dPVS: An Occlusion Culling System for Massive Dynamic Environments

Timo Aila
Hybrid Graphics and Helsinki University of Technology

Ville Miettinen
Hybrid Graphics and University of Helsinki

A platform-independent occlusion culling library for dynamic environments, dPVS, can benefit such applications as CAD and modeling tools, time-varying simulations, and computer games.

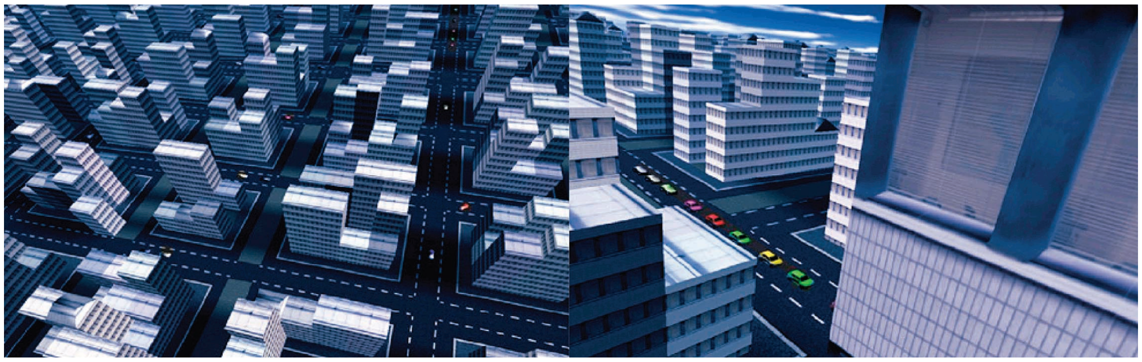
Visibility optimization is currently the most effective technique for improving rendering performance in complex 3D environments (see Figures 1 and 2). The primary reason for this is that during each frame the pixel processing subsystem needs to determine the visibility of each pixel individually. Currently, rendering performance in larger scenes is input sensitive, and most of the processing time is wasted on rendering geometry not visible in the final image.

In this article we concentrate on real-time visualization using mainstream graphics hardware that has a *z*-buffer as a de facto standard for hidden surface removal. In an ideal system only the complexity of the geometry actually visible on the screen would significantly impact rendering time—3D application performance should be output sensitive. Furthermore, opportunities exist for using lower accuracy for artificial intelligence, physics computations, and collision detection in the hidden parts of the scene.

A vast majority of applications have at least partially dynamic environments. Modeling packages, computer games, and military simulations have moving objects such as characters, vehicles, or destructible buildings. Our main goal was to develop a system that adapts to runtime changes in the environment. The primary issue in dynamic adaptation is maintaining a spatial subdivision. Strict real-time requirements pose limitations on the choice of subdivision algorithm. For example, generic BSP trees or partitioning into convex cells are difficult and expensive to maintain under dynamic changes. Additionally, to retain output sensitivity, the database must provide an approximate front-to-back traversal of the scene's visible subset.

An industrial-strength visibility determination system must not pose artificial limitations on the input geometry: the objects can't be assumed static, convex, two manifold, or closed. Also, modeling packages might decompose scenes into objects in a manner not suitable for efficient rendering or visibility determination. For example, they might group all windows of a large building into a single object. We optionally perform an automatic restructuring of the input data to obtain spatially coherent objects.

Another important application of our system is the

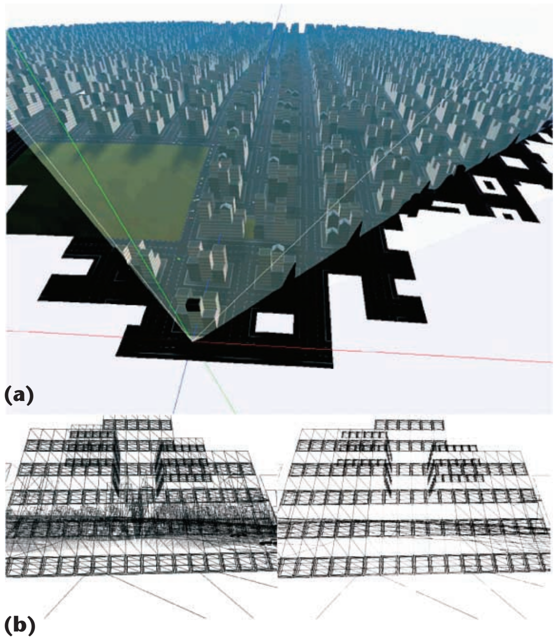

1 **Traffic Jam. Large dynamic simulation of 16,000 buildings and 20,000 moving cars. Occlusion culling avoids processing hidden objects and level-of-detail mechanisms simplify the visible geometry. Our system can render walkthroughs of the scene at 30 Hz on a low-end PC.**

out-of-core rendering of massive models. Only the bounding volumes of the hidden parts of the environment need to be loaded into memory. This is a key feature when a scene requires more memory than available, or when it's visualized on a remote machine over a limited-bandwidth connection.

A level-of-detail (LOD) mechanism can further improve rendering performance or reduce memory footprint. Visibility optimization and LOD are complementary techniques; both are required for real-time visualization of complex scenes. Our system provides information for selecting an appropriate LOD level. Optionally, visual artifacts can be traded for higher overall performance by enabling contribution culling, which discards barely visible objects.

One common characteristic of our target platforms is limited memory bandwidth. Also, the connection between the CPU and the graphics processing unit (GPU) is practically unidirectional—requesting data from the GPU is slow and has a high latency.

These factors have a great effect on the design of efficient culling algorithms and have motivated our choice of a purely software-based approach.



2 Occlusion culling is to minimizes a frame's rendering time by quickly finding a tight superset of the visible objects. (a) Parts of an urban environment intersected by a view frustum. (b) Wireframe renderings of the buildings from the same viewpoint.

Related work

Output-sensitive visualization systems performing view frustum culling and occlusion culling require an acceleration structure, the spatial database. In certain regularly structured scenes, a uniform subdivision is sufficient, but in practice, the database needs a hierarchical organization. Additionally, output-sensitive occlusion culling requires approximate front-to-back traversal of the database. A visibility solver is responsible for answering occlusion queries—that is, "Would this object contribute to the final image if it was rendered?" The visibility solver can be implemented either in software or using modern graphics hardware. Due to its complex nature, the spatial database needs to be implemented in software.

Researchers have proposed numerous occlusion culling systems for the limited case of urban environments. Wonka covers many of these in his recent thesis.[1] These systems assume that buildings are essentially boxes with different heights and therefore reduce the visibility problem into a 2D subproblem.

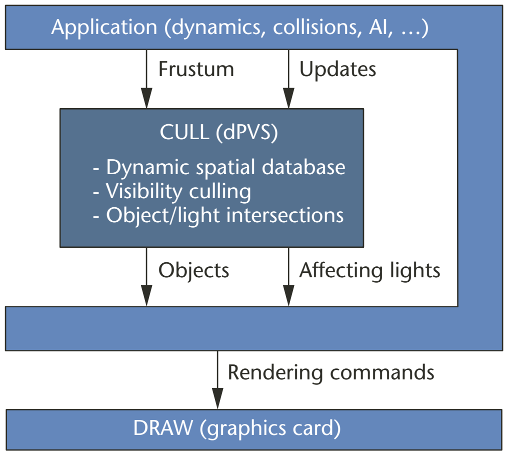
Airey et al.[2] and Teller and Séquin[3] propose subdividing the scene into cells connected by portals. This subdivision is particularly well suited for architectural scenes. Most intuitively this is explained with cells corresponding to rooms, and portals to doors and windows through which other rooms can be seen. Computer games often incorporate this approach, with scenes mostly consisting of closed indoor environments.

Several object-space algorithms have been proposed for static scenes; Cohen-Or et al.[4] cover many of these in their comprehensive surveys. An object is often blocked from view by the combined effect of several occluders. In practice, such occluder fusion is required for retaining output sensitivity. Exploiting occluder fusion is difficult in object space whereas image-space discretization makes it relatively simple. Therefore we concentrate on image-space algorithms.

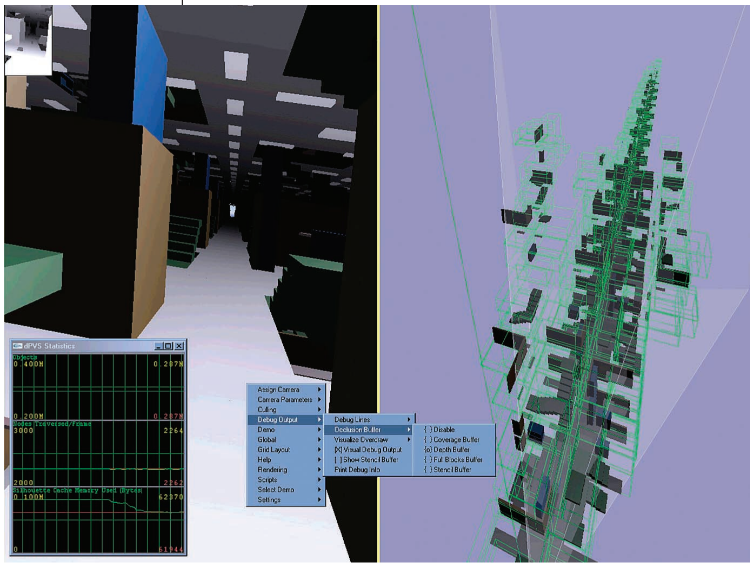
Greene and Kass propose extensions to the z-buffer hardware for avoiding per-pixel depth comparisons.[5] They organize the scene into an octree for approximate front-to-back traversal. An occlusion query is performed for each octree node. If a node is not visible, neither it nor its children need processing. The z-buffer is organized into a z-pyramid, the lowest level of the hierarchy being the traditional z-buffer. A primitive can be quickly determined hidden if its 2D bounding rectangle with the primitive's closest depth value is hidden. In many cases a single comparison on the appropriate z-pyramid level can accomplish this. Because the visibility information is generated as a side product of the actual rasterization, it's possible to use all visible objects as occluders.

ATI, Hewlett-Packard, Nvidia, and possibly others have implemented hardware occlusion queries. Their occlusion culling extensions provide a simple way for confirming the visibility of a complex model before it's rendered. Output-sensitive, front-to-back traversal of a spatial database typically requires hundreds or even thousands of occlusion queries every frame. HP's implementation effectively synchronizes the CPU and the graphics card, causing them to operate at a suboptimal performance level. Nvidia's implementation is more flexible, issuing several queries simultaneously and wasting less time waiting for the results. Hillesland et al. note that efficient use of hardware requires careful scheduling of occlusion queries and rendering of objects, as otherwise excessive fill rate problems are encountered.[6] The major cause for the increased fill rate is that the occlusion query semantics in Direct3D 9.0 require returning the number of visible pixels; implementations can't thus early exit a test when the first visible pixel is found.

Morein describes a hardware implementation of a two-layer z-pyramid (see http://www.merl.com/hwws00/presentations/ATIHot3D.pdf). A z-pyramid with two or three layers is implemented in commodity graphics hardware, such as ATI Radeon and Nvidia



3 **Our system maintains a dynamic spatial database internally. The application notifies dPVS about changes in the scene geometry. A visibility query returns a list of potentially visible objects for the submitted view frustum. The application then sends the related rendering commands to the graphics hardware.**



4 **dPVS'profiling and debugging tool collects hundreds of different statistics and provides visualizations of the algorithms and data structures. The right part of the image shows a bird's eye view of the visible subset of the Naked Empire scene.**

GeForce3. In addition to memory bandwidth optimizations, parts of the incoming triangles can be culled before rasterization. The occlusion query can also execute faster because of the hierarchical representation. However, this approach works well only if the scene is drawn in a front-to-back order.

Klosowski and Silva[7] first render an approximation of a scene using a fixed budget of resources. They subdivide the scene into convex cells as a preprocess and compute a solidity estimate for each cell. Their front-to-back traversal is controlled by a priority queue, which orders the nodes according to their expected importance. The algorithm estimates a node's importance from the currently accumulated solidity values of the corresponding screen-projection area. A second pass finds the rest of the visible objects using hardware occlusion queries until the visible set converges.

The algorithm can guarantee output-sensitive visualization and impressive real-time results have been published. The authors did not design the system for dynamic scenes. In particular, a scene can't be easily subdivided into convex cells at runtime and meaningful solidity estimation is tricky when objects are moving.

Zhang et al. point out that the occlusion query can be divided into two separate subtests: one for coverage and one for depth.[8] The occlusion query is conservatively correct as long as the coverage test is performed at full resolution and the depth test is performed conservatively. This is a valuable observation, as a lower resolution can be used for the memory and bandwidth-consuming depth buffer. Their hierarchical occlusion maps (HOM) framework finds the prominent occluders as a preprocess. At runtime this set of occluders is used for culling the other objects. Although the results are convincing, the system can't guarantee output sensitivity in dynamic scenes due to the static occluder classification. Our approach is most closely related to this work and to that of Greene and Kass.[5]

Wonka et al.[9] compute the visibility simultaneously for multiple viewpoints by shrinking the occluders sufficiently. Their results for large scenes are impressive, but due to the required preprocessing the approach does not lend itself easily to dynamic environments. Baxter et al. introduce a system for interactive walkthroughs of complex environments.[10] It runs on two SGI Infinite Reality pipelines in parallel and uses HOM as its occlusion culling algorithm. The system supports out-of-core visualization but dynamic updates to the database are not considered.

System overview

Our goal was to build a framework into which multiple visibility determination algorithms could be incorporated. We wanted the system to be portable to a number of different platforms and to be independent of the underlying graphics hardware and rendering API. The implementation should have minimal overhead in simple scenes while being capable of handling extremely large ones—even such that can't be fit into the available memory. The system should be extendable to existing and upcoming hardware culling methods.

We implemented dPVS as a library placed between the application and the rendering layer (see Figure 3). The library consists of approximately 70,000 lines of highly optimized C++ code divided into 150 files and 100 classes. The interface of the library is tight; it has only a dozen classes with a total of 120 public member functions.

The code is reasonably portable—we have successfully built and tested the library on all mainstream CPUs and operating systems including game consoles as well as high-end 64-bit multiprocessor workstations. The initial implementation required work hours adding up to 48 months. We have also developed a profiling and visualization tool that allows interactive analysis of new culling algorithms (see Figure 4).

5 **dPVS has been used over the last three years in various academic research projects, massive multiplayer games, and commercial visualization tools. (Examples of products using our system are courtesy of Sony Online Entertainment/LucasArts Entertainment and Artifact Entertainment.)**

Using the system

Because dPVS is an external module we can use it as a runtime optimizer with any 3D rendering application (see Figure 5). In most cases we only need only a few hundred lines of code to integrate it into an application. The possibility for interactive scene editing is a fundamental requirement for CAD and other modeling software. The system can accomplish fast WYSIWYG model editing. Time-varying simulations—especially military simulations—have a lot of moving entities and often need to modify the environment dynamically. Yet another application area is the remote visualization of massive datasets.

Figure 3 illustrates the use according to the common APP-CULL-DRAW model. When the application loads a scene description, it also submits the geometry data and the related transformation matrices to our system, which then internally computes bounding volumes for the models. We use a mixture of axis-aligned (AABB) and oriented (OBB) bounding boxes.

A hierarchical spatial database is built during the visibility queries. At the end of each query, a bounded amount of time is allocated for refining the database. Objects can be inserted or deleted at any time, and their transformation matrices and associated model data can be updated even during a visibility query.

When the application issues a visibility query for a view frustum, our system returns a list of potentially visible objects and optionally a list of affecting light sources. The application then sends the related rendering commands to the underlying graphics subsystem. Coarse-grained parallelization is possible by running the visibility determination in a dedicated thread on a multiprocessor machine.

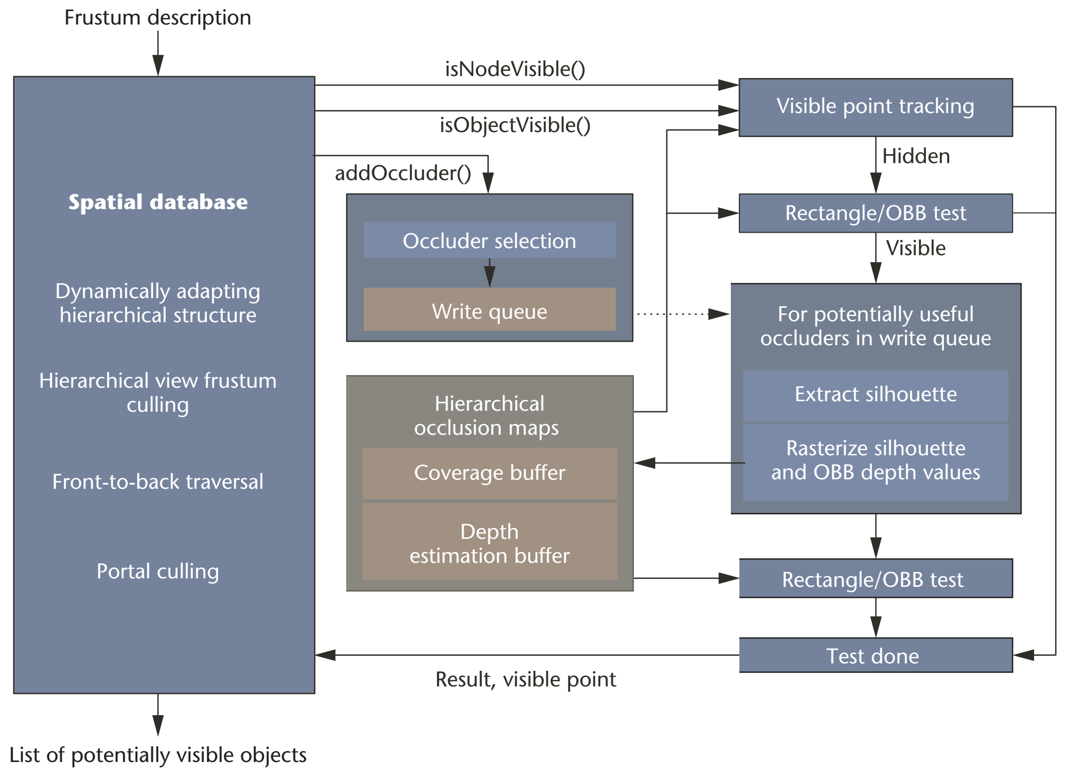
Our system needs to keep a separate copy of the scene geometry. However, for visibility determination, simplified versions of the models can be used—for example, deforming occludees may be represented using conservative bounding volumes. Most physics and collision detection packages take a similar approach. Additionally, only the geometry for the visible and nearly visible parts of the world need to be kept in memory. To further reduce the memory overhead, we support sharing model data between objects. Also, we internally perform a custom lossless compression of the connectivity information and vertex position data achieving on average a 4:1 reduction in memory consumption.

The system has two high-level components: a spatial database and a visibility solver (see Figure 6, next page). When the application provides a view frustum description, the spatial database begins a hierarchical traversal of the scene in an approximate front-to-back order. Visibility of each database node is queried from the visibility solver. If the node is determined to be visible, its children and the contained objects are tested for visibility. Otherwise the entire subhierarchy is skipped. For visible objects we decide whether they should be used as potential occluders and placed into the write queue. When the contribution of the potential occluders is needed by the subsequent occlusion queries, the occluders are fetched from the write queue and rasterized into the occlusion maps.

The visibility solver performs all operations using progressive and lazy computations. For example, objects are selected as occluders at runtime and their rasterization is postponed until their contribution is absolutely needed.

Contributions

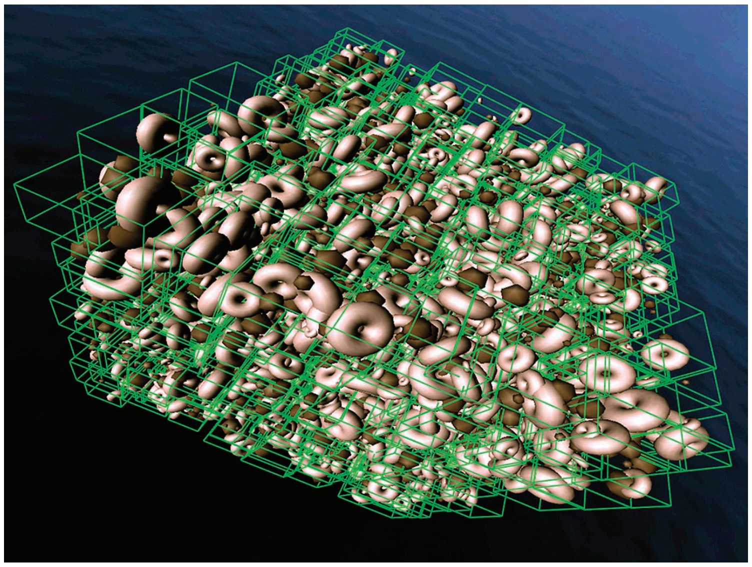
Our main contribution is a framework that unifies a number of culling methods into a system that works efficiently with massive dynamic environments (see Table 1). For example, we can use moving occluders to cull portals and light sources. We also support various precomputed visibility solutions: potentially visible sets (PVS) can easily be used in conjunction with our system, as our database supports fast insertion and removal of objects. The user can also define additional visibility relations between objects. Pregenerated coverage and depth buffers are also accepted as input, as some computer games and modeling packages use them to accelerate rendering. A more detailed discussion of the algorithms and their implementations can be found in the dPVS reference manual.[11] The library and the test scenes are freely available for research projects and academic institutions.



6 The database maintains a hierarchical representation of the scene and provides an approximate front-to-back traversal by using a priority queue. It performs hierarchical view frustum culling whereas the visibility solver manages both conservative and aggressive occlusion culling.

Table 1. We combine into a single framework a number of different culling algorithms that operate seamlessly together.

Culling Method	Handled by
View frustum	Spatial database (hierarchically)
Occlusion	Silhouette rasterization
Contribution	Image-space coverage information
Portal	View frustum culling and scissoring
Potentially visible sets	Fast spatial database updates
Light source	Sweep-and-prune over visible object



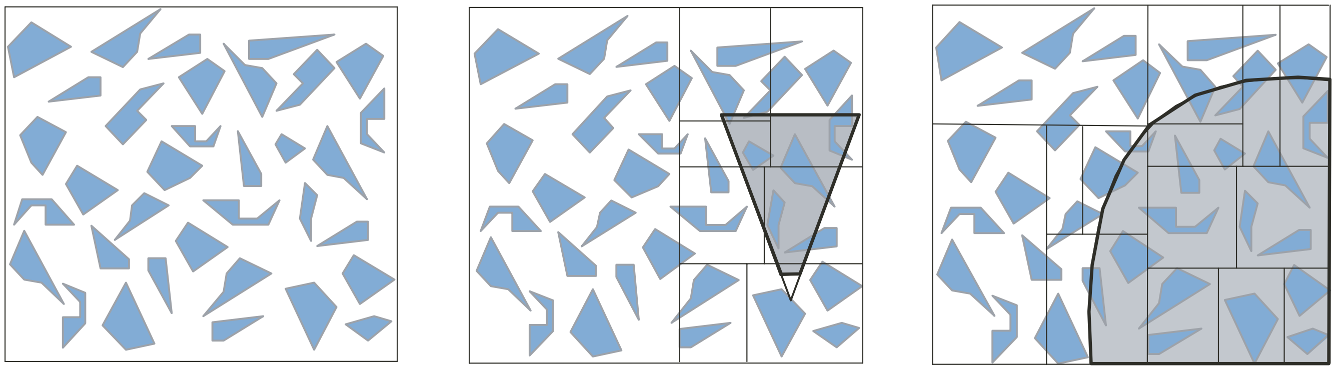
7 The picture shows the partitioning of 5,000 randomly oriented torii and icosahedra. The green boxes are the tight AABBs of the hierarchy's leaf nodes.

Spatial database

To achieve hierarchical traversal and culling of objects, we organize them into a spatial database. Although a variety of methods for representing object hierarchies exist, few meet our needs. The most important requirements for the hierarchy are that it can be built and maintained at runtime, must adapt to dramatic changes in the scene, and can handle all kinds of scenes robustly, even if the input data is degenerate. The database management should be output sensitive and scale to extremely large environments.

We use an axis-aligned BSP tree as our subdivision model (see Figure 7). We permit an arbitrary order for the partitioning planes, for example *x*-*y*-*z*-*z*-*y*, and allow each splitting plane to vary in its location along the axis. Such a choice of hierarchy with adjustable partitioning planes adapts more tightly to the scene geometry than octrees while being orders of magnitude faster to build and maintain than freely oriented BSP trees. Our strategy for selecting partitioning planes for nodes is to minimize the surface areas of the resulting child nodes.[12]

In our implementation, building and updating the hierarchy are closely tied to the database traversal. Initially, we create a single root



8 **Using lazy construction of the spatial database, all objects are initially placed into a single node. The first visibility query (the frustum is shown in gray) refines the hierarchy in visible areas. After the camera has moved around a while, explored parts of the database are refined but the unseen areas stay relatively coarse.**

node that encloses the bounding volumes of all objects in the world. The visibility query then traverses visible nodes within the database and subdivides them, pushing objects downward in the hierarchy. This means that after the first query we have a refined hierarchy for the visible parts of the database (see Figure 8). As we move the camera around, new high-level nodes become visible and are subdivided. This lazy approach for building the database effectively amortizes the creation cost over a long period of time. In many walkthroughs only a small part of the world is explored; the lazy construction ensures that hidden parts of the world are never accessed.

Objects are placed into a node if their bounding volumes intersect the node's bounding box. An object can belong to multiple nodes—we use timestamps for processing each object only once per query—and is pushed down the hierarchy as long as it's smaller than the current node. We use the length of the diagonal of the bounding volumes for size comparison. Nodes are subdivided until they contain less than 10 objects.

Dynamic objects

If an existing object is modified—that is, its transformation matrix is updated or its model data is changed—we don't remove and reinsert the object into the database. Instead, we first find out if the new bounding volume occupies the same nodes as before; in that case no updates are needed. Otherwise we push the object up in the hierarchy until its new bounding volume fits completely inside a node. We tag this node as modified but take no further action until it's reached by a subsequent visibility query. This approach ensures that objects moving in hidden parts of the world require minimal processing.

Furthermore, each object is internally classified as either static or dynamic. Any object that has not been modified for a while is considered static and placed into the hierarchy by using its exact bounding volume. For a dynamic object we construct a temporal bounding volume (TBV)[13] that we expect to enclose all the exact bounding volumes of the object over a certain period of time.

We use TBVs only for placing dynamic objects into the spatial database and for coarse view frustum culling. The tight OBBs of the objects are used for object-level culling tests. Our main motivation for using TBVs is to reduce the amount of database modifications caused by hidden dynamic objects. We construct the TBVs for hidden objects by a simple feedback-based heuristic. Whenever an object moves outside its temporal bounding volume, we grow the TBV. We shrink the TBV whenever an object's TBV is visible but its exact OBB is hidden. After a few visibility queries we end up with a large TBV that stays hidden.

We also use history-based motion prediction for selecting the directions in which the TBVs are grown or shrunk. This additional rule produces much better TBVs for objects with limited motion in certain directions, for example, vehicles, elevators, and people. Some additional heuristics are also needed for keeping the system stable. For example, if an object's position changes by a considerable amount between two visibility queries, we consider it to be teleporting and ignore the motion prediction information. The TBVs of visible dynamic objects are constructed by predicting the path of the object over the duration of one second.

By using temporal bounding volumes combined with lazy updates of hidden parts of the scene we can manage worlds with tens of thousands of moving objects. For example, in our Traffic Jam test scene (see Figure 1) only 0.2 percent of all object updates caused actual modifications to the database and less than 5 percent of the CPU time used by the visibility queries was spent in database maintenance.

Implementation issues

We measure the number of clock cycles taken by each visibility query and use 2.5 percent of the time to update hidden database nodes. These updates might subdivide nodes containing many objects or collapse nodes containing only a few objects. The main purpose for this maintenance work is to avoid sudden slowdowns when previously hidden parts of the world become visible for the first time. The same refining process recomputes partitioning planes for nodes, thus adapting the spatial hierarchy to modifications in the world. We collapse nodes that have been hidden for a long time to reduce the memory requirements of the database. The strict time budget for this additional maintenance work ensures that our visibility queries remain output sensitive. To facilitate out-of-core rendering, the system provides hints for the application so that well-hidden parts of the world can be swapped out of the memory.

When profiling our code, we noticed that on practically all platforms the visibility queries were limited by the memory bandwidth rather than the available CPU power. The majority of our code optimizations are therefore memory and cache related. The most important one is the use of a custom memory manager that provides efficient and highly coherent pool allocators for all of our small data structures, for example, nodes and object instances. The data structure sizes are carefully tuned so that all data is exactly aligned on cache lines. Many parts of the structures are overlaid in memory and infrequently used parts are allocated separately. For example, hidden objects consume less memory than visible ones and static objects less than dynamic ones. Additionally, data structure member order is based on measured memory access patterns.

Sophisticated level-of-detail algorithms might require scene modification during the visibility query. For example, a building might be initially stored in the database as a single object.

When the building becomes visible for the first time, the application might replace it with an object hierarchy consisting of individual rooms, furniture, and other contained objects. Performing this substitution before the visibility query is difficult, as it would require predicting the outcome of the query. On the other hand, if the update is made afterwards, output sensitivity might be lost, as the objects can't be used as occluders during the query. Due to many requests from the users of our system, we added support for database modifications during visibility queries.

Portals

Certain scenes—such as architectural environments—are best represented by using multiple spatial databases connected by portals. Portals are placed into the databases the same way as other objects; they are also culled by the same criteria. For example, dynamic objects can occlude portals. Our portals can have arbitrary shapes and be freely moved or deformed. Each portal also has an associated warp matrix applied when the visibility query proceeds through the portal. This matrix can help produce special effects—for example, mirror reflections for planar surfaces.[14]

Portals participate in the culling process in two ways. First, the view frustum of the camera is shrunk to tightly fit the current portal sequence. This means that the hierarchical view frustum culling automatically rejects objects and database nodes not seen through a portal. Second, we compute a screen-space axis-aligned bounding rectangle for the portal sequence and allow the application to submit it to the rendering hardware as a scissor rectangle. This lets the GPU further limit the rendering of visible objects to the area specified, effectively performing triangle and pixel-level culling.

Optimizing lighting computations

Determining the lights affecting each object can optimize the geometry processing of scenes containing many local light sources. The light attenuation models commonly used in rendering applications allow the determination of the region of influence (ROI) of a light source. In general, point light sources can be modeled using a sphere primitive with a radius equal to the far attenuation range and spot lights can be modeled using cones.

Our system allows tagging objects as ROIs. Once we have determined the visible objects and the visible ROIs, we perform a separate sweep-and-prune pass that finds out the spatially overlapping object and ROI pairs in an average of $O(objects + ROIs)$ time. We supply this overlap information to the application, which can in turn reduce the amount of lighting computations done by the GPU by disabling the noncontributing light sources. The light sources are culled the same way as the other objects, that is, completely hidden ROIs are discarded.

Visibility solver

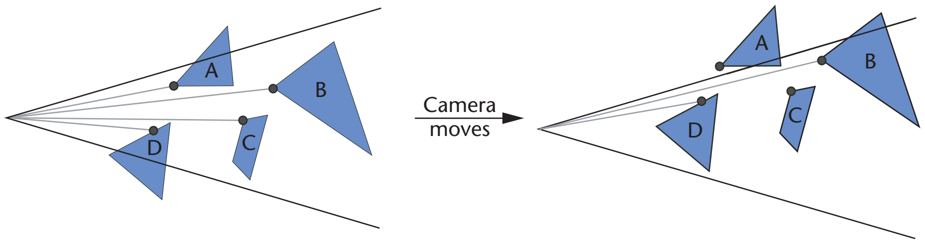
A visibility solver is a high-level component responsible for occlusion culling and contribution culling. Implementations of visibility solvers differ mainly in how occluder selection is done, occlusion queries and occlusion writes are scheduled, and occlusion information is stored.

Our approach is to maintain separate hierarchical buffers for coverage and depth information. Rasterizing silhouettes of meshes using a custom software renderer generates the coverage information. The depth values are conservatively estimated from the objects' bounding volumes. The visibility solver collects information about the usefulness of occluders, allowing feedback-based occluder selection. Our occlusion queries have no fill rate problems even with large screen resolutions as we use hierarchical tests that early exit when the first visible pixel is found. Visible point tracking (VPT) further optimizes occlusion queries of visible objects.

Visible point tracking

Every visible object or database node must have one or more visible points (VPs), see Figure 9. If such an object-space point is known in advance, we can confirm that the object is visible with a single ray cast; this operation can be implemented by comparing the depth value of the projected VP with the corresponding depth buffer value. Alternatively, we can perform a single-pixel hardware occlusion query. In our implementation, we receive VPs as feedback from the occlusion culling system. In the absence of such information, visible point candidates can be generated by randomly selecting a point inside an object's bounding volume. If the point is determined visible, it's cached and retested during the next visibility query. Otherwise, a new random point is selected during the next query.

The main motivation for using VPs is to replace most of the costly occlusion queries by an inexpensive, fixed-cost operation. Many objects and nodes tested during a hierarchical database traversal are visible and thus visible point tracking can optimize their processing. In our test scenes, VPT replaced more than 90 percent of the occlusion queries of visible objects. Visible point tracking can be seen as a unified early-exit test for all kinds of visibility queries.



9 **An object-space visible point has been cached for each object. After the camera is moved, the visibility of objects B and D can be proven with a single 3D ray cast to their cached VPs. The VPs of objects A and C have become hidden and therefore a more involved occlusion query is required.**

Occlusion queries

Due to the large number of occlusion queries, both their execution speed and accuracy play a major role in application performance. Using a lower accuracy in the queries might significantly increase the number of objects reported as visible. It makes sense to search for the optimal break-even point where the processing time lost due to increased accuracy roughly equals the rendering time won due to reduced rendering work. The optimal solution is dependent on the relative performances of the CPU and the GPU.

If the visible point candidate is hidden, the visibility of an object is tested using an axis-aligned rectangle with a single depth value. If the rectangle is visible, a more accurate test is executed using the silhouette of the OBB and interpolated depth values. The OBB test is required because a flat rectangle tends to exaggerate the size of the object depending on the camera orientation. In our test scenarios the use of a more accurate test increases the cost of visibility determination by 10 percent while reducing the number of objects reported as visible by 25 percent.

Occluder selection

Not all visible objects serve as meaningful occluders. For example, a clock on a wall is redundant as the wall acts as an efficient occluder or glasses on a table are too small and because of their transparency will not occlude even if viewed up close. Our occluder selection algorithm attempts to avoid such objects by collecting history information about their effectiveness as occluders. Therefore selecting a subset of the visible objects as occluders generally improves performance. Hardware implementations can equally benefit from occluder selection because postponing the rendering of nonoccluders to the frame's end can reduce the latency of the occlusion queries.

Our selection algorithm is based on cost/benefit estimation. An object is considered a potential occluder if its expected benefits exceed the expected costs. An object's rendering cost is estimated from its triangle count, approximate projection area, and a complexity term. The occlusion cost is the predicted time spent on rasterizing the object into the hierarchical occlusion maps. This estimate is based on the number of silhouette edges and the screen size of the object. Anticipated benefits are the combined rendering costs of the other objects that can be culled if the object is used as an occluder.

We subdivide the screen into 64×64-pixel tiles to estimate the benefits. For each tile, we maintain a list of occluders that have contributed to the occlusion inside the tile during the current query. The spatial database keeps track of the combined rendering costs of the objects contained by the database nodes. When an object or a node is found to be hidden, we consider its rendering cost as gained benefits. The gained benefits are distributed evenly to all tiles intersected by the axis-aligned rectangle of the object.

Inside each intersected tile the benefits are divided evenly to all of the objects that have contributed to the occlusion before the object was tested. Front-to-back traversal guarantees that the objects close to the camera generally receive more benefits than the far objects.

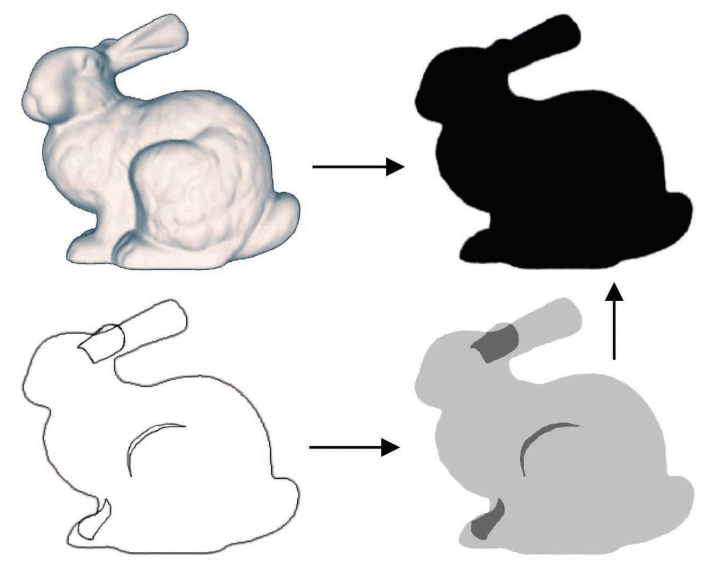
We must use an object as an occluder to validate its expected benefits. Therefore we maintain a short history of the measured benefits for every object. Whether an object is considered as a potential occluder is based on a weighted average of the most recent benefit information and the history data. Objects whose status changes from hidden to visible are always considered to be potential occluders until proven otherwise. Additionally, a small random term avoids over adaptation of the selection process.

As illustrated in Figure 6, we delay the processing of potential occluders up to the point when their contribution is actually needed. The process is conceptually similar to the lazy occlusion grid by Hey et al.[15] This reduces the overhead caused by occlusion culling in cases when the scene has a very limited depth complexity. For example, if a terrain is viewed from above, occlusion culling is next to useless and we can discard 97 to 99 percent of potential occluders.

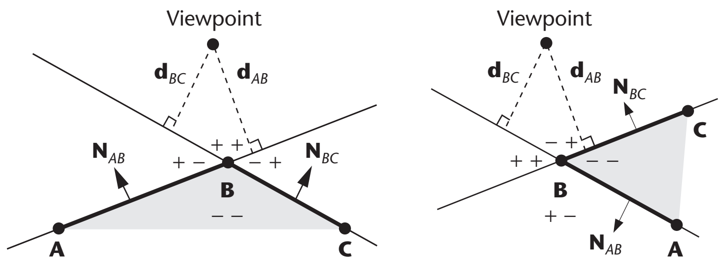
In our test scenes, the occluder selection algorithm used 30 to 80 percent of the visible objects as occluders. However, due to its heuristic nature, the algorithm might fail to block some lines of sight. Therefore we track the overall costs of the visibility queries. If a significant increase is observed between consecutive queries, we abort the database traversal and re-resolve the query using all visible objects as occluders. This usually happens when the camera accidentally penetrates an object and sees through it.

Silhouette rasterization

We introduce a new way for generating coverage information. Instead of rendering an object using triangles, we extract its silhouette in object space and rasterize the silhouette directly (see Figure 10, next page). A high-performance implementation is possible since



10 Instead of scan converting triangles, we generate our coverage information by extracting model silhouettes and rasterizing them directly. This approach lends itself to an extremely fast software implementation. Both methods generate the same coverage information.



11 2D illustration of our temporally coherent silhouette extraction algorithm.

the coverage rasterization needs to produce only a single bit per pixel and can thus be easily parallelized. Also, the average number of silhouette edges in many complex models is roughly the square root of the number of edges of the object.[16] This approach lends itself to an extremely fast software implementation.

Silhouette extraction

The complexity of brute force silhouette extraction is linear to the number of edges in the mesh. When a scene contains complex objects, the extraction operation is too expensive for time-critical use. To overcome this problem, we amortize the extraction work over several frames by using temporal coherence. Although other algorithms with sublinear complexity exist,[16] we chose a different approach due to its simplicity and efficient implementation.

An edge is a silhouette edge if a front- and back-facing triangle share it (see Figure 11). A triangle is front facing if the signed distance from the viewpoint to the plane defined by the triangle and its surface normal is positive. The distance indicates how much the camera can move before the visibility status of the triangle can change. We define the visibility event distance (VED) of an edge as the smaller of the two distances to the defining planes. Figure 11 shows two lines, spanned by edges **AB** and **BC** and their normal vectors $\mathbf{N}_{AB}$ and $\mathbf{N}_{BC}$, that define four subspaces. The signs of the signed distances $\mathbf{d}_{AB}$ and $\mathbf{d}_{BC}$ from the viewpoint to the two lines categorize the subspaces. A viewpoint in the + + subspace means that the vertex **B** is a front-facing interior vertex, − − indicates a back-facing interior vertex, and + − and − + are silhouette vertices. The distances indicate how much the viewpoint can move before potentially entering another subspace. Our algorithm keeps track of these distances, and only updates the status of an edge when the viewpoint has potentially entered another subspace. A 3D generalization of this algorithm is straightforward.

In the caching part of our algorithm we compute the VEDs for all edges of an object from the current camera position. The edges are classified as either silhouette edges or interior edges. We store all silhouette edges and *N* interior edges that have the smallest VED.

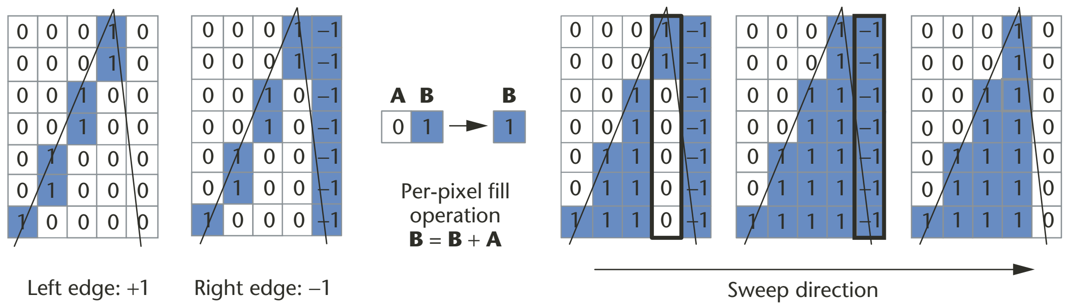
The value *N* has been empirically determined to be twice the number of silhouette edges. The greatest absolute distance among the stored interior edges is the validity radius of the cached representation. As long as the viewpoint stays inside the validity radius, the resulting silhouette can be fully reconstructed from the cached representation.

The extraction pass computes the distance from the cached viewpoint to the current viewpoint. The stored silhouette edges having VEDs less than the viewpoint movement are still parts of the silhouette and are trivially accepted. Interior edges with VEDs larger than the viewpoint movement remain interior edges. The rest of the cached edges might be part of the silhouette and are tested for inclusion. This test requires two dot products per edge. On average we need to perform less than 3*N* dot products to extract a silhouette of *N* edges from the cached representation.

To bound the memory consumption of the silhouettes, we place all extracted silhouette data into a silhouette cache. This cache stores up to 256 Kbytes of silhouette data and has a least-recently-used policy for discarding old data. In our test scenes the cache is rarely congested, as a typical frame requires 20 to 100 Kbytes of silhouette information. We have measured cache-hit ratios ranging from 60 to 100 percent with the average being around 95 percent. The silhouette caching and extraction consume on average 10 percent of the total time spent on visibility determination. Meshes with less than 50 vertices are always brute-force processed and their silhouettes are not cached.

Parallel silhouette rasterization

The silhouette of a mesh always consists of closed loops of edges and can be seen as an arbitrary 2D poly-



12 In the first pass of our parallel rasterization scheme, the silhouette edges are rasterized using the value +1 for left-side edges and −1 for right-side edges. Then, each scan line is filled independently by accumulating the stored values from left to right.

gon. We fill the silhouette as described in Figure 12. A pixel is covered if the corresponding accumulated depth complexity is greater than zero. Since the scan lines are independent of each other, we can process multiple scan lines simultaneously. Our implementation fills 64 scan lines in parallel. The depth complexity indicates the number of overlapping parts in the silhouette. With convex meshes, the depth complexity of a pixel is always zero or one. For arbitrary meshes the maximum depth complexity is not bounded, but rarely exceeds five in practice. We implement the depth complexity counter using 3 bits per pixel. As a result, a depth complexity value of eight is incorrectly interpreted as zero—that is, the pixel is not covered. This simplification might lose some of the existing coverage, but due to its conservativeness it does not lead to visual artifacts.

A 3-bit accumulator can be implemented using roughly 10 logical bitwise operations. Thus a 64-bit processor can simultaneously execute 64 three-bit accumulators using a sequence of 10 integer operations. As a result, our parallel filler has a peak fill rate exceeding 5 billion pixels per second on a 1.1-GHz Intel Pentium 3 using MMX instructions.

Implementation Issues

We have implemented several optimizations to save the limited memory bandwidth. First, we subdivide the screen into 64 × 64 pixel tiles and perform the filling operation using a tile-sized cache. The edges intersecting a tile are rasterized into the cache when the tile is being filled. Three bit planes are needed for the cache and the full-screen buffer stores only 1 bit per pixel.

If the tile is already full or there are no edges intersecting it, the filling operation can be skipped. Secondly, if a block corresponding to an 8 × 8-pixel screen area is fully covered, we store the information into a lower resolution coverage buffer instead. Occlusion queries always use the lower resolution buffer before resorting to the full resolution.

Edges or parts of edges outside the viewport can be ignored with the exception of the left plane of the view frustum. The *x*-coordinates of the parts outside the left plane have to be clamped to the leftmost pixel column to guarantee correct coverage after the execution of the filler.

Because the silhouette rasterizer generates only coverage information, the depth values are conservatively estimated from the back faces of the OBB of a mesh. The depth estimation buffer stores one floating-point depth value for each 8 × 8 block of pixels. To facilitate faster occlusion queries we further organize the depth estimation buffer hierarchically.[5]

Results

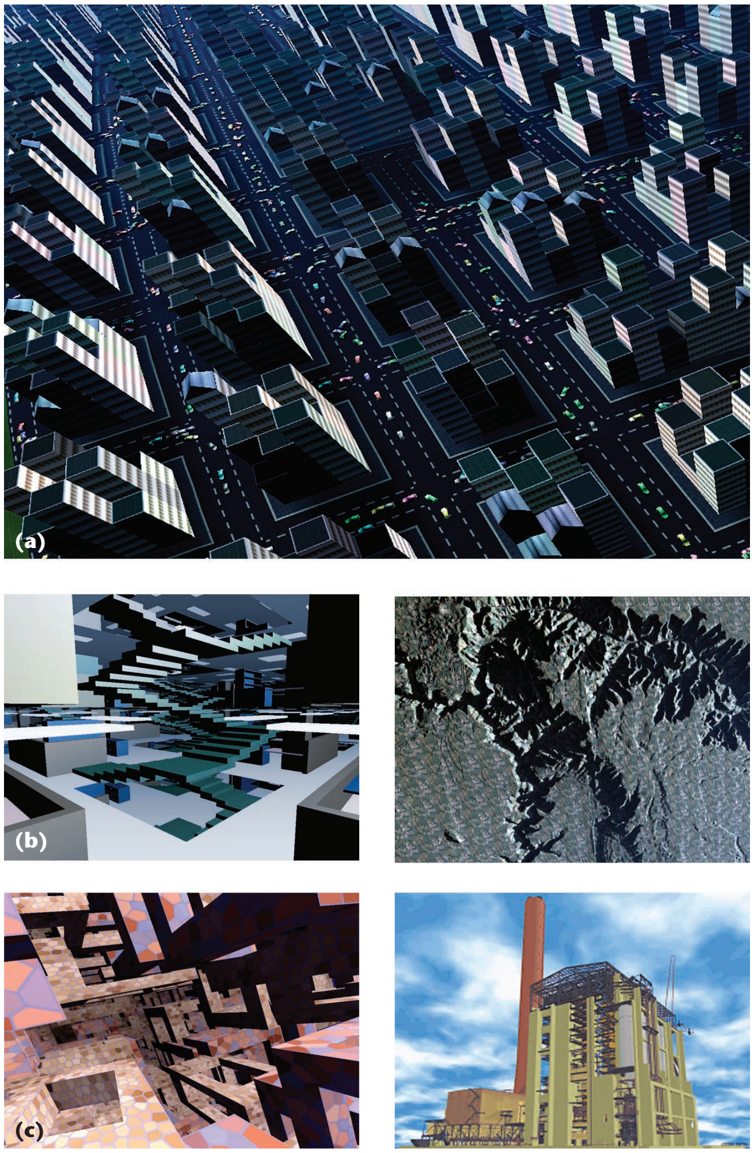
We have measured the speedups gained by using dPVS in a number of complex environments (see Figure 13, next page). We made the performance measurements on a 1.3-GHz AMD Athlon workstation with an Nvidia GeForce4 graphics card. The scenes and demo applications are available online at http://www.hybrid.fi/main/dpvs/download.php. In our tests, we constantly modified the environments by moving objects around. These modifications had no impact on the performance.

Table 2 shows the average number of frames rendered per second, the number of visible objects per frame, and the number of rendered triangles per frame for the scenes with and without occlusion culling, respectively. The rightmost column lists the average time used by the visibility query. Each test run lasted two minutes and was rendered in 1,280 × 1,024 resolution without antialiasing. (The Wonderland scene could not be rendered without occlusion culling.)

CPU usage breakdown of the visibility determination framework for the five test scenes was: occlusion writes (34 percent), visibility tests (24 percent), miscellaneous (11 percent), database traversal (10 percent), silhouette extraction (10 percent), occluder selection (7 percent), and database maintenance (4 percent). All of the queries were conservative—that is, contribution culling was not used although it would significantly increase the performance in the more complex scenes.

The time-varying Traffic Jam simulation has 16,000 buildings and 20,000 moving vehicles. Both the cars and the buildings use level-of-detail rendering. In our test the camera was placed into one of the cars.

The Naked Empire architectural scene consists of 167 million triangles. Walls and most floors are missing and therefore the visibility function is extremely complex. As a result, precomputed solutions would not be feasible. Although we rendered the scene in real time with occlusion culling enabled, we did not manage to render a single frame without it. Therefore we decided to make the comparisons using only eight floors of the building.



13 Complex environments used to test our system. (a) Traffic Jam simulation (courtesy of Hybrid Graphics). (b) Naked Empire (courtesy of Ned Greene and Gavin Miller, Apple Computer) and Grand Canyon (courtesy of the United States Geological Survey with processing by Chad McCabe of the Microsoft Geography Product Unit). (c) Wonderland (courtesy of Hybrid Graphics) and Power Plant (courtesy of the Walkthrough Group at the University of North Carolina at Chapel Hill).

The resulting data set was comprised of 300,000 objects.

The Grand Canyon scene is a 4-million triangle model of the Grand Canyon. Level-of-detail rendering was used for the terrain, as it would be in a flight simulator. If the terrain is viewed from high above, occlusion is virtually nonexistent and therefore occlusion culling is useless. We measured the performance loss due to the overhead of our system in such view positions. However, if the camera is lowered into the canyon, occlusion is plentiful and large parts of the scene can be culled (we named this scene Grand Canyon 2).

Wonderland is a procedural scene containing 2 million objects; each model has 108 triangles and 258 vertices making the total triangle count 216 million. Out-of-core rendering was used for controlling the memory consumption. The database was partially dynamic: 5 percent of the objects were moving around every frame.

The Power Plant model contains 13 million triangles. Our automatic objectification algorithm converted it into 60,000 objects. No level-of-detail rendering was used.

Future work

Our system performs efficiently in various test scenarios, employing several new algorithms. There are a few problematic cases common to almost every visibility determination system up to date, including ours. We can't handle complex deforming occluders efficiently. However, most deforming objects in typical applications tend to be characters and would not be good occluders in any case. The exact representations of displacement maps, parametric surfaces, or highly complex objects such as trees can't be efficiently used as occluders without simplification.

Other limitations include alpha matte objects. For example, a branch of a tree is often rendered using two intersecting polygons that have the shape of the branch encoded into the alpha channel. As our silhouette rasterization scheme does not support textures, we can't use such objects as occluders. A workaround is to generate a simple mesh from the alpha matte.

Hardware occlusion queries have been recently included in Microsoft's Direct3D API. Most parts of our framework and the majority of the algorithms discussed in this article will remain unchanged when hardware occlusion queries are used. Although our system will support hardware queries on selected platforms, we will also continue using the silhouette rasterization scheme due to the large installed base of graphics hardware without occlusion query functionality.

Extending the system to handle shadow volumes in an output-sensitive fashion is an interesting possibility. For example, our silhouette extraction algorithm could be used for fast construction of shadow volumes. Should dependent hardware occlusion queries become available in the future, we could significantly reduce the query latency by providing a point test condition in the

Table 2. Average frame rates achieved with and without occlusion culling.

Test Scene	Frames Per Second	Objects	Triangles	Time (ms)
Traffic Jam	36.1/2.2	110/11.3K	150K/15.2M	7.1
Naked Empire	18.5/1.4	800/60K	25K/2M	38.9
Grand Canyon	65/68	140/140	180K/180K	1.8
Grand Canyon 2	81/40	20/301	25K/155K	8.6
Wonderland	31.1/-	204/-	22.1K/-	19.9
Power Plant	15.8/4.8	750/7.6K	155K/1.2M	25.3

fashion of our VPT algorithm to avoid the execution of a more involved query. ■

Acknowledgments

We thank the R&D team at Hybrid Graphics. We also thank the anonymous reviewers and John Airey, Jeff Grills, Eric Haines, Theodore Jump, Janne Kontkanen, Jaakko Lehtinen, Kari Pulli, Lauri Savioja, Tapio Takala, Matthias Wloka, and Hansong Zhang for their valuable feedback. The development of dPVS was partially financed by Tekes, the National Technology Agency of Finland.

References

1. P. Wonka, *Occlusion Culling for Real-Time Rendering of Urban Environments*, doctoral dissertation, Inst. of Computer Graphics, Vienna Univ. of Technology, 2001.
2. J. Airey, J. Rohlf, and F. Brooks Jr., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," *Computer Graphics (Proc. 1990 Symp. Interactive 3D Graphics)*, vol. 24, no. 2, ACM Press, 1990, pp. 141-150.
3. S. Teller and C. Séquin, "Visibility Preprocessing for Interactive Walkthroughs," *Computer Graphics* (*Proc. ACM Siggraph*), vol. 25, no. 4, 1991, pp. 61-69.
4. D. Cohen-Or et al., "A Survey of Visibility for Walkthrough Applications," *IEEE Trans. Visualization and Computer Graphics*, vol. 9, no. 3, 2003, pp. 412-431.
5. N. Greene and M. Kass, "Hierarchical *Z*-Buffer Visibility," *Proc. ACM Siggraph*, ACM Press, 1993, pp. 231-240.
6. K. Hillesland et al., *Fast and Simple Occlusion Culling Using Hardware-Based Depth Queries*, tech. report TR02-039, Univ. of North Carolina, Chapel Hill, 2002.
7. J.T. Klosowski and C.T. Silva, Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm," *IEEE Trans. Visualization and Computer Graphics*, vol. 7, no. 4, 2001, pp. 365-379.
8. H. Zhang et al., "Visibility Culling Using Hierarchical Occlusion Maps," *Proc. ACM Siggraph*, ACM Press, 1997, pp. 77-88.
9. P. Wonka, M. Wimmer, and F. Sillion, "Instant Visibility," *Computer Graphics Forum*, vol. 20, no. 3, 2001, pp. 411-421.
10. W. Baxter et al., "Gigawalk: Interactive Walkthrough of Complex Environments," *Proc. 13th Eurographics Workshop on Rendering*, Eurographics Assoc., 2002, pp. 203-214.
11. T. Aila and V. Miettinen, *dPVS Reference Manual*, Hybrid Graphics, Sept. 2000.
12. J.D. MacDonald and K.S. Booth, "Heuristics for Ray Tracing Using Space Subdivision," *The Visual Computer*, vol. 6, no. 3, 1990, pp. 153-166.
13. O. Sudarsky and C. Gotsman, "Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality," *Computer Graphics Forum*, vol. 15, no. 3, 1996, pp. 249-258.
14. D. Luebke and C. Georges, "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets," *Proc. 1995 Symp. Interactive Computer Graphics*, ACM Press, pp. 105-106.
15. H. Hey, R. Tobler, and W. Purgathofer, "Real-Time Occlusion Culling with a Lazy Occlusion Grid," *Proc. 12th Eurographics Workshop on Rendering Techniques*, Springer-Verlag, 2001, pp. 217-222.
16. P. Sander et al., "Silhouette Clipping," *Proc. ACM Siggraph*, ACM Press, 2000,pp. 327-334.

Timo Aila *is a PhD student in the Laboratory of Telecommunications Software and Multimedia at the Helsinki University of Technology and he also works at Hybrid Graphics. His research interests include visibility and shadow algorithms, and graphics hardware architectures. Aila has an MSc in computer science from the Helsinki University of Technology.*

Ville Miettinen *is the head of visual research at Hybrid Graphics in Helsinki. He also studies computer science at the University of Helsinki. His research interests include visibility determination, real-time processing of massive dynamic environments, and rendering on mobile phones and other limited-capability devices. He is a member of ACM Siggraph, the International Game Developers Association, and the Khronos Group.*

Readers may contact Timo Aila at the Helsinki Univ. of Technology, P.O. Box 5400, FIN-02015 HUT, Finland; timo@tml.hut.fi.