An Approach For Computing Intervisibility Using Graphical Processing U

2004

Judd Tracy
University of Central Florida

STARS Citation

Tracy, Judd, "An Approach For Computing Intervisibility Using Graphical Processing U" (2004). *Electronic Theses and Dissertations*. 251.
https://stars.library.ucf.edu/etd/251

AN APPROACH FOR COMPUTING INTERVISIBILITY
USING GRAPHICAL PROCESSING UNITS

by

JUDD TRACY
B.S. University of Central Florida, 1997

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering (MSE)
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2004

ABSTRACT

In large scale entity-level military force-on-force simulations it is essential to know when one entity can visibly see another entity. This visibility determination plays an important role in the simulation and can affect the outcome of the simulation. When virtual Computer Generated Forces (CGF) are introduced into the simulation these intervisibilities must now be calculated by the virtual entities on the battlefield. But as the simulation size increases so does the complexity of calculating visibility between entities. This thesis presents an algorithm for performing these visibility calculations using Graphical Processing Units (GPU) instead of the Central Processing Units (CPU) that have been traditionally used in CGF simulations. This algorithm can be distributed across multiple GPUs in a cluster and its scalability exceeds that of CGF-based algorithms. The poor correlations of the two visibility algorithms are demonstrated showing that the GPU algorithm provides a necessary condition for a "Fair Fight" when paired with visual simulations.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AOI   Area of Interest

CORBA   Common Object Request Broker Architecture

CPU   Central Processing Unit

DIS   Distributed Interactive Simulation

GPU   Graphics Processing Unit

LOD   Level of Detail

OSG   Open Scene Graph

OTB   OneSAF Testbed

SAF   Semi-Automated Forces

2D   Two Dimensional

3D   Three Dimensional

CHAPTER ONE: INTRODUCTION

In Distributed Interactive Simulation (DIS) there has been a need over the years to execute simulations with an ever increasing amount of entities. In large scale DIS simulations there can be hundreds of simulators operating together over a network with the number of entities in the tens of thousands. These individual simulators vary and can consist of man-in-the-loop vehicle simulators, dismounted infantry stations and Semi-Automated Forces (SAF) simulations among others. For the most part, each simulator is solely responsible for simulating itself and presenting the information about other entities to the user(s) of the simulator. But due to the high cost of simulator platforms and their limited number for certain types platforms such as helicopter and tanks, a virtual entity or a virtual force must be used to represent the missing simulations. It is the SAF's job to represent these virtual forces and in large scale simulations the SAFs are responsible for representing a majority of the entities.

Since the SAFs represent a virtual force on the battlefield there are a multitude of computations that must be performed which would otherwise be handled by a human operator. Some of these tasks normally performed by humans include entity movement; path planning; weapons firing; and intervisibility calculations. However, when the simulation size increases, or more importantly, when the number of entities increase some of theses calculations become bottlenecks to the performance of the simulation.

One such calculation that exhibits this behavior is the intervisibility calculation. Intervisibility is the process of determining if one entity can see another entity and, if so, how much of that entity is visible. The calculation involves searching the synthetic environment. A synthetic environment is typically comprised of a terrain database and any manmade or natural feature on that database such as buildings or trees. The calculation also involves searching other entities that are operating in the simulation to see if any one entity occludes the visibility of another. The algorithm is also dependent on the type of sensor that is being simulated. Due to sensor variances between types of entities as well as position-related environmental issues for entities within the terrain database the intervisibility algorithm is not commutative, so it cannot be assumed that if one entity sees another that the reciprocal is true. Because the intervisibility algorithm is not commutative the worst case complexity of the algorithm dependent on the number of entities is $O(n^2)$. The algorithm is also dependent on the resolution of the synthetic environment used in the simulation. As the terrain resolution increases there is the corresponding increase in the amount of geometry represented in the terrain, especially in urban, mountainous, or wooded environments. With the increase in geometry represented in the terrain there is a subsequent increase in the search space used to find occluders that directly corresponds to increased search times.

Another issue related to intervisibility in DIS simulations is the correlation of intervisibility between what is calculated in SAF-based simulations and what could be perceived in visual simulations, such as man-in-the-loop simulators. One of the most important determinations when dealing with mixed DIS simulations is whether the

simulation results in a fair fight. Intervisibility plays a primary role in this determination.

Therefore trade-offs must occur in the SAF to determine how accurate the calculation

needs to be versus how much computation is needed to calculate intervisibility. If too

much time is used to calculate intervisibility in the SAF then the man-in-the-loop

simulator may have an advantage in being able to react more quickly to another entity.

Conversely, if the algorithm makes trade offs for speed instead of accuracy then the SAF

might have an unfair advantage and incorrectly determine the visibility of another entity.

In this thesis a new technique for calculating intervisibility is presented using

Graphical Processing Units (GPU). Recent advancements in GPU technology make it

possible to implement the intervisibility algorithm using traditional rendering techniques.

The intervisibility process also fits very well into the paradigm of the GPU. For the most

part intervisibility is a search for intersections with the polygonal data of the terrain,

features and entities of a simulation in three-dimensional (3D) space. On the other hand,

GPUs are built for the sole purpose of efficiently rendering 3D polygonal data into a 2D

plane. In the past, to determine visibility, techniques such as searching framebuffers

were employed. Unfortunately the time it takes to perform a framebuffer search in this

manner does not allow the algorithm to perform efficiently. What is needed is a method

for determining visibility using a 2D projection of a 3D world. With the latest GPUs this

calculation can be performed in hardware using Occlusion Querying extensions.

The other interesting effect of using a visual system to calculate the intervisibility

concerns the correlation issue between SAFs and visual simulations. SAF-based

simulations and visual based simulations have separate needs for representing the

3

synthetic environment in each simulation. For SAF-based simulations there is a need for constructive synthetic environments. The terrain database is more than a collection of polygons that need to be displayed. Information about the terrain such as road segments, contour lines, and soil type also need to be stored as the simulation queries this information in the database. Conversely, visual simulations often only store information about how to display data at hand so that typically there is no need to store information such as road segments or soil types. This information is already encoded into the database by the use of polygons and textures. Due to these differing data requirements between constructive and visual databases, it is often found that the different formats generally do not correlate well.

New standards have emerged such as SEDRIS [21] and EDCS [19] as well as tools such as See-It [23] and Side-by-Side [24] viewer that are helping to address the correlation issues. However, one area that has not been sufficiently addressed is that of intervisibility correlation between visual and constructive representations of the synthetic environment. Some notable observations and examples addressing this particular problem were made by Ashby, et al. [18] and by Wannacott [20]. By using a visual technique for calculating intervisibility it stands to reason that the visibility calculated should represent a good metric to what is perceived.

CHAPTER TWO: LITERATURE REVIEW

Visibility Algorithms

In the 3D visualization world there is extensive research into visibility determination. In the past there have been two typical roles of visibility determination in visual simulations. The first and probably most researched role is occlusion culling. Occlusion culling is the process of determining what geometry in a scene is occluded by other geometry and then culling away the unneeded geometry due to its lack of relevance to the scene. Occlusion culling is often used to increase the frame-rate or decrease the latency of applications therefore rendering large scenes by eliminating large amounts of geometry that would otherwise not be visible. Another use of visibility determination in visual simulation is for the selection of Level-of-Detail (LOD). In this form the visibility information is used to determine how relevant the geometry in question is to the scene. If only a small portion of the geometry is visible then it may suffice to render it at a lower LOD and reduce the geometry that is required to be rendered.

Hierarchical Z-Buffer

In 1993 Green and Kass[4], [5] proposed a hierarchical way of representing the Z-buffer to help accelerate visibility determination. In this method the Z-buffer is represented as a pyramid of buffers that increase in resolution. At the finest level of detail, the buffer represents the same information as a traditional Z-buffer. Each level of the pyramid is created by decreasing the resolution of the previous level. Each depth value in the new buffer represents the furthest value in the previous 2x2 window that maps to the current value.

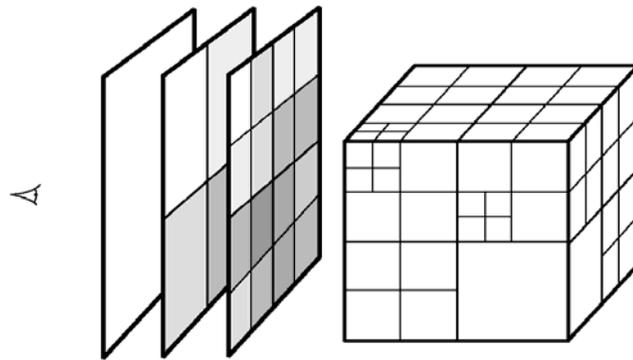Figure 1: Hierarchical Z-Buffer (from Durand 99)

The scene is also stored in a hierarchical data structure. By using an octree for the scene geometry, elements of the scene can quickly be checked for visibility. To determine visibility the nodes from the octree are rendered in a front to back order. Each node in the octree is then subsequently checked for visibility in the hierarchical Z-buffer.

The nodes are checked against each level of the Z-buffer starting with the coarsest level. If a node is found to be occluded at that level, all geometry in that node and children are discarded. If not, the next level of the hierarchy is checked until either the node is discarded or rendered. Due to the front to back ordering of these checks, this procedure allows for efficient occlusion culling since typically only the coarsest levels of the Z-buffer need to be checked.

Unfortunately, this method is not directly applicable to what is needed to calculate intervisibility. The goal of this algorithm is fast elimination of geometry by focusing on whether the object in question is not visible. So if an object is found to be visible, the traditional rendering techniques will be applied. However, the algorithm could be used in conjunction with other algorithms to calculate intervisibility. Aside from a form of this algorithm in ATI's HyperZ, this algorithm is currently not implemented in most graphics hardware and doing this in software would not be feasible.

Stencil Buffer Occlusion Querying

In 98 and 99 Bartz et al. [1], [2] described a technique for performing visibility queries using a virtual occlusion buffer. The scene was partitioned using sloppy n-ary space-partitioning-trees (snSP-trees) to provide a hierarchy that is easily checked for occlusion. The algorithm involved first performing view-frustum culling using the OpenGL selection mode to determine which nodes exist in the frustum. The remaining

nodes' bounding volumes are then rendered into the virtual occlusion buffer which is then mapped to the stencil buffer. Any occluded nodes will subsequently be rejected during the depth test and no record of them will exist in the stencil buffer. Once the bounding volume is rendered the stencil buffer is read and checked to determine visibility. Instead of reading the entire buffer at one time, Bartz et al. addressed the lack of speed by sampling the stencil buffer while reading back to the stencil buffer from the hardware. This process helps alleviate some of the issues with reading buffers but introduces the possibility to misidentify visibility.

Hardware Occlusion Querying

In 93 Green et al. [6] discussed a hardware implementation using the Kubota Pacific Titan 3000 workstation with a Denali GB graphics subsystem. It involved using a graphics library to determine whether any pixels in a set of polygons were visible using the current z-buffer. They concluded that the cost of the operation was too high to be effectively used.

In 98 Bartz et al. [3] described a method for extending the OpenGL pipeline to allow for hardware-based occlusion querying. They describe creating a new mode of operation similar to the OpenGL selection mode where tests are performed without affecting the contents of the framebuffer. Their methods provide a wealth of information that can be useful in occlusion culling, but has not been implemented in hardware.

8

There are currently two forms of hardware-based occlusion queries that have been implemented. The first is the HP_occlusion_query [11] extension. It provides a way to query the visibility of a set of geometry that is rendered. It returns a true or false answer to whether any fragment has passed the depth-test. The HP query does not allow multiple queries to be performed at the same time and causes the rendering pipeline to be stalled while the results are being returned. The other extension currently implemented on some hardware is the NV_occlusion_query [12]. It solved the two major failures of the HP extension in that it returns the total number of pixels that pass the depth test and allows multiple occlusion queries to be pending at the same time.

In 2002 Micikevicius [9] described a technique for determining the Level-of-Detail (LOD) at which to render trees in a forest walk-through simulation. The simulation uses the NV_occlusion_query extension to calculate the visibility of a tree. The LOD used to render a tree is then selected based on the percent visibility and the projected size of the tree in pixels. In 2003 Martens [8] presented a method of occlusion culling using the NV_occlusion_query to determine visibility of bounding volumes of the scene hierarchy. And also in 2003 Govindaraju et al. [10] presented a similar method of using the same extension for occlusion culling. Their algorithm includes a stage in which known occluders are initially rendered to perform occluder fusion in image-space. This fusion of occluders is then used to perform the visibility determination using the hardware extension.

CHAPTER THREE: METHODOLOGY

GPU Intervisibility

The GPU Intervisibility algorithm presented here relies on the recent advances in 3D video hardware. Current hardware released by NVidia and ATI have a new OpenGL extension called the NV_Occlusion_Query. This extension was originally created to determine if a grouping of geometry is occluded. This algorithm takes advantage of the results and calculates a visibility metric. In this algorithm intervisibility is defined as a normalized ratio of the number of pixels actually rendered versus the number of pixels possibly rendered in the range between zero (not visible) and one (completely visible).

There are specific requirements that must be maintained for this algorithm to operate properly. First, the synthetic environment is rendered since there is no interest in calculating the visibility on objects in the terrain. And the cost for sorting the entire terrain to render it from front to back would be prohibitive.

Second, all entities must be rendered in a front to back order. This order is required so that occlusion of distant entities from closer entities occurs. All entities must also be rendered with back face culling enabled. Without back face culling, the pixel counts generated by the occlusion query may be significantly different from what is

actually rendered do to an indeterminate order of rendering causing possibly occluded fragments to be counted.

The OpenGL alpha function is used to handle transparency in the scene. This function controls how OpenGL renders transparent fragments. In this algorithm the alpha function is used to not render transparent fragments above a certain alpha value. This causes the scene to be rendered differently from traditional visual simulations where the alpha values are blended while still providing a compromise allowing the algorithm to work with transparency.

Given all of the requirements defined above here is an overview of operation of the algorithm. All entities are sorted based on distances calculated between each pair of entities. Any geometry not in the view frustum or outside of a sensor's Area of Interest (AOI) is culled away. The synthetic environment is rendered with an alpha function set to not render transparent fragments. Each entity is rendered twice in a front to back order with back-faced culling enabled. The first time the entity is rendered depth testing, depth writing, and color mask are disabled. The rendering is wrapped with occlusion querying start/stop functions. This rendering will give a baseline of how many pixels would be rendered if no geometry occludes the entity. The second rendering is performed with the depth testing, depth writing, and color mask enabled and is also wrapped with occlusion querying start/stop functions. This rendering provides the actual amount of pixels rendered with occlusion. Given these two calculations a ratio of visibility can be calculated.

To implement the intervisibility algorithm a visualization framework is used. The framework chosen is Open Scene Graph (OSG) [13]. OSG is an open source, high performance, 3D graphics toolkit written in C++ and OpenGL that is used to create applications in fields such as visual simulation, virtual reality, scientific visualization, and modeling. This framework provides a scene graph that is highly modular, infinitely customizable and extremely fast.

OSG was chosen as the framework for the intervisibilty algorithm for three primary reasons: a complete feature set, extensibility, and speed. OSG's toolkit provides a strong feature set for rendering geometry and has many modules available to perform tasks that are not part of the core system. It is fairly simple to create an application, input geometry and have that geometry rendered in only a few minutes. Another feature that OSG provides is an extensive set of database file loaders and images such as OpenFlight and TerraPage.

Of primary importance with respect to this project is OSG's extensibility. Due to the nature of the algorithm there is a need to control many rendering system factors as well as to be able to manipulate OpenGL's underlying graphics system. OSG provides an extensive callback system which allows manipulation of how the scene graph is traversed; how the cameras are updated; and the order in which the scene is rendered. Due to the object-oriented design of the system elements, they can be reused or extended and also be transparently injected back into the framework of OSG. The final reason OSG was chosen is the raw speed of its rendering process.

The intervisibility algorithm uses the OSG framework. This framework rendering is performed using three discrete stages. The first stage is the Update Stage. In this stage the scene graph is free to be updated. Any modification that must be made such as updating the position of an object or changing a color must be completed. The second stage in the rendering process is the Cull Stage. In this stage the scene graph is traversed and all geometry is culled against the view frustum as well as any occluders or clipping planes that might exist in the scene graph. During the Cull Stage all geometry is placed into bins and the geometry is stored based on common states to ensure proper rendering. The final stage in the rendering pipeline is the Draw Stage. The Draw Stage takes the geometry from the bins and renders the bins in order providing an effective way to render transparent geometry.

With this algorithm the issue of transparent or translucent objects needs to be addressed. Transparency or translucency exists in a scene when a polygon or a texture has an alpha component. The alpha component represents a degree of translucency that is used to blend color values of other geometry. It is also used with textures to build complex objects by using simple polygonal models along with textures that have transparent sections so that when the polygonal model is rendered it looks like complex geometry. This is often used in visual simulation to model objects such as shrubbery and trees since using geometry for rendering each leaf may prove too costly. The problem with transparency or more appropriately, alpha values, is that when alpha values are rendered the depth buffer is still updated even though there might not be any visible pixels. To address this issue the OpenGL Alpha Test [22] operations are used. The

Alpha Test lets the user specify an alpha function that controls how fragments are rendered based on the alpha value. The alpha function lets the user specify a function to apply and a reference value to check to determine if a fragment will be rendered. Using this function a threshold can be set that will cause alpha values above this threshold not to be rendered. This in effect creates holes in the scene where alpha values are in excess of the threshold and allows geometry that would otherwise be occluded to be rendered.

Update Stage

During the Update Stage the intervisibility algorithm performs two main tasks. The first task of the update stage is to calculate distances between entities. For every entity in the simulation a distance to every other entity in the simulation is also stored for later use in the cull stage. This information is necessary to ensure proper visibility determination as the entities are rendered in a front to back order. The second task of the update stage is to update the locations of the cameras that represent the sensors of the entities. The sensors of the entities are tied to geometry in the model that represents the sensors. If there are updates to the position or orientation of either the entity or the sensor those updates are reflected in the absolute position or orientation of the geometry representing the sensor.

```
UpdateStage ()

    Foreach entity in entity_list:

        UpdatePositionOrientation (entity)

        Foreach target_entity in entity_list:

            entity->distances = CalculateDistance (entity, target_entity)

            // Sort map of entities to render_bin_indexes based on distance.

            entity->renderbin_index_map == BuildEntityToRenderBinIndexMap (entity, distances)

    Foreach sensor in entity:

        UpdateOrientation (sensor)

        CalculateOpenGLViewSettings (sensor)
```

Figure 2: Pseudo Code for the Update Stage

Cull Stage

The primary operation of the cull stage is the determination of which geometry

will be sent to the graphics card and what geometry will be thrown away. The scene

graph is traversed and each node is tested against the view frustum to determine if

rendering is required. If the geometry is determined to be visible, it is placed in a render

bin appropriate for its type of geometry. For this algorithm the cull stage is extended in

order to accommodate the algorithm's requirements.

15

The distance information collected in the update phase is converted into render bin indexes that are mapped to the entities as they are rendered in a front to back order. When the render bins are built during the cull traversal they will be placed in front to back order based on the entities contained in the bins. While the map of render bin indexes is being generated, the distance to the entity is being checked against the Area of Interest (AOI) of the sensor to see if it is outside of the AOI. If the distance found is outside the AOI, there is no need to render the map and therefore no mapping will be generated.

Every node in the hierarchy is checked to determine potential visibility during the cull traversal of the scene graph. The default setting is used for the terrain and all static features on the terrain, but a slightly modified version is used for the traversal of the entities in the scene graph. During the traversal of the entities, the cull stage detects an occlusion query node and performs extra operations. The view frustum checks are first performed to quickly determine if the entity should be rendered. Then the system attempts to find a render bin index mapped to the entity. If a render bin index is found a new render bin is created with that index and is designated the current render bin. The cull stage then traverses all children and subsequently places all geometry belonging to the mapped entity in this new render bin. Finally a callback is placed on the render bin so that during the draw stage the rendering can be modified.

```
CullStage ()

    Foreach sensor in scenegraph:

        Foreach node in scenegraph:

            If node is frustum culled:

                Next      //Node is culled

            If node is type entity:

                If entity_distance > area_of_interest:

                    Next      //Out of range and does not need to be rendered

                render_bin = CreateRenderBin (entity, entity_renderbin_index_map)

                AddElement (render_bin, render_bin_list)
```

Figure 3: Psuedo Code for the Cull Stage

Draw Stage

The draw stage is responsible for drawing all geometry not culled out during the

cull stage. All render bins generated during the cull stage are now rendered in order. The

cull stage places all geometry for the terrain in lower-indexed rendering bins so that the

terrain will be rendered first. Then sequentially, each entity is rendered in a front to back

order as the render bins were built. When the draw stage attempts to perform the actual

rendering of the geometry the callback placed on the render bin is executed. Inside this

17

callback the entity is rendered twice. The first time the entity is rendered the depth and color buffer are disabled and the rendering is wrapped between occlusion query start/stop calls. This rendering is used to generate a baseline value of the number of pixels the entity would have rendered if there was nothing occluding its view.

Then the entity is rendered again with the depth and color buffers enabled and also wrapped between occlusion query start/stop calls. This rendering is used to determine the actual number of pixels rendered with terrain and other occluding entities. At the end of the draw phase all pixel counts are collected and ratios of visibility are calculated.

```
DrawStage ()

    Foreach sensor in scenegraph:

        Foreach render_bin in render_bin_list:

            If render_bin contains entity:

                occlusion_query = GetOcclusionQuery (occlusion_query_list)

                DisableDepthBufferWritesAndTests ()

                DisableColorBufferWrites ()

                StartOcclusionQuery (occlusion_query->non_occluded)

                DrawRenderBin (render_bin)

                StopOcclusionQuery (occlusion_query->non_occluded)

                EnableDepthBufferWritesAndTests ()

                EnableColorBufferWrites ()

                StartOcclusionQuery (occlusion_query->occluded)

                DrawRenderBin (render_bin)

                StopOccusionQuery (occlusion_query->occluded)

            Else:

                DrawRenderBin ()

        Foreach occlusion_query in occlusion_query_list:

            pixels_non_occluded = GetQueryResults (occlusion_query->non_occluded)

            pixels_occluded = GetQueryResults (occlusion_query->occluded)

            visibility = pixels_occluded / pixels_not_occluded
```

Figure 4: Pseudo Code for the Draw Stage

Screen Partitioning

When rendering in OpenGL, geometry is sent to the GPU and is processed in parallel with processing on a CPU. Certain operations in OpenGL cause the CPU to wait for the processing on the GPU to finish in order to proceed with processing on the CPU. One such operation is the swap_buffers [22] function which forces the geometry to be rendered and displayed on the screen. This operation happens every time a frame is rendered in OpenGL and creates a contention point in the application. Because of this issue it might be advantageous to render multiple sensors per frame.

The approach taken in this thesis is to partition the screen using an equal split kd-tree with a maximum depth setting. When there are more sensors then are allowed by the maximum depth of the kd-tree another screen is created. The screens are then filled in a breadth first order to maximize the sizes of the rendering areas in each screen. By rendering multiple sensors per screen the issue mentioned above can be minimized.

Distributed Algorithm

The distributed algorithm is an extension of the GPU Intervisibility algorithm to execute across multiple computers on a network. Because the algorithm is embarrassingly parallel no modifications had to be made to the underlying algorithm. The only things added were a way to distribute the load, a way to control the simulation, and a way to collect the results. CORBA [14] was used to implement the networking.

20

Instead of having to focus on the underlying networking code, CORBA was chosen because it provides an easy framework to generate network interfaces at a high level that are simple to work with.

The entity distribution is a simple iterative approach designed to evenly distribute the number of sensors across all nodes in the simulation. All nodes in the simulation must first register with the control node. Once all nodes are registered the control node then continuously iterates over each node and assigns it one entities' sensor until all sensors have been assigned to the nodes. Once all sensors have been assigned and the control node sets the database that is being used the simulation can begin.

The control node then signals all nodes that the simulation should begin. Each node performs the GPU Intervisibility algorithm on each sensor that has been assigned to it and the visibilities are collected. A list of visible entities is generated and transmitted back to the control nodes. It is important to note that only visible entities are reported to the control node. If an entity is not in the list, it is assumed to not be visible. Once the control node receives information back from each node a signal is sent out to start another iteration.

Correlation between GPU Intervisibility and Constructive Intervisibility

For correlation between the GPU Intervisibility algorithm and the algorithms used in SAFs, the LibCTDB algorithm was chosen. LibCTDB is a library that provides intervisibility calculations among other things to common SAFs such as OneSAF and

ModSAF. The intervisibility algorithm provided by LibCTDB is quite complex and is described in [16]. The algorithm has many modes of operation but the more commonly used case of the point-to-point test or more appropriately point-to-cone test is going to be used.

In the point-to-cone test the following data points are required for operations: 1) The X, Y, Z in absolute coordinates of the sensor, 2) The X, Y location in absolute coordinates of the entity being looked at, 3) The upper and lower Z values of the entity Z_h, Z_l, and 4) The width of the 2D projection of the entity from the sensor. With these values two constructs are setup for calculating intervisibility. First a triangle is generated using the position of the sensor and the two points generated by bisecting the width of the entity at Z_h and Z_l. This is called the Intersection Triangle as seen in Figure 5. Second a visibility rectangle is generated using the width of the entity and the Z_h and Z_l values that is perpendicular to the intersection triangle. Using these two constructs the algorithm can calculate a visibility for the query. The algorithm calculates two values used to calculate visibility: visible area and linear transmittance. Visible area represents the aggregate fraction of the visibility rectangle that can be seen from the sensor ranging from 0.0 to 1.0. Linear transmittance represents the aggregate fraction of light that is seen from the visibility rectangle and ranges from 0.0 to 1.0.
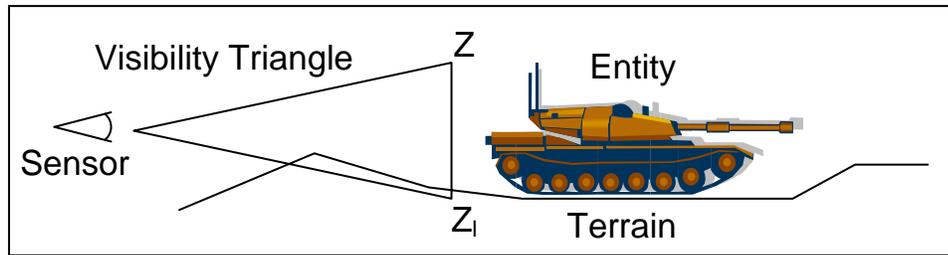
Figure 5: Intersection Triangle

The first step in calculating visibility in LibCTDB is performing intersection testing of the terrain with the intersection triangle. The terrain is iterated and each polygon is tested for intersection with the triangle. If an intersection with the triangle occurs the lower point of the triangle is adjusted to the intersection point and intersection testing continues. Once all geometry in the terrain is tested the percentage of the height of the new triangle to the original triangle is calculated and this is considered the visible area.

The second step in calculating visibility is determining the linear transmittance. This operation can be performed using either statistical measures or rasterization techniques but since rasterization is hardly ever used only the statistical method will be discussed. In the statistical method features of the database are tested for overlap with the visibility rectangle. If overlap occurs the linear transmittance is modified by multiplying itself with the percentage of overlap.

Once these two operations are performed for an entity the visibility is calculated using the following equation:

23

$$Visibility = VisibleArea \cdot LinearTransmittance$$

Given the algorithm described above there some issues regarding how it calculates intervisibility and how the GPU algorithm works. First, the LibCTDB algorithm only considers polygons of the terrain that intersect with the intersection triangle. It is possible for polygons to occlude the sides of the entities without significantly intersecting the triangle. While this situation will probably not occur very often it does exists and will properly be accounted for in the GPU algorithm.

The second issue deals with the statistical calculating of the linear transmittance. Features such as trees and buildings are checked to see what percentage of visibility rectangle they occluded and are statistically factored into the linear transmittance. The algorithm takes no account of if some other features were already occluding the same area and therefore can affect the linear transmittance even though it does not provide any further occlusion.

Finally there will be significant differences between visibilities through tree lines using the two different algorithms. In LibCTDB this visibility is calculated through statistical methods using some distribution of tress. But tree lines are represented differently in visual simulations and hence in the GPU algorithm. In visual simulations tree lines are often represented by large polygonal areas that use textures to represent the trees. These textures are images of trees lined up next to each other with alpha transparency used around the edge at the top to simulate the tree line. However there is

often no way to see entities on the other side of a tree line because of the way tree lines

are represented in visual simulation.

CHAPTER FOUR: FINDINGS

GPU Intervisibility

To test the GPU intervisibility algorithm a scenario with 300 entities was generated using correlated versions of the Ft. Polk Shugart-Gordon Database. The urban features were removed, since the early version Multiple Elevation Surface (MES) urban structures is not compatible with the OTB Version 1 that was used for comparative testing. The sensor view frustum parameters were set at 0.619406 radians horizontal and 0.508736 radians vertical, with no far clipping plane applied. The results for the visual algorithm were generated using a maximum 1600 x 1200 screen resolution and the algorithm described. The scenario was generated manually so that a significant number of entities are expected to be within view of each other. Seven different configurations of the scenario were executed using entity counts of: 25, 50, 100, 150, 200, 227 and 300. Each scenario was run 5 times with 100 iterations per run resulting in 500 sample points and the results shown are the averages of the samples.

In this scenario view, the sensors are blue, and it is evident the area is densely populated with individual trees and has significant variations in the terrain elevation.

Intervisibility between all entities was compared, using an AOI in the form of 1km radius.
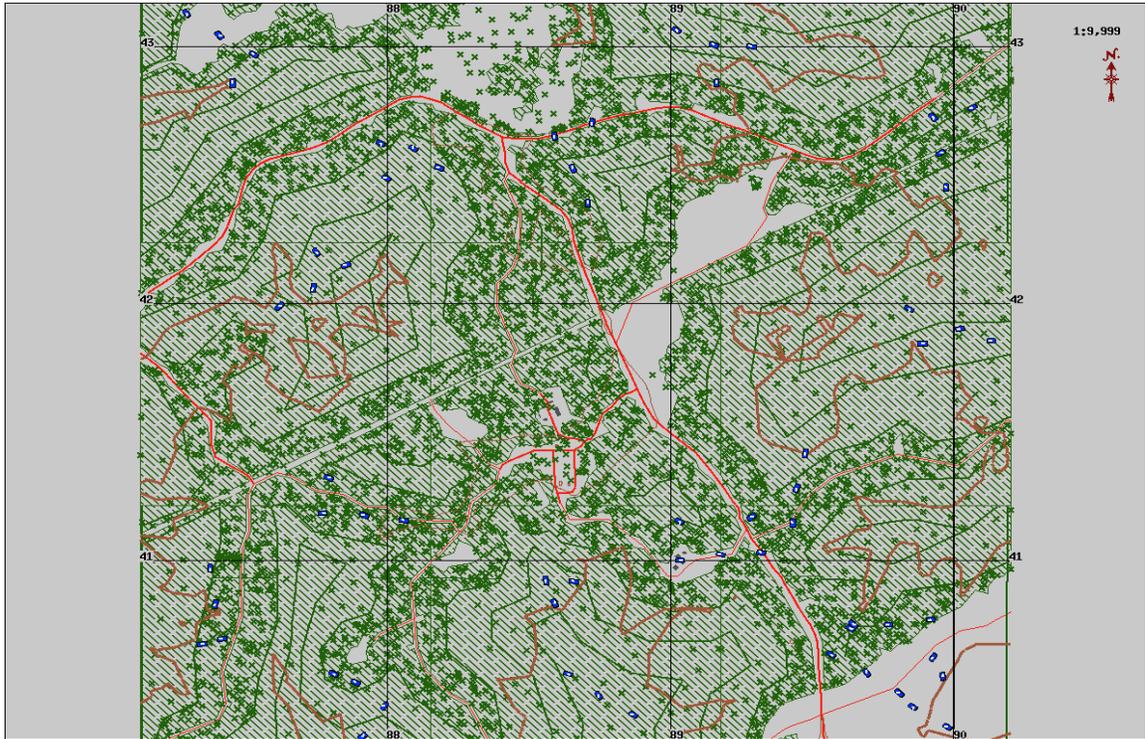

Figure 6: Scenario used for initial experiment.

Three screen partitioning layouts were also tested to see how they would effect rendering times. The first screen partitioning layout used was the simple case of a single sensor rendered per screen. The second screen partitioning uses a kd-tree with a maximum depth of two which allows a maximum of four sensors to be rendered per screen. Finally the third screen partitioning uses a kd-tree with a maximum depth of four which allows a maximum of sixteen sensors to be rendered per screen.

The results generated were obtained using the hardware described in Table 1. In the cases where multiple GPUs were used, identical hardware was provided for each GPU.

Table 1: Compute Node specification

Processor:	Dual AMD Athlon MP 1500+ 1.33Ghz
Memory:	512MB DDR 2700
GPU:	NVIDIA GeForce FX 5900 256MB
Network:	Fast Ethernet 100Tx

Intervisibility between all entities was compared using an AOI in the form of a 1km radius. Figures [8, 9, 10] show the number of entities rendered per second versus the number of sensors in the system using the three screen partitioning algorithms above. The figures show a fairly linear increase in the entities rendered per second versus the number of sensors and shows no asymptotic trend.
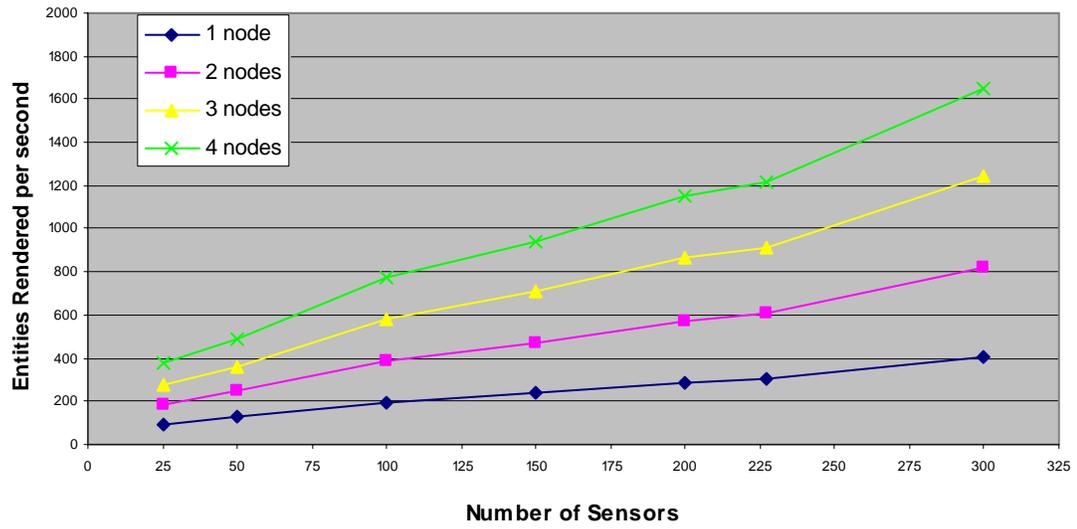
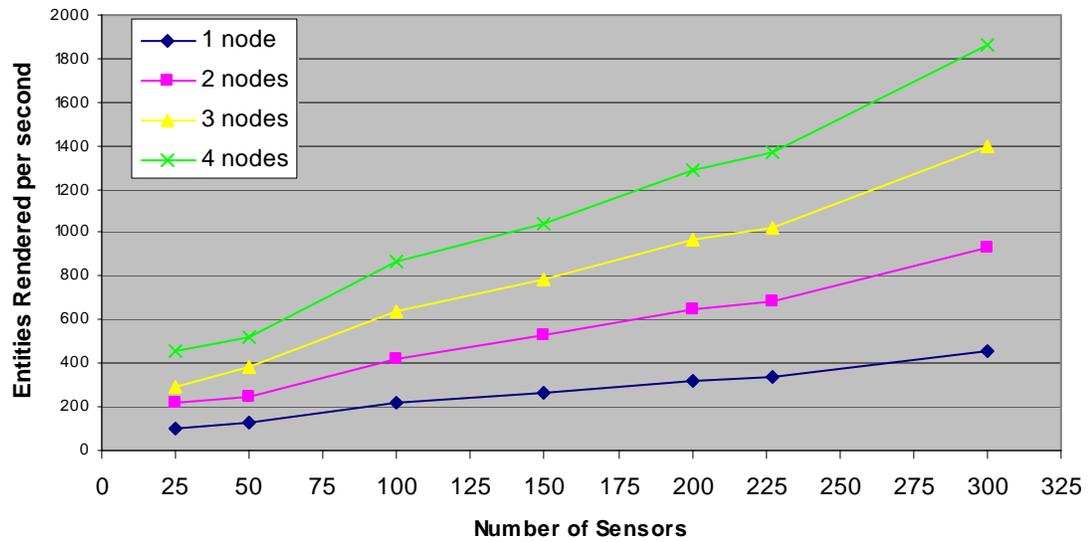Figure 7: GPU Intervisibility results with a single sensor per screen.

Figure 8: GPU Intervisibility results with a maximum of four sensors per screen.
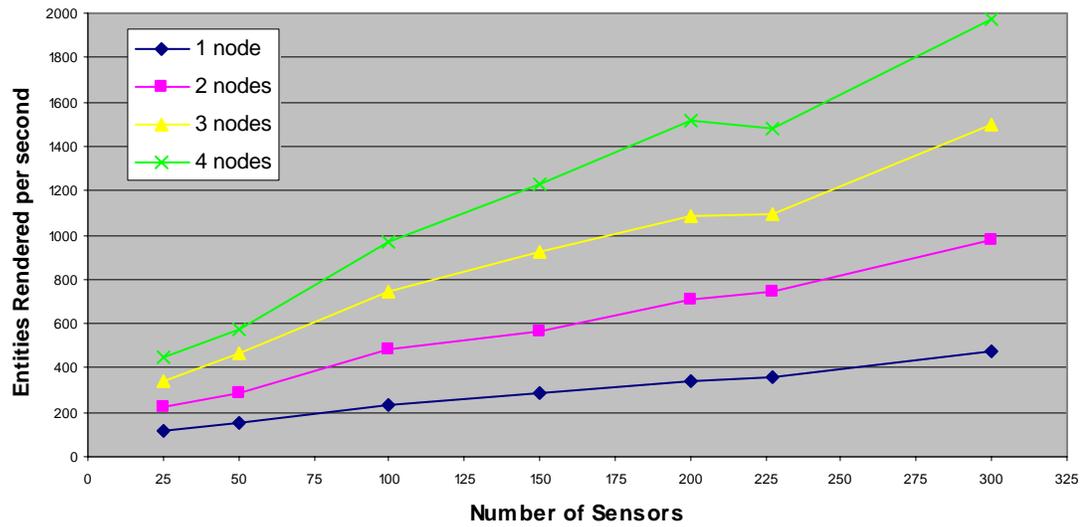
Figure 9: GPU Intervisibility results with a maximum of 16 sensors per screen.

Figure 10 shows the average speedup of the GPU intervisibility algorithm as the number of nodes increases for the three different screen partitioning algorithms. As one can see the speed up is linear as the number of nodes increases. One can also see that the differing screen partitioning algorithms are not providing any real benefit in rendering performance.
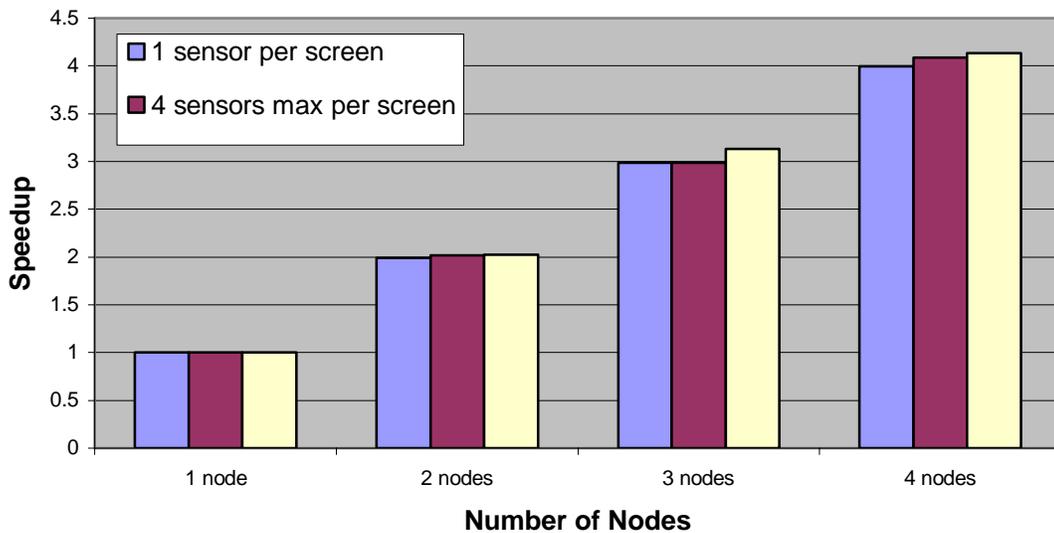
Figure 10: Average Speedup of intervisibility versus increase in node count

Figure 11 shows the number of intervisibility calls in OneSAF versus the number of sensors being processed. One should note that OneSAF operates on an internal scheduler that runs at a specific "tick" rate. All operations, such as intervisibility, run at this tick rate. But as the simulation running in OneSAF becomes overloaded the tick rate is changed to accommodate the increased load. One can see from the figure that as the number of entities increases the number of intervisibility calls per tick increases too. But if intervisibility calls are measured in real-time the number of calls reaches an asymptote and levels off.
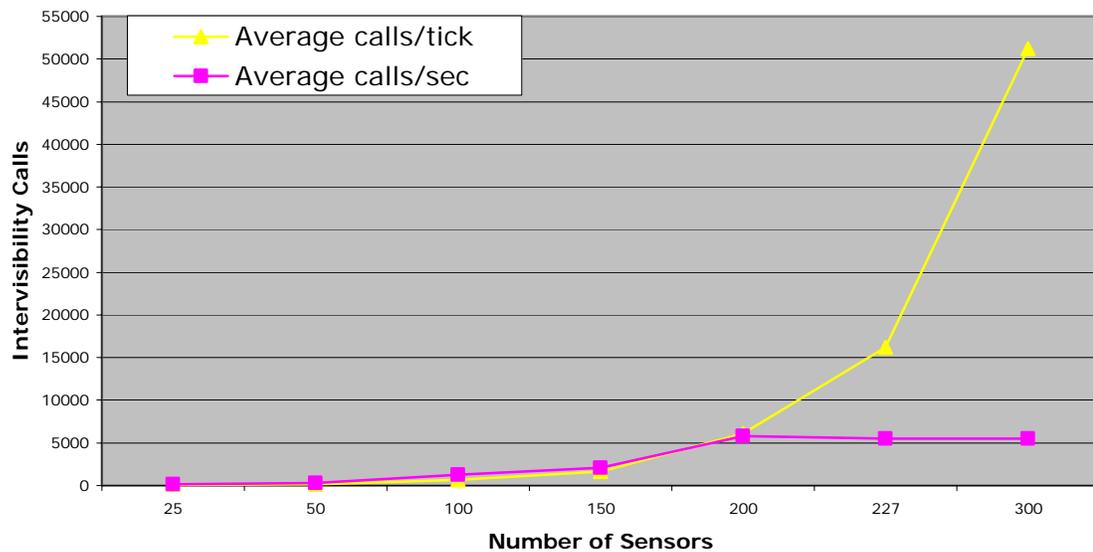
Figure 11: LibCTDB Intervisibility results

Correlation

For the correlation results the same databases were used as in the GPU Intervisibility results. In this test a scenario with 68 entities was generated. In GPU intervisibility algorithm the view frustum was set to have a horizontal field of view of 0.619406 radians and a vertical field of view of 0.508736 radians. The resolution used to render the sensors was 1600 x 1200 pixels.

Intervisibility between all entities was compared. Although in practice an AOI would be used to limit the number of entities being looked at and to reduce the computational load during rendering, one was not used in this case. This resulted in a

32

total of 4556 intervisibility calculations for each algorithm, neglecting the trivial case of

self visibility. Using a threshold of zero for visibility determination the results of the test

are shown in Table 2:

Table 2: Visibility results with 68 entities

Not visible in both algorithms	4453
Visible in both algorithms	40
Visible in LibCTDB algorithm Only	64
Visible in GPU algorithm Only	3

The Correlation Coefficient was calculated using Equation 1 where μ is the mean

and σ is the standard deviation. Using this equation a correlation coefficient of 0.518

was calculated. The samples used to calculate this correlation were taken from the results

generated above. Only visibility of entities that were within the sensors' view frustum

and AOI were used. This resulted in a dataset of 738 visibility queries that were used to

calculate the correlation.

$$
\rho_{xy} = \frac{\dfrac{1}{n}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y)}{\sigma_x \sigma_y}
$$

CHAPTER FIVE: CONCLUSION

Although the performance of the GPU intervisibility algorithm did not outperform the OTB intervisibility algorithms in our tests, the scaling of the algorithm, and since the communication overhead scales better than the computational problem as the number of entities increases, it appears highly likely that by using a larger number of GPUs the performance of the method developed in this thesis has the potential to greatly outperform the OTB approach.. The GPU algorithm scales linearly through four nodes and demonstrates perfect scalability. There are also some important observations to note about the algorithm. First there have been no attempts to optimize the algorithm. The goal was first to get it working properly and then to get it working efficiently. LibCTDB on the other hand has been around for over a decade and has had plenty of optimization work over the years.

In this thesis it has been shown that correlation between the two different algorithms is rather poor. This can mainly be attested to the statistical versus visual algorithms and the way features are represented in the two different simulations. Given that the GPU algorithm is closely tied to how visual simulations operate, such as in man-in-the-loop simulators, it can be interpreted that the combination of SAFs and man-in-the-loop simulators in the same DIS simulation might not offer a fair fight. However, since a GPU accelerated intervisibility server would be operating on a common data

source, this approach would effectively solve the intervisibility correlation problem between OTB and visual simulations.

Suggestions for Further Study

Larger Scale Clustering

For this thesis only four GPUs have the NV_Occlusion_Query extension in hardware available for use. Four GPUs only begins to show the trend of scalability but does not offer any insight into performance on larger sized clusters. Ideally a cluster of at least 16 GPUs should be used to show good scalability.

Algorithmic Optimizations

As stated earlier there were no attempts in optimizing the GPU algorithm but several areas have been identified and should be addressed. First, some form of load balancing technique should be applied when distributing the entities to nodes and to rebalance the load while the simulation is running. There are two metrics that might be useful in implementing this load balancing. The first metric to look at is the amount of data that must be rendered on each node. If the amount of data is reduced then it might be possible to store it all directly in the GPU's memory so that no transfers need to occur

during run-time. In order to achieve this reduction of data the distribution of the entities

must take advantage of the clustering or locality of entities. The second metric to look at

is the geometric complexity of the terrain. If the complexity of a region of terrain is high

and there are multiple entities operating in this area it is possible for these entities to

overwhelm the processing power of the GPU. In this type of case it might be better to

have multiple nodes covering the same terrain region to divide the load of possible

problem areas.

Perception of Visibility

The visibility that is calculated using this new method only calculates the

percentage of visible pixels based on boolean tests of if a fragment passed the depth and

stencil tests. It does not however take into account what the environment is immediately

around the pixel. There could be aspects such as camouflage that effect the visibility of

an entity in the real world that would be perceived by a human. This perceived visibility

could be significantly different then what is calculated and is an important area the needs

to be addressed.

LIST OF REFERENCES

[1]   D. Bartz, M. Meißner, and T. Hüttner, "OpenGL-assisted occlusion culling for large polygonal models", Computer and Graphics, pp. 667-679, 1999

[2]   T. Huttner, M. Meißner, and D. Bartz, "OpenGL-assisted visibility queries of Large Polygonal Models", Dept. of Computer Science (WSI), University of Tübingen, 1998,

http://www.gris.uni-tuebingen.de/~meissner/publications/pdf/TRwsi98-6.pdf

[3]   Dirk Bartz, Micheal Meißner, and Tobia Hüttner, "Extending Graphics Hardware For Occlusion Queries In OpenGL", Proccedings of Eurographics/SIGGRAPH workshop on graphics hardware, pp. 97-104, 1998

[4]   Hanson Zhang, Dinesh Manocha, Tom Hudson, Kenneth E. Hoff III, "Visibility Culling using Hierarchical Occlusion Maps", Proceddings of the 24th annual conference on Computer graphics and interactive techniques, pp. 77-88, 1997

[5]   Hanson Zhang, "Effective Occlusion Culling for the Interactive Display of Arbitrary Models", Ph.D. Dissertation, Departmant of Computer Science, UNC-Chapel Hill, July 1998

[6]   Ned Green, Michael Kass, Gavin Miller, "Hierarchical Z-Buffer Visibility," ……

[7]   Gerald Schroecker. Visibility Culling for Game Applications. Thesis, Technische Universitat Graz, 2002

[8]   Roel Martens. Occlusion Culling for the Real-Time Display of Complex 3D

        Models, Masters Thesis, Limburgs Universitair Centrum

[9]   P. Micikevicius, C. Hughes, M. Moshell. Interactive Forest Walk-through.

        Computer Science Technical Report CS-TR-02, 2002.

[10]   Naga Govindaraju, Avneesh Sud, Sung-Eui Yoon, Dinesh Manocha, "Interactive

        Visibility Culling in Complex Environments using Occlusion-Switches,"

        Proceedings of the 2003 Symposium on Interactive 3D Graphics, 103-112, 2003

[11]   HP_occlusion_test, http://oss.sgi.com/projects/ogl-

        sample/registry/HP/occlusion_test.txt

[12]   NV_occlusion_query, http://oss.sgi.com/projects/ogl-

        sample/registry/NV/occlusion_query.txt

[13]   Open Scene Graph, http://www.openscenegraph.org/

[14]   OmniOrb CORBA, http://omniorb.sourceforge.net/

[15]   Guy Schiavone, Judd Tracy, Eric Woodruff, Mathew Gerber, Troy Dere, Julio de

        la Cruz, "Scalability of Intervisibility Testing using Clusters of GPUs",

        Simulation Interoperability Standards Organization (SISO) Spring 2004

        Simulation Interoperability Workshop, April 2004

[16]   Guy Schiavone, Mathew Gerber, Judd Tracy, Eric Woodruff, Todd Kohler, "A

        Software Tool for Comparing Intervisibility Correlation between CGF and Visual

        Systems", Simulation Interoperability Standards Organization (SISO) Spring

        2004 Simulation Interoperability Workshop, April 2004

[17]   Guy A. Schiavone, S. Sureshchandran, Genneth C. Hardis, "Terrain Database

Interoperability Issues in Training with Distributed Interactive Simulation", ACM Transactions on Modeling and Computer Simulation (TOMACS), 332-367, 1997

[18]   Jill Ashby, Brian Mannlein, and Kirk Thomas, "Correlating Large Area Visual and SAF Databases: Challenges and Solutions", Proceedings of the Seventh Conference on Computer Generated Forces and Behavioral Representation. Orlando, FL, (1998).

[19]   Robert F Richbourg, Dale D. Miller, Paul G. Foley, and Guy A. Schiavone, "Standardizing the Codification of Environmental Objects: The Environmental Data Coding Specification", 00F-SIW-071, Presented at the *Simulation Interoperability Standards Organization (SISO) Fall 2000 Simulation Interoperability Workshop*, September 17 - 22, 2000, Orlando FL.

[20]   Paul Wannacott, "Assessing the consistency of multi-resolution terrain database", Paper# 00S-SIW-041, Presented at the Simulation Interoperability Standards Organization (SISO) Spring 2000 Simulation Interoperability Workshop, March 26-31, 2000, Orlando FL.

[21]   Birkel, Paul A., "SEDRIS Data Coding Standard", Presented at the Proceedings of the Spring Simulation Interoperability Workshop, March 1999, 99S-SIW-011.

[22]   Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, *OpenGL Programming Guide*, 3rd ed. Massachusetts: Addison Wesly, 2000

[23]   SEE-IT, *http://tools.sedris.org/seeit.htm*

[24]   Side-by-Side Viewer, *http://www.acusoft.com/products/sbs*