an algorithm for interpolation between two levels of detail.

Figure 7. Split and merge operations in a binary triangle tree.
3.2.3. Continuous LOD Algorithms

Continuous (or dynamic) LOD algorithms have a much greater impact on a real-time system because the reduced-detail mesh is calculated on-the-fly before the mesh is rendered. These systems work via the mechanism of the edge collapse and its dual, the vertex split (see figure 8). A note on terminology: “edge collapse” is the term used by [Hopp96]; [Garl98] calls this operation “edge contraction.”

![Figure 8. The edge collapse (ecol) and vertex split (vsplit) operations.](image)

[Hopp96] defines the collection of a base mesh $M^0$ and a sequence of vertex splits $vsplit_0, ..., vsplit_{n-1}$ a “progressive mesh”. It describes a mesh at its simplest form and a series of operations to restore the original fully-detailed mesh. Meshes are preprocessed to produce this progressive mesh representation, and then at runtime a selected resolution is produced from the progressive mesh. The sequential nature of the vertex split operations make it easy and efficient to interpolate smoothly between levels-of-detail, [Hopp96] defines this interpolation as “geomorphing”.

A progressive mesh is generated by taking the full-detail input mesh $M^n$, defining a weight over each edge of the mesh using an energy function $E$ which includes terms to help preserve the surface geometry, preserve scalar attributes such as vertex colors, normals, and texture coordinates, and preserve discontinuity curves (hard edges, material boundaries). Then, the edge with the greatest energy is chosen and an edge collapse is performed. Once the energy function has been minimized, the resulting mesh is stored as the base mesh $M^0$ and the sequence of edge collapses is reversed and inverted to generate the sequence of vertex splits which form the progressive mesh (along with $M^0$).

The energy function used by [Hopp96] is view-independent, and as such the meshes it generates may be sub-optimal for some viewing transformations. This energy function can be replaced by a view-dependent function; Hoppe does this in [Hopp97]. The advantages of this view-dependent construction is that using the new energy function, we can avoid edge collapses which are outside of the viewing frustum or...
otherwise not visible (and therefore should not be rendered). This new function can also be used to implement selective refinement in the part of the mesh nearest to the camera.

Progressive meshes are currently being used by the NetImmerse real-time graphics library produced by Numerical Design Limited [NDL99] and in Microsoft’s DirectX 8.0 Graphics library [Micr00].

A faster scheme than the energy functions is the quadric error scheme described by [Garl97b]. Michael Garland defines the cost of an edge collapse by associating a 4x4 symmetric matrix Q with each vertex and then defining the error at a vertex \( v = [x\ y\ z\ 1]^\top \) to be the quadric form \( \Delta(v) = v^\top Qv \). This error is a measure of the sum of the squared distance from the set of planes defined at each vertex.

The WildMagic real-time graphics library [Eber00] implements Garland’s quadric error metrics as its simplification primitive, and Hoppe adds a quadric error metric to his progressive mesh framework in [Hopp99].

Both progressive meshes and Garland’s quadric error scheme require that the mesh and its simplified versions reside in memory during simplification. Peter Lindstrom has developed a scheme [Lind00] built around the vertex clusters of [Ross93] and the quadric error metric which does not require the full-resolution input mesh to be in main memory – the vertex cluster under consideration is the only thing that is required to be in memory. This is useful for systems which are memory-limited, as many game consoles have a minimum of hardware in order to keep their price down, and to interactively explore very large datasets, such as the Visible Human project [Acke98], whose dataset is well over 10 million voxels, and the Digital Michelangelo project [Levo99], which has generated models containing up to 2 billion triangles.

4. Culling

Another way to reduce the number of polygons being drawn is to not draw meshes which aren’t visible. A mesh isn’t visible if it isn’t within the view frustum, or if it is obscured by other elements of the scene being rendered. Algorithms which test an input mesh to determine if it should be rendered or not are known as culling algorithms, and there are two kinds of culling algorithms: frustum culling algorithms, which test a mesh to determine if it’s within the viewing frustum or not, and occlusion culling algorithms, which test meshes to determine if they’re visible (i.e. not obscured by other elements of the scene being rendered).
4.1 Bounding Volumes

All culling algorithms work with a simplified version of the polygon mesh being tested known as a bounding volume (BV). The bounding volume can be one (or more, depending on the application) of several standard shapes: an oriented bounding box (OBB), which is a box oriented in world space to provide a tighter fit than an AABB, a bounding sphere which is the sphere which encloses the mesh entirely, or some more specialized geometry such as a capsule or ellipsoid. Bounding volumes are critically important to any algorithm which requires a high-speed intersection or distance test for a polygon mesh, such as a culling algorithm or collision test.

Bounding volumes are not merely limited to enclosing polygon meshes; much work has been done developing hierarchical bounding volumes, which is a directed acyclic graph whose nodes are bounding volumes, and whose leaves are the BVs enclosing individual polygon meshes. Interior nodes are BVs which enclose the BVs of their child nodes, so that the root node is a BV which encloses the entire part of world space which contains renderable geometry.

4.1.1 Oriented Bounding Boxes

Oriented bounding boxes are some of the most frequently used bounding primitives. There are several kinds of OBBs available, ranging from the axis-aligned bounding box (AABB), which is merely the box defined by the minima and maxima of the mesh’s coordinates, to fitting a box to the points of a polygon mesh using an anisotropic Gaussian distribution [Gott96], to computing a best-fitting minimum-volume box through a minimization algorithm such as Powell’s direction set method [Pres88]. Computing a minimum-volume box is an expensive algorithm which should not be done at run-time, or used for deformable geometry, but is an excellent candidate as an ahead-of-time preprocessing step for static geometry.

4.1.2 Bounding Spheres

Intersection tests using bounding spheres are faster than intersection tests using bounding boxes, but spheres may not fit the geometry as tightly as a bounding box would. A bounding sphere can be easily derived from a bounding box [Eber00], but the fit of the sphere to the polygon mesh is mediocre at best because it encloses the bounding box, which itself encloses the geometry. A better fit may be obtained by
centering the sphere at the average of the points of the polygon mesh, and setting the sphere radius to be the smallest value for which the sphere at that center and radius encloses the mesh’s points. An optimal fit can be calculated in expected linear time [Welz91], but Welzl’s algorithm has polynomial worst-case performance, so it should not be used in real-time applications except as an ahead-of-time preprocessing step, and should not be used for deformable geometry.

4.1.3 Capsules
Capsules are an extension of the sphere BV to better accommodate long, thin polygon meshes. A capsule is defined as the set of all points that are at a distance \( r > 0 \) from a line segment with end point \( P \) and direction \( D \) (placing the other end point at \( P + D \)). This makes a capsule a cylinder with two hemispheres as end caps. Capsule BVs are about as expensive to generate as OBBs, which mean that they should not be used for deformable geometry but rather be computed ahead-of-time, but they offer faster intersection tests as separating axes do not need to be calculated [Eber00]. A capsule can be generated either through a least-squares fit, which is faster but produces a larger BV, or through an iterative process involving calculating the minimum of minimum-area projected circles, which is much slower but provides a tighter fit.

4.2 Frustum Culling
Algorithms which determine if a polygon mesh being rendered is within the view frustum are referred to as frustum culling algorithms. View-frustum culling involves walking the hierarchical bounding volume tree, testing each BV to see if it intersects the view frustum or not. If a BV does not intersect the frustum, then we can stop traversal because we know that the child BVs are also outside of the frustum [Clar76]. Use of frustum culling can improve other stages of the graphics pipeline: if a BV is entirely within the frustum, it does not need to be clipped against the frustum, so the polygon-clip stage of the pipeline can be disabled [Moll99]. We can also make this test more fine-grained, determining which planes of the view frustum which intersect the BV and then enabling clipping for those planes only [Bish98].

In cases where the majority of the polygons being rendered are static, a faster way to implement frustum culling is through the use of a spatial data structure such as an octtree [Hana89], or a Binary Space Partitioning (BSP) tree [Fuch80]. Another advantage of the spatial data structure approach is that occlusion culling (see section 4.3) can be combined with frustum culling, so that another traversal of the scene
database is not necessary. If the scene is highly dynamic, the cost of updating the data structures swamps the savings provided through their use, so the bounding-volume approach is better.

In animated scenes, we can take advantage of temporal (or frame-to-frame) coherence by storing the distance from a BV to the nearest plane of the frustum for all BVs outside of the frustum. This allows us to skip the intersection test for some rigid-body transformations, as we only need to update the distance [Assa99].

Most scene-graph based products implement some form of frustum culling; Direct3D [Micr00], IRIS Performer [Rohl94], and OpenInventor [Wern94] have frustum culling enabled by default. Frustum culling is also used in conjunction with the other kinds of culling; for example, portals can be culling using a frustum culling algorithm before portal culling is done (see section 4.5), and occlusion culling is usually performed after frustum culling has been done (see section 4.4).

### 4.3 Backface Culling

The simplest form of occlusion culling is known as the backface test, also known as backface culling. This is an inexpensive test usually done through either determining the winding order of the polygon being tested (the winding order is the direction (clockwise or counterclockwise) the vertices of the polygon are specified), or by computing the dot product of the polygon’s normal vector with the vector to the eye point. The winding order can also be calculated by computing the signed area of the polygon [Moll99]. Backface culling is usually turned on or off at a global level, and many products allow you to invert the test and perform front-face culling as well [Micr00, ARB93]. One caveat to be aware of is that mirroring transformations such as a negative scaling operation can make back-facing polygons appear as front-facing polygons and vice versa [Blin96], so the backface test needs to be updated after such as transformation has been applied.

While backface culling is effective for single polygons, it is more efficient to be able to perform a single test and use that to prevent the processing of an entire set of polygons. This is the focus of what are called *clustered culling* algorithms. Kenny Hoff has developed an algorithm [Hoff96] where he sorts a groups of polygons into bins called “clusters” by their normal vector, so that polygons with similar normals are
grouped together. He then performs one backface test for each cluster, and based on the results of that test either renders or ignores every polygon in that cluster.

Kumar et al. take a similar approach [Kuma96] where he builds a hierarchical tree of clusters, and each node contains a FrontRegion, a BackRegion, and a MixedRegion. Culling is then performed on a region-by-region basis, allowing for large numbers of invisible polygons to be pruned in a single operation.

### 4.4 Occlusion Culling

More sophisticated algorithms which determine if a polygon mesh being rendered is visible to the camera are known as occlusion culling algorithms. Most occlusion culling algorithms start with a spatial data structure (e.g. an octree) or a hierarchical representation of the scene being drawn and calculate the visibility of a given polygon from the eye’s point of view.

#### 4.4.1 The Hierarchical Z-Buffer

In [Gree93], Ned Greene et. al. developed an algorithm called the hierarchical z-buffer which takes the scene, represented using an octree, and an image pyramid built using the z-buffer called the z-pyramid. When traversing the octree, for each node that node’s bounding box is tested against the z-pyramid to determine if that box is visible. Within a node, each front-facing polygon is tested against the z-pyramid to determine visibility. For each face, the coarsest z-pyramid cell which encloses the face’s screen projection is selected, and then the z-value stored within the cell is compared against the face’s nearest z-value. If the face’s z value is less than the cell’s z-value, that face is potentially visible and the test recurses down to finer levels of the z-pyramid. Otherwise, the face is occluded and testing can stop at this point. For densely occluded scenes, this algorithm is able to often cull faces based on a single depth comparison.

#### 4.4.2 Hierarchical Occlusion Masks

Due to its reliance on an octree, the hierarchical z-buffer does not offer real-time performance for dynamic scenes due to the overhead of managing the octree. This limitation is addressed by Hansong Zhang’s Hierarchical Occlusion Masks (HOM) [Zhan97]. HOM is an image-space algorithm which works by restructuring the graphics pipeline some (see figure 9).
Figure 9. The modified graphics pipeline used by HOM.

All occlusion tests are done after the viewing transform has been applied, so all polygons are represented in the same space, with the eye located at the original looking down the negative z-axis, the y-axis pointing up, and the x-axis pointing right. HOM divides the occlusion test into two parts: a 1-dimensional depth test in the z-direction and then a two-dimensional overlap test in the xy plane. The overlap test can be extended to support approximate visibility, where an object which is “occluded” by some translucent geometry (e.g. a window) can still be drawn. Before either test is performed, a database of potential occluders is built, and the occlusion representation is developed from this database. Objects are selected for the database based on a few criteria: small objects are excluded, since they only cover small portions of the image unless the viewer is very close. Objects with a high polygon count (fine detail) are also excluded, because they will slow down the generation of the occlusion map too much. Objects with large or ill-shaped bounding volumes (e.g. a long skinny rod with a spherical BV) are excluded because they may cause the depth map to be too conservative. Last, redundant objects are excluded such as pictures on a wall – the wall is all that’s necessary to calculate occlusion.

For the overlap test, the occluders are first rendered into the color buffer using a white color on a black background, which can be done using the conventional graphics pipeline. In fact, it can be speed up because the occluders do not need to be textured, lit, or z-buffered. Another advantage of rendering into a color buffer is that several small occluders can be combined into one larger occluder by rendering them into the same buffer. This image, which is called an occlusion map, is then used as the base for the occlusion representation. This base occlusion map is then used to build a hierarchy of occlusion maps (HOM) by building an image pyramid through repeated filtering by 2x2 averaging filters (see figure 10). The HOM is
very similar to a mipmap, so texture-mapping hardware can be used to accelerate HOM generation by using a bilinear interpolation minification filter.

Figure 10. A Hierarchical Occlusion Map, and the original torus from which the HOM was built. The overlap test against the HOM starts by projecting the input mesh’s OBB onto the screen. A 2-dimensional bounding rectangle is then calculated from the projection, and that rectangle is tested against the HOM for overlap. This overlap test starts at the HOM level where the HOM pixel size is approximately the size of the rectangle. If all pixels are opaque (fully white), then the rectangle is occluded and the input mesh passes the occlusion test. Otherwise, we recurs down to the next greater resolution level in the HOM.

In the variant which computes approximate visibility against transparent or translucent objects, a threshold opacity is used to determine occlusion (i.e. a grey value).

The z-depth test is used to determine if an object is behind a selected occluder. Zhang implements the depth test using a depth estimation buffer, which does not require the presence of a z-buffer in the graphics pipeline. The depth estimation buffer is a software z-buffer whose elements are large relative to the pixel size (i.e. each depth buffer element spans several pixels). Each occluder is then rendered into this buffer, and the farthest z value at each element is stored (as opposed to a z-buffer, which would store the nearest z value). Note that this buffer must be recomputed each frame that something moves.
4.4.3 Shadow Volume Culling
A third form of occlusion culling uses intersection tests against the shadow volumes of occluders. This form of occlusion was developed by Hudson et. al. [Huds97] and it operates in a manner very similar to a frustum culler. This is due to the similarity between a shadow volume and a viewing frustum – the only differences are that a shadow volume has no far plane, and it may have more than four side planes. A hierarchical data structure (e.g. a hierarchical bounding volume) is then used to perform frustum culling, identify occluders, and then test each remaining object against those occluders. Good occluders are identified by calculating the solid angle subtended by the occluder’s bounding volume when projected onto the view plane. Larger angles denote better occluders.

4.5 Portal Culling
Indoor scenes can be rendered quickly through the use of a portal culler. Portals, which were introduced by John Airey in 1990 [Aire90], work through the development of a potentially visible set (PVS) for a given set of viewpoints. Portal-culling systems work through the classification of the scene into a set of 2-dimensional convex cells bounded by walls connected by portals (see figures 11a and b).

Figure 11a, a sample room layout, and Figure 11b, the sample layout divided into cells (white, grey, and striped), walls (black lines), and portals (blue with double arrows).
As figure 11b displays, cells are always convex polygons, so to support non-convex rooms such as the room on the right connected to the hallway it must be divided into several convex cells connected by portals (the white areas and the horizontally striped area). Airey builds a BSP tree from this cell graph, aligning the partition planes along the walls of the cell graph, and then uses point sampling or shadow volumes to build the PVS from the BSP tree.
Point sampling is not dependable, and shadow volumes can cause an excessive amount of overdraw, so Teller and Sequin [Tell91] developed an improved method of calculating the PVS from the BSP tree in conjunction with a cell-to-cell visibility graph. This graph is built by projecting a straight line from a point in one cell to another point in a different cell. If that line intersects no geometry, then those points are visible from each other. Cell-to-object visibility is also computed. Construction of the graph is very time-consuming, but for static geometry it can be calculated once and then stored for later queries. At runtime, the cell-to-cell and cell-to-object graphs are queried to determine what is visible from the eye point. The identified geometry is then culled against the frustum and rendered.

Luebke and Georges [Lueb95] have developed a simple method which can reduce the amount of preprocessing necessary to display a set of cells, through the use of a frustum culler (see figure 12). Starting with the cell $E$ containing the eye point, define a 2-D bounding box $P$ in screen space encompassing the entire screen. Render the cell $E$ containing the eye. Now, for each portal in that cell, project that portal on to the screen, and calculate its 2-D axis-aligned bounding rectangle in screen space. Intersect the portal’s bounding rectangle with $P$ and if that intersection is non-empty, then the portal is visible, render the cell $B$ beyond that portal, culling against the frustum that starts at the eye point and extends through the visible portal. Given that further cells may be visible, we must recurse to the step where we iterated over the portals of the cell $E$ but using the cell $B$ in $E$’s place.

![Figure 12](image.png)

**Figure 12.** A portal-based scene, with the portal frustums shown in white.

Portal culling has been used to great effect in the PC game industry. id Software’s game Doom was built around a portal-based renderer, as has its successors Quake and Quake 2. Epic Megagames’ bestselling
game Unreal Tournament is also built around a portal-culling system, and the NetImmerse library from Numerical Design, Ltd. [NDL99] features a portal-culling package.

5. Image-Based Rendering
Another way to reduce the polygon count in a scene is to replace large static meshes with simple quadrilaterals containing an image of the mesh being replaced. There are two ways in which image-based rendering (IBR) techniques are used to improve frame rates in real-time systems: through the use of screen-aligned polygons (also known as “billboards” [McRe98]), and through the replacement of geometry with an image of the geometry [Alia98].

5.1 Billboards
One of the oldest tricks in the book for displaying complex effects without rendering thousands of polygons (e.g. smoke, fire, explosions, or complex objects which occur hundreds of times such as trees) is through the use of screen-aligned or world-aligned polygons containing texture maps called billboards. To screen-align a polygon the polygon must be oriented so that its normal points in the opposite direction of the view-direction vector, and its up vector must be aligned with the camera up so that the billboard is parallel to the view plane. A world-aligned polygon skips the up-vector alignment step – it’s up vector is always equal to the world-up, hence the name “world-aligned”. Trees and other cultural features would use world-aligned polygons, while explosions would typically be screen-aligned. Individual particles in a particle system [Reev83] are usually screen-aligned billboards.

5.2 Impostors
Imposters, also called sprites, are a rendering primitive similar to a billboard which is used to display some complex geometry efficiently in cases where the viewer and/or the objects move slowly relative to each other [Scha95]. The complex geometry is rendered into an image, and then that image is texture-mapped onto the impostor. This allows the renderer to completely ignore the complex geometry and has been used to great effect in the Walkthru project at the University of North Carolina [Alia98].

Imposters have several severe limits on their usefulness, however. If the viewing is moving normal to the impostor, that is it’s getting closer to or further away from the impostor, then the fixed resolution of the
image may generate undesirable aliasing artifacts. Also, if the viewer moves so that it’s now viewing the impostor at an angle greatly different from the original angle when the impostor’s geometry was first rendered, the warping of the image can damage the illusion that the geometry is actually there (the presence of the impostor will be plainly visible). This can be rectified if we measure the resolution of the image relative to the resolution of the screen buffer and regenerate the image if those two resolutions diverge too much, or if the viewer moves so that the angle between the viewer and the image exceeds a threshold value.

5.3 Hierarchical Image Caches
Schaufler and Sturzlinger [Scha96] and Shade et. al. [Shad96] both independently developed the hierarchical image cache, which is a system where a hierarchy of imposters are used to reduce the number of times an imposter needs to be regenerated. The hierarchical image cache is built by first partitioning the scene into a hierarchy of bounding boxes, through the use of an axis-aligned BSP tree, and then generating an imposter for each tree node. Care is taken to minimize the number of objects which span nodes, as it is difficult to prevent cracks and other visible artifacts when an object is divided between two or more impostors. The imposter for a parent box is typically generated by rendering the impostors of the child boxes, rather than directly from the scene geometry, since the scene is typically too large and complex to render directly. Dynamic objects can be added to the scene if the impostors are nailboards [Scha97], which are impostors with a depth buffer associated with it so that objects can appear to move among the objects rendered into the imposter, or layered depth images [Shad98], which are nailboards with several depth buffers attached to the imposter so that the impostor can be warped more due to movement of the camera.
6. Conclusion
Maintaining near-real-time frame rates is a difficult thing, but a lot of research has been done to provide an interactive experience when viewing rendered scenes. This research can be broken down into two categories, efficient implementation of the graphics pipeline, and reducing the number of polygons displayed, either through a LOD solution, culling invisible objects, or replacing complex geometry with an imposter containing an image of the complex geometry. When several of these techniques are used in conjunction it becomes possible to render scenes that are richer and more complex than was possible before.
7. References


