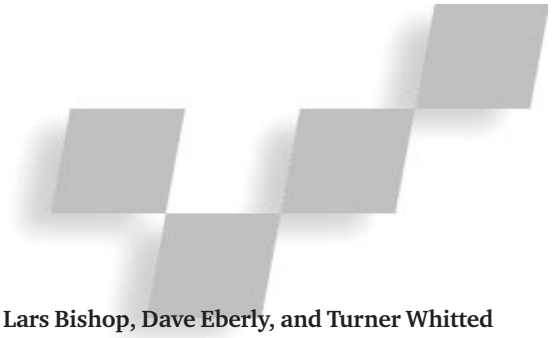


Designing a PC Game Engine

Lars Bishop, Dave Eberly, and Turner Whitted
Numerical Design

Mark Finch
HeadSpin Technologies

Michael Shantz
Intel

We outline here the requirements of a 3D game engine, illustrated by describing a particular engine's components. We designed the game engine, marketed as NetImmerse, to run on PCs with a broad range of performance levels, both with and without 3D graphics acceleration. We have found that a high-level programming interface need not compromise performance.

Computer game engine requirements change in response to evolving game platforms and demands for reduced development costs. We propose a 3D game engine for varying performance levels.

What's in a game?

Writers of early 2D computer games seemed obsessed with creating meaningful patterns on low-resolution screens in real time with essentially no memory. The tight link between game programs and game console hardware compelled programmers to write games in assembly language with little or no intervening system software. Such was the skill and patience of these "old-time" game writers that their product was usually entertaining and occasionally compelling.

Today, to write games this way would be prohibitively expensive. Programmers now write games in high-level languages to run on PCs with relatively adequate memory, fast access to secondary storage, powerful CPUs, high-resolution displays, fast audio processors, and in many cases hardware graphics acceleration. However, user expectations have outstripped PC performance increases, forcing developers to add content and features while struggling to maintain high performance.

Figure 1 schematically outlines an interactive game's vital elements. The items with solid outlines are essential to any game. As an example, consider *Adventure*, a successful game of the 1970s and 1980s. All input came from a keyboard and all output was text. *Adventure* consisted of little more than game logic in the form of a state table, minimal level data (mainly text strings), and a wrapper hardly recognizable as an event manager—yet it was an interesting and popular game. It is worth noting that a game is really a story, and the game logic is its plot; all other elements we add are just media through which we tell the story. As such, we can regard 3D graphics, audio, data management, dynamic behavior, and more complex input devices as extras. These "extras," however, hog almost all of a game platform's resources, creating an overwhelming need to execute them efficiently.

Building efficient components from the ground up for each game title is neither economical nor necessary. Developers reuse components from one game to another, so we customarily distinguish game *content* from the game *engine*. As we define it, a game engine includes all elements in Figure 1 that have no effect on actual content, that is, everything indicated by dashed lines plus an event loop. In practice, most game engines are tuned to a particular content style. For example, an engine tuned for flight simulators may not be appropriate for games that take place in tunnels and dungeons. We describe an engine that can be used for varying game styles while maintaining the efficiency of a tuned engine.

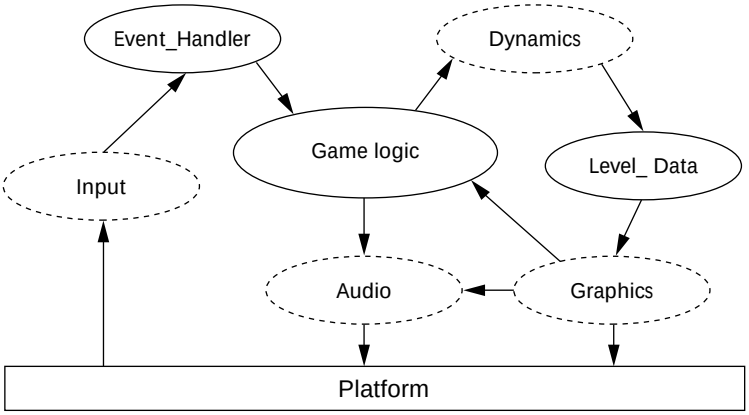
Background

Using scene management techniques to gain the greatest performance advantage from 3D display hardware has a long history, especially for visual flight simulator applications. Even early flight simulators were fed hand-optimized data at controlled levels of detail through just-in-time data prefetch channels to ensure that the graphical data working set was small and that the display hardware received only essential detail. Libraries such as Silicon Graphics' Performer[1] eventually provided this style of scene management to workstation applications as well.

PC games offer many examples of effective scene managers. The first 3D computer games used extremely simple and heavily constrained geometry. With no hardware or platform software support for 3D display, managing complexity was not as important as rapidly transforming and rasterizing polygons. For example, the graphics engine used in id Software's *Doom* includes a very efficient rasterizer, but the game's scene complexity is low

0272-1716/98/$10.00 © 1998 IEEE

to begin with. A few general-purpose retained-mode graphics packages such as Criterion Software's RenderWare,[2] Argonaut Technologies' BRender,[3] and Rendermorphics' RealityLab introduced scene management to PC applications. id's *Quake*[4] includes reasonably sophisticated scene management, most notably a culling mask to indicate which parts of a level are potentially visible to a viewpoint in some other part. For potentially visible areas, surrounding fixed surfaces such as walls are rendered first with no z-compare, and moving or decoration elements second with z-buffering enabled.



1 Schematic of a game.

These engines all yield remarkable real-time performance on modest PCs with no graphics hardware. However, the growing popularity of 3D computer games has spurred redefinition of the PC to accommodate them. Next-generation PCs include built-in graphics acceleration hardware and a memory port dedicated to graphics functions.[5] While these faster platforms promise richer games, they do not lessen the need for effective scene management and in fact make scene management programming more challenging by expanding the target platform's performance range.

Design goals

Somewhere in the back of our heads a voice tells us that speed is what really matters in a game engine. No matter what other design decisions we make, the resulting implementation must be fast. This has led us to employ very simple techniques universally, as simple and fast seem to go hand in hand.

However, we have some other goals in mind as well. For our sake as well as users', we ensure compatibility with standard low-level interfaces for graphics and audio. For spatialized audio, the engine is layered on top of Intel's RSX sound library. For drawing, the engine provides interface layers for both Direct3D and OpenGL. This strategy also helps us deal with target platform evolution, since it forces us to abstract the low-level display and audio functions.

We also provide a consistent, high-level application programming interface (API) for the entire package, largely for economic reasons. Controlling game development costs becomes more important as game players demand more complex content. Ideally, game developers should be able to create new titles quickly without having to learn the low-level details of sound and graphics programming.

At the extreme end, we might dispense with the API, provide a generic game executable, and have developers add data in a prescribed format. The *Doom* engine took this approach and produced a flock of level extensions to the original game, but they all look and play like *Doom*. This may prove reasonable for future titles and resembles loading virtual reality modeling language (VRML) files into a generic player for distributed applications,[6] but it does not currently provide the rich game design possibilities nor the degree of optimization needed for most new 3D titles. More recently, Java in VRML and a specialized language, Quake-C, for the Quake engine[7] have extended object behavior in games.

We took a more moderate approach by providing a common API to engine elements along with radically optimized implementations under the hood. This approach is more in the spirit of the SGI OpenInventor graphics package than that of a monolithic game engine. Some of the engine's novel "structural" features let game developers optimize for content in how they organize data and how the engine elements are invoked. Finally, the engine's object-oriented construction facilitates extension of the engine itself via subclassing. We detail these features below.

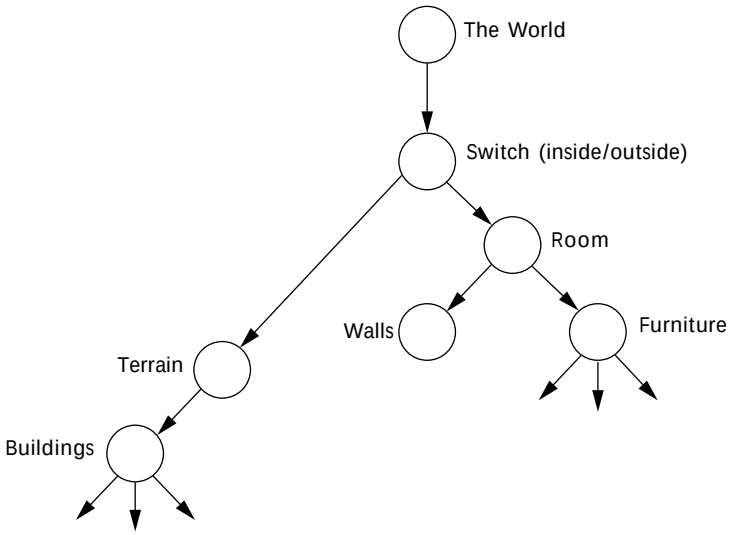
Scene management

We first discuss how our object-oriented, 3D commercial application engine incorporates scene management features in a novel fashion. The API for this library is high-level[8,9] but provides exceptionally fast execution.

Central to the graphical database is a tree-structured *scene graph*, common to traditional retained-mode graphics packages. (In fact, because the engine allows instancing, the scene graph is actually a directed acyclic graph (DAG), but we simplify most discussions of traversal by thinking of it as a tree.) The scene graph structure is not unique, but the manner of traversal for display is unusual. Missing from Figure 2 (on the next page) is the camera, an object not part of the scene graph. The root of a scene graph (there is no restriction on having a single scene graph in an application) is attached to a camera when both are initialized. It is sometimes appropriate to include a camera in the scene graph, for example, when mounting a camera in the nose of a missile that must be transformed along with the missile.

The *switch* node is a general selection mechanism. It includes a method for determining whether the viewpoint is inside or outside a room and selects one or the other subtree for traversal. The same mechanism with a different method provides control of discrete levels of detail or animation sequences.

The scene graph is an integrated data structure that includes audio emitters, bounding volumes, transformations, and properties as well as displayable surfaces. Audio and graphics rendering share a common traver-



2 Scene graph.

sal of the scene graph. Other functions such as geometric update and collision detection use local, partial traversals.

Efficient display

Display operates in two phases. First, the game application updates portions of the scene graph that have changed for any reason using `NeedUpdate()`. For efficiency this update is local to the scene graph's changing portions and does not require a global traversal. When the application invokes this method for any given scene graph node, geometric transformations are propagated downward through the node's descendants; bounding volume updates, which are much quicker to execute, are propagated upward.

The second phase of display is a "cull and show" traversal of the scene graph. `CullShow()`, a method of every scene graph node, operates as follows:

```
CullShow()
{
      cull();
      prepareDisplay();
      draw();
      undoPrepare();
}
```

We describe culling more fully in the next section. The `draw()` function is bracketed by calls that push and pop drawing state. While the API and function sequence are common to all nodes, each component's implementation is heavily tuned for the specific node type. This is a perfect example of object-oriented methodology both simplifying the programming interface and streamlining execution.

Efficient display technology depends heavily on level-of-detail management. Creating discrete levels of detail (LOD) lies in the artist's domain, and the LOD node, a subclass of the switch node, makes displaying them simple. Intense research focuses on how to create and display multiresolution geometric models with continuous level of detail.[10] Such models incorporate information and methods that permit them to adjust level of detail autonomously in much the same manner that the representation of procedural models can be expanded to an appropriate level of detail. Rather than treat this subject generally, we discuss a specialized procedural terrain model as an example.

Culling

As with almost any game engine, aggressively culling entire subtrees of the scene graph extracts the most efficiency from the graphics pipeline. Our engine maintains bounding spheres at each hierarchy level and updates them each time the scene changes. During traversal each sphere is tested against a list of clipping planes to determine on which side of the plane it lies. The list can contain more or fewer than the customary six boundaries of the view frustum.

The obvious advantage of hierarchical culling is that traversal can bypass whole subtrees that lie outside the view frustum. Less obvious is the potential performance advantage of eliminating clipping planes from the cull test as the traversal proceeds. For example, if a node's bounding sphere lies completely below the top clipping plane and completely beyond the near clipping plane, we need not test the bounding spheres of the node's children against these planes and can eliminate them from any further test in that subtree. If a node's bounding sphere lies completely within the view frustum, then we eliminate all clipping planes from the list and the overhead of the cull test is nil.

While eliminating clipping planes from cull tests provides a welcome speedup, its real value results from the low-level clipping operations for primitives in scene graph leaf nodes. Clipping against only one or two rather than six clipping planes provides a major performance improvement in low-level geometric operations. Current 3D graphics accelerator hardware for PCs relegates this part of the geometric processing to the host CPU; thus this efficiency directly affects a game's overall speed.

Dynamic collision detection

The bounding spheres for nodes in the scene graph can be used for various purposes such as detecting collisions between objects in the scene. We take advantage of the scene graph's hierarchical nature in computing object intersections.

A simple hierarchical approach to determining two objects' intersections, each represented by a scene graph subtree, follows. If the bounding spheres of the subtree's root nodes are disjoint, then the objects themselves do not intersect. If the bounding spheres intersect, the objects may or may not intersect. In this case, recursive descents of the subtrees are made and the bounding spheres of the children are compared for overlap. If at some point in the traversal the bounding sphere tests show that those portions containing one subtree's geometry are disjoint from the other subtree, then the objects

do not intersect. Otherwise, each subtree has a leaf node with geometry (that is, has vertices) for which the bounding spheres overlap. While the actual polygons those leaf nodes represent still might not intersect, the application may assume that an intersection has occurred.

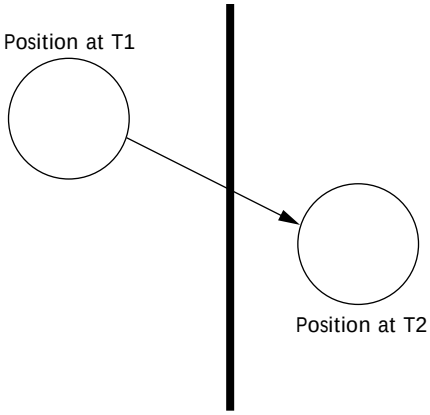
This simple algorithm is easy to implement but has a few drawbacks. First, if the application needs to compute collisions between N objects, the number of subtree comparisons is of order N^2. Second, if the geometry at a leaf node is complex and the bounding sphere does not accurately depict the represented object's shape, the application might need a more accurate intersection test. Third, the intersection tests are static, that is, they assume the objects have zero velocity. This is inadequate for a 3D game. For example, the quantum bowling ball in Figure 3 could pass through the wall without any collision being detected by a static overlap test. We could increase the sampling rate to lower the chance of this happening, but the computational cost would be prohibitive and it still might not work.

Our collision detection system avoids all three drawbacks. Detection occurs in three phases. The first allows the application to place objects in *collision groups*, each a collection of objects whose intersection is relevant to the game. A *collider* has a velocity (possibly zero); a *collidee* has no motion. For example, a collision group might consist of a bullet (a collider) and walls of a room (collidees). A tree corresponding to each collision group hierarchically orders the members based on axis-aligned bounding boxes. Under most circumstances, if the collision represents N objects, this tree allows $N \log (N)$ comparisons for intersections (rather than N^2 in the case of naive pairwise comparisons).

If the first phase does not rule out intersections among the members of a collision group, a second phase applies the hierarchical bounding sphere intersection tests to those pairs of objects that failed the first phase. If this still shows that two or more objects might intersect, then a third phase uses an extension of oriented bounding boxes (OBB),[11] which supports nonorthogonal oriented bounding boxes with velocities. Oriented boxes achieve a tighter fit than axis-aligned boxes on the geometry and thereby (possibly) reduce the number of box-box intersection tests.

Given a leaf node with geometry (represented as a list of triangles), the engine builds an oriented bounding box tree by analyzing the distribution of vertices at that leaf node. At the top level, it fits the vertices with a 3D non-circular Gaussian distribution. The engine uses the mean of the distribution as the center of the box and the eigenvectors of the covariance matrix for the box axes. It determines the axis lengths by processing the list of vertices and increasing each axis length as needed so that the final oriented box contains all the vertices.

The engine then partitions vertices for the entire leaf node into two sets based on the information obtained about the OBB. Each subset is fit with a Gaussian distribution and the corresponding OBBs are built. The algorithm proceeds recursively until the OBB tree has leaf nodes, each representing a triangle of the original scene graph leaf node. Comparison of two OBB trees resembles comparison of bounding spheres, but heuristically the tighter fit on the geometry reduces the number of comparisons.



3 Collision missed by static overlap test.

Moreover, the game engine allows the application to tune the system by specifying how many triangles should be stored in the OBB tree leaf nodes. One triangle per node gives more accurate intersection tests but requires more comparisons between OBBs. More triangles per node reduces the accuracy of the intersection tests, but generally the tree comparisons take less time. The engine also allows the application to specify how far down the OBB tree the tests should occur.

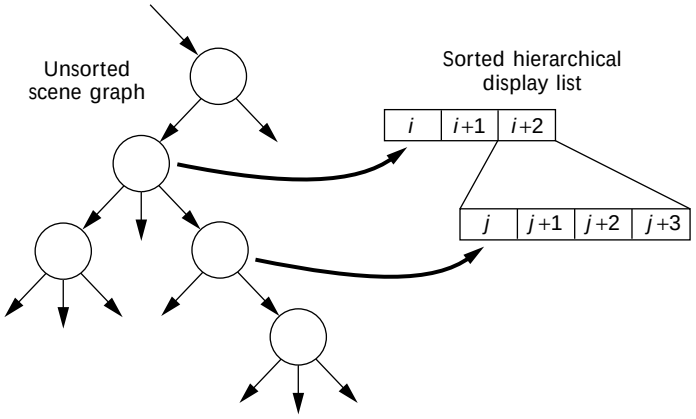
The static OBB comparisons proceed quickly because two boxes are tested for intersection by projection onto various *separating axes*. Determining intersections requires at most 15 axes,[11] but on average very few are needed to determine nonintersection. Surprisingly, the dynamic OBB comparisons require little additional work. Intervals represent the box projections on a separating axis. In the static case, two intervals are tested for overlap; if no overlap occurs on a single separating axis, the OBBs do not intersect. If overlap occurs on all 15 separating axes, the OBBs do intersect. In the dynamic case, the intervals corresponding to the box projections themselves have velocities. Determining an intersection of two moving OBBs means determining that for some time between the frame's start and end, the moving intervals either overlap or pass through each other for all 15 separating axes. The velocity of the OBBs are used directly in determining such overlap.

One of our experiments added 10 concave objects (each containing 24 vertices) as colliders to a collision group. We added a room's four walls, floor, and ceiling as collidees. The engine called the collision detection system during each frame processed for rendering (though it could call it less frequently). The amount of time spent processing the collision group approximately equaled the time spent rendering the scene graph to the display.

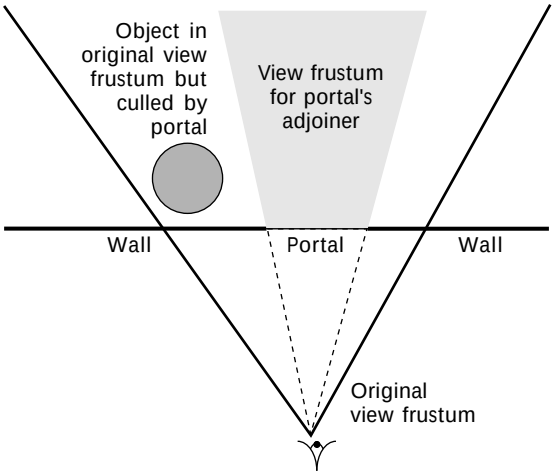
Hierarchical sorting

Objects in a scene are typically sorted in order of distance from the viewpoint to eliminate visibility tests and to enable transparency calculations. This by no means represents the only useful sorting order, nor does an entire scene need to be sorted using these criteria.

Traversing a tree-structured scene graph is straightforward unless we consider sorting. We can easily tra-



4 Hierarchical sorting.



5 Portal and culled object.

verse a scene graph constructed as a binary separating plane (BSP) tree in sorted order if the BSP tree is available. While it would be simpler to rely on z-buffered visibility tests and ignore traversal order, z-buffered hardware is not universally available and z-buffering has a significant memory footprint whether implemented in hardware or software.

In addition to BSP trees for sorting during traversal, our graphics processor also employs a deferred sort. When the deferred sort is enabled, a volatile sorted display list is constructed internally as traversal progresses. The display list encapsulates all information needed to render the object from which it is derived; otherwise, sorted rendering and instancing would not coexist gracefully within the engine. As Figure 4 illustrates, this is a hierarchical sort in which one element of a sorted list may be another sorted list or an entire scene subgraph.

For flexibility and efficiency, hierarchical sorting is enabled locally in the scene graph, that is, can be turned on or off at any node in the scene. We could enable sorting at the root and create a global sorted display list during traversal, but this would not be efficient for large scenes. A sorted list is flushed as soon as it and its descendants are completed. Using the scene graph in Figure 2 as an example, we do not need to sort the entire interior scene to resolve visibility if the walls form a simple box. Drawing all walls first and then drawing a sorted list of furniture suffices. Furthermore, we don't need to grow the sorted list with a hierarchy of furniture pieces if a BSP tree embedded in the scene graph describes each item.

Sorts are unrestricted. A sort method may obviously key on distance from the viewpoint, but a programmer may equally well sort based on display properties. For example, graph nodes may be sorted on both opacity and depth such that opaque objects are grouped but not depth sorted among themselves and transparent objects are grouped and sorted back to front. This method proves useful in efficiently rendering objects with transparency.

Specialized drawable objects

Each scene management technique described above is generic and works with a variety of shapes and scene types. However, we can gain significant speed by constructing specialized objects whose drawing methods take advantage of special cases. Two obvious examples are indoor scenes, such as a collection of rooms joined through portals, and outdoor scenes such as terrain.

Interiors and portals

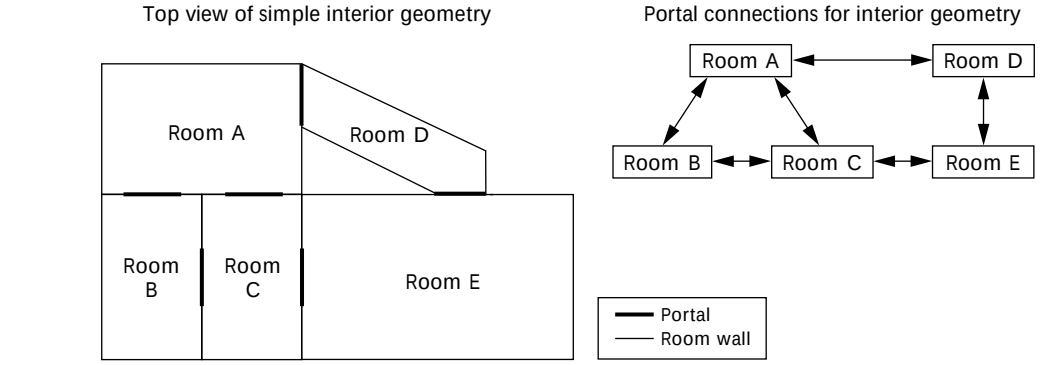
Two pieces of data represent portals, like any other scene graph object: the portal polygon itself, a convex polygon in space; and the *adjoiner*, a piece of geometry to be seen through the portal.[12] Note that the adjoiner is not a child of the portal in the scene graph sense. The adjoiner attachment method resembles a camera's scene graph—no transformations are inherited.

Portals draw themselves by adding a clipping plane to the camera for each edge of the portal polygon. These planes are each defined by a portal polygon edge and the camera center. Thus, each portal adds a new frustum to the set of culling/clipping planes, as shown in Figure 5. Having pushed the new clipping planes, the portal draws its adjoiner and then removes the clipping planes it added to the camera. Geometry in a portal's adjoiner is thus culled and clipped to the portal.

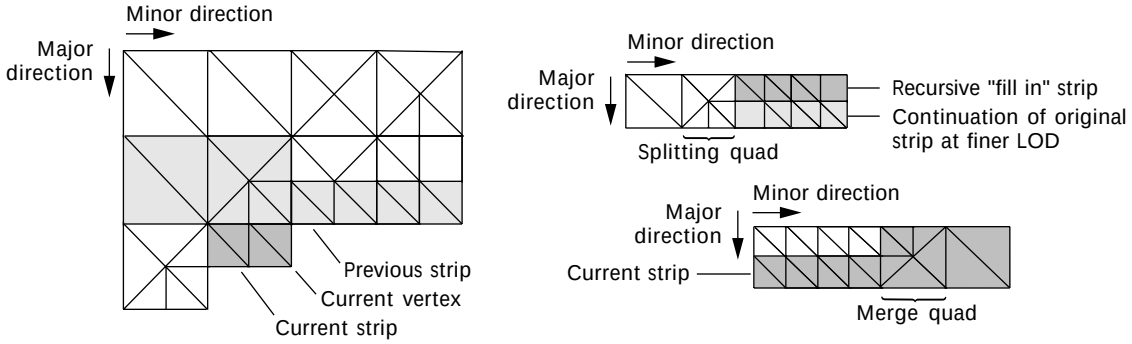
Portals are the enabling elements of a higher level room-to-room system, represented at the top level by a *room group* object. This is a scene graph node that represents an entire set of adjoining *rooms* and the portals between them. The room group uses an undirected graph structure, shown in Figure 6, in which the nodes are rooms and the edges are windows and doorways between them.

In this context, a room is a convex polyhedral space. Each room has three object classes: (1) the walls and (2) the portals, representing the shape of the convex polyhedron itself, and (3) the fixtures, the room's interior objects and furnishings. Each portal in a room references another room, and a full set of rooms forms a connected graph. We can create non-convex rooms by joining together sets of convex rooms, fitting the seams with a portal (or portals).

We draw a set of rooms as follows. First, we find the



6 Simple interior geometry and portal connectivity.



7 Sorted height field traversal.

room containing the camera using an application message, full search of the room group, or an incremental search starting from the last room known to contain the camera. Once we locate this room, we create a depth-first drawing of the room graph using the following (this example is for back-to-front sorting, but front-to-back sorting is done analogously). Starting with the room containing the camera,

```
if the current room is being visited
  return to calling room
else
  mark the current room as being visited
  disable all of the room's incoming
   portals
  for each portal in the room
  test the portal against the current
   clipping planes
       if the portal is visible
            push the portal's
             clipping planes
            and recursively draw the
             room
  endfor
  draw the current room's walls
   (unsorted)
  draw the current room's fixtures
   (sorted)
  re-enable all of the room's incoming
   portals
  mark the current room as not being
   visited
  return to calling room
```

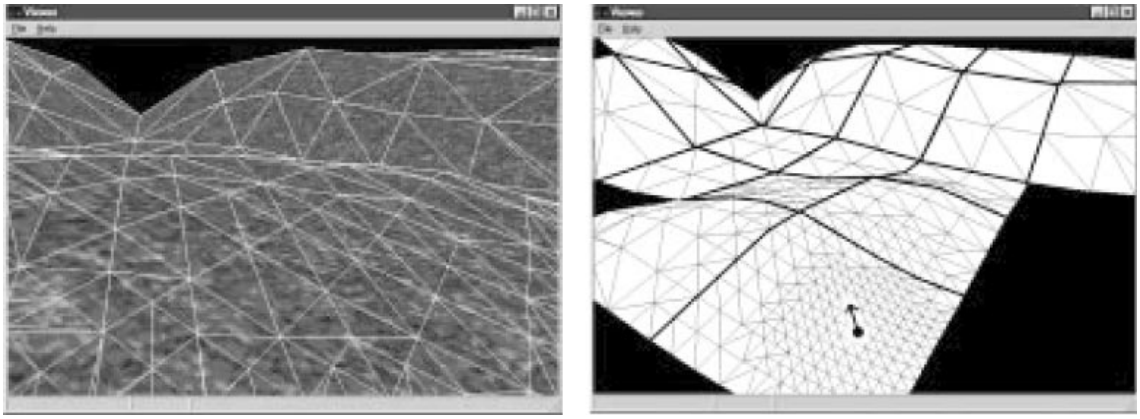
All objects in the rooms are listed as being placed in the room containing them, but this need not be the case. If using z-buffering "write-only" as the room group is drawn, then z-buffer testing can be turned on after all the rooms are drawn, and objects physically in the rooms can be drawn. This allows the application to avoid keeping track of an object's current room while still achieving efficient drawing and correct visibility.

This system allows for correct back-to-front or front-to-back sorting of a large set of interior rooms. It also keeps the depth complexity of all drawn walls themselves very close to 1, an important consideration with software rasterizers. Finally, only potentially visible rooms are drawn (as the system visits rooms through visibility portals and stops when a given room is culled by any of the clipping planes pushed by other portals).

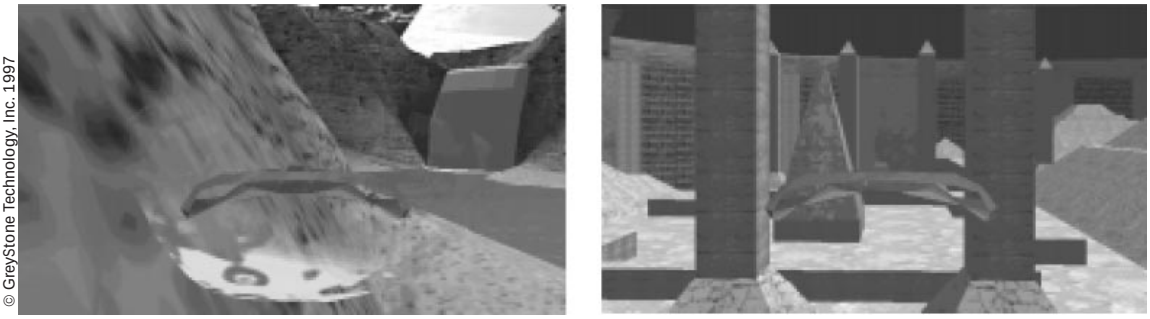
Exteriors and terrain

Because terrain can be represented so compactly as a height field, we store terrain as a collection of pages, each of which stores a 65×65 array of elevation values at the densest resolution available. Since only a few pages, typically 30 or 40, are visible in any given frame, the cost of storing the highest resolution data remains low. Maintaining a triangle mesh data set, essential for accuracy in terrain display systems,[13] would be impractical in a game. Instead, the terrain object generates triangle meshes at the appropriate level of detail on the fly as each height field page is traversed in back-to-front order. This turns out to be far more efficient than it sounds.

Depending on the camera's position and direction of view, the engine defines a major and minor direction of scan for each cell, illustrated in Figure 7. As it scans the



8 (a) User's view of (b) the active data set.



9 Two scenes from *Canyon Runner* by GreyStone Technologies.

elevation grid along the minor direction, it uses the level-of-detail measure to determine how many elevation values to skip over when choosing candidate triangle vertices. The engine applies fixed rules for splitting or merging meshes, shown on the right in Figure 7, at points where the level of detail changes. The rules produce no cracks at a transition between levels of detail, and frame-to-frame morphing of elevation values selected for triangle vertices mitigates the annoying "popping" artifact at a level of detail shift.

Figure 8a shows a line drawing of polygon boundaries superimposed on the rendered terrain, indicating that the terrain object maintains nearly constant screen space area for each triangle. This is important for driving rasterizers that do not compute perspective-corrected texture map indices, but is less important for most hardware rasterizers. Figure 8b illustrates the scene's geometry viewed from a different perspective (the original viewpoint and view direction are indicated by the arrow). Only a few pages (bold outlines) are visible in the frame; distant pages are triangulated with very few triangles, and the dynamic range of object-space polygon sizes is large.

This terrain object, though by no means perfect, represents a tradeoff between accuracy and speed that is appropriate for games. Furthermore, in keeping with the game engine's design goals, it scales very well across a broad range of PC performance.

Results

Developers at GreyStone Technologies used NetImmerse to create the PC version of *Canyon Runner*, an arcade game. NetImmerse also forms the basis for other titles currently in development. These development efforts share a common thread: faster prototype programming. The developers' experience justifies our goal of designing a high-level, object-oriented API for fast, efficient game programming.

Figure 9 shows scenes rendered with generic components only, without any specialized objects such as terrain or portals. We can obtain 30-Hz frame rates on a 200-MHz Pentium Pro CPU with a 3Df/x Voodoo Graphics accelerator card. This rate includes collision detection, shown in Figure 9, left, where the spacecraft's proximity to the canyon wall triggers the translucent force field surrounding the craft.

The efficiency of the specialized drawing objects is lost if game developers cannot create the objects. An important, often overlooked need is for modeling features to be matched to runtime features. Game developers employ a filter to process content because most games mix content from multiple sources and because most modeling tools' external data format does not necessarily work as a game's internal format. As procedural geometry and specialized game elements become more common, tuned modeling filters become even more important parts of the game authoring process.

The engine users' learning curve is as important as runtime performance but proves much more difficult to summarize quantitatively. Invariably, initial prototypes are not optimized and require that developers bring frame rates up to expected levels. In other words, the engine does not magically apply the techniques described in this article. Ironically, tuning in NetImmerse requires considerable programmer attention because of the flexibility the system offers. However, developers who learn and understand the

engine's scene management techniques can apply them successfully within a few weeks (in some cases days) and achieve the runtime performances needed to make games visually exciting. ■

References

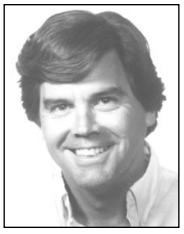
1. J. Rohlf and James Helman, "IRIS Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Proc. Siggraph 94*, ACM Press, New York, 1994, pp. 381-394.
2. Criterion Software Ltd., *RenderWare Technical Specification*, http://www.csl.com/RenderWare/rwtech.htm.
3. Argonaut Technologies Ltd., *BRender Technology Features*, http://www.argonaut.com/.
4. M. Abrash, *Michael Abrash's Graphics Programming Black Book, Special Edition*, The Coriolis Group, Scottsdale, Ariz., 1997.
5. Intel Corp., *Accelerated Graphics Port Interface Specification*, http://developer.intel.com/pc-supp/platform/agfxport/.
6. A. Ames et al., *VRML 2.0 Sourcebook*, John Wiley and Sons, New York, 1997.
7. Olivier Montanuy, *Unofficial Quake-C Specification*, http://www.gamers.org/dEngine/quake/spec/quake-spec34/index1.htm.
8. M. Shantz, "Building Online Virtual Worlds," *Proc. Graphicon 96*, Grafo Computer Graphics Society, St. Petersburg, Russia, Vol. 2, July 1996, pp. 12-17.
9. Numerical Design Ltd., *NetImmerse Programmers Manual,* http://www.ndl.com.
10. M. Eck et al., "Multiresolution Analysis of Arbitrary Meshes," *Proc. Siggraph 95*, ACM Press, New York, 1995, pp. 173-182.
11. S. Gottschalk, M. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection," *Proc. Siggraph 96*, ACM Press, New York, 1996, pp. 171-180.
12. S.J. Teller and C.H. Séquin, "Visibility Preprocessing For Interactive Walkthroughs," *Proc. Siggraph 91*, ACM Press, New York, 1991, pp. 61-69.
13. P. Lindstrom et al., "Real-Time Continuous Level of Detail Rendering of Height Fields," *Proc. Siggraph 96*, ACM Press, New York, 1996, pp. 109-118.

Lars M. Bishop *is a technical staff member at Numerical Design Limited, where his interests include consumer-level real-time 3D graphics. He received a BS in mathematics and computer science from Brown University and an MS in computer science from the University of North Carolina at Chapel Hill.*

Dave Eberly *is the director of engineering at Numerical Design Limited. His technical interests include real-time 3D computer graphics and computational differential geometry. He received a BA in mathematics from Bloomsburg University, an MS and PhD in mathematics from the University of Colorado, and MS and PhD degrees in computer science from the University of North Carolina.*

Mark Finch *received a BS in physics from Georgia Institute of Technology and an MS in computer science from the University of North Carolina at Chapel Hill. He is currently with HeadSpin Technology, where he focuses on 3D ambients for autonomous environments.*

Michael Shantz *is a principal engineer in Intel's Microcomputer Research Laboratory. He received an MS in biomedical engineering from Drexel University and a PhD in information science from the California Institute of Technology in 1976. His research interests include computer graphics, scene management architectures, articulated body dynamics, modeling autonomous interacting behaviors, and genetic programming.*



Turner Whitted *recently joined Microsoft as a senior researcher. He has been a research professor of computer science at the University of North Carolina at Chapel Hill for the past 14 years as well as a cofounder and director of Numerical Design Limited. Prior to that he was a technical staff member in Bell Labs' computer systems research laboratory. He earned BSE and MS degrees in electrical engineering from Duke University and a PhD from North Carolina State University. He is an editorial board member of* IEEE Computer Graphics and Applications*, was papers chair for Siggraph 97, and is an ACM Fellow.*

Contact Bishop at Numerical Design Limited, 1506 E. Franklin St., Suite 302, Chapel Hill, NC 27514, lmb@ndl.com.