REAL TIME VISIBILITY CULLING WITH HARDWARE OCCLUSION QUERIES AND UNIFORM GRIDS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ilya Seletsky

June 2013

COMMITTEE MEMBERSHIP

TITLE:	Real Time Visibility Culling With Hardware Occlusion Queries and Uniform Grids
AUTHOR:	Ilya Seletsky
DATE SUBMITTED:	June 2013
COMMITTEE CHAIR:	Zoë Wood, Ph.D.
COMMITTEE MEMBER:	Foaad Khosmood, Ph.D.
COMMITTEE MEMBER:	Aaron Keen, Ph.D.

Abstract

Real Time Visibility Culling With Hardware Occlusion Queries and Uniform Grids

Ilya Seletsky

Culling out non-visible portions of 3D scenes is important for rendering large complex worlds at an interactive frame rate. Past 3D engines used static prebaked visibility data which was generated using complex algorithms. Hardware Occlusion Queries are a modern feature that allows engines to determine if objects are invisible on the fly. This allows for fully dynamic destructible and editable environments as opposed to static prebaked environments of the past. This paper presents an algorithm that uses Hardware Occlusion Queries to cull fully dynamic scenes in real-time. This algorithm is relatively simple in comparison to other real-time occlusion culling techniques, making it possible for the average developer to render large detailed scenes. It also requires very little work from the artists who design the scenes since no portals, occluders, or other special objects need to be used.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Real Time Graphics

Real time computer graphics are needed for CAD applications, simulations, video games, and even for just displaying the user interface of an operating system. Computer graphics can be non real-time, for example when doing CGI for a movie. Each frame can take hours to render, and the result is a very photorealistic image that looks like is real life itself. Millions of frames are generated only for each still frame to be visible for a fraction of a second until the next frame to give the illusion of motion. Real time graphics also display a frame for only a fraction of a second but are rendered on the fly right as the user is controlling the application. This means the computer only has 33.33 milliseconds if trying to render at 30 frames per second (FPS). Most computer monitors update at 60 Hz so real-time graphics typically strive for 60 FPS giving an even smoother experience, and this means the computer now only has 16.66 milliseconds to render a frame. Maintaining a smooth, also known as interactive, framerate is important for ensuring the user has a good experience controlling the application.[26]

Real time graphics are a challenge because a scene may need to be rendered with the quality that looks as close to non real-time graphics as possible like the scene from Battlefield 3 in Figure 1.1. Many calculations need to be performed to produce the resulting geometry and lighting for a frame. Photorealistic real time graphics get away with many shortcuts to look as real as possible. The result may not be completely realistic and does not simulate the physics of light as accurately as non real-time graphics, but the resulting image still looks good. It is possible to have non-photo realistic graphics as well like the character from Borderlands 2 in Figure 1.2, but those too require a large amount of processing power.

1.2 The Power of Graphics Processing Units

Computers have dedicated Graphics Processing Units (GPU) to help with rendering these images. Modern GPUs are built to perform a single operation on large amounts of data in parallel, which makes them useful for non-graphics applications as well with the help of tools like CUDA.[37] The graphics pipeline of current GPUs is built to work with triangles.[13] Geometry is represented by a series of vertices making up the triangles. The GPU then puts the geometry through various stages where all of the vertices are transformed from 3D world space to 2D screen space, and are then rasterized to the screen. There are large amounts of vertices and large amounts of pixels to be rasterized. The GPU runs a single algorithm at a time on these large amounts of data in parallel. This is why modern hardware can render many triangles with relative ease. For example, the dPVS paper has one scene displaying approximately 180,000 triangles at about 68 FPS.[18]

Figure 1.1: The photorealistic real-time graphics of Battlefield 3.[10]

Figure 1.2: The comic book style of Borderlands 2.[11]

1.3 Visibility Culling

Modern graphics hardware is becoming better and better at rendering large detailed environments in real-time. Games like Rage by id Software, Battlefield 3 by Dice, Crysis by Crytek, and Just Cause 2 by Avalanche Studios, have maps on an large scale with drawing distances of several kilometers. These kinds of environments are achievable by avoiding drawing as much of the world as possible.[38, 24, 33, 9, 12, 29, 18, 34, 25, 21, 30] This is the job of a visibility culling algorithm. The scene is viewed through a virtual camera. Objects visible to the camera fit within the frustum as shown in Figure 1.3. It is relatively easy to avoid drawing portions of the scene that are not in the camera's field of view with view frustum culling by checking if the object is inside this frustum.[6] If a camera is looking at a wall or a mountain in the distance, it is also best to avoid drawing the objects behind them. Figuring out that objects are behind other objects in real-time is a challenge but can save many unnecessary draw calls to the GPU.

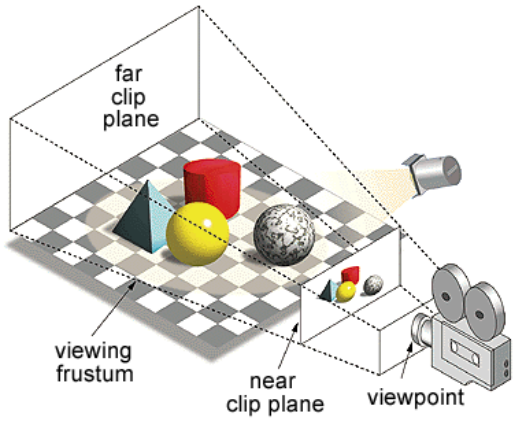

Figure 1.3: A camera and its view frustum. A frustum is a pyramid with its top cut off. Objects within the frustum are visible.[7]

A visibility culling algorithm should do its best to determine which objects are not visible to the camera. The algorithm may generate some false positives and determine that a few objects are visible when they really are not. Since the GPU is able to render geometry very quickly, a few extra objects will just cause a negligible performance drop.[24] The visual quality starts to suffer if objects go missing that should actually be there, so false negatives on visibility should be avoided. Most of the visibility culling work happens on the CPU while the GPU does all the rendering. If the CPU is doing too much work finding which objects the GPU should not render, the performance may be worse than if there is no occlusion culling algorithm in the first place. A good algorithm must strike the perfect balance where the CPU is not working too hard but is still saving the GPU a lot of work, giving a considerable performance increase.

1.4 Dynamic Environments

Static environments of the past used prebaking steps to compute expensive operations such as lighting and visibility. Nowadays more is possible with fully destructible environments and WYSIWYG level editors built right into the game. Environments can be made out of modular set pieces and put together like Lego blocks. Figures 1.4 and 1.5 show an example. A wall, column, railing, ladder, or tree may be reused many times in the level, saving artists a lot of work as well as keeping memory down due to loading in one resource and reusing it many times.

While designing levels, the artists would work with these objects by throwing in new ones, moving them around, and deleting them. The level editor's performance can be greatly increased if it uses a real-time visibility culling algorithm. In a game with destructible environments, these same objects can now be moved

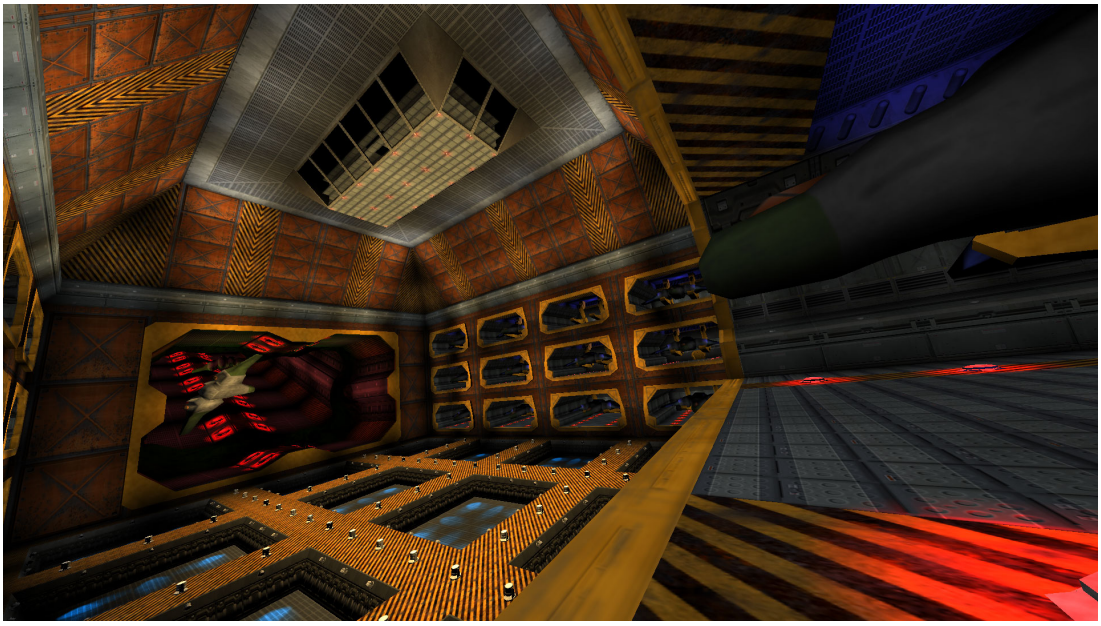

Figure 1.4: A view of the Hangar in the environment made for this project.

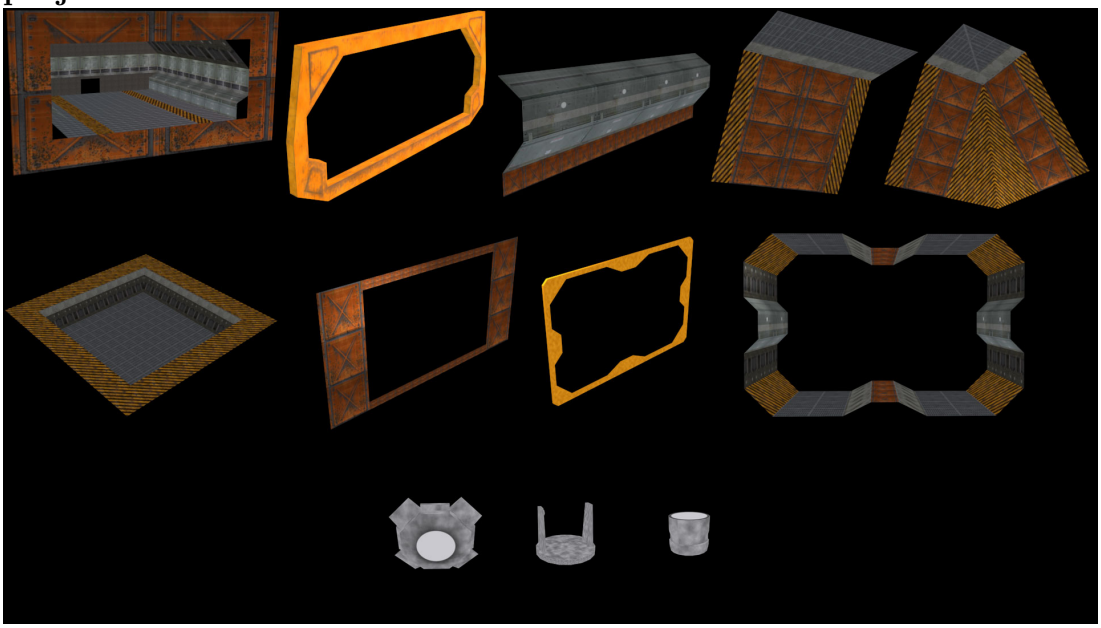

Figure 1.5: Some of the modular pieces that make up the hangar.

around and destroyed on the fly. With all of this happening, visibility data needs to update automatically and in real-time. In fact, the level editor can be either built right into the game or just use the same 3D engine that that game uses. This lets artists see exactly how the environment looks as if they were currently playing the game.

1.5 Artist Placed Hint Objects

Sometimes an algorithm can not figure out the best way to perform visibility culling in every situation so artists need to place hint objects. Hint objects allow an engine to avoid making calculations since the information is given by the designer. This creates a lot of extra work for the artists and can be really tedious. Artists now need to be educated on the technical inner workings of the engine so they can place the hint objects in the best way. Even then, artists may still not do it the best way.

Portals are objects that are placed in windows and doorways linking different rooms or zones of an indoor environment.[33, 32, 9] They help the engine render objects only visible to the camera through the portal geometry and not through the walls. They are effective in static environments where the artists know a door or window will always be around. However if someone takes a rocket launcher and blows up a hole in a wall, there is no artist placed portal there to help the engine. The engine will not necessarily be smart enough to know how to best place a portal there since any kind of arbitrary destruction can take place. The zones divided by portals are often static as well and the engine would need to somehow figure out how to create new zones on the fly. Portals can also be very tedious for the level designers to keep track of as they are constantly iterating on

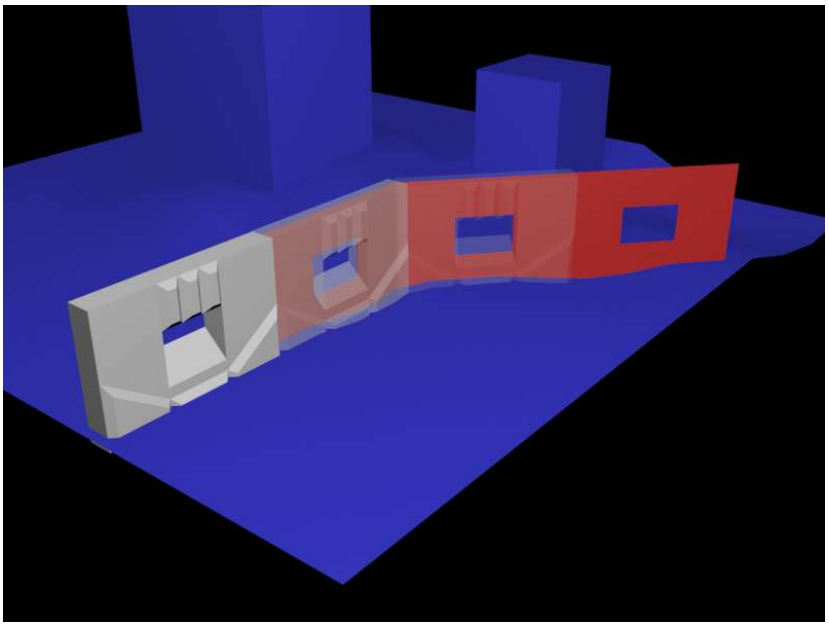

Figure 1.6: The detailed wall and its associated low detail occluder geometry shown in red.

the design of a level.

Occluder geometry is also often used in modern game engines. Modelers can make a low detail version of a mesh as shown in Figure 1.6.[25] The engine can then do ray intersection tests against all triangles in this mesh to test if some object is behind the geometry.[12] Engines that do software occlusion culling render these meshes CPU side rather than rendering the detailed geometry the GPU powered hardware occlusion queries would render. Making these low resolution meshes that are used specifically for occlusion culling creates extra work for modelers. If an object is destructible, the low resolution mesh must somehow be deformable along with the high resolution mesh it represents.

1.6 Our Contribution

This paper presents a simple high performing visibility culling algorithm that any 3D engine can integrate. It allows rendering of large outdoor worlds that are seamlessly combined with detailed indoor architecture. It requires hardware that supports occlusion queries which allow the GPU to report how many pixels of an object are not occluded by other objects.[20]

This algorithm avoids the added complexity of a software rasterizer. Similar algorithms use software occlusion queries in order to avoid the delay when retrieving hardware occlusion query results.[25] This would require software rasterizers to be written that run on the CPU instead of making use of the hardware that performs this task far more efficiently. The paper also presents a method for combining the use of queries with effective render state sorting.

There is very minimal artist intervention required when designing environments using this algorithm. The high detail objects that are rendered are also automatically occluding parts of the environment without the need for artists to manually tweak things with portals or special occluding geometry.

A scene traversal heuristic is also presented. The heuristic traverses the cells of the spatial data structure in the scene in a front to back order relative to the camera view. If the objects in front are drawn before the objects behind, the objects behind can be tested against those already drawn objects. This reduces false positives in visibility tests because objects in back are not mistakenly determined to be visible if the objects in front are processed first.

Chapter 2

Background

This section provides an overview of computer graphics concepts relevant to this paper. First, we describe how software interfaces with the GPU in Section 2.1. Sections 2.2 and 2.3 describe tests that are part of the graphics pipeline for preventing pixels from being written to the screen. Next, the application of the pipeline's tests for use in Hardware Occlusion Queries is explained in section 2.4. Section 2.5 is an overview of spatial data structures that are essential in most graphics algorithms.

2.1 3D Graphics API

There are many Application Programming Interfaces (APIs) for allowing software to interface with the graphics pipeline. The pipeline may be running in software on the CPU or hardware accelerated by the GPU. The APIs allow software to change states in the graphics pipeline and to issue draw calls. Two widely used APIs on personal computers are OpenGL and Direct 3D.[4, 15] The APIs may be different but the computer graphics concepts remain the same. The project

in this paper was built with OpenGL but is portable to Direct3D.

2.1.1 OpenGL

OpenGL is a cross platform standard for rendering 3D graphics maintained by the Khronos group.[4] OpenGL APIs are available on many systems such as Windows, Linux, and MacOS. OpenGL ES is the mobile version bringing 3D graphics to iPhone and Android devices.[17, 5]

2.1.2 Direct3D

Direct3D is part of Microsoft's DirectX package.[15] It is available on platforms such as Windows, Xbox, and Windows 8 phones. At the time of this writing, DirectX 11 is the latest version and has some features that make it easier for multithreaded applications to issue draw calls in parallel.[14]

2.2 Z-Buffer and Depth Test

The z-buffer is allocated along with the Frame Buffer where the image is rendered. A depth value gets computed for each fragment, which corresponds to a pixel of a triangle as it gets rasterized by the graphics hardware. The early z test happens right before the work of the fragment shader where the color of the pixel gets computed and written.[13] The graphics hardware compares the depth value of the fragment about to be written with the value already written. By default, OpenGL is set to allow a fragment to pass if the new value is less than the current value.[8] This test is used for solid geometry only, which usually makes up most of the geometry in the scene. This means solid polygons do not

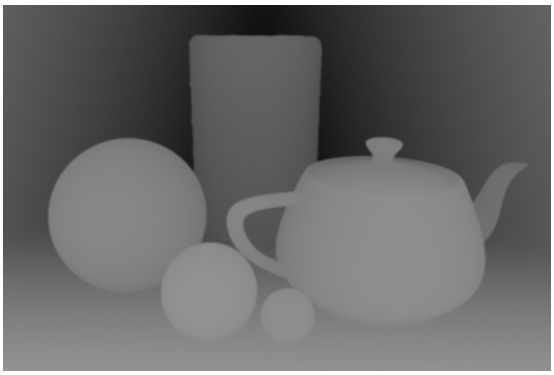

Figure 2.1: Example of a Z Buffer

need to be presorted by depth to have the scene look correct. The depth test is also essential for hardware occlusion queries which will be described later.

2.3 Other Tests

There are a few other tests done by the graphics pipeline that can eliminate a fragment.[2] The scissor test allows rectangle on the render target to be set. Anything outside the scissor rectangle is not rasterized. The stencil test uses a buffer similar to the z-buffer.[13] Values can be rasterized to the stencil buffer and tested in some way. The alpha test prevents a fragment from being rendered if the alpha, also known as opacity, portion of the color is below a certain threshold. These tests, along with the previously mentioned depth test, can be enabled or disabled depending on what is needed for a draw call.

2.4 Hardware Occlusion Queries

The hardware occlusion query is a relatively new feature that is essential to this paper. The query measures how many fragments have passed for a set of

draw calls. The fragments must pass all tests of the graphics pipeline described in Sections 2.2 and 2.3 to contribute to the query results.

A possible use of a hardware occlusion query is to first render an object's bounding box. The box is composed of only 12 triangles so it has very little impact on performance. The box should be rendered during a query with color and depth writing disabled. This means the box will never affect the final image, but it will still go through the full pipeline. The depth test should be enabled so fragments of the box can be rejected if other objects already rendered are occluding the box. Objects in the scene should be rendered front to back so occludees in back are tested against the occluders in front.

After testing with the invisible bounding box, retrieve the results of the query. If any fragments of the box passed, the real object is probably visible. Now color writing and depth writing should be enabled as the real object is rendered. With depth writing enabled, the object is written to the z-buffer and can now serve as an occluder for more objects that are behind it.

2.4.1 CPU Stalls

The approach just described negatively affects performance. The problem with hardware occlusion queries is their results may not be ready to read back when expected.[2, 20, 21, 30] The GPU has its own memory and processor. Data must travel across a bus on the motherboard between the CPU and GPU. Draw calls issued by the CPU are non-blocking and are stored in a buffer for the GPU to execute whenever it gets around to it. However, retrieving a value back from the GPU can cause a stall since the application must wait a few milliseconds for the GPU to execute the buffered commands and for the query result data to

travel across the bus. The query result may be available about 3 to 4 frames later. When there are thousands of objects to draw, the stalls add up. The challenge with using hardware occlusion queries is finding a way to effectively use them in later frames to reduce CPU stalls but still save the GPU from drawing too many unnecessary objects.

2.4.2 Conditional Rendering

Conditional rendering allows the results of a query to be used in the same frame. The GPU knows the results of a query after a draw call since the data is local and does not need to travel across the bus. The CPU still issues all potentially unnecessary draw calls but the GPU ignores them if the draw call depends on the results of a query. If hardware support for conditional rendering is not available, it is forced to run in software and causes a drop in performance. This does not prevent unnecessary traversals of portions of the scene by the CPU that may be invisible. It is simply a way to tell the GPU to ignore an already issued draw call.

2.5 Spatial Data Structures

Spatial data structures accelerate spatial queries by reducing the search space. View Frustum Culling described earlier is an example of a spatial query that finds objects in the scene that are within the bounds of the frustum. Another example is to find all objects within a radius or axis aligned bounding box. Many spatial data structures exist, each with their own pros and cons. The right structure should be chosen depending on the application. Chapter 3 provides an overview

of other visibility culling techniques and what spatial data structures were chosen.

The uniform grid is one of the simplest structures. It provides a uniform subdivision of space. Objects have a bounding box, and are binned into cells of the grid that overlap with the bounding box.[25, 31, 19] Querying for objects in the grid simply requires indexing into cells of the grid which intersect with the spatial query. This is an optimal structure for cases when objects are evenly distributed in the world. This could be an issue if the scene has hundreds of objects in one cell while the rest of the cells have very sparse amounts of objects. Querying within the bounds of that cell would not reduce the search space by very much.

A spatial hash is similar to a grid but allows the scene to not be constrained within predefined bounds. The hash has buckets just like a grid has cells, but multiple cells can hash to one bucket. For example, if the grid is allocated to hold 5 by 5 cells, grid cell (1, 3) would hash to the same bucket as (6, 3). A standard grid would not be able to handle cell (6, 3) since that is outside of the grid.

The quad tree is a hierarchical data structure that handles unevenly distributed objects better.[31] It distributes the objects by progressively dividing each cell into four smaller cells until each cell has a small enough number of objects for the application. Since it is structured as a tree, many cells get filtered out along the traversal before reaching the leaf cells. Once a path down a traversal is not taken, all child nodes in the hierarchy are automatically ignored. This helps filter out many more potential objects than a grid earlier on in the search. Octrees are a 3D extension of quadtrees and divide each cell into 8 children rather than 4. This can be extended to other dimensions beyond that as well, to help with problems mapped to multiple dimensions.[31]

Bounding volume hierarchies are another common hierarchical structure.[36] This organizes the scene by having each node be a bounding box that contains the bounding boxes of all of its children. The leaves in the tree contain the objects themselves. As the tree is traversed, the bounding box of the query is used to determine what set of children to follow if their bounding box also intersects the query object. A well balanced tree structure gives the best overall performance. This kind of structure allows any arbitrary scene layout while grids and quadtrees are constrained to hold the scene within predefined bounds.

K-D trees recursively divide up a space into two halves with axis aligned planes.[33, 32] They can look a bit similarly to Octrees in some situations. The space is divided in a way that evenly distributes the geometry between the two planes, handling the poorer distribution of Octrees and Uniform Grids. Binary Space Partitioning trees are similar except they divide space with arbitrary planes. The final result is an even better distribution of geometry that results in a more balanced tree, but the algorithm for computing the tree is more time consuming.

Chapter 3

Related Work

Visibility Culling has been performed with a variety of methods over the last decades. Many clever tricks have been found that take advantage of certain assumptions about a scene, sometimes limiting what kinds of scenes are possible. Keeping a scene static while performing a walkthrough of an indoor area is common.[33, 29, 32] Some software-based occlusion culling solutions have been used as well in order to avoid the CPU stalls of hardware occlusion queries.[27, 25]

One visibility culling algorithm uses mathematical properties of convex polyhedral objects to determine visibility on the fly.[24] Coorg and Teller find objects in the scene that serve as large occluders, and avoid testing against smaller objects that would not serve as effective occluders. Since the environments are made of collections of convex geometry, it is possible to compute separating and supporting planes from pairs of objects and compute whether or not one object is occluding another. The algorithm was developed at a time when the z-buffer may be unavailable or done slowly in software, so this algorithm was able to avoid relying on the z-buffer.

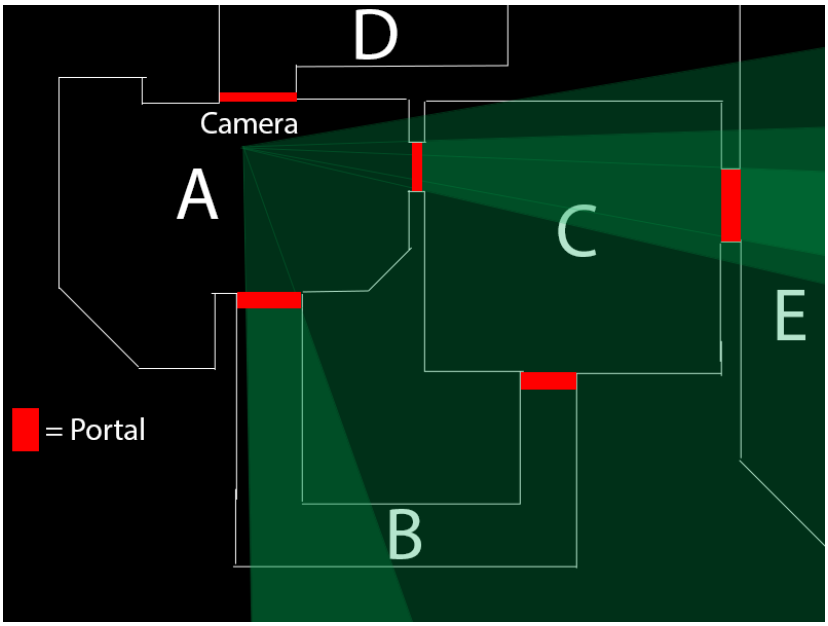

Figure 3.1: An example of portal culling at work. Zones A, B, C, and E are rendered.

Portal culling, which was mentioned earlier, is effective for indoor environments but not as effective in outdoor environments.[29] If the environment is subdivided into zones, only geometry in zones that are determined to be visible gets rendered. The camera's initial view frustum gets computed. The geometry in the zone the camera is located is then rendered. If the geometry of a portal is visible to the camera, a new smaller view frustum spawns off that simulates the view through the portal into the adjacent zone. The scene's zones are traversed this way until no more portals are determined to be visible.

Teller finds another way to compute visibility in environments made of convex polyhedral geometry.[33] This time the scene is preprocessed ahead of time with his algorithm. The algorithm is applied in id Software's idTech 4 engine used in games like Doom 3 and Quake 4.[32] Areas of the environment are subdivided and stored in a BSP tree. Portals need to be placed by the level designers in crucial

areas like doorways and windows to allow the algorithm to compute zones. The algorithm finds mini areas and they are joined into zones with a flood filling algorithm. Solid geometry and artist placed portals determine the bounds of zones. Without artist placed portals, Doom 3 renders the entire level rather than only visible rooms. In 1992, this algorithm was said to take hours of CPU time, but was able to store the processed scene in a fraction of the space. Id software's previous titles, Quake 1 through 3, used to run a similar algorithm offline in a separate utility. Artists made levels in a separate tool and converted the levels to a format loadable by the game. Doom 3 can now run this algorithm right before loading a level and the process is built right into the game. This way the game can ship with the levels in their editable form for people to look at with the built in map editor.

Zang, Manocha, Hudson, and Hoff have developed an algorithm that uses a hierarchy of occlusion maps.[38] Occlusion maps are similar to the z-buffer. In this algorithm a hierarchy of buffers is created, which are similar to the z-buffer at different levels of detail. Occluders are rendered to this buffer. Occludees then are tested against the right buffer depending on how big they are on the screen by testing and rasterizing a bounding rectangle. An algorithm similar to this was applied in Ubisoft's Splinter Cell Conviction.[27] They were able to achieve about 20,000 queries every frame on the Xbox 360.

The Frostbite 2 Engine developed by Dice is a great example of modern occlusion culling.[25] This engine effectively supports destructible environments in both indoor and outdoor settings as shown in the game Battlefield 3. It shows off the capabilities of modern high end hardware as well as managing to run on dated hardware with a lower spec profile. Typically hierarchical data structures such as Octrees, Bounding Volume Hierarchies, or BSP Trees are used for 3D

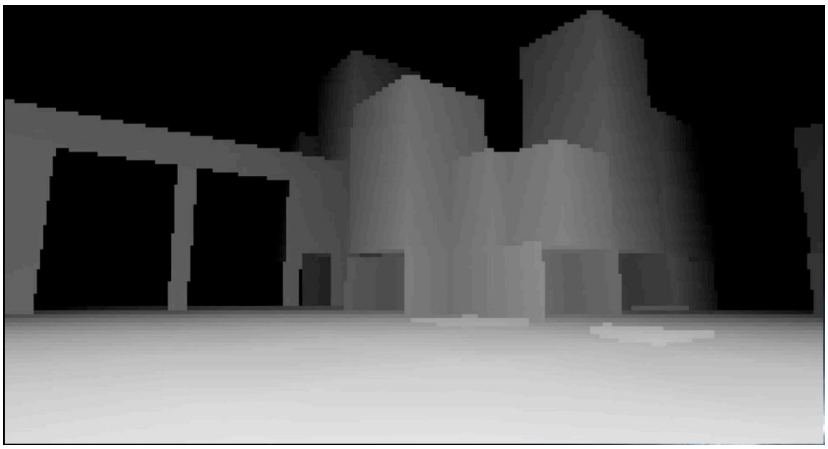

Figure 3.2: The low resolution software z-buffer from a scene in Battlefield 3.[25]

scene management.[21, 30, 20, 18] However, the developers of Battlefield 3 found they were able to push more performance with a uniform grid. Modern parallel computing architectures handle the linear arrays of a grid much better than the branch heavy traversal of the hierarchical structures.[25] The lack of branching creates predictable data access that is cache coherent and allows various parallel algorithms to operate more efficiently. The visibility culling algorithm relies on artist generated low detail large occluder geometry that is rasterized software-side to a low resolution depth buffer. This requires a software rasterizer capable of rendering arbitrary geometry to be implemented. Objects such as tanks or soldiers have their bounding rectangle tested against this buffer similarly to how a hardware occlusion query would operate before being queued up for the hardware to render.

CryENGINE 3, used for games like Crysis 2 and Farcry 3, is another modern game engine capable of rendering large indoor and outdoor environments effectively.[34, 12] Cryengine uses what is called a coverage buffer but avoids implementing a full arbitrary geometry rasterizer by reading back the z-buffer from

Figure 3.3: The coverage buffer used in CryENGINE 3. The red portions are the holes after reprojecting from previous frames to the current camera angle.[34]

the hardware and using it in later frames. As the camera moves around, the coverage buffer needs to be adjusted to correspond more closely to how the current frame looks. CryENGINE reprojects the buffer with the transform of the current camera and then fills in any holes. After this, like Frostbite 2, objects' bounding rectangles are rasterized and tested against the contents of the coverage buffer. CryENGINE also allows for use of portals and occlusion geometry that they refer to as antiportals to further augment the visibility culling algorithm.

The dPVS algorithm developed by Aila and Miettenen is another similar algorithm capable of rendering large dynamic environments.[18] Their system allows for a combination of various occlusion culling algorithms such as portals and queries. The scene is organized by a K-D tree. This data structure is normally used best in static environments, but this algorithm is able to effectively recompute the structure on the fly. They too use a coverage buffer for occlusion queries

where the silhouettes of objects are rasterized. Queries are then run against this coverage buffer.

The Coherent Hierarchical Culling algorithm uses the same hardware occlusion queries that this paper uses.[21, 30] This algorithm allows a scene managed by a Bounding Volume Hierarchy to effectively use queries. Temporal coherence is taken advantage of, meaning the results of queries from previous frames are not likely to change. As the tree of the bounding volume hierarchy is traversed, the visibility results of nodes are used in later frames. Since this is a hierarchy, entire portions of the scene can be culled out for a number of frames. They make no mention on whether or not objects noticeably pop in late with this algorithm.

Chapter 4

Visibility Culling Algorithm

In this chapter, we describe the steps in the visibility culling algorithm. Section 4.1 is a general overview of the entire algorithm. Then, Section 4.2 describes the scene traversal algorithm. In Section 4.3 we describe the steps the algorithm takes to prepare the objects in the scene to be rendered. Then, in Section 4.4, we describe how to add support for multiple viewports for split screen support.

4.1 Algorithm Overview

This algorithm stores the scene in a uniform grid. A grid is easy to keep updated with the contents of a fully dynamic scene. Objects that move around can use their Axis Aligned Bounding Box to determine which cells they need to be tracked in. A grid's structure does not need to be recomputed as the scene changes in real-time like the other static structures would. There is no branching as the structure is traversed, making it easier to process the scene in one pass with occlusion queries. The results of visibility changes in a hierarchical structure would take a few extra frames to propagate, possibly increasing the likelihood of

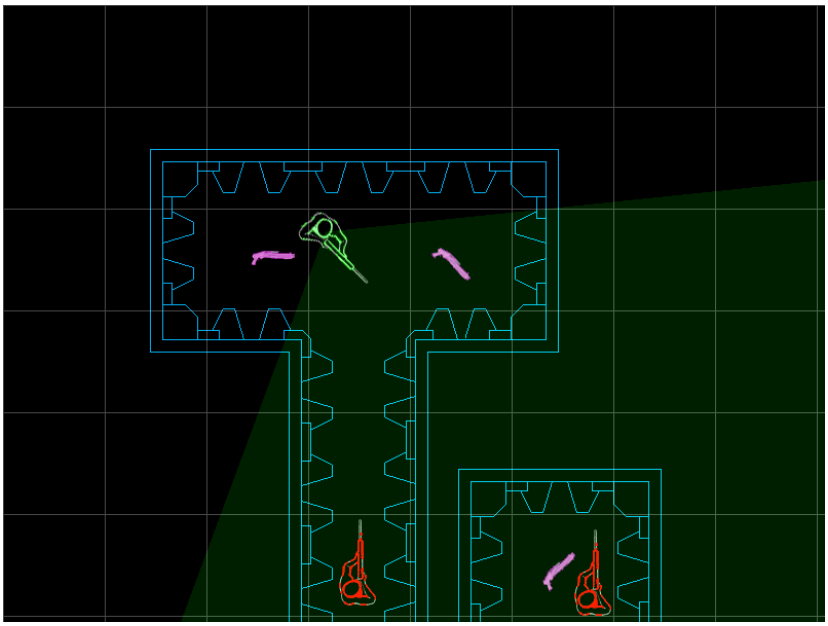

Figure 4.1: The overall scene partitioned by a uniform grid. The green cone represents the camera's field of view.

noticeable popping artifacts.

Figure 4.1 shows a top view of a full scene and the field of view of a camera that is looking at the scene. The grid cells that intersect the camera's view frustum are traversed in approximate front to back order relative to the view from the camera. For each cell that gets traversed, an occlusion query is issued. Color and depth write are disabled and just the box of the cell is rendered.

If the cell is marked visible in a previous frame, the cell's objects are now rendered with depth write only but color write turned off. This way objects are put into the z-buffer to occlude cells farther back but none of the expensive lighting and shading calculations are performed. The cell's objects are also queued up to later be rendered in full color. Figure 4.2 shows a moment in time during the frustum traversal and Figure 4.3 shows a fully traversed scene.

Now that the scene is fully traversed and all objects that are from visible cells

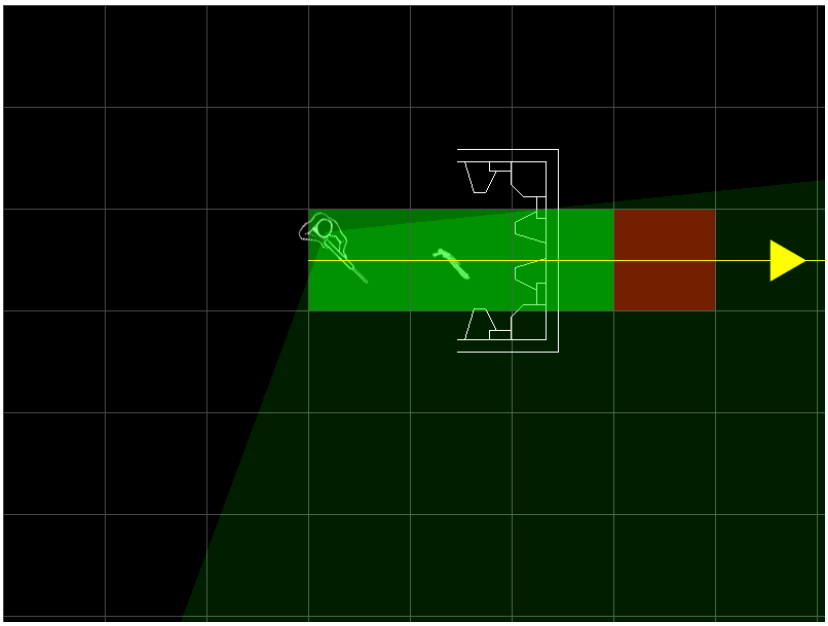

Figure 4.2: A moment in time. The objects are white to show that they are only rendered to the depth buffer at this stage. The cells of the grid within the camera's view are being traversed. Green represents cells visible to the camera. Red represents occluded cells.

are queued up, they get rendered with color write on. Figure 4.4 shows a full color scene with objects being rendered only if their cells were determined to be visible.

4.2 Grid View Frustum Culling

4.2.1 Overview

The view frustum culling occurs on the grid cells only, rather than on each individual object in the scene. This allows all objects grouped in a cell to be culled out at once rather than making the CPU check every object individually. If a cell is right at the border of the view frustum and a few objects inside the

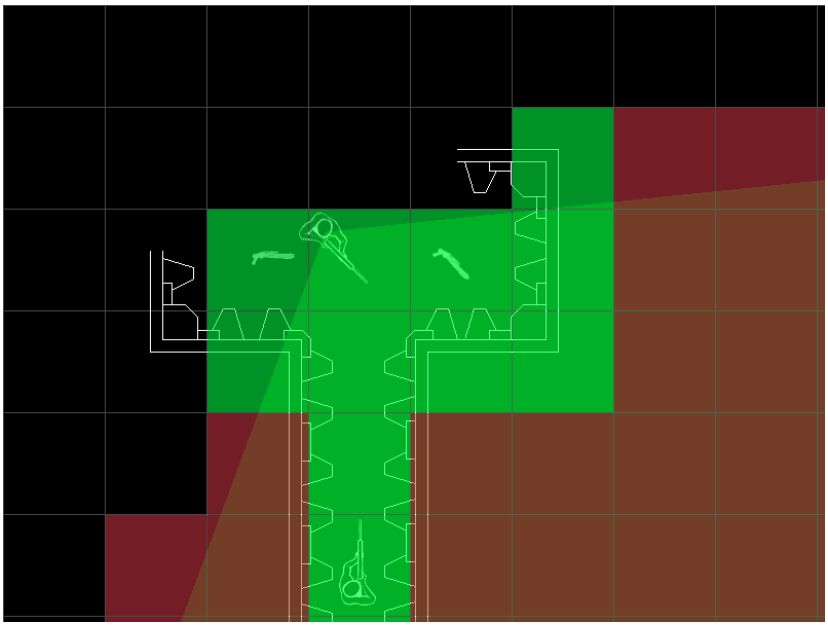

Figure 4.3: The scene fully traversed.

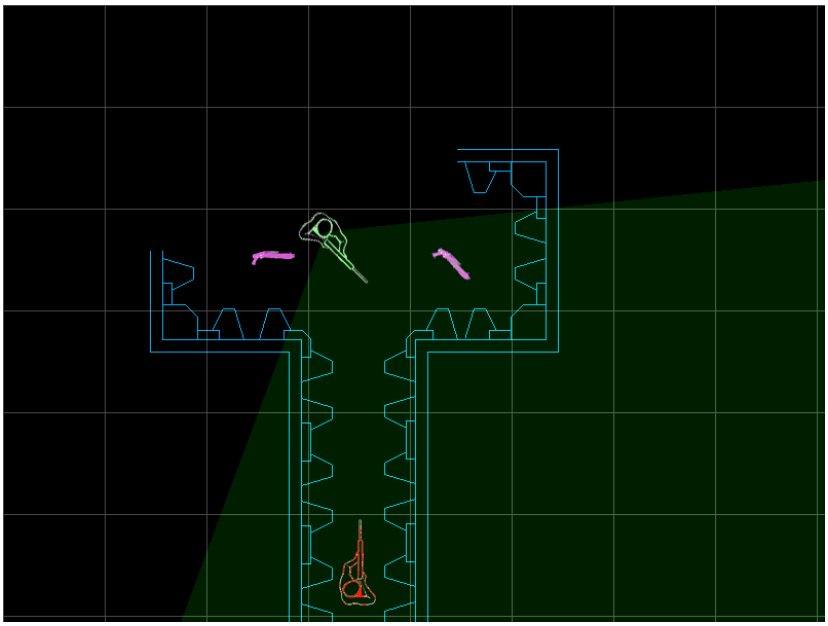

Figure 4.4: The scene is now rendered in full color.

cell are not in the frustum, it is not a problem to have the GPU needlessly render them. The work done by the CPU checking all objects in a cell will cause a higher performance drop than having the GPU draw a few wasted objects. This algorithm allows for large scenes with fine enough grids that there will usually be very few wasted objects compared to the immense number of objects that do get rendered. The space should be subdivided by the grid in a way that renders as few wasted objects as possible. This applies to the occlusion culling as well. For example, if the camera is located inside a room, the cells at the border of the room should contain as few objects on the other side of the wall as possible or else a large portion of invisible objects get rendered along with the visible objects in the current room.

The cells in the grid should be traversed front to back relative to the view. Figure 4.5 shows a top down view of a scene. Objects closer to the camera get rendered to the z-buffer before objects in the back do. Cells farther from the camera get tested against the objects in front that are potentially occluding them since they are in the z-buffer.

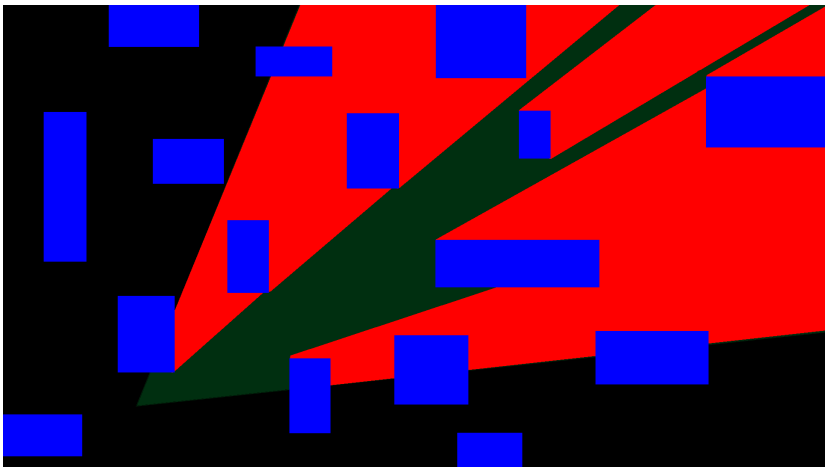

Figure 4.5: A view frustum, and the objects inside occluding each other.

A true front to back traversal of a grid is very difficult to do well. One attempt involved traversing out with rays from the center and stopping the traversal when reaching an occluded cell or the far plane as shown in Figure 4.6. A 3D DDA line rasterizing algorithm would be used to traverse the cells along a ray.[23] This method was rejected early on. It was non-trivial to find out the best way to align the rays so that duplicate cell traversals were minimized and absolutely no cells were ever missed.

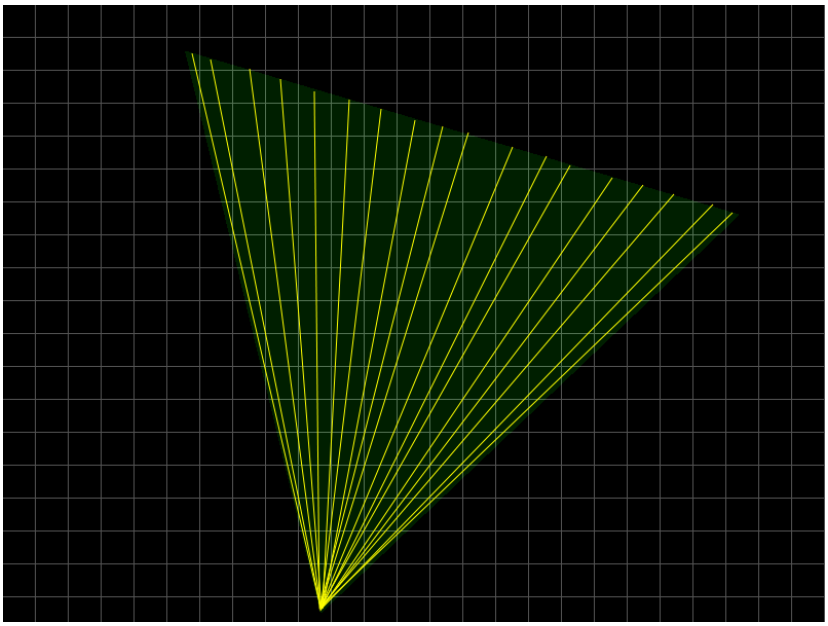

Figure 4.6: Showing an example of ray based frustum traversal. The cells in front get traversed multiple times.

Another method for grid traversal uses the chessboard metric.[19] Each slice of the frustum is determined by traversing cells that are a certain grid distance away from the origin as shown in Figure 4.7. The distance is increased by 1 each slice. The view vector itself is rasterized as a line in 3D using the DDA algorithm. The cells that make up the points of the rasterized line are used as origin points from where traversal should fan out for that slice. This algorithm

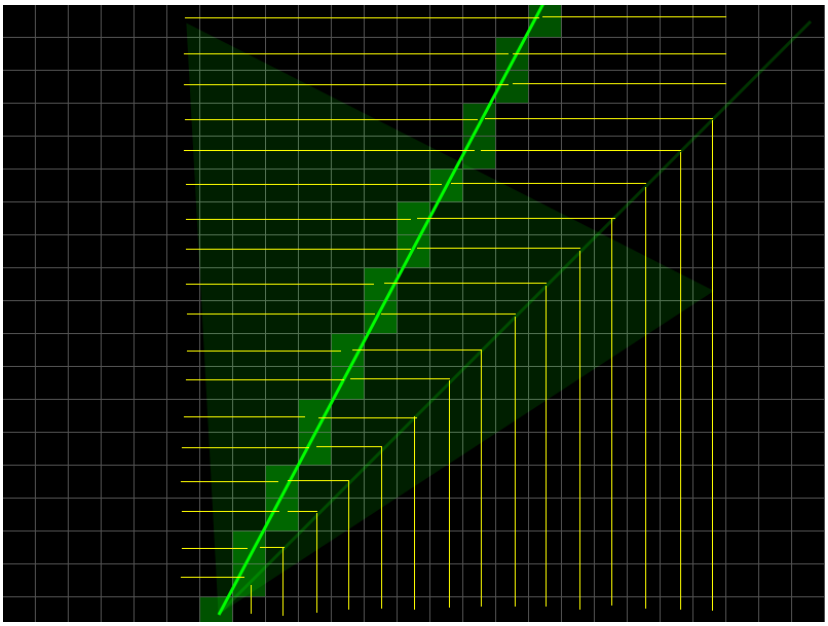

Figure 4.7: The chessboard metric in action.

made it difficult to avoid traversing cells completely outside of the view frustum. When the view is at a 45 degree angle, there is a huge space past the far plane that ends up being needlessly traversed. Modifying the algorithm to avoid this, especially extended to 3D, was not trivial enough.

The final idea involves taking axis aligned slices of the grid perpendicular to the lowest magnitude component of the view vector. It is a simple heuristic compared to the previously tried solutions and can be based on existing convex polygon rasterization algorithms but extended to 3D. This is not a true front to back traversal but it provides the needed results. In Figure 4.8, cell 6 is traversed after cell 5, but cell 6 is closer in distance. However, cell 6 will never occupy space in front of cell 5, so any objects in cell 6 will never be drawn in front of cell 5. Cell 6 will be occluded by objects in cells 2 and 0, while cell 5 will most likely be occluded by objects in cells 0, 1, and 4. As long as a cell is traversed after the cells that could potentially occlude it, the query results will

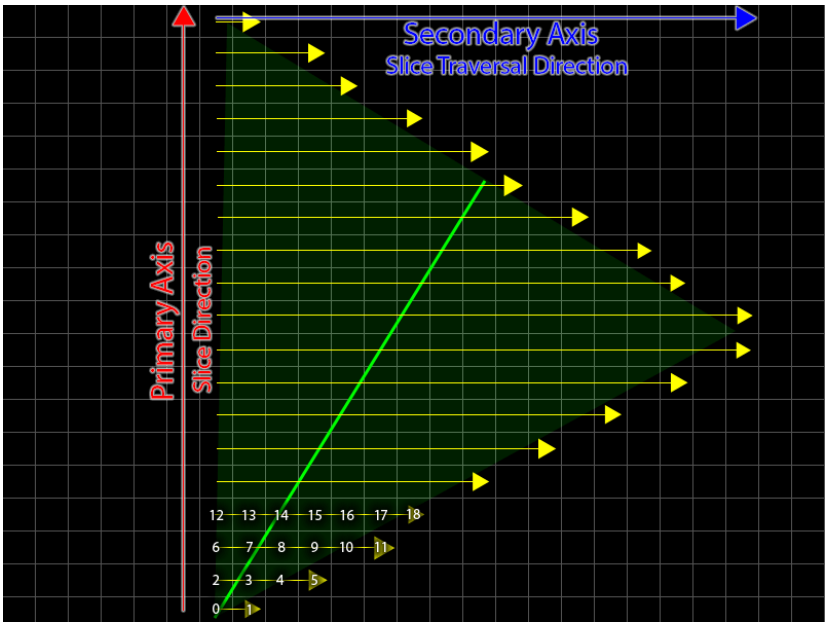

Figure 4.8: Occlusion order traversal.

be more likely to be correct and not create false positives. This gives a good approximation of occlusion order traversal but not a true depth order traversal. This same traversal order is also perfect for roughly depth sorting alpha blended objects that will need to be rendered later. A shorter sub sort per cell rather than sorting over the entire list of objects to render can give the correct rendering order for transparent objects. This depth sorting can even be parallelized by using a separate thread per cell, however we leave this for future work.

4.2.2 Handling Wide Fields of View

In practice, view frustums are very wide. A camera using a standard 90 degree vertical field of view on a 16:9 wide screen monitor will create a horizontal field of about 160 degrees. Wide frustums cause cells that should have been occluded to be traversed before the cells that occlude them as shown in Figure 4.9. In this

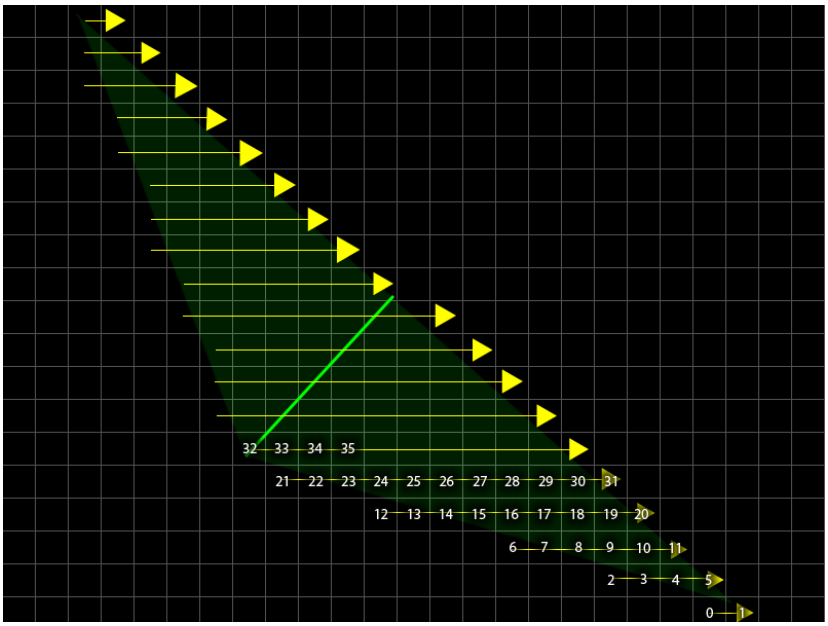

Figure 4.9: Incorrect traversal of angled wide view frustum.

case cell 0 is most likely occluded by cells 32, 21, 22, 23, 24, 12, 13, 14, 15, and so on. Even if the camera is staring straight at a wall, cells 21, 22, 23, and the others will falsely pass the occlusion queries because the wall in cell 32 has not been rendered to the z-buffer yet.

Splitting the frustum into pieces solves this. The splitting point is centered around the cell where the eye is located as shown in Figure 4.10. This creates new convex volumes out of the split view frustum. In 3D, the frustum is split into 8 pieces by the x, y, and z planes. A convex polygon plane clipping algorithm will do the trick. The volumes are now traversed in a fan out order around the center cell. In Figure 4.10, first the central volume is traversed, and then cells 60 and onward are traversed afterwards. This keeps a better occluder order and prevents more cells from being falsley determined to be visible.

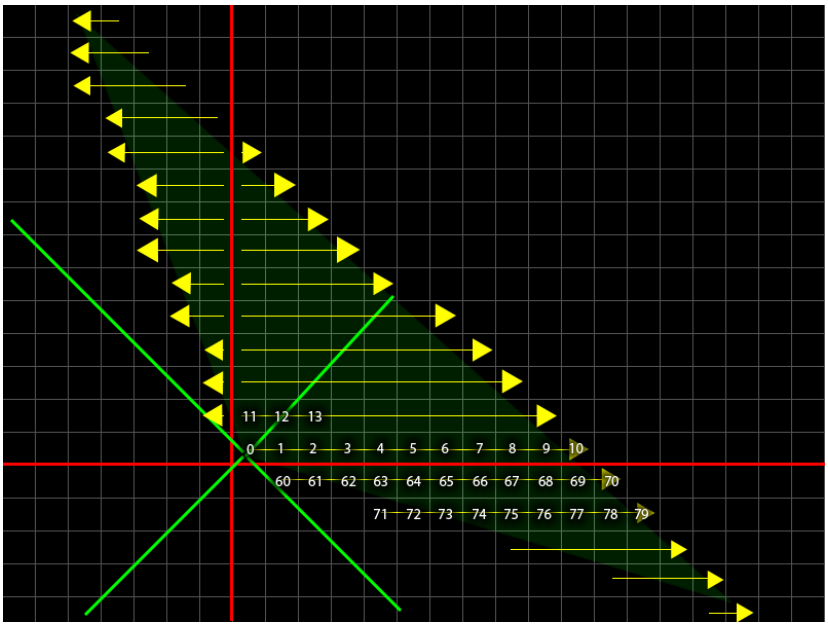

Figure 4.10: Correct traversal of split angled wide view frustum.

4.2.3 Objects at Cell Boundaries

When doing spatial queries against a grid, some group of cells gets traversed and all objects in those cells get returned in the query. If an object occupies multiple cells it can potentially get added multiple times as a query result. Keeping track of a 64 bit unsigned query id counter can prevent this. The main scene can store a query counter, and increment it every time a query occurs. The objects returned by the query can keep a query counter within them as well. If the object's counter is less than the scene's counter, then the object has not been added to the query results yet. As the object is added, its counter is set to the scene's counter and any other cells that contain the object will not needlessly duplicate the object in the query.

In a multithreaded engine, when many queries can occur in parallel, a lockless set of query ids per object can work. If a query id is already in the object's query

set, the object can be ignored. After the query results are done, the query id can be removed from the set, keeping the set from growing. Lockless sets that are part of Intel Thread Building Blocks do not allow safe lockless removal, so it may be best to clear these sets at some synchronization point such as the end of a frame to prevent them from growing too large.[16]

4.2.4 The Rasterizing Algorithm

The front to back traversal heuristic uses this 3D rasterizing algorithm. The algorithm does not handle just frustums, but any convex polygon with any arbitrary amount of points and edges. This allows the algorithm to handle the frustum being clipped against the level bounds and sliced into 8 pieces to handle wide and angled views.

The algorithm is optimized enough that it is able to quickly process very dense grids. It has been tested on grids of 1,000 x 1,000 x 1,000 cells and the framerate stayed at about 40 FPS in the worst case, and at about 100 in other cases. This means there are one billion cells total. The algorithm was tested by making OpenGL render a point at the center of what would be a cell in 3D space.

In practice, the scene is never subdivided by that fine a grid anyway. Grids would typically have dimensions of not more than 100 x 100 x 100. It would take over 59 gigabytes in RAM to store one billion 64 bit NULL pointers for a completely empty scene with a grid of 1,000 x 1,000 x 1,000 cells. Since the hardware must also render a box for each cell, the performance would drop significantly rendering that many boxes and later retrieving occlusion query results even if a system had that much memory. Thus, the CPU is able to rasterize dense grids with relative ease, while memory and GPU power are the bottlenecks requiring

the grids to be even less dense in practice.

It is possible to avoid implementing this algorithm and use a simpler axis aligned box rasterizer. The axis aligned bounding box of the view frustum can be traversed in occlusion order row-by-row by just using three nested loops, one for each dimension. In this case cells that are outside of the view frustum get traversed, creating some wasted effort by the CPU as well as requiring either wasted cell queries or to check if each cell is in the view frustum.

The algorithm works by taking a convex polygon as shown in Figure 4.11 and processing it slice-by-slice down the grid as shown in Figure 4.12. Each slice is now a 2D polygon, so a modified 2D rasterizer is used similar to existing 2D convex polygon rasterizing algorithms. For each row in the slice, the range of cells to be traversed is computed as shown in Figure 4.13. The full implementation is presented in the Appendix in Chapter A.

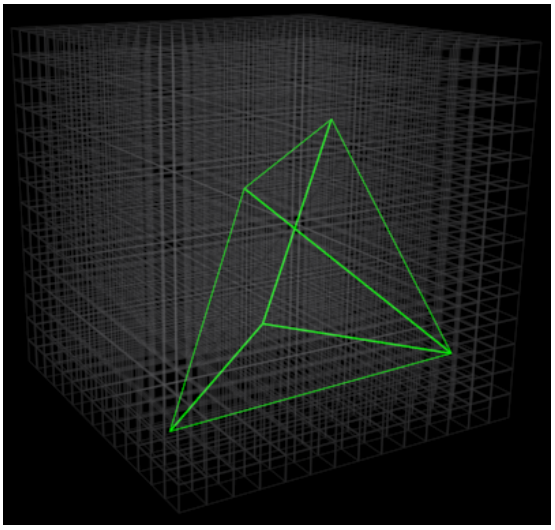

Figure 4.11: A view frustum within the bounds of a scene.

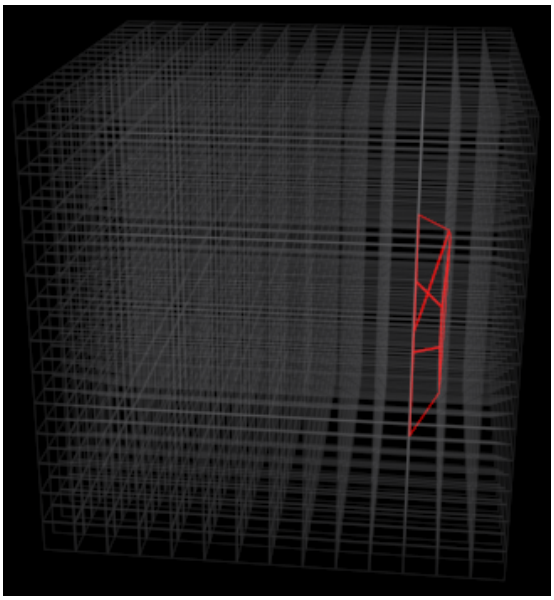

Figure 4.12: A 2D slice of the same frustum seen from the side.

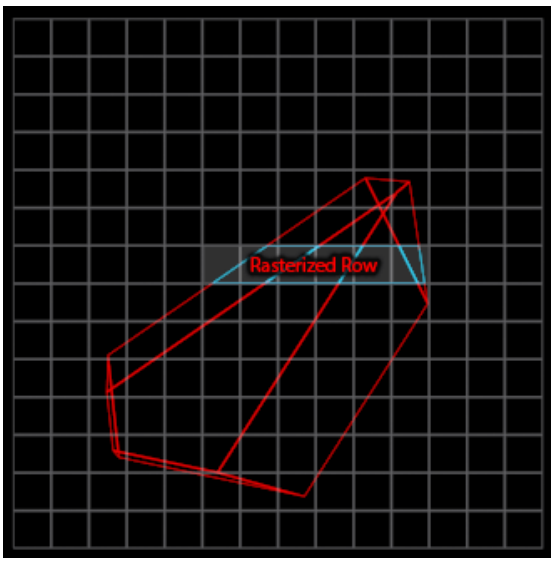

Figure 4.13: A row from the slice being rasterized.

4.3 Rendering

4.3.1 Render State Sorting

Other rendering algorithms that make use of hardware occlusion queries usually benefit from having all objects sorted front to back. In Figure 4.14 Tree A should be drawn before the Tank since it can potentially occlude the tank. After that the Person should be tested against the Tank and Tree A. Then Tree B gets tested against everything else. This would require the objects to be drawn in the order, Tree A, Tank, Person, Tree B.

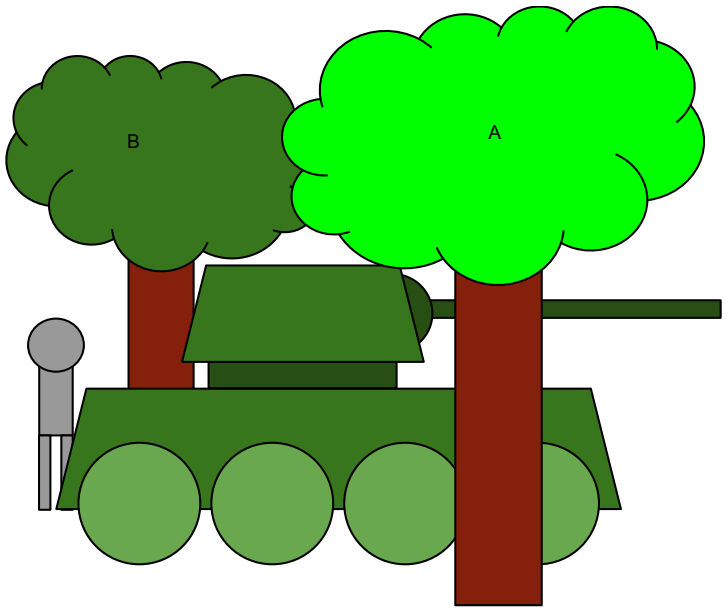

Figure 4.14: A scene.

Rendering performance would suffer since these objects are not sorted in an optimal render state order, but in occlusion order. When rendering a large number of objects, they should be sorted in a way that causes the least amount of change in the render state. It would be better to render Tree A, Tree B, then

switch to the shader and textures that render the tank, then do the same for the person. The algorithm in this paper avoids this problem and allows objects to be queued up and then sorted by their render state.

The renderer used for this project sorts objects first by shader, then textures, then by the instance of mesh. This is done in order of which changes cause the worst performance hit for the GPU.[22] It should also be possible to quickly depth sort the meshes being rendered to avoid some overdraw but the objects are already sorted well enough when the view frustum is rasterized front to back. True depth sort is only needed for alpha blended objects.

4.3.2 The Visibility Culling / Depth Pass

The first phase involves performing a depth pre-pass while performing the occlusion queries. The depth pre pass renders solid objects to the z-buffer with color writing disabled so the complex pixel shaders that compute lighting and color are not run. This is used for the occlusion queries during this stage as well as for preventing pixel shader overdraw when the scene is later rendered with lighting and color calculations. For each unoccluded cell that is traversed during this phase, its objects are rendered to the z-buffer as well as being added to the main render queue.

Retrieving Cell Query Results

First, the query results from last frame are gathered. This updates the visibility status of cells for the current frame. Since the cell query results are used next frame, objects may pop in a frame late. This is unnoticeable at high frame rates due to how little time a frame lasts. This is a common sacrifice other engines

make when using occlusion queries. Query results may take up to four frames to be ready in some cases, so even using the results next frame rather than the current frame causes a CPU stall. This makes retrieving query results from last frame take up about 25% CPU time. The frame rate still remains high, sometimes reaching 300 FPS if running only queries but not rendering anything. It is possible to first check if the query result is available before making the blocking call. This is avoided to keep the algorithm simpler and avoid rendering frames where the object is not visible yet. This stall can be inconsequential if a separate thread retrieves the query results while the engine is left to run Physics, AI, sound, and other subsystems.

Storing Query Results

The cells keep track of their last visible frame by storing a 64 bit unsigned frame counter. A 64 bit unsigned integer can count more atoms than there are in the universe. Even if the renderer is running at 500 FPS, it will take 15,768,000,000 years for this counter to roll over. The scene itself keeps track of the current frame counter in its own 64 bit unsigned integer. When a cell query is retrieved, it writes the current frame counter of the scene as the last visible frame. This way it is unnecessary to iterate over all cells every frame to reset their visibility status.

Occluder Effectiveness

In order to prevent too much work during the depth pre-pass, the artists can specify how the object should be rendered during this phase. This, along with grid dimensions, is the only setting artists need to set in order to achieve high

performance. The three settings for an object are to always render, sometimes render, and to never render. Figures 4.15 and 4.16 provide good examples of what objects are given what setting. An object can be marked as an essential occluder that must always be rendered if its cell is visible. This is for objects like walls, terrain, or any other objects that keep the world water-tight. These are objects that must populate the z-buffer in order for cells behind to be properly occluded.

The second setting of "sometimes" is reserved for fairly large detail adding objects that would be in the middle of rooms, such as crates, columns, or trees. The renderer in the game Space Marine limits the depth pre-pass to only 75 objects in order to prevent too much work during this phase.[28] This engine only adds objects marked to render sometimes if there are not already 75 objects rendered in the depth pass. Since the scene is traversed front to back, the objects close to the camera and likely to contribute to occlusion usually make it while the further away objects are avoided.

The third setting of "never" is for the tiny objects that add detail, such as a clock on a wall. In a fully detailed environment it is likely that there will be very many of these objects. They are likely to not contribute much to the occlusion and rendering them during this phase can be a waste.

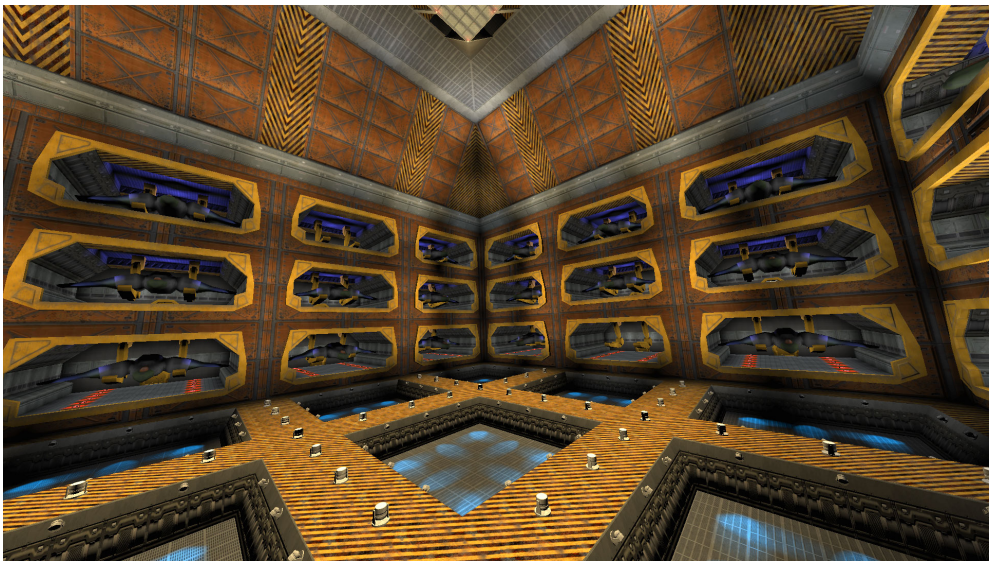

Figure 4.15: Here the walls, ceiling, and floors are marked to always render during the depth pre-pass. The fighter jets inside the hangars are marked to sometimes render. The lamps on the floor and clamps holding the jets in place are marked to never render.

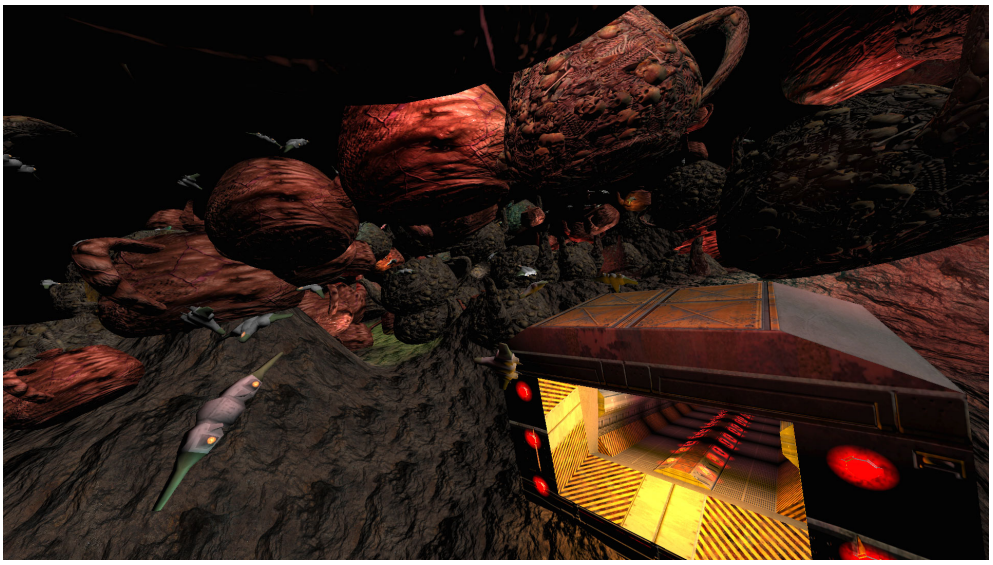

Figure 4.16: Here the building, underground tunnel walls, and the terrain are set to always render during the depth pre-pass. The teapots and jets are set to sometimes render. The red lamps on the building are set to never render.

Pseudocode

The following pseudo code shows the algorithm as described earlier. The scene stores a 64 bit unsigned frame counter to store the current frame. For each cell, the scene also stores a corresponding 64 bit unsigned integer storing the last frame the cell was marked to be visible.

```
class Scene {
  uint64_t frameCounter;
  uint64_t cellLastVisibleFrames[numberOfCells];
};
```

The cell queries are queued up and retrieved next frame. The cell index needs to be stored as well as a reference to the hardware occlusion query for what ever graphics API is used.

```
struct Query {
  int cellIndex;
  void * queryReference;
};
```

The algorithm starts by retrieving results from all queued up cell queries from last frame. If a cell query result was positive, the cell is marked to have been visible during the frame for the query.

```
//retrieve cell queries from last frame
for each query from last frame
  get query result
  if any samples passed
    cellLastVisibleFrames[query.cellIndex] = frameCounter

//now advance the current frame counter
++frameCounter
```

Now the rendering of objects to the z-buffer is about to start, as well as the querying of the traversed cells for this frame. Color write should be disabled.

Blending should be disabled since only solid geometry is rendered during this stage. Depth test should be enabled with a depth function of less than. Depth write should be enabled when rendering objects, and disabled when rendering a cell during a cell query. Backface culling should be disabled when rendering a cell when the camera is inside of a cell or else the cell will not be visible.

```
disable blend
//we do not need to write colors in this phase, just populate the z-buffer
disable color write
enable depth test
depth func less

for each grid cell traversed
  if cell is not empty
    disable backface culling
    disable depth write

    //cell is completely invisible, but still tested in queries
    begin occlusion query
    draw cell
    end occlusion query
    queue up the query to be retrieved next frame

    //if cell was marked visible last frame
    if cellLastVisibleFrames[cell.index] == frameCounter - 1
      for each node in cell
        add node to render queue
        if node has a solid unblended material
          if node marked to always render
              or (sometimes render and numObjectsForDepthPass < 75)
            add node to depth pass queue
            ++numObjectsForDepthPass

      //now that we are drawing the objects, reenable depth write
      enable depth write

      sort depth pass queue by shaders, any needed textures, and meshes
```

```
//use the cell query just performed to do a conditional render and
//avoid drawing these objects if the cell is not visible
begin conditional render using cell query
for each mesh in depth pass queue
  draw mesh
end conditional render from cell query

clear depth pass queue
```

4.3.3 The Rendering

The occlusion querying and depth pre-pass is the heart of the algorithm in this paper. All objects that are most likely visible are now queued up and ready to be rendered in color. They should be sorted by render state as mentioned earlier. This is a great opportunity to use the batching API available on modern graphics cards to reduce draw calls and render all objects of one type in one call. Any kind of renderer can be used at this point. The depth write and depth test should still be enabled at this time since not all objects were added to the z-buffer during the depth pre-pass. The depth test function should be set to less than or equal in order to avoid fragment shader overdraw by using the already populated z-buffer.

4.3.4 Spreading Queries

Tight detailed indoor environments benefit from the scene using a finer grid. The hallway in 4.18 is built around the hangar pictured in 4.17. If the grid is too sparse, large chunks of the hangar get rendered that should be occluded by the walls of the hallway because the cells are too big and contain too many objects.

When inside the hallway of 4.18, the frame rate is about 60 FPS with a fine

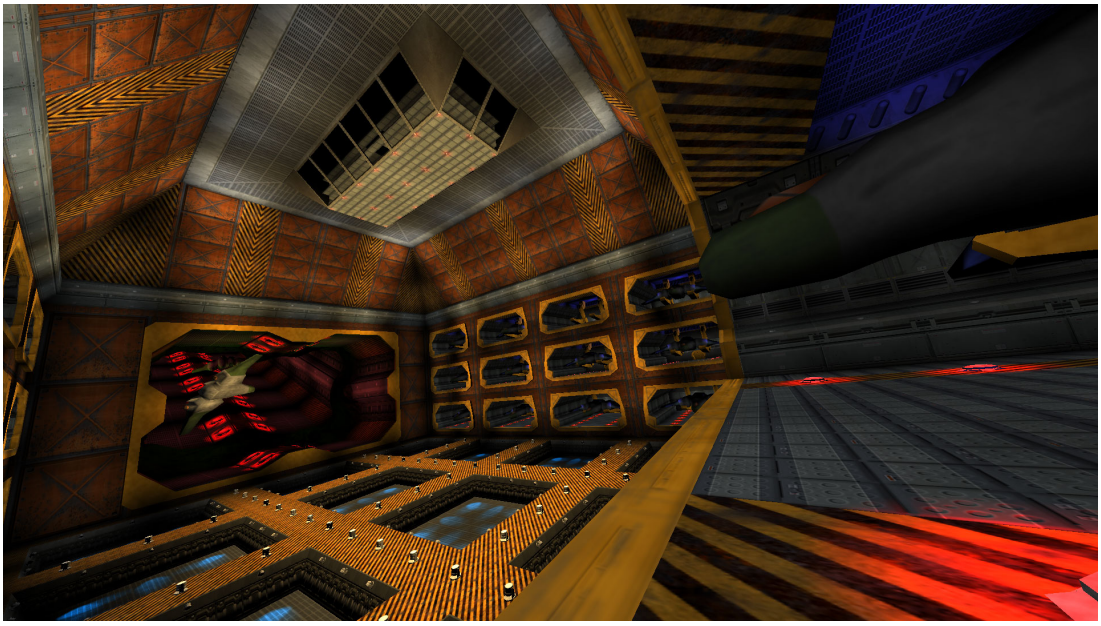

Figure 4.17: Hangar Bay

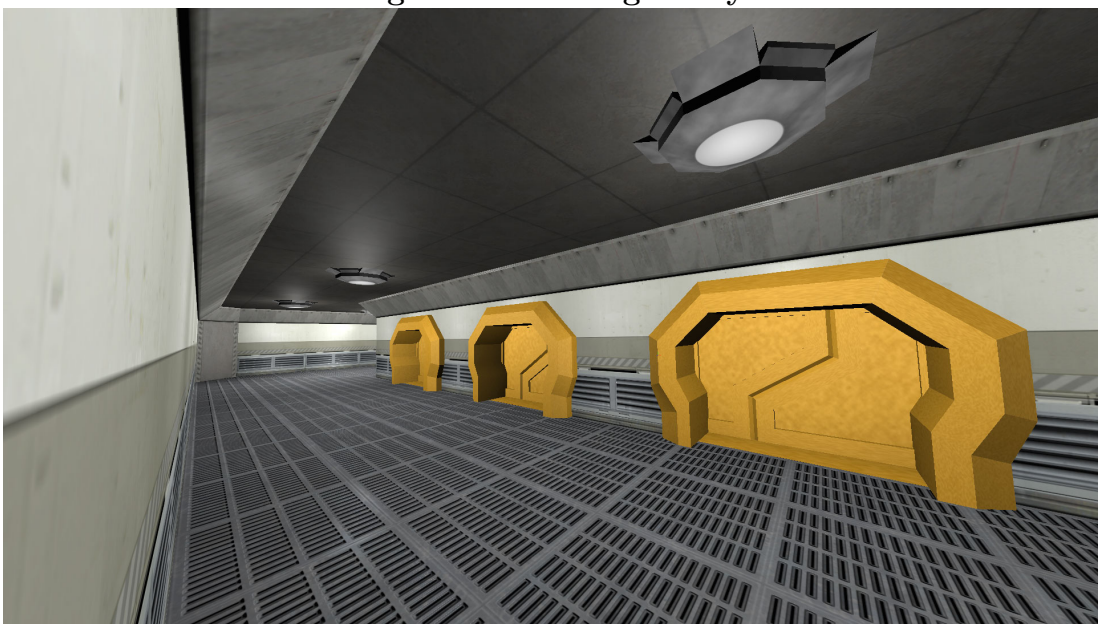

Figure 4.18: Hangar Halls

grid, and can drop down to 25 FPS with a more sparse grid. However, having too fine a grid can also be bad for performance overall. The frustum rasterization algorithm has no problem with traversing grids as large as 1000 x 1000 x 1000 cells. The querying and rendering of individual cells is the bottleneck. Depending on the camera position and viewing angle, the number of traversed cells in a fine grid can range from about 500 to 60,000. There is not a good way to batch all of the cubes that get rendered for the cell queries into a single draw call. The queries happen in between rendering a group of objects to the z-buffer and there seems to be no way to have a query for each individual object in a batch at the moment. In the worst case, that many draw calls being made causes severe performance drops down to about 4 FPS.

One solution is to try to group far away cells into one query. It may be possible to modify the frustum rasterizing algorithm to support this in the future. The grid could be rasterized more like a hierarchical octree. Any far way grouped cells that are visible can then be subdivided into finer cells in later frames to filter out the portions of the scene that are not visible. This would take considerable effort due to the way the rasterizing algorithm is currently built to traverse row-by-row in the grid.

A simpler solution is to limit the number of queries per frame. A limit of about 3000 to 4000 produces good results. Query results now need to last multiple frames. The algorithm is becoming similar to the Coherent Hierarchical Culling algorithm.[21, 30] We are now assuming that results of queries are likely to not change too much between frames that are close by in time.

The current algorithm allows successful visibility queries to last for about 30 frames and failed invisible cell queries to last about 2 frames. A lower number for invisible cells is needed to avoid waiting too long to let previously invisible

cells become visible. Too high a number would cause noticeable object popping in scenes full of motion. Now if a query result is still valid, the cell does not need to be re-queried. Cells that have not been queried in previous frames have a chance to be queried now. This can cause slightly more noticeable pop-in of objects while moving since some cells get queried a few frames later at the cost of keeping the frame rate higher as shown in 4.19. The frame rate being high reduces the real-time it takes for an object to pop-in, making it less noticeable anyway, so this tends to be a good tradeoff.

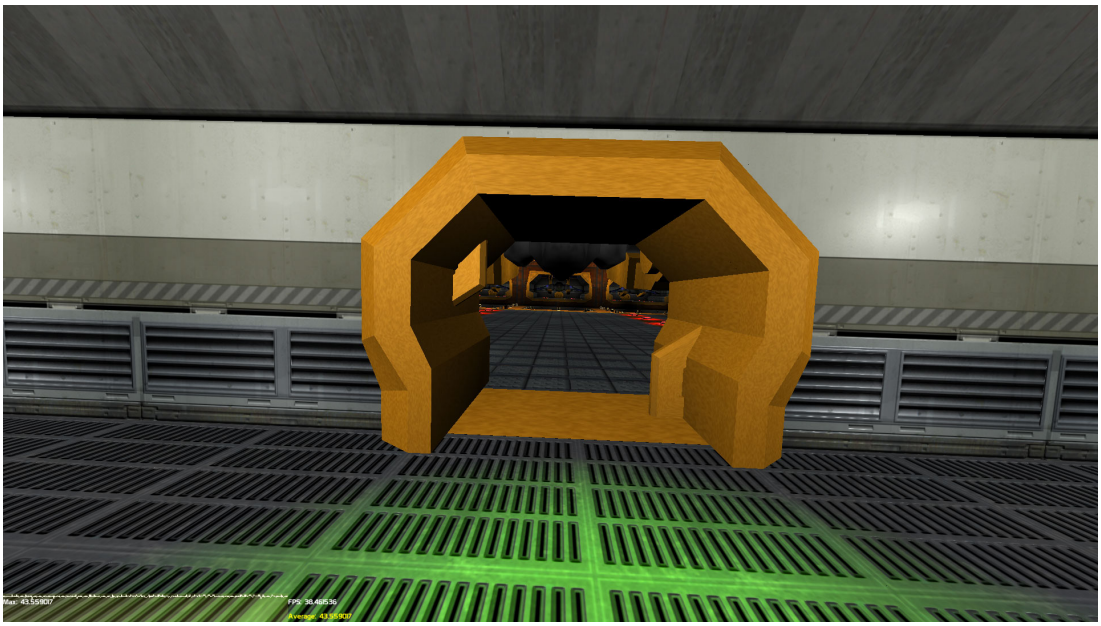

Figure 4.19: Moving sideways can make far away objects seen through the door sometimes appear to pop in visibly if someone is really paying attention. A high frame rate and an action packed game help ensure this artifact is hard to notice.

4.3.5 Query Starvation

When a large amount of cells needs to be queried, some cells never have a chance to be queried and patches of the world go missing as shown in Figure 4.20. Cells early in the frustum traversal whose query results have expired eat up the queries that farther away cells need to become visible. If 40,000 cells need to be queried, and non-visible cell results last only 2 frames, those same non-visible cells will keep executing queries.

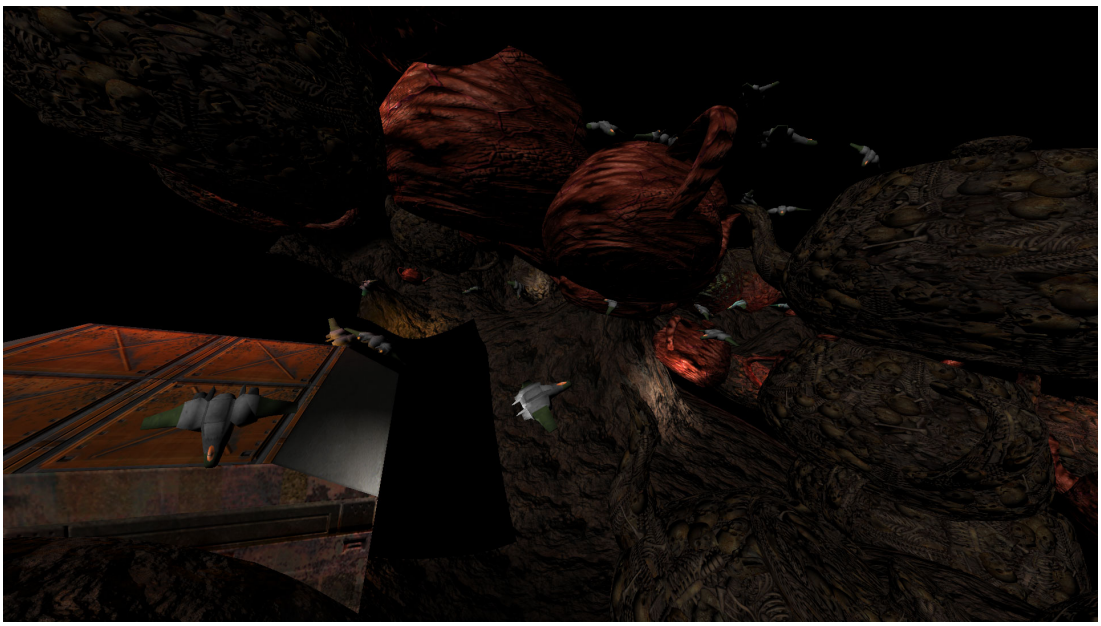

Figure 4.20: Parts of the terrain go missing. Cells containing those objects never have a chance to be queried.

Extending the duration of query results when the query cap is hit, as well as offsetting the duration by a random amount, helps avoid the problem of query starvation. It is possible to keep track of how many frames in a row have hit the query cap and base the duration extension on this amount. After a few frames, all cells that should be visible have had a chance to be queried and the scene

looks correct. The popping artifacts stay relatively unnoticeable.

We are now storing the last frame that the cell query will be valid ahead in time. It is also necessary to store if the cell was visible or not. The 64th bit can just be a one or a zero to store a boolean of the query result. Now there are only 63 used bits to store the query frame, which is still more than enough. The query result storage is illustrated in Figure 4.21.

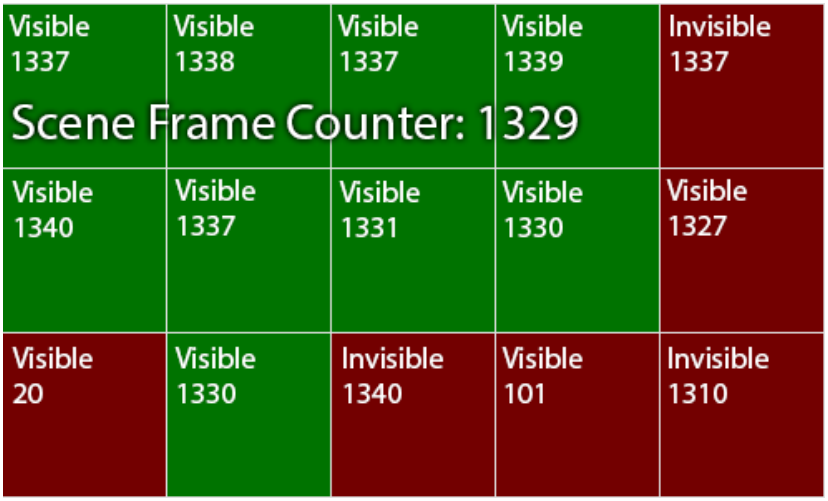

Figure 4.21: For each cell, the query visibility status and frame are stored. Notice how some cells are marked as visible but their frame is less than the scene's frame, meaning the cell is still invisible. It is time to requery that cell.

The following pseudo code shows how the current implementation takes into account the number of frames in a row the query cap was hit. This is a slight modification of the previous pseudo code that was retrieving cell query results. The invisibilityDurationGrowth variable should be smaller than the visibilityDurationGrowth variable to keep the previously described popping artifacts at a minimum while moving.

```
//encodes or decodes the cell's frame given a 64 bit integer
uint64_t codeFrame(uint64_t frame) {
  return 0x7FFFFFFFFFFFFFFF & frame;
}

//encodes the 64th to store a boolean value
uint64_t encodeVisible(bool frame) {
  return frame ? 0x8000000000000000 : 0;
}

//decodes the 64th bit to retrieve a boolean value
bool decodeVisible(uint64_t frame) {
  return (0x8000000000000000 & frame) != 0;
}

//retrieve cell queries from last frame
for each query from last frame
  get query result
  if any samples passed
    cellFrame = frameCounter
      + (visibilityDuration + visibilityDurationGrowth * numFramesOverflowed)
      + rand(0, numFramesOverflowed * 2)

    visible = true
  else
    cellFrame = frameCounter
      + (invisibilityDuration + invisibilityDurationGrowth * numFramesOverflowed)
      + rand(0, numFramesOverflowed * 2)

    visible = false

  //now store the query result and frame
  cellLastVisibleFrames[query.cellIndex] = codeFrame(cellFrame) | encodeVisible(visible)

//now advance the current frame counter
++frameCounter
```

Sometimes a few random objects can start blinking every couple frames because a visibility query has expired. This ends up being noticeable even at high

frame rates. This happens when the query cap has been reached so the visible cell is now invisible until next frame when it can be queried. This is easily avoided by ignoring the visibility cap for visible cells that need to be re-queried. This happens rarely enough that only about 2 to 30 cells may sometimes need to violate the cap and there is no negative impact on performance.

The following pseudo code shows the previous algorithm but taking into account the new modifications.

```
//retrieve cell queries from last frame
for each query from last frame
  get query result
  if any samples passed
    cellFrame = frameCounter
      + (visibilityDuration + visibilityDurationGrowth * numFramesOverflowed)
      + rand(0, numFramesOverflowed * 2)

    visible = true
  else
    cellFrame = frameCounter
      + (invisibilityDuration + invisibilityDurationGrowth * numFramesOverflowed)
      + rand(0, numFramesOverflowed * 2)

    visible = false

  //now store the query result and frame
  cellLastVisibleFrames[query.cellIndex] = codeFrame(cellFrame) | encodeVisible(visible)

//now advance the current frame counter
++frameCounter

disable blend
//we do not need to write colors in this phase, just populate the z-buffer
disable color write
enable depth test
depth func less

//keep track of whether or not we hit the query limit this frame
```

```
recordedOverFlow = false

for each grid cell traversed
  if cell is not empty
    disable backface culling
    disable depth write

    visible = decodeVisible(cellLastVisibleFrames[cell.index])
    frame = codeFrame(cellLastVisibleFrames[cell.index])

    //check if it s time to requery a cell
    if frame <= frameCounter
      if numQueries >= queryCap
        //record that the query cap has been hit for the frame
        if(!recordedOverflow)
          ++numFramesOverflowed
          recordedOverflow = true

      //perform the query if the query cap has not been hit or if the cell was visible
      if numQueries < queryCap or visible
        //cell is completely invisible, but still tested in queries
        begin occlusion query
        draw cell
        end occlusion query
        queue up the query to be retrieved next frame

    //check if a cell is marked as visible and the result is still valid
    if cellLastVisibleFrames[cell.index] >= frameCounter
      for each node in cell
        add node to render queue
        if node has a solid unblended material
          if node marked to always render
              or (sometimes render and numObjectsForDepthPass < 75)
            add node to depth pass queue
            ++numObjectsForDepthPass

      //now that we are drawing the objects, reenable depth write
      enable depth write

      sort depth pass queue by shaders, any needed textures, and meshes
```

```
    //use the cell query just performed to do a conditional render and
    //avoid drawing these objects if the cell is not visible
    begin conditional render using cell query
    for each mesh in depth pass queue
      draw mesh
    end conditional render from cell query

    clear depth pass queue

  if numQueries < queryCap
      numFramesOverflowed = 0
```

4.4 Supporting Multiple Viewports

Some bookkeeping is necessary for keeping track of visibility queries between frames. To support split screen multiplayer with multiple viewports, a separate set of query data needs to be allocated. Each cell stores a 64 bit unsigned integer. When allocating a new viewport, a new array of 64 bit integers per cell needs to be allocated.

Since there is a limit to the number of queries per frame, the viewports would have to share that limit and distribute the number of queries in some way. Draw distance may have to be lowered to reduce the number of draw calls from objects rendered as well as the number of cells that need to be queried. We leave a solution for how to distribute the queries between viewports for future work.

4.5 Review

This is a recap of the algorithm presented in the section. First we retrieve queued up occlusion query results from the last frame. Next we traverse the cells of the scene's uniform grid in occlusion order using a 3D convex polygon rasterizing algorithm. For each cell traversed, we perform an occlusion query for the cell. If the cell was determined to be visible in previous frames, objects from that cell are rendered to the z-buffer only, and queued up to later be rendered in color. Once the scene is traversed, the objects can now be rendered in color.

Chapter 5

Results

5.1 Environment

The current implementation does not take advantage of multithreading due to the difficulty in trying to have a good multithreaded design while experimenting with different ideas, so the number of cores is not as much a deciding factor in performance as the speed of the processor itself.

The system this was tested on is a custom built desktop tower with an Intel i5-750 processor running at 2.67 GHz and is running Windows 8 64 bit. It has 6 Gigabytes of RAM. The graphics card is an EVGA Geforce GTX 480 with Geforce 311.06 drivers installed and 1.5 Gigabytes of video memory.

SDL 2.0 revision 7140 is used as the window management library. The glm library was used for all of the 3D math. DevIL was used to help load various texture formats. The engine runs with OpenGL 3.3 features while also using deprecated features like immediate mode to make it easier to draw debug visualizations. GLEW 1.9.0 was used to help with the OpenGL function bindings.

The tested executable is built for 64 bit using Visual Studio 2012.

A limited deferred shading renderer is used to display the scenes. A deferred shading renderer performs the lighting in a separate pass from rendering the geometry, allowing for a very large number of dynamic lights. The renderer is capable of performing normal mapping, specular highlights, and rendering only solid geometry. It is able to light the scene with point lights, spot lights, and a directional volume light for lighting up large areas. More features are planned for the future.

5.2 Test Scene

The engine treats 1 unit as 1 meter. The scene was designed keeping in mind the average height of a person being 1.7 meters or units. 3DS Max 2012 and a custom Max Script were used for exporting the scene into a format readable by the engine.

The scene is a large outdoor area with a detailed underground indoor area. Figures 5.1, 5.2, 5.3, 5.4, 5.5 show the different parts of the scene. The scene has also been pictured throughout the paper in Figures 1.4, 1.5, 4.15, 4.16, 4.17, 4.18, and 4.19. All objects are textured, causing extra stress to the deferred shading renderer by making it use more demanding shaders and sorting by more render states. The textures are borrowed from the Perfected Doom 3 mod by Vgames, who made high resolution versions of textures for id Software's game Doom 3.[35] A large number of point lights and spot lights are used for the hangar area. The outdoor area is a combination of several directional volume lights and randomly placed point lights. There is currently no heightmap terrain rendering system in place, so the terrain is composed of a set of meshes from a model that was split

into 100 pieces as a temporary solution. This scene is approximately 1250 x 750 x 1250 meters in size.

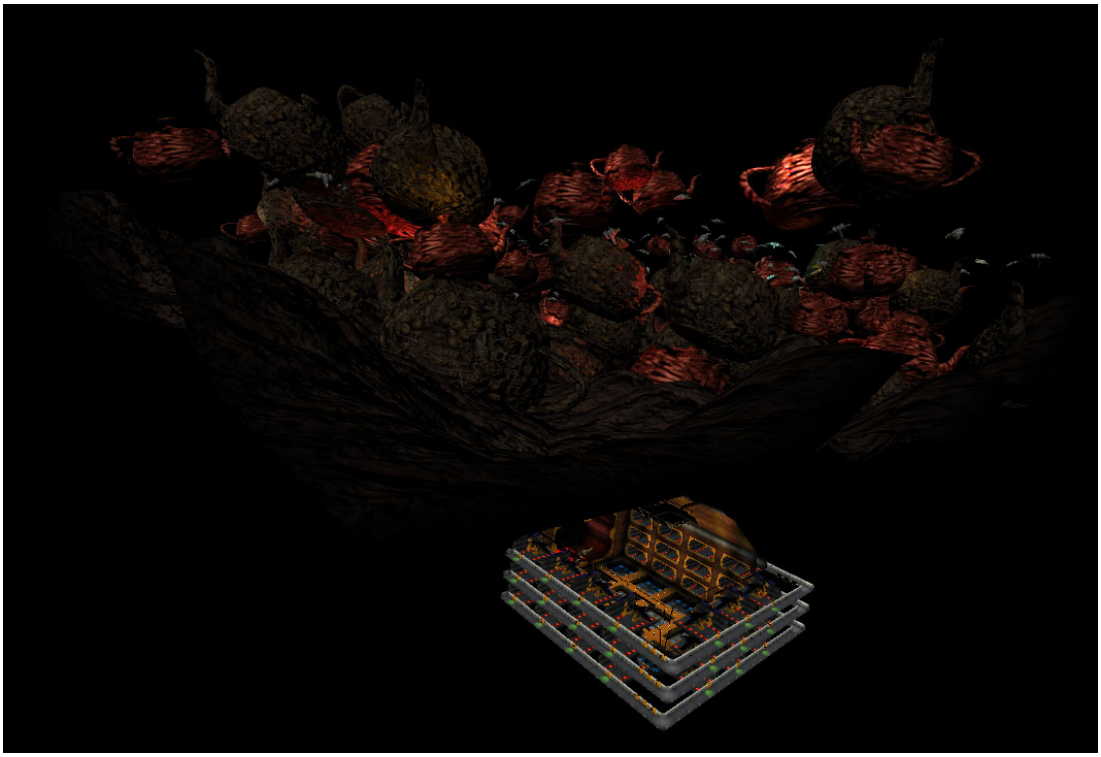

Figure 5.1: Overview of Teapot Graveyard.

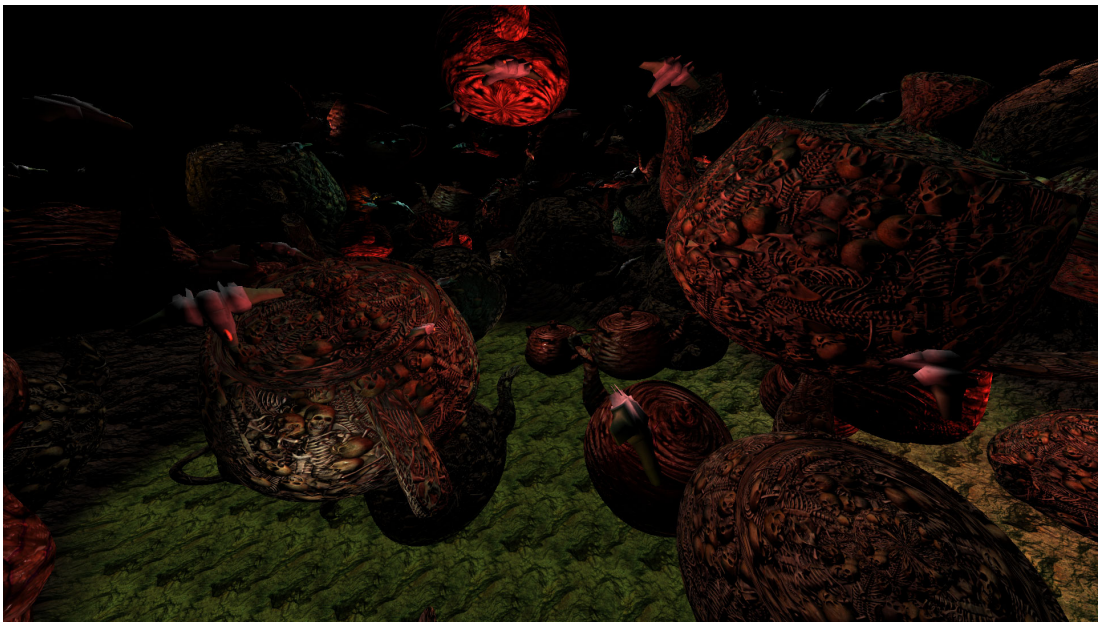

Figure 5.2: A view of the outdoor portion of the Teapot Graveyard map.

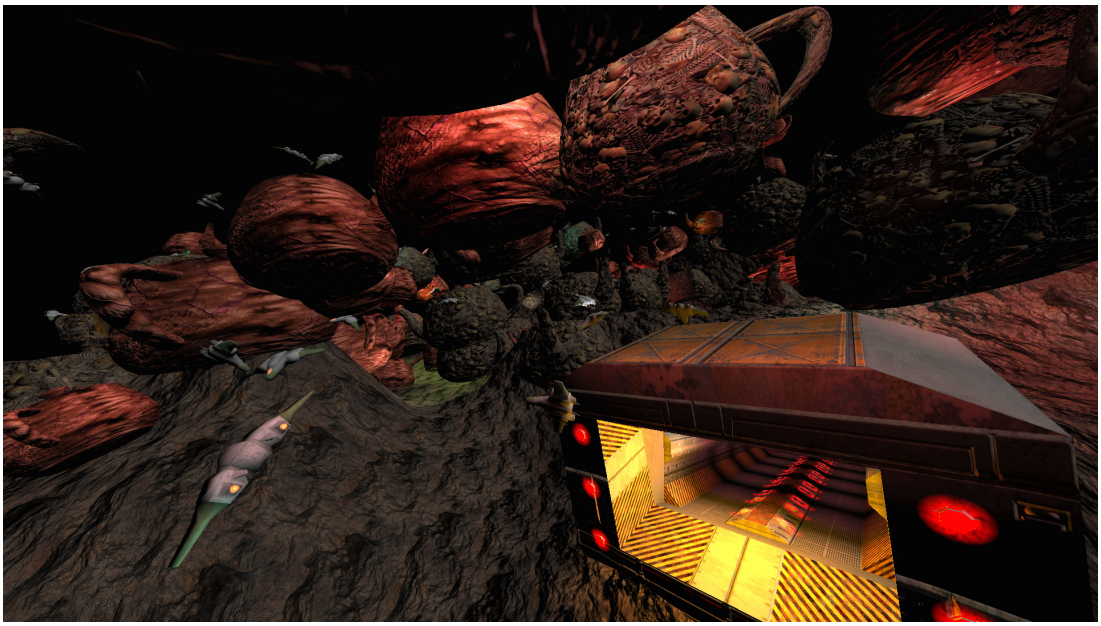

Figure 5.3: Entrance into the Hangar area.

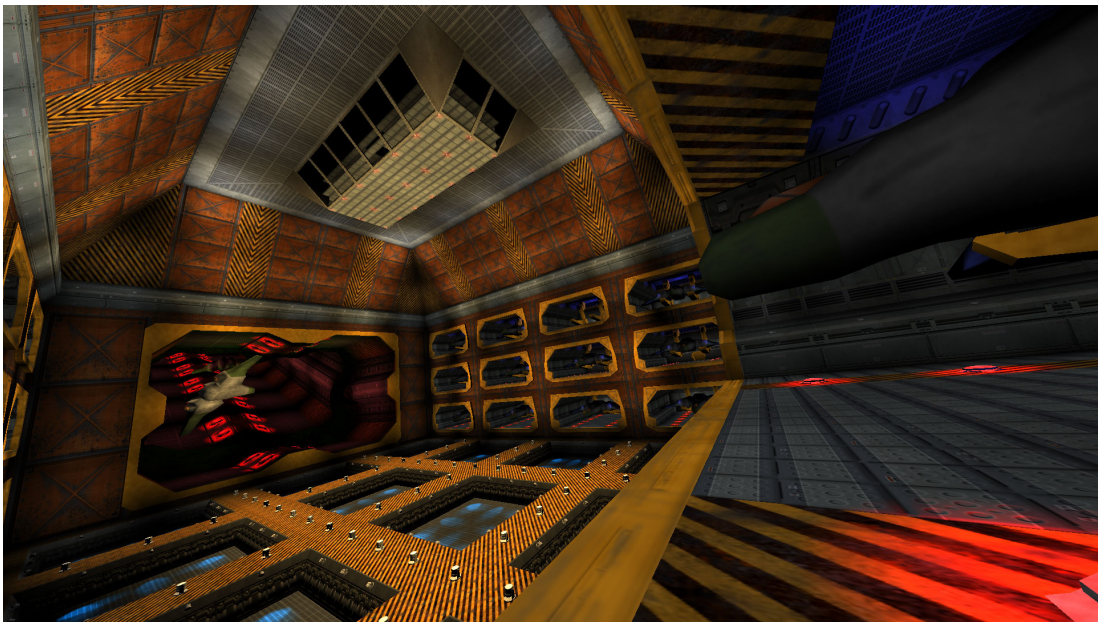

Figure 5.4: A view of the underground Hangar of the Teapot Graveyard.

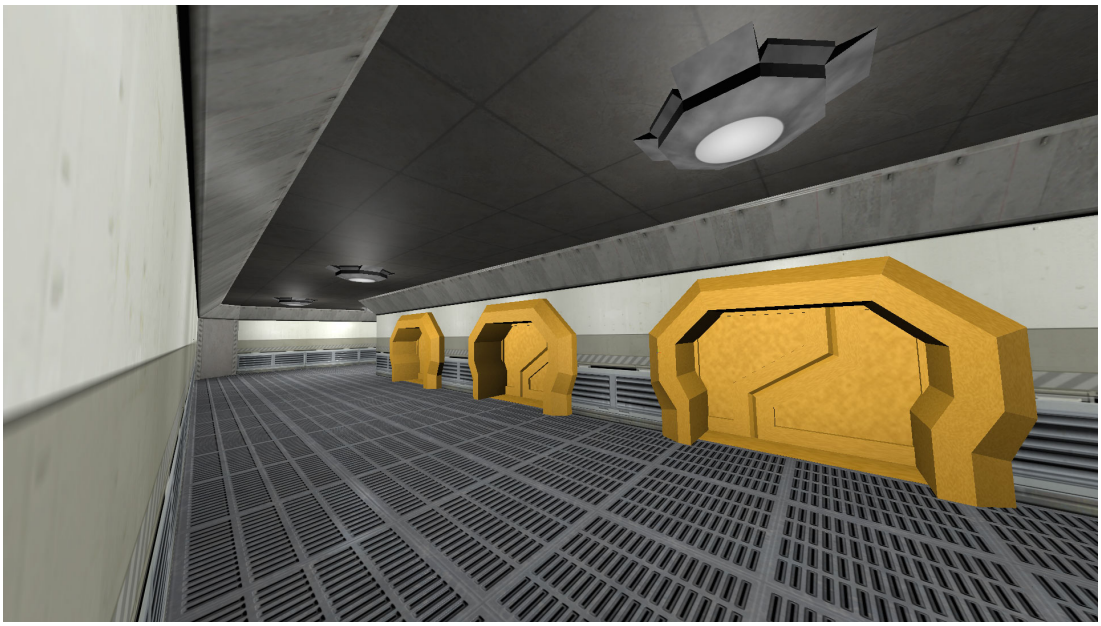

Figure 5.5: Side hallways of the Hangar in Teapot Graveyard.

5.3 Data

5.3.1 Framerate

The scene was tested with various grid dimensions and with two different query caps. The aim is to find the best overall framerate, while keeping it above 30 FPS at all times. The framerate changes depending on where the camera is looking in the scene. It is better to have one area running at 40 FPS and another running at 35 FPS than to have one area running at 100 FPS and another running at 10 FPS. A framerate of 100 FPS is not much more noticeable than 40, while a framerate of 10 would provide a poor experience.

5.3.2 Grid Dimensions

The grid dimensions affect how well parts of the scene are occluded. The more a scene is subdivided, the less objects there are in the cells. This can positively affect performance by drawing less objects.

More cells also means more queries need to be executed for the same draw distance. In this way more subdivision can also negatively affect performance due to the number of queries being executed. This can also cause more noticeable popping since the query cap can be hit, requiring a longer time until some objects become visible.

5.3.3 Maximum Queries Per Frame

The query cap is meant to improve performance for high density grids by spreading the queries out over multiple frames. Lower numbers improve perfor-

mance, but cause more noticeable popping than higher numbers.

5.3.4 The Testing Method

To test the engine, a flythrough is recorded storing camera positions and orientations at different times. The engine plays this flythrough back, ensuring the same areas are revisited at every test during the experiment. Various statistics are recorded each frame and logged so they can be displayed in graph form in Figures 5.6 and 5.7. Figure 5.8 shows the framerate relative to the average of the entire run.

The graph shows the performance metrics at different points in time during the flythrough. Portions of the graph are labeled to describe the general area the camera was at that moment.

Outside Area

This is the large outside area that makes up the majority of the map. Most of the scene is visible out here, causing a large portion of the scene to be drawn. However, the outdoor area is also not as detailed as the underground hangar.

Cave

There is a tiny underground cave to simulate a tight indoor environment. Very few objects get rendered when the camera is located here since so little of the world is visible.

Hangar Tunnel

This is the portion of the map that joins the outdoor area and the indoor portion of the hangar. This is often an area that causes poor performance under sparser grids. With tighter grids, only a small portion of the hangar gets drawn when looking down the tunnel. When the grid is sparse, the entire hangar gets drawn, causing a very low framerate.

Hangar

This is a large open indoor area with many objects and lights to provide high detail. This is an area of the map that often causes the most stress due to the large amount of objects to render.

Halls

These are halls that border outside of the hangar itself. With sparser grids, more of the hanger gets rendered even when just looking right at a wall. Tighter grids typically provide the best performance in this area since more objects get occluded.

Door Test

In this portion of the flythrough, the camera is positioned in the hallway looking through a doorway out into the hangar. The camera then moves side to side so that it looks at the wall around the doorway for a bit. This is a good test of how noticeable popping artifacts are since objects may pop in when the camera suddenly can see into the hangar when it was previously looking at a

wall.

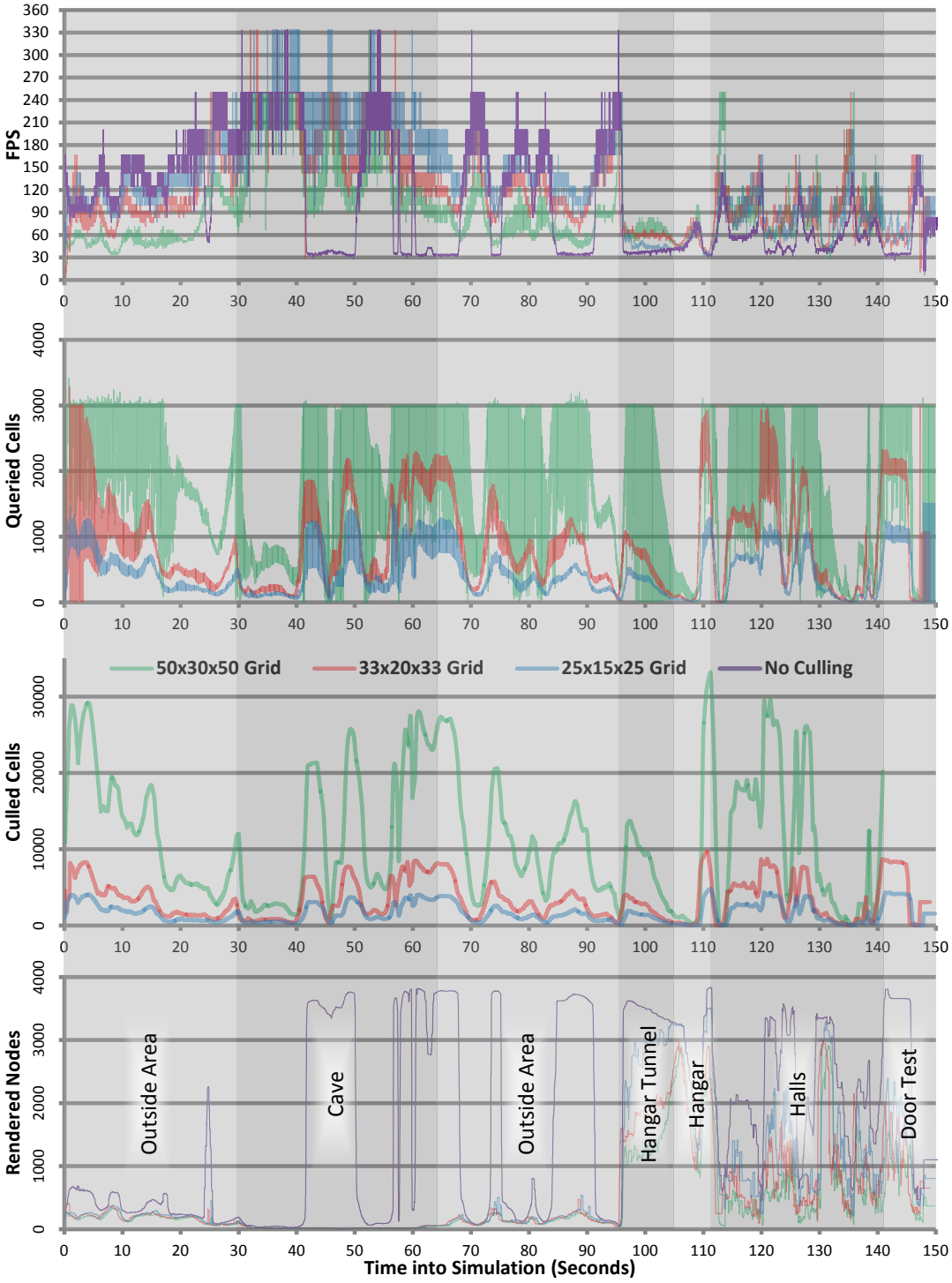

Figure 5.6: Data for a scene with a query cap of 3000.

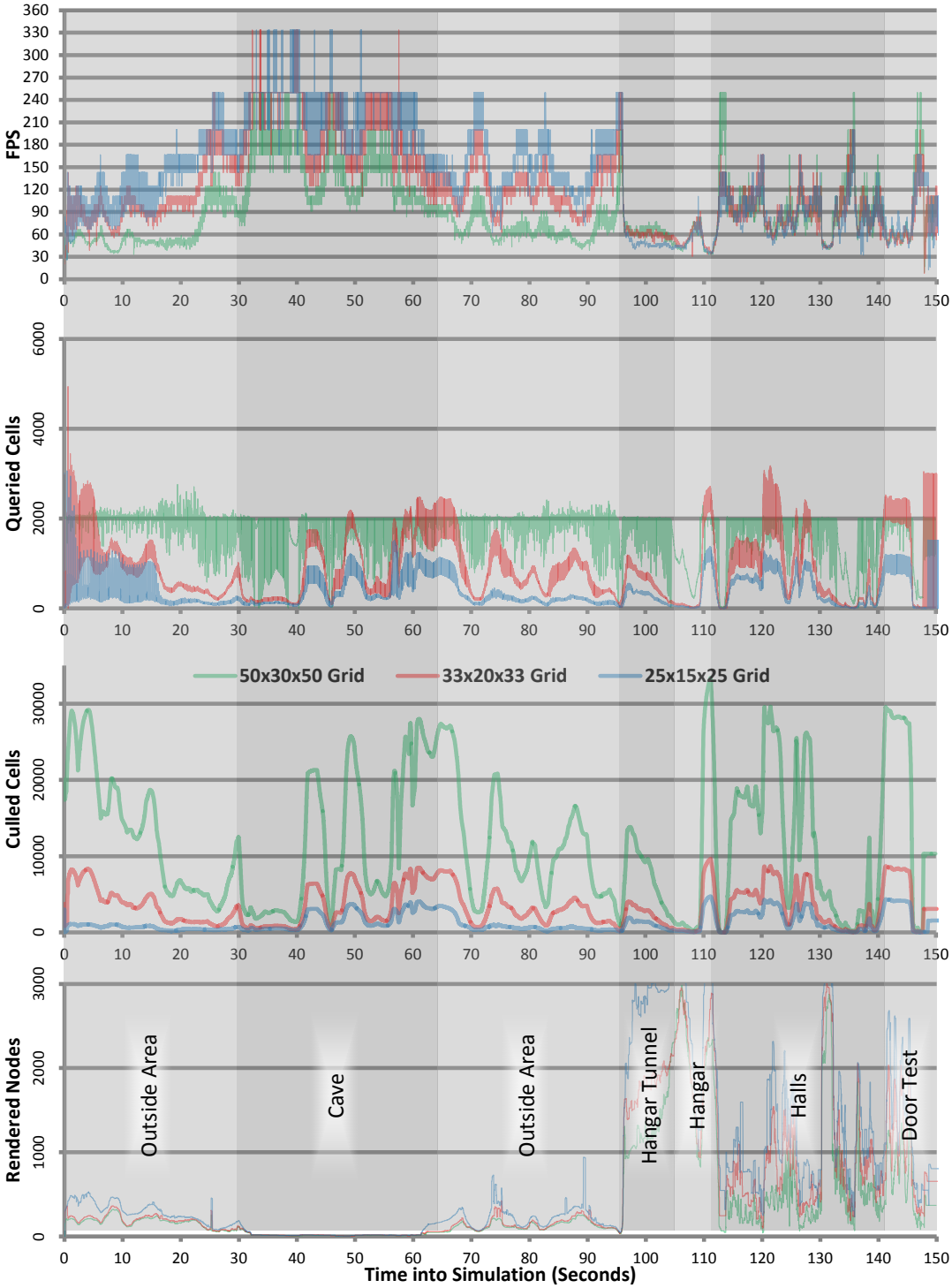

Figure 5.7: Data for a scene with a query cap of 4000.

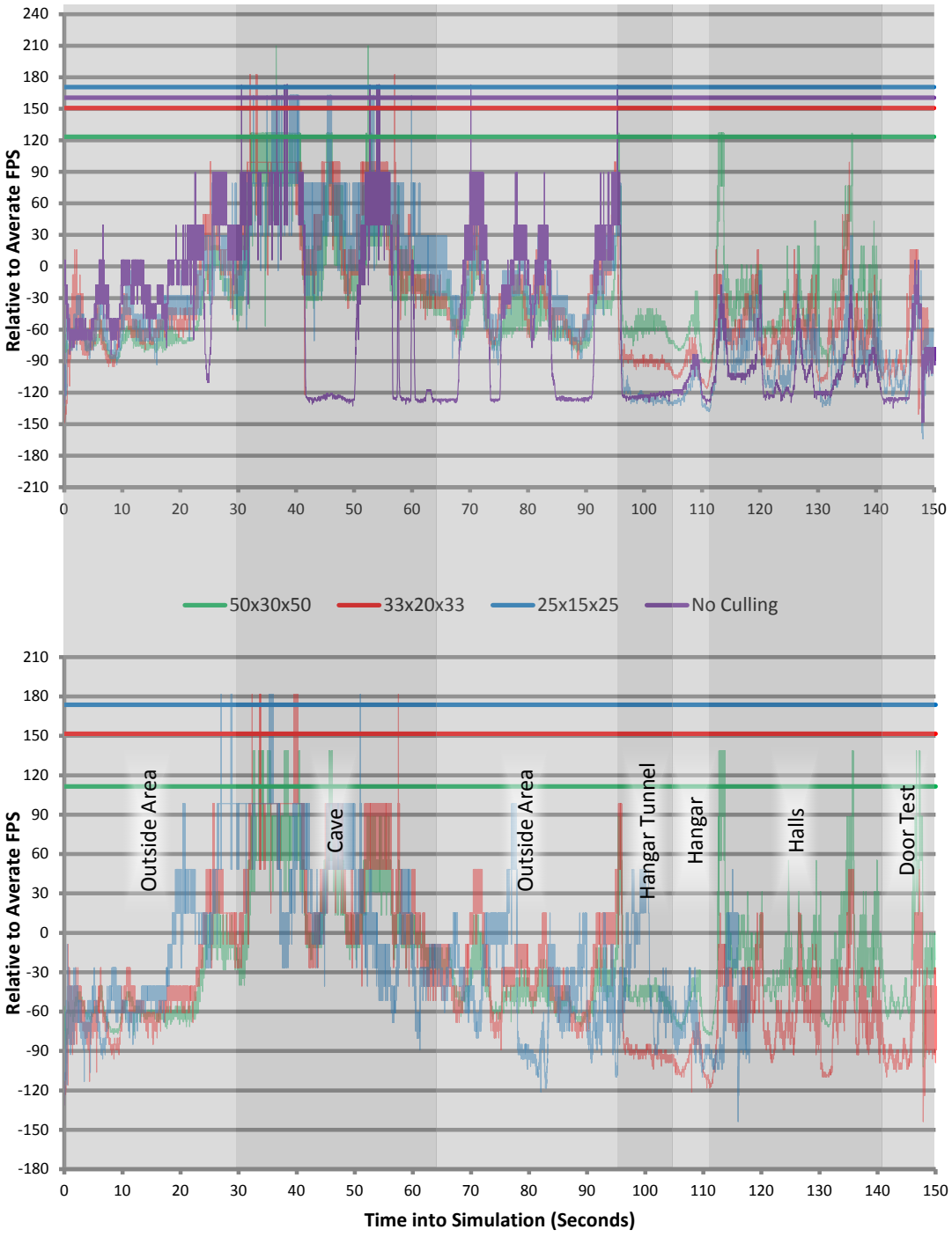

Figure 5.8: Showing the framerate relative to the average framerate. The top graph is for a query limit of 3000 and the bottom graph is for a query limit of 4000. The horizontal lines mark the average framerates.

5.4 Conclusions

These results show that there is a relationship between the framerate and the number of objects rendered. It is hard to see a difference in performance between a query cap of 3000 and 4000 from the graphs, but visually, there are a few areas that produce popping artifacts more noticeably with a query cap of 3000. The scene was also tried with a query cap of 2000 but the popping artifacts were too noticeable leading to unacceptable compromises in visual quality.

When occlusion culling is turned off, the framerate can fluctuate between high values and back down to values around 30 and at times go even lower. The framerate can be seen to dip when the number of rendered nodes jumps up to high values. The graph in Figure 5.8 shows that with occlusion culling turned off, the framerate tends to jump down to below average and fluctuate much more noticeably. This would create very jittery performance which is undesireable. If the scene had more objects to add even more detail, occlusion culling being off would produce even worse performance.

The different grid sizes can be seen to produce different performance depending on if the camera is looking at outdoor portions of the scene or at indoor portions. The sparser grid performs better outside, while the tighter grid performs better inside. The grid size in between seems to produce a more even framerate which is typically desirable.

These tests have shown that the visibility culling algorithm presented in this paper is effective for both indoor and outdoor environments. Although hardware occlusion queries can cause CPU stalls, using them wisely can be very beneficial. By taking advantage of spatial and temporal coherence, we were able to push performance out of an often overlooked feature of the graphics pipeline.

Chapter 6

Future Work

As mentioned earlier, the current deferred shading renderer is limited to rendering only solid objects. Static multimaterial triangle meshes are the only type of object renderable, which was enough to create the detailed scenes used for testing the algorithm. Transparent objects would require a forward shading renderer. The approximate front to back frustum traversal algorithm would work great for sorting alpha blended objects in the correct order since only occlusion order is needed, not true depth order. Skeletal animations, particle systems, shadows, and various post processing effects would add even more stress to the engine, giving more opportunity to find ways to optimize the algorithm.

Cascaded shadow maps are often used for large outdoor environments to cast shadows from sunlight or other global outdoor sources.[1] The paper mentioned earlier a method for supporting multiple viewports for split screen multiplayer in section 4.4. A viewport can be created for the point of view of the outdoor global light source since a large portion of the scene needs to be rendered to the shadow map. For smaller light volumes, it is best to avoid using the hardware occlusion queries and to just render objects that are within the view frustum

from the light's point of view. The bookkeeping required for each viewport that uses this algorithm would be impractical for that many lights.

The graphics engine was originally a fully featured multi threaded game engine. It was driven by a combination of Nvidia PhysX 3.0 and Intel Thread Building Blocks. Those features were stripped out for the current implementation in favor of a single threaded graphics only engine in order to avoid being distracted by other features. Seeing a fully functional world with moving objects being driven by this graphics engine would be a great stress test. Being in an action packed world would likely reduce how noticeable popping is. The CPU stall from gathering query results from last frame mentioned earlier would be less impactful since the engine has more work to do. The query results would likely be ready by the time the engine is done updating the rest of the game. There may also be room for parallelizing some portions of this algorithm, such as the sorting of alpha blended objects mentioned earlier.

There are some portions of the scene where the culling algorithm brings down the framerate rather than boosting it. This is usually in areas of the scene where there are very few objects on the other side of a wall. It is strongly dependent on the camera position and viewing angle. It may be worth finding a way to turn culling off in some areas for certain points of view, especially if this is an area where the environment will not change too much. This can possibly be done by comparing the number of non-empty cells versus the number of cells traversed. If the number of non-empty cells is a low enough percentage, the algorithm can cache the approximate camera position and viewing direction to be reused in later frames to increase performance in those areas.

It should also be possible to keep a hierarchy of grids to support very large environments with uneven distribution of geometry as shown in Figure 6.1. The

top level of the hierarchy is very sparse and is used to organize the grids beneath it. The leaves of the hierarchy are the grids themselves and can be at varying degrees of sparseness. The frustum rasterizing algorithm first traverses the outermost grid using the front to back heuristic. For each cell in the outermost grid, the frustum is clipped down to fit within the cell. Then the front to back heuristic is performed on the grid contained within the outer grid cell.

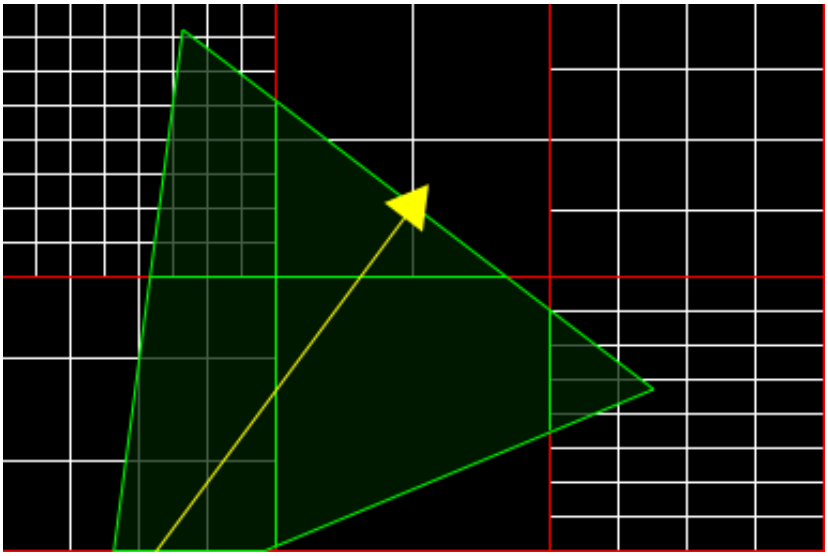

Figure 6.1: A grid hierarchy.

Appendix A

The 3D Rasterizing Algorithm

Here we go over the 3D rasterizing algorithm used for scene traversal in greater detail. This algorithm works on any convex 3D mesh. The view frustum happens to be a convex 3D mesh, but is usually split into pieces to create smaller convex meshes.

A.0.1 Mesh Edge List

First a mesh edge list is needed. When dealing with view frustums, the eight points that make up the shape are needed. The frustum consists of six planes. Each point is derived from finding the intersection point between the three corresponding planes adjacent to that point. Once the points are found, a list of edges that joins those points gets built with an example shown in Figure A.1. The edge list is hardcoded and not computed automatically since we know the shape of the frustum. For example, an edge is formed between the top-left-near point and the top-left-far point. This mesh edge list is now the stored representation of the convex 3D mesh that will be rasterized.

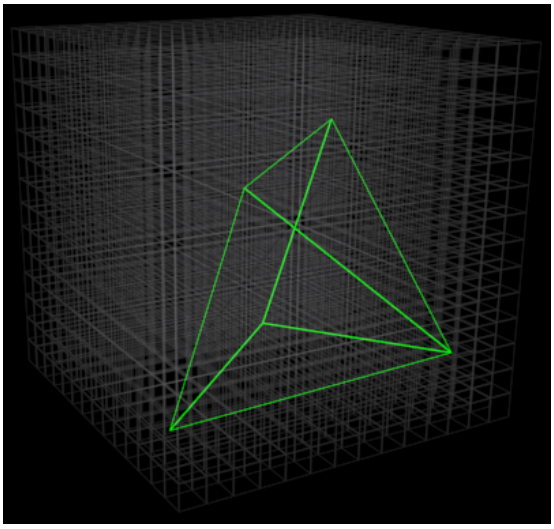

Figure A.1: A view frustum converted to a point edge list.

A.0.2 Clipping Algorithm

The mesh edge list needs to be clippable by a plane to remove portions of the mesh or to split it into pieces. After the view frustum is turned into a mesh edge list, it needs to be clipped against the bounds of the 3D uniform grid to avoid traversing cells outside of the scene bounds. Section 4.2.2 also relies on this clipping algorithm when splitting the frustum into pieces.

This is done by first finding which side of the splitting plane all of the points are on. The points and all other geometry behind the splitting plane get discarded. To split the mesh without discarding what is on the back side, the algorithm is performed twice on the original mesh with two oppositely facing planes.

Now there is a list of which points are on the back side of the plane. For the corresponding edges whose other point is on the front side of the plane, the edge is clipped by the plane. This creates a new point for the edge, and the old point

is discarded. If both points of an edge are on the back side of a plane, the entire edge is discarded.

This now leaves a partly complete convex mesh edge list. New edges need to be computed between the newly created points to close the shape and form the new face. This is done by performing a 2D convex hull algorithm projected onto the clipping plane. The monotone chain convex hull algorithm finds edges between the new points.[3]

A.0.3 World to Algorithm Space

The view frustum needs to be traversed in an order relative to the view. However it is easier to write the rasterizing algorithm to not know about this traversal order. The slices are always assumed to go down the z axis. The slices are then rasterized left to right along the x axis and bottom to top along the y axis.

This means that the points of the mesh edge list need to be converted to a different coordinate space. For example, if the camera is looking down the x axis in the world, the frustum is sliced down the x axis as well. Since in algorithm space, the slices go down the z axis, the x coordinates and the z coordinates of the points in the mesh edge list get swapped.

As the convex mesh is rasterized in algorithm space, the grid cells are computed in algorithm space as well. The grid cells then need to be converted back to world space to map to the grid cells in the world that are being traversed by the algorithm.

A.0.4 Slicing The Mesh Edge List

The 3D mesh edge list now has slices of it taken and then projected into 2D to later be rasterized with a 2D polygon rasterizing algorithm. This is done in increments based on the grid size as shown in Figure A.2.

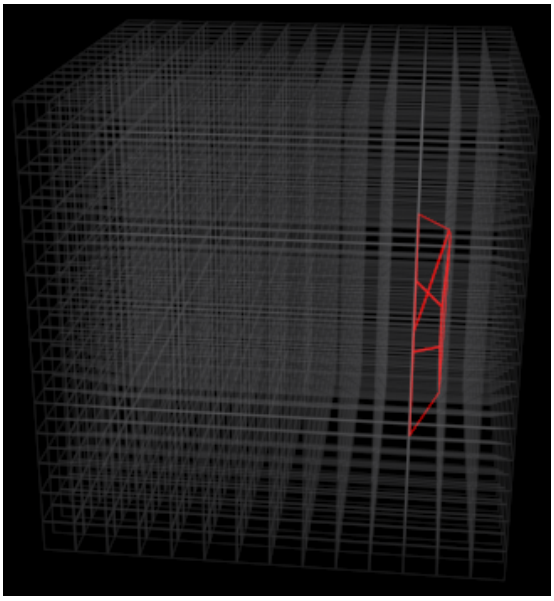

Figure A.2: A mesh edge list slice with the dimensions of the grid.

The mesh edges that intersect with the grid edges get clipped to leave behind sets of points. These points are projected into 2D and then the monotone chain algorithm is used again to create a 2D convex mesh as shown in Figure A.3.

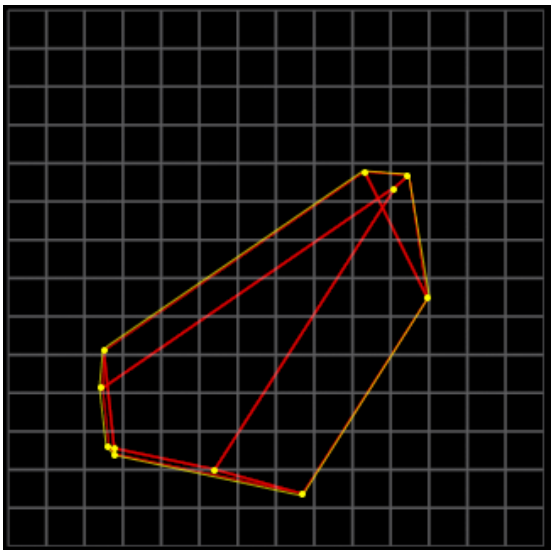

Figure A.3: The slice and its points projected to 2D. The yellow lines show the convex polygon that will be rasterized after performing the monotone chain convex hull algorithm on the projected points.

A double buffer of point lists is kept. The intersection points from the front side of the last slice are kept to now be used as the back side of the current slice. This avoids unnecessary recomputation of the intersection points against edges of the mesh if those points were already computed last slice.

First the active edges to be clipped are found to avoid checking all edges every slice. This is done initially by finding all points within the first slice being rasterized. Those points get added to the point buffer.

The corresponding edges from those points now become the active edges that will be clipped. For each newly found edge, the number of slices until the edge's other point gets computed. A counter is stored for each active edge, and is counted down at each slice. Once the counter runs out, the corresponding edges for that other point become the new active edges. When the counter runs out that point also gets added to the point buffer for the current slice in order to

potentially contribute to the shape of the 2D rasterized polygon built from the slice.

A.0.5 Rasterizing The Slices

At this point it can be possible to use standard 2D polygon rasterizing algorithms, but a few edge cells may sometimes be missed. To ensure all cells are traversed that intersect with the edges, a custom 2D rasterizing algorithm is used. The 2D convex polygon found in Section A.0.4 is scan converted row by row in the grid as shown in Figure A.4. The range of cells to be rasterized for each row needs to be found.

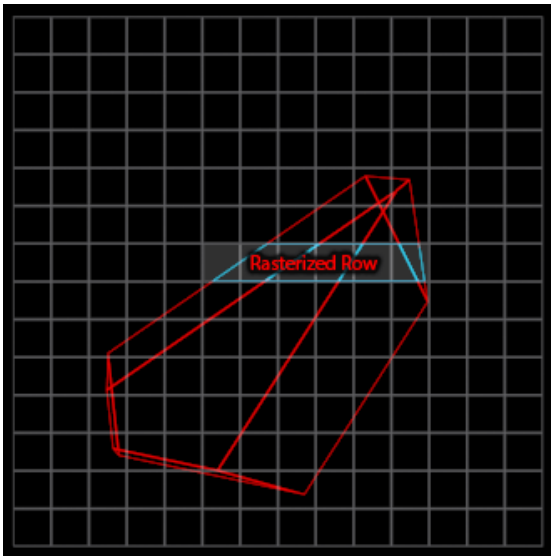

Figure A.4: A row of the slice being rasterized.

The monotone chain algorithm used in the previous step in Section A.0.4 gives two point lists. The lists are in an order that represents a chain of edges. One is the left side of the polygon, and the other is the right side. The left side and right side are traversed independently.

The active edge for each side is found by starting at the beginning of the list. Next, the number of rows is found until the next point in the point list. Again, a countdown is formed until the end of the current active edge based on the number of rows until the next point. Now, if an edge is facing outward, it is clipped against the top of the row and if facing inward it is clipped against the bottom. The range of cells to rasterize for the row is derived from these points as shown in Figure A.5 by computing which grid cell the point is in.

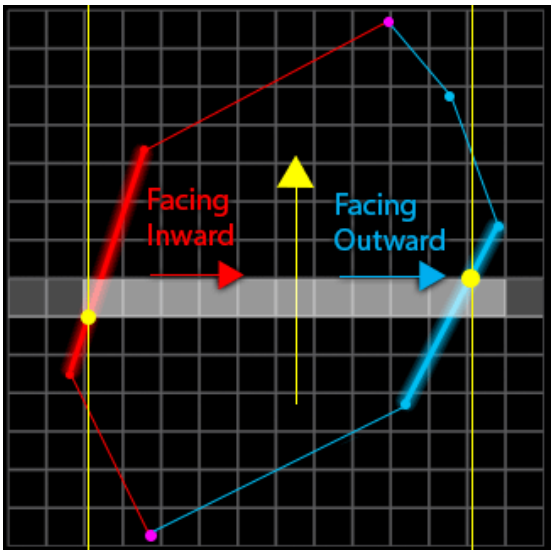

Figure A.5: The two sides of the polygon are being clipped against the row bottom and top. Here the left red side is facing inward, while the right blue side is facing outward. The active edges are boldened.

As the point list is being traversed, the edge may switch from facing outward to facing inward. Due to being a convex polygon, the edge will never face outward again. The point between an outward facing edge and an inward facing edge as shown in Figure A.6 is now used to determine the row range by computing which grid cell the point is in.

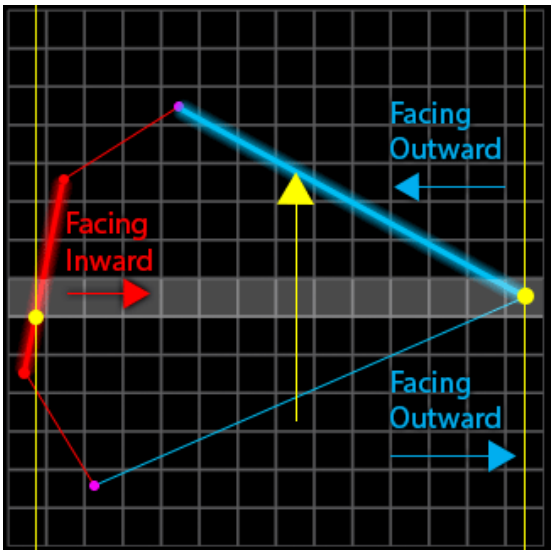

Figure A.6: Now the point between two active edges is used to determine the row range when the edge orientation changes.

Bibliography

[1] Cascaded shadow maps. http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx. Accessed: 2013-05-27.

[2] *Chapter 29. Efficient Occlusion Culling.*

[3] Monotone chain convex hull. http://www.algorithmist.com/index.php/Monotone_Chain_Convex_Hull.

[4] Opengl overview. http://www.opengl.org/about/.

[5] *OpenGL—Android Developers.*

[6] View frustum culling. http://www.lighthouse3d.com/tutorials/view-frustum-culling/.

[7] Viewing frustum. http://www.pcmag.com/encyclopedia/term/61771/viewing-frustum.

[8] *OpenGL 4 Reference Pages glDepthFunc*, 2006.

[9] Portal culling. http://www.3dkingdoms.com/weekly/weekly.php?a=26, 2006.

[10] Battlefield 3. http://www.battlefield.com/battlefield3, 2011.

[11] Borderlands 2. http://www.borderlands2.com/, 2012.

[12] *Culling Explained*, 2012.

[13] *Graphics Pipeline*, 2012.

[14] *Immediate and Deferred Rendering*, 2012.

[15] Getting started with direct3d. http://msdn.microsoft.com/en-us/library/windows/desktop/hh769064(v=vs.85).aspx, 2013.

[16] *Intel Threading Building Blocks Documentation*, 2013.

[17] *OpenGL ES Programming Guide for iOS*, 2013.

[18] T. Aila and V. Miettinen. dpvs: an occlusion culling system for massive dynamic environments. *Computer Graphics and Applications, IEEE*, 24(2):86–97, 2004.

[19] H. Batagelo and W. Shin-Ting. Dynamic scene occlusion culling using a regular grid. In *Computer Graphics and Image Processing, 2002. Proceedings. XV Brazilian Symposium on*, pages 43–50, 2002.

[20] J. Bittner and M. Wimmer. *Chapter 6. Hardware Occlusion Queries Made Useful*, chapter 6. GPU gems 2 : programming techniques for high-performance graphics and general-purpose computation. 2005.

[21] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, Sept. 2004. Proceedings EUROGRAPHICS 2004.

[22] D. Blythe. The direct3d 10 system. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 724–734, New York, NY, USA, 2006. ACM.

[23] P. N. Chaudhuri. Implementation of the dda line drawing algorithm. 2012.

[24] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, I3D '97, pages 83–ff., New York, NY, USA, 1997. ACM.

[25] D. C. (Dice). Culling the battlefield: Data oriented design in practice. 2011.

[26] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 247–254, New York, NY, USA, 1993. ACM.

[27] S. Hill. Rendering with conviction. 2010.

[28] P. Kim and D. Barrero. Rendering tech of space marine. Korea Game Conference 2011, 2011.

[29] D. Luebke and C. Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, I3D '95, pages 105–ff., New York, NY, USA, 1995. ACM.

[30] O. Mattausch, J. Bittner, and M. Wimmer. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008)*, 27(2):221–230, Apr. 2008.

[31] H. Samet. Applications of spatial data structures. Addison-Wesley, 1990.

[32] F. Sanglard. Doom 3 source code review. http://fabiensanglard.net/doom3/index.php, 2012.

[33] S. J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, University of California at Berkeley, 1992.

[34] N. K. Tiago Sousa and N. Schulz. Secrets of cryengine 3 graphics technology. In *ACM SIGGRAPH 2011 Courses, Advances in Real-Time Rendering in 3D Graphics and Games*, SIGGRAPH '11, 2011.

[35] Vgames. Perfected doom 3 texture pack v2.0. http://www.moddb.com/mods/perfected-doom-3-version-500/addons/perfected-doom-3-texture-pack-v20.

[36] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), Jan. 2007.

[37] C. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, 2011.

[38] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff, III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 77–88, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.