Fast Line-of-Sight Computations in Complex Environments

David Tuft Brian Salomon Sean Hanlon Dinesh Manocha

Dept. of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
{tuft, salomon, wanderer, dm}@cs.unc.edu

Abstract

We present an algorithm and implementation for solving the line of sight (point-to-point visibility) problem for simulations with many moving entities. This problem arises in military simulations and can bottleneck such systems. We employ the concept of region based visibility to precompute visibility for the environment. The simulation environment is segmented into regions and a visibility map is constructed for each region. The visibility map indicates portions of the environment that are definitely blocked from any point within a region. Once this computation is performed an entity needs only to perform line of sight ray-cast tests for entities in the unblocked part of its region's visibility map. Using our implementation on an existing simulation dataset we achieved a three times speedup.

1. Introduction

Modern simulations of computer generated forces (CGF) in virtual environments are sophisticated and complex simulation systems relying on many simulation steps. One of the most taxing for many simulations is line of sight (LOS) calculations.

An LOS query simply requires determining whether two entities in the environment can see each other with respect to all sources of occlusion. Occlusion may be caused by environment obstacles such as the terrain or man-made structures, atmospheric effects, or other simulation entities. These queries are used extensively in entity AI processing allowing entities to react to other entities within their visible range (or the range of various sensors).

Although a single LOS query is a fairly simple geometric problem LOS queries can account for upwards of 40% [Salomon 2004] of total simulation time. The total number of LOS queries grows as $O(n^2)$ where n is the total number of simulated entities. Thus, as the desired complexity of simulation increases, the fraction of CPU cycles used to compute LOS rises. Moreover, advances in acquisition and modeling technologies have allowed simulation designers to create more complex environments, thus increasing the number of obstacles against which an LOS query must be tested.

LOS is typically solved as a point-to-point visibility problem using ray-casting. A ray is traced through environment and tested against obstacles. Such an algorithm can borrow many of the techniques of raytracing used to generate images in computer graphics such as acceleration data structures (e.g. grid or kd-tree). Raytracing acceleration algorithms often rely on coherence between rays. Raytracing begins with coherent rays exiting the eye into the environment. However, LOS requires tracing many non-coherent rays through the environment making it difficult to leverage more advanced raytracing algorithms.

Region based visibility (RBV) algorithms have been developed in computer graphics as solutions for various problems. RBV algorithms determine the visible portion of the environment from a given environment. Because of the high complexity of this problem, most practical algorithms determine an approximation or overestimation of this set.

We employ RBV to reduce the number of ray-casts that must be performed to resolve simulation LOS queries. The environment is divided into a set of regions. Visibility is precomputed for each region using the obstacles of the environment. We track the region containing each entity. We then use the visibility information of an entity's region to determine whether a ray-cast test must be performed. If a second entity falls outside the visible set of the region then no ray-cast test needs to be performed as it will certainly find an intersection with an environment obstacle.

We have implemented our algorithm and integrated it into OneSAF, a next generation CGF. In test scenarios 70 to 90 percent of queries can be culled using our technique. The average LOS query time in our implementation takes 2.7 microseconds.

The rest of the paper is organized as follows: Section 2 describes an approach to high entity count LOS calculations described in [Salomon 2004] and how this approach can be combined with RBV-based LOS. In Section 3 we describe region based visibility and two algorithms appropriate for the LOS problem. Section 4 discusses how we have applied RBV to the LOS problem. Section 5 provides details about the runtime computations. Initial results are presented in Section 6.

2. Previous LOS Algorithm

The algorithm presented in [Salomon 2004] used graphics rendering hardware to render the LOS ray between two points as a line segment. Computer graphics cards contain built-in hardware that can perform comparisons between the LOS ray and the terrain in order to determine whether or not the ray passes below the terrain. This method culls rays with definite visibility and works best in scenarios in which most of the entities have LOS, such as wide open fields. In this case, many LOS calls can be culled as visible. Non-culled queries are tested using ray-casting.

Like this previous algorithm our LOS algorithm is a culling approach. In face these algorithms are orthogonal and can be used in combination. While the previous algorithm conservatively accepts trivially visible queries, our new algorithm rejects trivially blocked queries using Region Based Visibility.

3. Region Based Visibility

If it can be shown that there are no unblocked rays between region *A* and region *B* of the virtual environment we can be certain that all LOS queries between all entities e_A in region *A* and e_B in region *B* will be blocked. We can decompose the virtual environment into regions and then use RBV to determine which regions are fully blocked from a given region.

Given a subspace of a virtual environment, RBV algorithms compute a visible subset of that environment. RBV algorithms have many applications in computer graphics. One main contribution is that they can increase rendering speeds by culling invisible geometry. RBV has also been used to decrease network traffic for remote renderings. However, it has been shown [Plantinga, 1990] that computing RBV exactly is an $O(n^4)$ problem.

To alleviate the computational complexity, approximate and conservative algorithms have been proposed. Approximate algorithms compute a possibly bounded estimate of the visible set. Conservative algorithms compute a superset of the actual visible set. Our aim is to conservatively cull LOS queries that are blocked by the environment so we explore only conservative algorithms.

Early work in region-based visibility focused on using a single occluder [Cohen-Or et al. 1998; Saona-Vazquez et al. 1999], or approximated multiple occluders [Gotsman et al. 1999]. In order to perform more effective visibility culling, umbras of multiple occluders must be joined to create larger occluders. The concept of merging the umbras of occluders or merging occluders into each other is called occluder fusion. An example of occluder fusion is presented in Figure 1. In this figure, the occluders are represented by the gray bars. The object being viewed is not blocked by any one occluder alone. However, the occlusion umbra cast by both occluders together occludes the object.

Two recent practical RBV algorithms for 3D environments that perform occluder fusion are: ray space factorization ([Leyvand, 2003]) and volumetric ([Schaufler, 2000]). We explore the advantages and disadvantages of these algorithms for LOS acceleration in CGF environments.

These RBV algorithms perform spatial decomposition on the world, such as a grid or hierarchy (e.g. octree or kd-tree). RBV is then computed for every spatial region.

In the next two subsections we describe ray space factorization and volumetric. We describe our implementation in section 4.

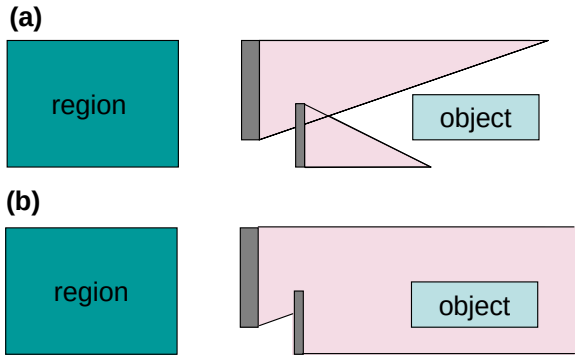

Figure 1: Occluder fusion. In (a) neither of the individual umbras occlude the object. However, in (b) the fused umbra fully occludes the object.

3.1 Ray Space Factorization

We present an overview of ray space factorization and its relation to LOS computations and refer the reader to [Leyvand et al. 2003] for the details of the algorithm.

The algorithm performs visibility computations in ray space. This is the four-dimensional space of rays in the environment. The parameterization algorithm of [Leyvand et al. 2003] separates the horizontal and vertical components of the ray.

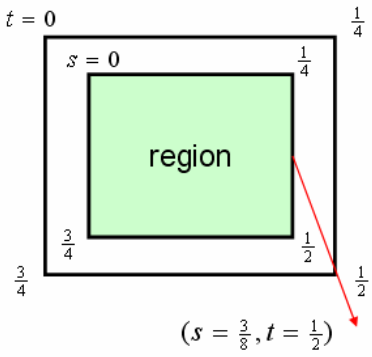

Figure 2: Ray parameterization. In this top view of a region the horizontal component of the ray is parameterized in 2D as (s, t) coordinate where s is the intersection of the perimeter of the region

and t is the intersection with an enlarged perimeter. A sample ray is shown in red.

When considering the rays emanating from a cell, the origin of the ray can be ignored, thereby removing one dimension. The horizontal direction of the ray is parameterized by its intersection with the bounding rectangle (s) of the region and an outer rectangle (t) as shown in Figure 2.

Each parameter-pair (s,t) represents a vertical plane or slice of rays leaving the region. These parameters are used as coordinates to a framebuffer which stores the vertical parameterization of rays blocked by an occluder. Four angle values are used to represent an occluder within a vertical slice. These angles are the angles of the supporting lines (shown in red in Figure 3) and separating lines (blue) with respect to the ground plane.

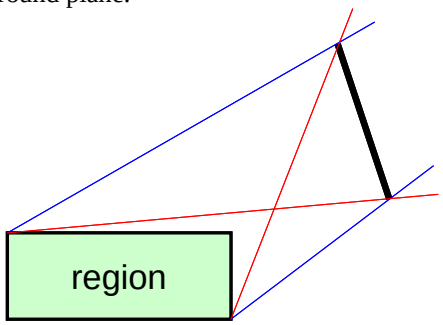

Figure 3: Side view of a region showing a vertical slice and an occluder. The occluder is stored in the frame buffer as four angles that locate its end points. These are the angels of the separating (red) and supporting (blue) lines.

Culling proceeds by processing the environment in a front to back order with respect to the region. The (s,t) footprint of a bounding volume or primitive is computed. Within each pixel of the footprint angle comparisons determine whether the object falls within the umbra of previously encountered occluders. If so, it is occluded. Otherwise if the object is a scene primitive, the angle values are adjusted to include this primitive as an occluder. These computations can be implemented as fragment programs that utilize the GPU occlusion queries to determine whether an object is visible.

The major limitation of this approach is that only one occluder can be stored for each vertical slice in a single 4-component color buffer. A GPU supporting k color buffers can store k occluders in each vertical slice. However, this limitation does not preclude occluder fusion. Occluders that overlap in the vertical direction can be combined as a single virtual occluder that accounts for their fused umbra. This property makes ray space factorization appropriate for environments with less complexity in the vertical direction, such as terrain, architectural, and urban environments.

3.2 Volumetric

This section provides an overview of the algorithm and we refer the reader to [Schaufler 2000] for further detail.

The algorithm operates on volumetric occluders in object space rather than ray space. The volumetric approach requires all occluders to be submitted as 2D manifolds. A special hierarchy is imposed on the environment. The cells of the hierarchy are marked as interior if they are completely inside a 2D manifold occluder, exterior if they are completely outside all occluders and boundary if they are partially inside the occluders. This is done one time before computing visibility for any region.

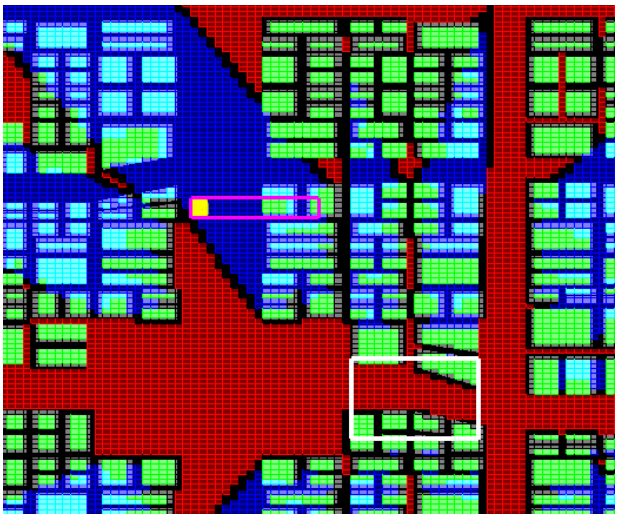

Figure 4: Visualization of volumetric approach. This image shows a region in white, the leaf level of the hierarchy and an occluder. The red cells are exterior cells, the green cells are interior cells, and the blue cells are blocked cells. The yellow cell is current cell in the traversal. It is a blocked cell that is being used as an occluder. The extents have been enlarged to cover neighboring blocked and interior cells to create a larger occluder (magenta).

To calculate occlusion for a region the hierarchy is traversed in a front to back hierarchical manner. As the traversal proceeds each cell visited is marked as either visible or blocked with respect to the region. If the cell being visited is an interior cell its umbra with respect to the region is computed and all cells within that umbra are marked as blocked. The traversal skips over the sub-tree of the current cell if it has previously been marked as blocked.

In [Schaufler 2000] it is shown that blocked cells can be used, in addition to interior cells, as occluders to facilitate occluder fusion. Thus the traversal is modified to compute umbras for interior and blocked cells. Furthermore, rather than using just the cell as an occluder the extents of the cell are enlarged to enclose neighboring cells which are also interior or blocked. For example the yellow cell in Figure 4 is extended to the

magenta rectangle. It is by this occluder extension that the algorithm is able to fuse disjoint occluders.

Once an umbra is calculated it is necessary to determine which cells of the hierarchy fall within it. This is done by traversing the hierarchy in a depth first order. Each cell is tested against the planes defining the umbra. The traversal descends until a max depth is reached or the current cell is completely contained by or outside of the umbra. If the cell is completely contained by the umbra it is marked as blocked.

This algorithm over comes several limitation of ray space factorization. Ray space factorization assumes that the occluder complexity along one axis is simpler than the other two. Also, ray space factorization is subject to robustness problems and is sensitive to the order in which occluding primitives are processed. However a major limitation of the volumetric algorithm is that it requires closed 2D manifold occluders. In general occluder will not line up perfectly with the divisions of the hierarchy. Because of this the interior cells will be a subset of the true occluders.

4. Implementation

After experimenting with rays space factorization and volumetric we chose volumetric because of its simplicity and robustness. We used an octree as our special hierarchy. Adjustments were made to optimize the algorithm for speed and limit the storage space required for the visibility results. In Section 4.1 we describe how the lookup tables were created. Section 4.2 explains how the volumetric pre-process can be run in parallel. Section 4.3 describes the integration into OneSAF.

4.1 Lookup Table Creation

In order to compute visibly for the virtual environment, the virtual environment must be divided up into regions. In order to attain *O (1)* lookups we use a uniform grid of regions. Each region stores a visibility table as a uniform grid at a finer resolution. Figure 5 shows the grid of regions in white. The visibility is superimposed in green and black for the red region. This figure shows that the visibility table is a much finer resolution than the grid of from regions. We call the cells of the fine grid *visibility cells*.

The visibility for a region is computed using an octree as described in Section 3.2. To store the visibility table as a uniform grid, each cell of the grid is checked against the octree. If the cell is completely enclosed by occluded nodes the cell is marked as occluded.

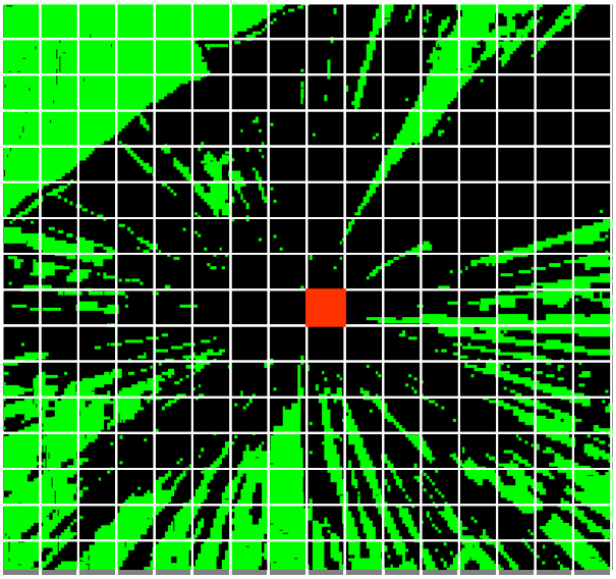

Figure 5: The white grid cells are regions that visibility is computed for. The region in red is the selected region. The areas in green are the finer visible cells from the selected region.

In modern military simulators the majority of entities are ground based. Some entities are aerial entities. This gives three cases: ground-to-ground, aerial –to-ground, and aerial-to-aerial. The aerial-to-aerial query is a fast query and is determined to be visible in most cases. The GPU culling method of [Salomon 2004] will cull most of these queries. We do not use RBV culling for aerial to aerial queries.

In the ground-to-ground case and aerial-to-ground case one of the entities is on the ground. We only store visibility data for visibility cells along the ground. Visibility events occur more frequently due to horizontal movement than vertical. Therefore our grid is much coarser in the vertical direction (e.g. seven in the z direction vs. 64 in x and y). These modifications reduce both the storage space and the computation time for the algorithm.

4.2 Parallelization

Calculating the visibility for each region is extremely parallelizable. In order to generate the visibility lookup tables we used several computers to compute different portions of the uniform grid of regions. We manually partitioned the work among multiple machines. A client server architecture could also be used to manage the distribution of the region computations. All of the clients would initialize their own copy of the octree. Each client would then ask the server for the next region to be computed. The region would be computed and the list of visibility cells would be sent back to the server. The server would create the final table from the data sent back by the clients.

4.3 OneSAF Integration

The OneSAF refers to a composable, next generation CGF that can represent a full range of operations, systems, and control processes from individual combatant and platform to battalion level, with a variable level of fidelity that supports all models and simulation (M&S) domains. LOS calculations can account for 40% of simulation time ([Salomon 2004]). Using the lookup tables mentioned in section 4.1, our Region Based Visibility LOS implementation was integrated into OneSAF. These tables were computed for one area of the world. Section 5 explains the run-time of our algorithm. This run-time was turned into a library and integrated into OneSAF.

5. Run-Time

The run-time section of the algorithm consists of calculating LOS between all entities in the simulation. Figure 6 shows the run-time program flow. Queries are culled first by the RBV lookup tables. Next they are culled by [Salomon 2004]. If both of these culls pass, then the ray-cast computation is performed.

Ground-to-ground entities require two table lookups. First *entity A* and *entity B* are looked up in the uniform grid of regions. Then *entity B* is looked up in the visibility cells for the region *entity A* is in, and *entity A* is looked up in the visibility cells for the region *entity B* is in. If either of the lookups return blocked then the LOS query is culled. Both of these lookups are *O(1)*.

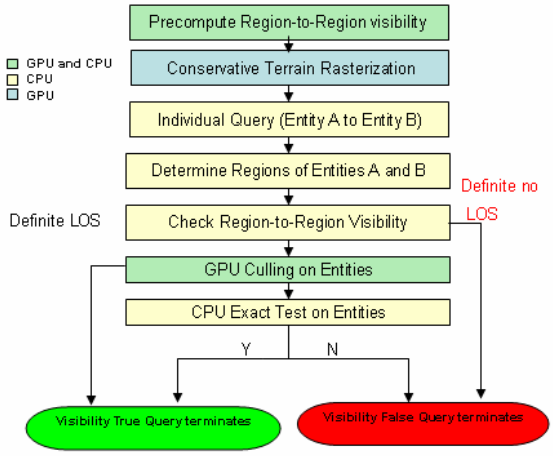

Figure 6: After we determine the region for Entity A and Entity B we can determine their respective regions with the visibility table. If there is possible visibility we can employ the visibility culling done in Salomon's LOS algorithm. Any LOS calls that are not culled by both techniques will default to an exact test on the CPU.

For calculations between aerial units and ground units we use one table lookup. The ground unit is looked up in the visibility cells of the region containing the aerial unit.

6. Results

Our implementation involves an extensive preprocess that produces tables that can be used at runtime to cull LOS queries. We implemented two version of this algorithm. The first version was 2D. We implemented it on a synthetic data set. Figure 7 shows the data set. On this data set our query time went from 5 microseconds to 1 microsecond when the RBV based culling method was turned on.

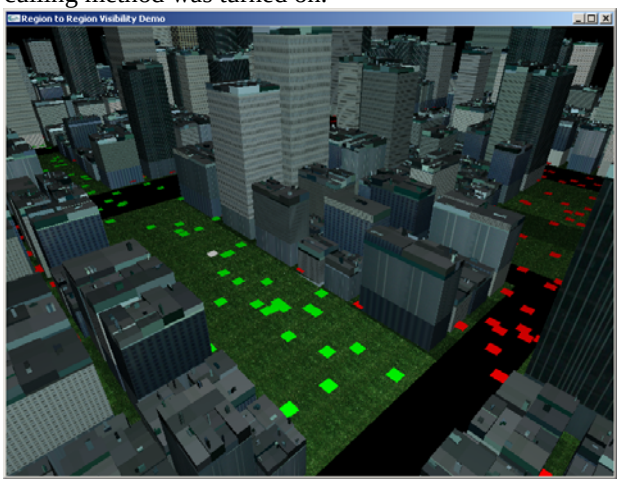

Figure 7: This simulated urban environment was a test simulation for region based visibility as a method for culling line of sight queries. The green squares are visible entities. The red squares are not visible entities.

The scenario that was integrated into OneSAF is now explained. This scenario is an 8 by 8 kilometer region of terrain. This region includes a highly occluded urban region surrounded by a hilly countryside. The urban environment consisted mostly of one and two story buildings.

In order to test this algorithm, entities were randomly distributed across the scenario. Entities moved in a random walk. LOS queries were computed between all pairs of entities. We timed all of the LOS queries with RBV culling on and again with RBV culling off. Without culling the average time of a query is 6.2 microseconds. With culling, the average time drops to 2.7 microseconds. In this scenario an average of 70% of queries were culled.

Results will vary based on the distribution of the entities. If entities are close together they are less likely to be culled than when they are more spread out. Also, the nature of the environment affects the performance of our algorithm. Greater culling can be achieved in densely occluded environments.

References

Salomon, B. Govindaraju, N. Sud, A., Gayle, R, Lin, M., Manocha, D, 2004, Accelerating Line of Sight Computations Using Graphics Processing Units, *24th Army Science Conference Proceedings* 2004.

Plantinga, H., Dyer, C. 1990, *Visibility, Occlusion and the Aspect Graph International* Journal of computer graphics Volume 5, Issue 2, 1990, pp 137-160

Cohen-Or, D., G. Fibich, D. Halperin and E. Zadicario, Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes *Computer Graphics Forum*, 17 (3) 1998, pp 243-253.

Saona-Vazquez, C, Navazo, I., and Brunet, P., *"The Visibility Octree. A Data Structure for 3D Navigation,"* TR LSI-99-22-R, Universitat Politecnica de Catalunya, Spain.

Gotsman, C., Sudarsky, O., and Fayman, J. 1999. *Optimized occlusion culling*. Computer & Graphics 23, 5, 645–654.

Leyvand, T., SORKINE, O., Cohen-Or, D. 2003. *Ray space factorization for from-region visibility*, Proc. of ACM SIGGRAPH.

Schaufler, G., Dorsey, J., Decoret, X., and Sillion, F. X. 2000. *Conservative volumetric visibility with occluder fusion.* Proceedings of ACM SIGGRAPH 2000 (July), 229–238.

Cohen-Or, D., Chrysanthou, Y, Silva, C, Durand, F. 2005, *Survey of Visibility for Walkthrough Applications*, to appear in *IEEE TVCG*.