Massive model visualization: An investigation into spatial hierarchies

by

Jeremy Scott Bennett

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Human Computer Interaction

Program of Study Committee:
James Oliver (Major Professor)
Yan-Bin Jia
Eliot Winer

Iowa State University

Ames, Iowa

2009

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The current generation of visualization software is incapable of handling the interactive rendering of arbitrarily large models. While many solutions have been proposed for Massive Model Visualization, very few are able to achieve the full capabilities needed for a computer visualization solution. In most cases this is due to overly complex approaches that, while achieving impressive frame rates, make it virtually impossible to implement features like part manipulation. What is needed is a simple approach with rendering performance bounded by screen complexity not model size, with primitive traceability to the original model to facilitate part manipulation, and capability to be modified in near-real-time. This thesis introduces MMDr, a simple system to achieve interactive frame rates on extremely large data sets, while retaining support for most if not all the features required for a computer visualization solution.

1 INTRODUCTION

1.1 Product Lifecycle Management

Product lifecycle management (PLM) is a set of software tools and processes by which a product is managed throughout its lifecycle; it is used by the Best-In-Class companies in order to achieve a high level of innovation. PLM focuses on designing the right product at a higher quality with reduced design, manufacturing, and maintenance costs. PLM accomplishes this by enabling globalization, ensuring that best practices are re-used from one product to the next, and enhancing the engineering design process.

The product lifecycle can be described in as a series of phases efficient and high quality product development. For most products this can be summarized into 6 phases: Inception, Requirements, Design, Manufacture, Support, and Retirement.

The product is first conceived in the inception phase. It could spawn from a "Eureka!" moment, brainstorming session, or a customer request. It is quite simply taking an idea and deciding to turn it into a product.

The requirement phase focuses on defining the product. Usually this requires talking with a customer and finding out what their needs are and then translating these into the set of the features that the product must contain. It is very important to ensure that the requirements are complete since it has been shown time and again that changing requirements in later phases is significantly more expensive than getting them right the first time.

In the design phase a solution is developed to satisfy the requirements. Many alternatives may be explored before a final one is chosen to be implemented. Further, a

prototype may need to be built in order to prove feasibility or to solicit customer feedback in order to ensure proper understanding of the requirements.

The manufacturing phase defines the processes by which the design is realized as a physical product. This encompasses everything from setup, fabrication, assembly, and shipping.

The support phase occurs once a product has been released and is in hands of the customer. Mechanisms must be put in place that allow customers reported issues and for any issues deemed severe enough to be addressed. As long as the product is manufactured support must be provided.

The retirement phase is the point at which a product is no longer manufactured or supported. The factory is closed or transitioned to a new product and any support mechanisms that were put in place are disabled. If it has not already been done, the company will take stock of what went well and what did not during the design process.

1.2 Visualization Software

Visualization software is a very important component of the PLM toolset, especially since it provides many opportunities for companies to reduce costs. The ability to visualize and analyze a product throughout the design phase means that the expensive physical prototypes of the past may not be necessary. Analysis routines that previously had to wait until after a physical prototype was manufactured can now be executed as soon as the required components are complete allowing for issues to be identified much sooner. Visualization allows for companies to transition to a paperless design and manufacturing process. Components are now capable of being designed in context. This reduces the

chances that they might accidentally interfere with one another, while simultaneously achieving much tighter integrations of components. Product manufacturing information like machining tolerances can easily be displayed in context. Complex assembly instructions can be implemented in full 3D with markups representing trouble areas or special instructions.

1.3 Motivation

The current generation of visualization software was designed for "Large Model Visualization" (LMV), or models on the order of magnitude of millions of triangles. While the implemented techniques were more than adequate for data of this size, it is quickly becoming apparent that these brute force techniques will not be able to keep up with the increases in model complexity. As illustrated in Figure 1, the problem stems from the fact that the computational power of both GPUs and CPUs is increasing at a far greater rate than memory bandwidth and data access speeds.

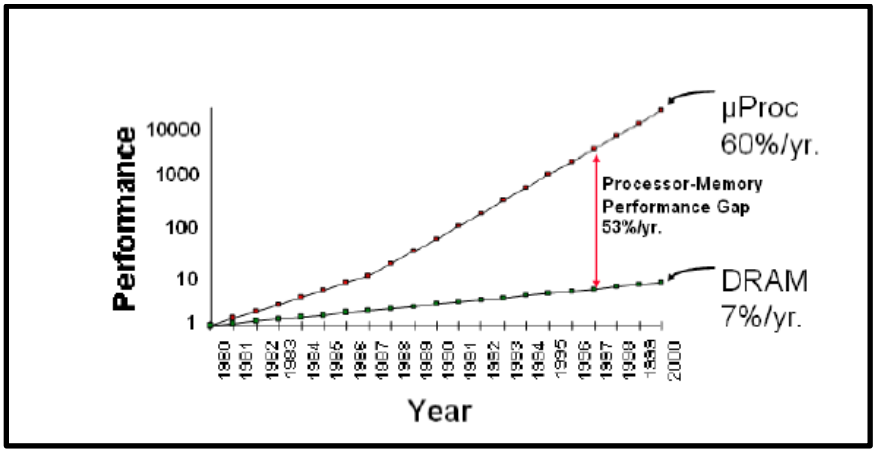

Figure 1: Growth in computational performance versus memory performance [1]

This means that while model complexity may be quickly approaching arbitrarily large sizes, the amount of data that the memory sub-system can handle is not approaching large sizes. "Massive Model Visualization" (MMV) is the term used to encompass the research

and techniques that address this problem and therefore it is of extreme interest to those in the field of computer visualization.

1.4 Thesis Organization

This thesis focuses on the use spatial hierarchies to facilitate Massive Model Visualization via a rendering approach called MMDr that is dependent upon output sensitivity and not model complexity. More specifically, it focuses on what can be achieved through a bare minimum approach to establish a better understanding and a framework by which more advanced research into MMV can be accomplished. Chapter 2 presents three different systems for handling arbitrarily large data sets, and briefly covers some of the more common techniques for accelerating rendering. Chapter 3 explains the algorithmic choices and implementation of the MMDr proof of concept. Chapter 4 presents the performance characteristics of MMDr and, where applicable, compares it against the highly optimized and commercially available DirectModel Toolkit from Siemens PLM Software. Finally, Chapter 5 summarizes the results and details many opportunities for future work within MMDr in particular, and MMV in general.

2 BACKGROUND

Research in massive model visualization, i.e., interactive rendering of arbitrarily large models on commodity hardware, builds on a relatively old and voluminous body of research in computer graphics. Many basic computer graphics techniques have been in continuous development for many years and have undergone significant enhancement to keep pace with the increases in data size, hardware acceleration, and other complimentary technological advances. However, no single graphics performance technique is capable of solving the problem of rendering arbitrarily-large models on commodity hardware and therefore these techniques are often grouped together. Thus, massive model visualization research typically focuses on grouping these fundamental graphics techniques to form complete systems that enable interactively frame rates to be achieved on the current and future generations of models.

2.1 Graphics Techniques

As stated earlier no single computer graphics technique is capable of achieving interactive frame rates on arbitrarily large models. Therefore it is necessary to take a broad look at the available methods presently in use in order to gain better understanding of their strengths, weaknesses, and why they may be included in a specific massive model visualization method. Below culling, spatial hierarchies, stand-ins, and memory management will be briefly introduced since they are the most commonly used techniques throughout the computer graphics literature.

2.1.1 Culling

One of the best ways to accelerate rendering is to render only the geometry that will contribute to a given frame. Thus, many methods have been proposed for culling data that does not significantly contribute to the current frame. While these methods are effective at reducing the rendering data set size, they often do so at a high computational cost. On large data sets these algorithms can become so expensive that it actually takes more time to cull the geometry than it does to render it. A common solution for this issue is to use a separate thread to perform the culling for the next frame while rendering the current frame. The problem with this approach is that the set of culled shapes between frames may be different enough that artifacts are introduced, such as the disocclusion or "popping" of shapes. A better solution is the use one of the spatial indexing approaches that are described below in order to significantly reduce the amount data that must be processed by allowing for larger chunks of data to be culled with a single calculation.

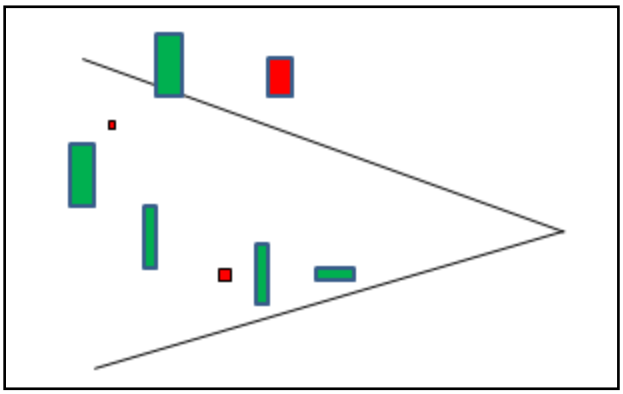

Figure 2: Diagram showing screen coverage, occlusion and view frustum culling

One of the earliest and easiest to implement methods for culling data is view frustum culling (VFC). The basic premise of VFC culling is that only the data that intersects or is contained within the current view frustum will likely contribute to the final image and

therefore it is safe to cull all other geometry. While this premise may not be true for all rendering methods, for example if reflections are being rendered, it does provide for an efficient and effective way to cull a significant amount of geometry. One of the earliest and most commonly referenced approaches was presented by Clark in 1976 and over the years there have been many enhancements with some being present as late as 1999 and 2000 by Assarsson and Mőller [2,3,4].

Another common method for culling data is screen coverage culling. Screen coverage culling is based on the fact that geometric data can only contribute to an easily-bounded fraction of the final image. The premise is that the maximum contribution of any geometric data is relative to its size in world coordinates compared to the size of the view frustum and that it is safe to cull any geometry whose contribution is so small it would not make significant difference in the final image. Screen coverage culling is not without its drawbacks. There is no clear-cut definition of the parameters of a significant contribution; further, in a non-static scene there can be a significant amount of geometric data popping in and out as it transitions between being culled and not culled. Screen coverage can also be referred to as adaptive, as further described by both Bartz et al and Zhang et al [5,6].

A more advanced method for culling is occlusion culling. The basic premise is that geometric data can be occluded by other geometric data during the rendering process and will not contribute to the final output image. In the past, methods like hierarchical z-buffers, occlusion switches, shadow frusta, and hierarchical occlusion maps were used to detect and cull occluded data [6,7,8]. However, the addition of hardware-based occlusion queries have made these algorithms all but obsolete, since replacing these approaches with ones that utilize the GPU almost always results in a higher level of performance [9,5].

2.1.2 Spatial Hierarchies

Most occlusion techniques require complicated calculations that are prohibitively expensive to calculate on the CPU. In some cases, these techniques are actually more expensive to perform on a shape-by-shape basis than it is to render the individual shapes. In order for these algorithms to be effectively used, it is necessary to find a means by which a single calculation can be applied to a much larger subset of the data, or rather a larger grouping of shapes. This is where spatial indexing data structures come into play [10,11]. Spatial indexing data structures allow for the data to be subdivided into a hierarchy of sub-regions such that when traversal stops at a region, it automatically results in all of its sub-regions being culled[12]. Over the years, many spatial indexing structures have been proposed with octrees, kd-trees, and bounding volume hierarchies being the most commonly used within the realm of massive model visualization [13].

Octrees are an indexing data structure that subdivides each region of space into eight equal-size sub-regions. This subdivision is recursively performed for each sub-region until a desired number of triangles are achieved within each region. The uniform nature of octrees allow them to be implemented in a very compact manner, however this comes at the expense of having less flexibility in choosing how the data is sub-divided because of the fixed nature of the octree subdivision.

KD trees are an indexing data structure that sub-divides a region of space into two sub-regions at a set distance along an axis. Like octrees this can be recursively done for each region; however the distance and axis of sub-division is changed on a region-by-region basis in order to maintain approximately the same number of triangles within the sub-regions. This allows the sub-division to more accurately match the underlying data structure; however

it comes at the expense of having to store more information to identify sub-regions since both the axis and distance of subdivision is needed. KD-trees are considered the data structure of choice within the realm of massive model visualization. The most common used and widely accepted approach for generating optimized subdivision in a kd-tree is the surface area heuristic that minimizes model-ray intersections. It was original proposed by Goldsmith and Salmon [14] in 1987 however it was further enhanced by MacDonald and Booth [15] in 1990 and then again by Havran [16] in 2000 .

Bounding volume hierarchies are a spatial indexing data structure that recursively sub-divides the data set into smaller and smaller bounding volumes. Bounding volumes have to store the greatest amount of information per region; however they are less inclined to require breaking the original data apart since sub-regions are allowed to overlap.

2.1.3 Stand-Ins

While the use culling techniques can significantly reduce the amount of geometric data that needs to be rendered, often it is not enough. In situations like this it is beneficial to replace highly complex geometric representations with a reduced representation for any geometric data that is considered to be far away from the viewer or is on the outer edge of the viewer's perception [17].

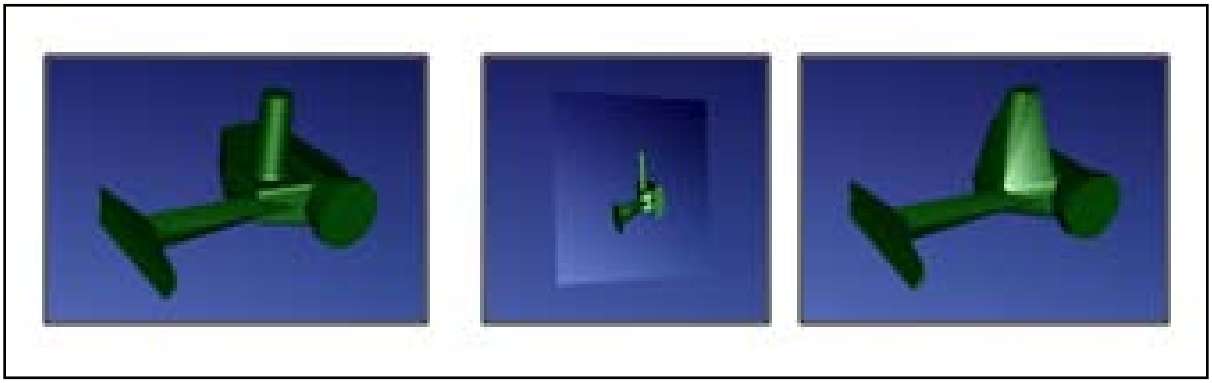

Figure 3: Billboard and reduced polygon count representation of the model on the left

One early form of stand-ins is image replacement. This goes back to the use of a billboard or textured quads to represent a complex object such as a tree [18]. As time progressed, these billboards were replaced with polygonal models in order to increase the realism; however it still made sense to use the original billboard in situations where the geometric data was far from the viewer.

Image replacement has come a long way since these early approaches. Rather than replacing single object, texture quads are now used to replace vast swaths of geometry. They are commonly combined with spatial hierarchies such that images are created for each cell looking out each of its sides. When rendering, only the geometry within the user cell needs to be rendered, the remaining geometry can be replaced by rendering the generated images. In order to avoid errors, the amount of rendered geometry can be expanded to neighboring cells. Further texture quads can be replaced with a textured depth mesh, which helps to alleviate some of the "popping" that inevitably occurs when transition from rendered geometry to a texture [19,20].

Another image replacement technique is splatting. Rather than generate images that look out from a cell, reduced images are generated that approximate the geometry when

looking into the cell. Early methods used an image for each of the sides, however more recent methods have come up with ways to use a single image shaders to generate the final result based upon the view direction. This method accelerates rendering by replacing far geometry with the rendered splats [21].

One of the biggest issues with almost all image replacement techniques is that they generally only work well with static scenes since the actual generation of the images can be prohibitively expensive.

Another stand-in method is to generate a reduced-polygon count representation for each geometric object using approaches like those proposed by Hoppe [22]. When geometric objects are far from the viewer or towards the edges of their visual perception these alternative representations can be rendered instead in order to accelerate rendering. Early methods would actually insert the alternative representation in to the scene graph as level of details (LOD) [3,23]; however this comes at the expense of having multiple representations in memory. The current generation of methods encodes efficient ways to generate new representations on-the-fly and therefore can create them at render time [13,22]. One of the more recent approaches Adaptive TetraPuzzles was able to clearly show that by utilizing the GPU this can even be accomplished in real time for even gigantic polygonal models [24].

2.1.4 Memory Management

Arbitrarily large models present a new set of challenges for managing memory since complete systems need to be more aggressive with the loading and unloading of memory in an adaptive manner. This requires transitioning to approaches like that proposed in MMR[19,20] and Intervis3D [1,25] in which data is prefetched based upon whether or not a

cell is predicted to be rendered in the near future using the cell's location relative to the current viewpoint and expected movement [19,1,26]. Further, lazy unloading of data will need to become proactive where only the cells that are expected to be used in the near future are kept in memory. Memory intensive algorithms need to be implemented in a cache-coherent manner in order to reduce the amount of memory thrash and maintain high performance [13]. The same holds true for how data is stored in memory, especially when it comes to storing meshes. Cache-oblivious layouts need to be utilized to their fullest, such as those described by Yoon et al [27] for optimizing data layout within the system and Sander et al [28] for optimizing the triangle vertex cache locality.

2.2 Complete Systems

Individual performance enhancing techniques are only part of the massive model visualization solution. Even the best techniques will fail if not used properly. Complete systems are end-to-end approaches that effectively make use of techniques in order to achieve a goal. In the realm of massive model visualization the goal is to achieve interactive frame rates on arbitrarily large models. Below, three complete massive model visualization systems will be briefly introduced. Although they may not all necessarily be the newest, each serves the purpose of demonstrating a different subset of the commonly used approaches and therefore is invaluable in terms of understanding the full scope of the problem.

2.2.1 MMR

MMR was designed and implemented as part of the Walkthru Project at the University of North Carolina at Chapel Hill [19,20,29]. MMR's developers realized that

CAD model sizes are increasing at a much faster rate than the rendering capability of commodity hardware. A new approach was needed to render these Massive Models that contain more than 1 million primitives at interactive frames of 20 fps frames per second or more. MMR was built from the ground up to be an extensible platform for enabling further research into arbitrarily large model that easily scales with model size. It is designed to allow researchers to interchange various techniques in order to facilitate interactive navigation (i.e., walking through) such massive models.

The basic strategy employed by MMR is to not render any geometry the user will not see. In order to facilitate this, a two-part technique is employed. The first part eliminates any geometry that is far away from the viewer by using an image replacement technique. The second part optimizes the rendering of nearby geometry through the use of common acceleration techniques such as occlusion culling and level-of-detail [3]. In order to make use of these techniques, the input data must be pre-processed.

The image replacement technique replaces faraway objects with textured depth meshes. In order to determine which objects are considered far from the viewpoint, the model space is sub-divided into a set of viewpoint cells through the use of a view emphasis function. The view emphasis function is a user-defined function, unique for each model, that returns a scalar importance measure for any point in the model space and therefore allows for the importance of any area to be easily determined. Each viewpoint cell is given a large cull box that may overlap with the cull boxes of neighboring cells. Any objects outside a cell's cull box are considered to be far from the viewer when they are standing inside the cell. For each view point cell, a set of depth maps are created that contain the resulting color and depth values for the geometry outside the cull box when looking from the center of cell through

each of its sides. This data is used to generate a simplified texture depth mesh for each of the sides that can serve as the necessary stand-in through the use of algorithms from Darsa and Sillion for the heavy reductions in complexity and Garland and Heckbert for the more fine tuning reductions [30,31,32].

In order to facilitate the use of the techniques required for rendering near-geometry, each object is processed such that large objects that intersect multiple viewpoint cells are split into a set of smaller objects. Further, alternate representations are created for each object using a slight variation of the method proposed by Erikson and Manocha [33]. Last, a list of all objects that are potential occluders is calculated for each cell.

MMR uses a four-process approach to handle the rendering of arbitrarily large models. The main process is devoted to rendering and handling any operations that need to occur between phases. Two sub-processes are devoted to handling culling, one of which is purely devoted to occlusion culling. The last process handles asynchronous I/O and attempts to pre-fetch any texture depth maps or tri-strips that will be needed in the following frames. [19,20,29]

2.2.2 Far Voxels

Far Voxels [21] is a volumetric approach for rendering arbitrarily large models. Realizing that model size is quickly outpacing the graphics capability of most commodity hardware, Gobetti and Marton set out to create a method that would work on a larger subset of models then the currently available output-sensitive approaches.

The approach uses a pre-processing step to generate a coarse volume hierarchy whose leaf nodes contains a fixed number of triangles and interior nodes contain a voxel

approximation of any underlying nodes. The hierarchy is generated using the surface heuristic approach defined by MacDonald and Booth to sub-divide the model into an axis-aligned BSP tree [15]. In order to improve cache locality, the resulting tree is stored in a memory coherent order similar to one proposed by Harvan [16]. Once complete, the tree is combined with a coarse hierarchical data structure by first removing any empty nodes and then associating sub-trees containing a set number of triangles of the BSP tree to the leaf nodes of the hierarchy. Once the data structures have been combined, the hierarchy is traversed to finalize each leaf node and immediately save it out to disk. Leaf nodes are finalized by extracting the triangles from the associated BSP-tree, culling out triangle that are outside the nodes bounding box, tri-striping the resulting triangles, and performing optimizations to increase the cache coherency of the resulting data. The interior nodes' view-dependent voxels are generated by casting a large random set of rays against the BSP tree at each node in order to generate a set of pixels samples from approximately all un-occluded directions. These samples are then fitted to a set of shader models in order to compress their size and allow for more efficient rendering.

In order take advantage of the massive parallelism that exists within the subdivision approach, the input data set is split into chunks containing between 20 and 30 million triangles. These chunks are then farmed out to a networked group of computers who generate a tree for their section of the model.

Rendering occurs in breadth-first front-to-back order on a set of trees. A priority queue is used to sort the current set of nodes to be potentially rendered in a front-to-back order, with the initial set of nodes for each frame being the root nodes of all trees. While there are still nodes in the queue, the front node is removed and processed for rendering. All

nodes are initially marked as invisible in the current frame, so during processing the active node can simply be discarded resulting in it and its whole sub-tree being culled from the current frame. For example, if the active node's bounding box is completely outside the view frustum, it is discarded without going through the occlusion query machinery. The occlusion query machinery handles each node based upon the node's state. Nodes that were invisible in the previous frame are initially queried against their bounding box. Leaf nodes determined to be below a certain screen coverage threshold, and nodes with children not yet loaded, are queried against their actual render. For all other nodes, their children are pushed directly into the potentially-render queue. The occlusion results are handled once they become available. Nodes whose queries result in zero visible pixels are immediately discarded. All other nodes, and their ancestors, are marked as visible and their children are pushed into the queue. Further, any node whose bounding box was rendered is rendered normally. If the queue should become empty while there are still outstanding queries, the node associated with the top-most query is handled without waiting on the query to complete.

Loading of nodes is handled through an asynchronous I/O mechanism. The list of nodes to be loaded is processed at each frame. Fetch requests are issued for as many nodes as can be handled in a given amount of time. Nodes are pre-sorted based upon the estimated voxel size of their parent in order to give a higher priority to nodes that will potentially contribute to a much larger region of the screen. [21]

2.2.3 Interviews3D

One of the most complete systems available for Massive Model Visualization is Interviews3D [1,25] supporting navigation, picking, manipulation, animation, and even collision detection. It is described by its creators, 3D Interactive [34], as a digital mockup

system that is capable of handling data sets with millions if not billions of triangles on commodity hardware through the use of a visibility-guided rendering approach.

Interviews3D was built around a set of pre-defined principles that helped to guide and influence its development. Some key principles that all systems can make use of include: rendering performance should be determined by the output complexity and not the data size; data should load and unload automatically and should only exist in memory when needed; temporal coherence between frames should be exploited at every opportunity possible; it is better to update than to rebuild; and make use of current and future technologies whenever possible.

In order to efficiently handle the visualization of an arbitrary large models Interviews 3D uses a preprocessing step to generate an axis-aligned bounding box tree out of the input data through a top down approach that results in leaf cells that contain between 1000 and 8000 triangles. The exact algorithm used for subdivision is not provided, however some key features are explicitly called out. The resulting tree can be selectively updated based upon changes in the original data set which can be shown to be much more efficient than rebuilding the whole tree. The available memory is taken into account when processing the tree and if necessary, a multi-pass approach that works only on subset of the input data is used. While splitting the side lengths of the cell, bounding boxes are kept as uniform in size as possible. Individual triangles are never split or duplicated which means there may be some overlap between the bounding boxes of child cells. Subdivision is based upon a minimum number of polygons in a given cell, which is calculated for the whole input data set based upon a variety of factors.

The use of a spatial hierarchy allows Interviews3D to achieve an interactive frame rate through the use of nothing more than an occlusion algorithm and a simple LOD [3] mechanism. The occlusion algorithm is very similar to the one presented in GPU Gems 2 [9], making full use of modern GPU capabilities and taking into account the temporal coherence in the geometry rendered from one frame to the next [9]. In order to do this, each cell in the spatial hierarchy is labeled as being 'visible', 'invisible', or 'untested'. Rendering uses these states to carry out a two-pass rendering approach. In the first pass all shapes marked as visible are rendered in order to prime the depth buffer for occlusion queries. In the second pass, the tree is recursively traversed in a front-to-back top-down manner using standard view frustum and occlusion tests. When a cell labeled as 'invisible' or 'untested' is encountered, an occlusion query is immediately executed and the traversal is pruned. Obtaining the results of each occlusion query occurs in parallel to the tree traversal in order to prevent stalls in the graphics pipeline. A post-render tree traversal propagates the visibility state up the graph. A simple LOD mechanism determines the cells that would contribute a pixel or less to the screen, and renders a single point instead of the cell's polygons.

Data management occurs asynchronously within Interviews3D and is coupled tightly with the rendering approach. The main objective is to load as many leaf cells as possible into memory based upon their priority order (respectively from highest to lowest: visible, invisible but within view frustum, invisible, and finally single-pixel cells). If cells should need to be removed from memory, the cells with lowest priority are removed first based which has been in memory longest. [1,25]

2.3 Research Issues

Massive Model Visualization is not considered a solved problem. Many advances have been achieved in trying to solve the problem of rendering arbitrarily large models at interactive frame rates on commodity hardware. However, there are few solutions that can actually achieve the full capabilities required for PLM and other advanced visualization applications and none that can do it without requiring time intensive pre-processing executed on the input data set. For a massive model visualization system to be considered complete, it must support advance features like part selection, part manipulation, advanced materials, shadows, and collision detection, while still maintaining interactive frame rates.

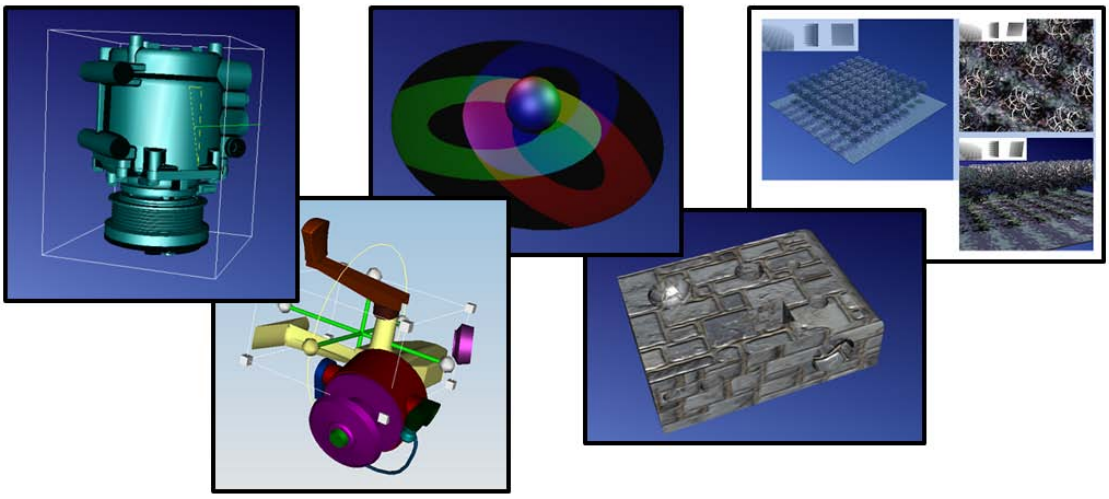

Figure 4: Advanced features commonly seen in visualization software

Systems like Far Voxels and MMR are capable of achieving rather impressive frame rates, however this comes at the cost of requiring an extremely time consuming pre-processing step. This primary cost is the generation of image-based stand-ins for most, if not all, of the cells. This cost is extremely important because advanced features like part manipulation require the ability to be able to dynamically update the spatial data structure in

real time. This is something that is not and may never be possible with image-based approaches. Further systems like Interviews3D have shown that interactive frame rates can be achieved through far simpler means and with significantly less time spent preprocessing. It has even shown that part selection, manipulation, and collision detection can be implemented on top of a simple spatial hierarchy in real time. However, Interviews3D does not support features like shadows or advanced materials and even though its preprocessing time is significantly less than other approaches it is still far from real time. These key observations can be summarized as follows and should be used as a guide by which additional research into massive model visualization is pursued:

- Complicated algorithms and highly specialized data structures are not needed in order to achieve interactive frame rates on extremely large datasets.
- Proposed solutions should not preclude the implementation of advanced features that are already present in modern day visualization solutions.
- Proposed solutions should not require an exorbitant amount of pre-processing time.

3 MMDR SYSTEM

In order to better facilitate research into the different aspects of massive model visualization, the MMDr proof-of-concept was proposed, designed, and implemented on top of the DirectModel toolkit as provided by Siemens PLM Software. MMDr consists of a simple MFC Viewer and a shared library that provides access to the classes and utilities that make massive model visualization possible.

MMDr's MFC Viewer provides a simple interface by which the capabilities of the shared library can be demonstrated and tested. The main setup is oriented around three windows that can be used to provide different views into the various aspects of MMDr's functionality. When utilizing the standard DirectModel rendering approach all windows display the final output. However, when utilizing the Spatial Hierarchy based rendering approach the right-hand window displays the final shaded output image while the upper left window displays rendered cells as green and culled cells as red. The lower left window displays loaded cells as green and unloaded cells as red.

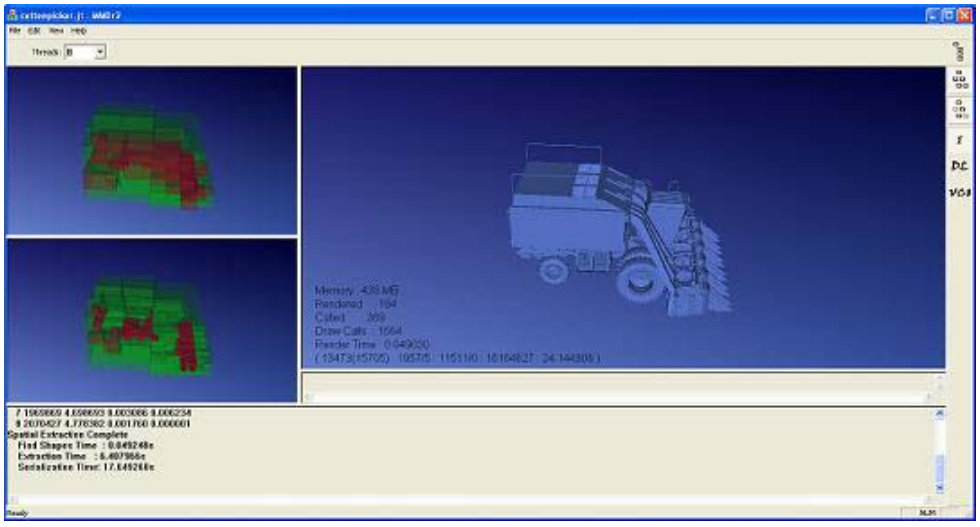

Figure 5: Screenshot of MMDr

A simple toolbar provides a means for toggling between various modes. The user can select whether rendering utilizes the DirectModel product structure or the spatial hierarchy. The user can also select whether occlusion culling is currently enabled. Lastly, the user can select which of three supported optimizations are being used. A second toolbar is provided to set how many threads of execution will be used when generating a spatial hierarchy. The number of available threads defaults to, and is automatically capped, at the maximum number of concurrent threads supported on the current machine.

The shared library, JtMMVis, provides the necessary functionality to build a visualization application based upon the use of a spatial hierarchy. The functionality can be organized into three main components (Spatial Partitioning, Rendering, and Memory Management) and the underlying data structure that allows them to work. It is proposed that when combined, these components form a complete system that should pave the way for the visualization of arbitrarily large models at interactive frame rates. At the very minimum the components should allow for a product structure to be converted into a spatial hierarchy and demonstrate that even the most basic rendering approaches using a spatial hierarchy are still superior to one that is optimized for rendering around a product structure.

3.1 Spatial Partitioning

One key problem with arbitrarily large size models is that they are often so large that it is impractical if not impossible to load them completely into memory. The use of product hierarchies helps to alleviate this issue by allowing for parts and assemblies to be dynamically loaded/unloaded and culled from what is actually rendered, however this scheme can go only so far. The assemblies and components of a product structure are often organized around systems, which often have parts that interleave and wrap around one

another. While this generally works well within the design process, it causes issues when trying to efficiently manage the data for interactive rendering. Often times, only a small subsection of a given sub-assembly or part is actually visible within a frame while the rest is obscured. This means that much of data loaded in memory is not being used. Furthermore, the sub-assemblies and parts have no implicit relationship to one another when it comes to performing advanced culling techniques like view frustum, screen coverage, and occlusion. When using a product hierarchy, the run-time of these algorithms is therefore bounded by the size of the model, which means that as the model grows to an arbitrarily large size, so will the amount of time it will take to complete these algorithms.

In order to move past these issues, the product must be transformed into a new data structure – one that helps limit the amount of loaded geometric data while at the same time bounding the culling algorithms to only those things that are currently visible. This is where the use of a *spatial hierarchy* comes into play. A spatial hierarchy is a way of organizing data such that two objects that are near to one another in space remain near one another *in the data structure*. Having such a data structure greatly aids in quickly performing the aforementioned culling tasks because large swaths of the hierarchy can quickly be culled and the detailed processing can be limited to only those areas where it is truly needed. Further, since the model is divided into spatial cells, the amount of data loaded into memory can be limited to only those cells that are visible and therefore exclude all the geometry that is not.

Spatial partitioning is the process by which a model is sub-divided into a spatial hierarchy. There are many different approaches for solving this problem; however they are not all equal. The approach below allows for: 1) arbitrarily large models to be extracted into a spatial hierarchy that, 2) maintains connectivity information between individual triangles

and their original shapes while, 3) maintaining high performance and a low memory turnover.

3.1.1 Basic Approach

The approach used for creating a spatial hierarchy from any given set of data can be written in a rather straightforward manner. While it may seem simple in nature, many considerations must be taken into account when executing each of the steps.

1) Find all shapes contained within the model and add them to the root cell of a spatial tree
2) For each unprocessed cell whose complexity would result in the memory limit being exceeded if the standard sub-division algorithm was used.
   a) Execute the secondary sub-division algorithm on the cell instead
   b) Write out the cell's geometric data unload it
3) While there are still unprocessed cells
   a) Select as many cells as possible that can processed without exceeding the memory limit
   b) Execute the multi-threaded sub-division algorithm on the selected cells
   c) Write out each cells' geometric data and unload it

Algorithm 1: Spatial Partitioning Approach

3.1.2 Data Gathering

The input data is typically organized in logical product hierarchy, which often contains more information than just the geometric information. In order to facilitate efficient partitioning, it is necessary to gather a list of all the shapes(i.e., groups of triangles). This operation can be expensive since it requires that all assemblies be opened and searched for their shapes and also carries the side effect of increasing the amount of memory in use; however this should be relatively minimal since the heavyweight shape data will not need to be loaded at this time. Once all the shapes have been collected the spatial hierarchy can be sub-divided.

3.1.3 Sub-division

Spatial hierarchies are powerful data structures because they allow a model be subdivided based upon the spatial locality of its data. The splitting of a cell simply entails subdividing the region encompassed by the cell into smaller sub-regions that are stored as its children cells. The triangles and shapes associated with the current cell are then processed. Any shapes that do not fit within a single child cell are split into triangles and handled as such. Any triangles or shapes that can be contained by a child cell are automatically removed from the parent and are pushed to the child. Triangles that do not fit into a single cell can be handled in many different ways depending upon the implementation of the spatial hierarchy. Some common approaches include splitting the triangles so that smaller triangles are added to each of the intersected cells, leaving the triangles on the current parent cell, and a hybrid where large triangles are left on the current parent cell and smaller triangles are added to all intersected cells.

One thing that should be pointed out about the above approach is that shapes are only split into individual triangles when absolutely necessary, which helps to keep the memory usage low. However when handling arbitrarily large models this may not be enough, especially if the number of shapes that need to be split is so large that it will result in the memory limit being exceeded. A secondary approach is therefore needed while the complexity of any given cell is considered to be so large that splitting the cell in the normal manner would result in a preset memory limit being exceeded. The main issue is that splitting a shape results in its heavyweight geometric data being loaded and copied throughout the children cells. Loading is not so much an issue since the shape's geometric data can be unloaded as soon as the algorithm is done with it; however the same is not true

about the triangles copied into the child cells. The solution is to add intersecting shapes into the children cells and add only triangles into the current cell. This approach is acceptable because the number of triangles that can potentially stay resident on the current cell is relatively small, and further, once the subdivision of the current cell is complete, its geometric data can be written to disk and unloaded.

3.1.4 Multi-Threading

In order to take advantage of modern computing resources, algorithms need to be designed and implemented to run in a multi-threaded fashion. A good design will isolate core operations such that the available work can be easily distributed evenly over an unknown number of execution threads. The spatial subdivision approach defined above takes this into account and thereby ensures that each cell in the spatial hierarchy can be processed independently of other cells during the splitting process once it has been initially set up. In other words, once a collection of cells has been established, they can easily be farmed out amongst multiple threads for further splitting without risk of collision between cells. In fact the only thing that will need to be taken into account is combining any resulting sub-cells into a common spatial tree data structure. This cost is minute compared to the cost of actually generating the sub-cells.

In order to facilitate multi-threading, the partitioning approach is designed to work on a spatial tree that is defined only by its root cell and bounding box. The power of this approach is that each cell in a given spatial tree is the root cell of its own spatial tree. This means partitioning only needs to split cells until it has a large enough pool of cells whose complexities are small enough that they can be effectively farmed out to the maximum

number of threads for processing as their own spatial trees. Once complete, the resulting trees from the processed cells can easily be combined back together into a single spatial tree.

One key obstacle to this approach is ensuring the initial set of cells is split sufficiently so as to ensure that the workload is evenly balanced amongst the available threads. The naive approach is to just split cells until the number waiting to be handled is greater than or equal to the number of available threads. It is naive because it does not attempt to balance the workload and therefore, some threads are likely to finish their work long before the others are done. One way to address this is to sort the cells in order of complexity so that different threads handle highly complex cells; however this can still result in some threads doing a lot more work than others. The best approach for addressing this issue is to split any cell whose complexity is greater than the complexity of the spatial tree divided by the number of available threads before sorting them in order of complexity, and then ensuring that all threads will continue working while there are still cells waiting to be processed. This ensures that even under a worst-case scenario of a thread spending its entire time handling a single cell that the others threads will have already handled or be handling the other cells.

3.1.5 File Format

In order to take full advantage of spatial hierarchy, one needs to create a file structure that allows for efficient loading and unloading of cell data based upon what is currently visible to the user. Furthermore, since massive models often times are so large that they are incapable of fitting into memory, this file structure needs to be designed such that it can be created in streaming fashion, with cells being serialized and unloaded as soon as they are handled during the spatial partitioning process.

The above tasks are accomplished by making use of data model and structures associated with the JT file format [35]. JT is regarded as the *de facto* standard for 3D visualization, finding widespread usage throughout automotive, aerospace, and other industries [35,36]. It is a high-performance; extremely compact persistent storage representation that can be customized to be extremely lightweight, containing only facet information, or full-featured geometric models depending upon the users need. Furthermore, JT natively supports the capability of being able to late-load both the facet information and heavyweight data as needed by the hosting application. These mechanisms provide a solid base from which the Spatial JT (SJT) File formation can be created.

SJT follows the same data model as the JT file format containing a header, table of contents (TOC), and multiple data segments, one of which is a logical scene graph (LSG). Segments are discrete data blocks that allow information that may not always be necessary to be loaded/unloaded as needed. For example heavyweight information such as shape geometric data is stored in its own segment and is only actually loaded into memory if needed for processing or rendering.

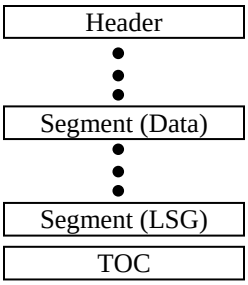

Figure 6: SJT File Segments

The header provides access to the version information, the byte order of the file, and the offset to both the TOC and the LSG.

The table of contents contains a list of all the segments in the file. The segment information is stored as TOC entries that contain a unique identifier, segment type, segment length, and the offset within the file to the segment. When possible, the TOC is stored immediately after the header in order to improve performance when a server hosts files. However, this is not always possible, especially when streaming data into the file, so the TOC can also appear at the end of the file.

The logical scene graph segment of an SJT file contains all the data required to reconstruct the spatial tree and dynamically load the other data segments. The header points to the LSG so that it can be loaded immediately upon opening a file.

The data segments in SJT file contain the geometric data associated with each cell. This data can be loaded or unloaded when needed by the application to conserve the amount of memory in use. In order to minimize the file size and limit the amount of data that must be written to and read from the disk, geometric data is written using Topological compression techniques introduced in the JT v9.x file format[37].

The segment data structure allows for segments to be written to the file in any order. Further, once a segment is has been written, its payload can be unloaded from memory since it can easily be loaded at a later point in time. This means that serialization can occur in an incremental fashion as cells are handled. Furthermore, by utilizing this capability to its fullest the amount of memory in use while partitioning can be minimized.

3.1.6 Part Associativity

One of the key disadvantages to partitioning a product structure into a spatial hierarchy is that the triangles associated with any given part can potentially be dispersed across multiple cells. This poses a potential problem since there are many common operations that must be supported in a spatial-hierarchy-based viewer that depend upon knowing part identity. Thus, the spatial partitioner is implemented so that the originating part of any triangle within the spatial hierarchy can be easily identified.

Each cell organizes its triangles into a number of *sub-groups*. Each sub-group contains triangles only from one part. (Note that multiple cells may contain different subsets of a given part's triangles) Each of these sub-groups, in turn, has an associated *unique part ID*. This unique part ID corresponds to the *part* from which the triangles came. Thus, any given triangle, anywhere in the spatial hierarchy can be quickly associated to its originating part.

3.2 Rendering

Another key problem when trying to handle arbitrarily large models is that the amount of geometric data available for rendering each frame is far greater than what could possibly be handled by the GPU while still achieving interactive frame rates. Most modern GPUs are capable of handling only 500 million triangles per second. The established approach for dealing with this situation is to render only that which is visible from the user's point of view. Many algorithms have been proposed and implemented in order to cull non-visible or barely-visible geometry in order to achieve this effect. While effective, these-brute force methods often do not scale well as the size of the geometric data increases. Often,

these methods require complicated calculations that actually take more time to compute than it takes to render the geometry in question – especially with the advances in modern GPUs.

3.2.1 Basic Approach

The rendering algorithm used by MMDr parallels the hardware occlusion algorithm presented in GPU Gems 2[9]. The basic approach is to interleave occlusion queries with standard rendering in such a way as to minimize both the amount of geometry rendered and number of queries executed on a given frame while avoiding stalls in the GPU rendering pipeline.

This is accomplished by using a priority queue to handle the next closest cell that is considered to be active. If the cell is labeled as being occluded in a previous frame, an occlusion query is executed on its bounding box and it is added to the FIFO queue of outstanding queries. If the cell is not labeled as being occluded in the previous frame, then its children are added to the active priority queue if they are not culled for being outside the view frustum or contributing too little to the frame. If it is an interior shape, it is rendered and marked as not being occluded in the previous frame. If it is a leaf shape, an occlusion query is executed when it is rendered and it is added to the FIFO queue of outstanding queries.

Before popping the next cell off of the active priority queue, any occlusion queries that have completed are handled. The head cell of the outstanding FIFO queue is checked to see if its occlusion query has completed. If so, it is popped from the outstanding FIFO queue and the occlusion results are retrieved. If the number of pixels contributed by the cell is so small that it is considered occluded, the cell is marked as such and discarded. Otherwise, it is

marked as being not occluded. If it was occluded in the previous frame, it is immediately rendered and its children are added to the active priority queue if they are not culled for being outside the view frustum or contributing too little to the frame. The cells with the outstanding FIFO queue will continue to be processed in this manner until it is either empty or the occlusion query of the current head cell is not yet done and the active priority queue is not empty.

After all cells in the two queues have been handled, a post traversal is executed on the spatial hierarchy to mark as culled any interior nodes whose children are labeled as culled.

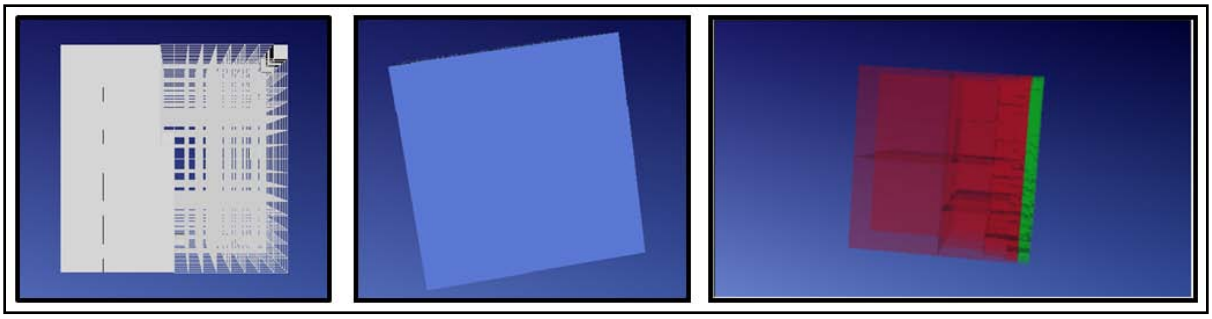

Figure 7: Original model (Left), Output (Center), Results of the occlusion query algorithm (Right)

3.2.2 Multi-view Rendering Support

While currently not enabled, the lightweight tree data structure and the parallel rendering data structure provide an easy means by which independent views on the same spatial hierarchy can be implemented. The obvious usage for this capability is to allow Massive Model Visualization in an immersive environment where it is necessary to render the scene from several independent viewpoints. Another application that is not quite as obvious is the ability to add real-time projected shadows.

3.3 Memory Management

In order for rendering to reach its full potential, the problem of how to manage a model that does not fit into host memory must be addressed. Culling techniques help by limiting the amount of data required at any given time for rendering, however this is only part of the solution. The model needs to be stored in such a way that the heavyweight data, such as the geometric shapes, can be loaded or unloaded based upon the current rendering needs. The loading of data should be prioritized and occur in such a way as to not impact rendering performance, and further the system should prefer unloading low-importance data first. These processes should occur automatically and in such a fashion that they go unnoticed by the user.

3.3.1 Basic Approach

The SJT file used for storing the spatial hierarchy is based on the normal JT file. This means that not only is heavyweight data already serialized out to its own segment that can be easily loaded, but also approaches for handling the loading and unloading of data that already exist within DirectModel can be readily adapted to work on the spatial hierarchy. These lazy approaches, while not necessarily ideal, can form a foundation by which more advanced methods that utilize predictive algorithms can be implemented.

3.3.2 Residency Request

Retrieving data from disk is an expensive proposition and definitely not one that should occur haphazardly. In order to coordinate data retrieval in DirectModel, a residency request manager (RRM) is implemented that allows for loading of heavyweight data to occur on an independent set of threads rather than always on the current one. Whenever any

heavyweight data is required a "will need request" is made to the RRM in order to initiate the loading process.

The residency request manager is implemented as a priority queue. Each request contains a priority value that determines the overall importance for that particular request. Requests are therefore handled in their perceived order of importance with more important requests being handled first. The biggest bottleneck with most requests is actually waiting on the I/O subsystem. In order to reduce this performance impact and take advantage of multi-core processors, the RRM manager supports running on multiple threads where several requests can be in flight at any given time. This allows the I/O operations and processing of heavyweight data to be farmed out to secondary threads while the main thread needs to focus only on hooking up items to their respective data structure once loading is complete.

In order to support loading heavyweight geometric data from the spatial hierarchy, a new derived class is implemented on top of the existing RRM. This class adds support for new types of requests that are capable of loading geometric data associated with each cell and attaching it to the spatial hierarchy.

Requests are currently issued as part of the rendering process. For any non-culled cell whose geometric data is not currently loaded, a load request is made to the RRM with the priority value set to the area of the cell's bounding box. This means that only the geometric data associated with render cells will ever be loaded. Further, larger cells are more likely to be loaded before smaller cells since they are more likely to take up greater screen space.

3.3.3 Disposable Objects

Unloading data is not as simple a task as loading it. While it may be fairly obvious when data is needed, the same cannot be said about when it is no longer needed. Even though the current thread may have not accessed a piece of data for a period of time there is no guarantee that other threads have not as well.

In order to properly handle unloading of data that may be accessed at random, DirectModel implements a new type of smart pointer that allows its contents to be discarded at will if it is not currently in use. All these so-called "disposable pointers" upon creation, register themselves with a global table. Furthermore, every time their payload is accessed the table is updated to contain a new last access time. This provides a mechanism by which heavyweight data can be easily unloaded starting with the oldest non-used data.

The table of disposable objects integrates directly with the DirectModel memory allocation system in order to facilitate the unloading any data regardless of its origin. This integration allows the disposable table to be automatically accessed and culled when a low memory condition is encountered during a memory allocation request. While this approach is lazy in nature, it provides an easy catchall mechanism by which memory can be bounded to an upper limit.

The spatial hierarchy takes advantage of this built-in feature by using disposable pointers to store its heavyweight geometric data. The unloading mechanism therefore can easily unload the geometric data of cells that have not been rendered recently when low memory conditions are met.

3.4 Spatial Hierarchy

A fast culling algorithm or clever rendering loop is nothing without an efficient and compact data structure for storing the spatial association information. There are many different types of spatial hierarchies, each with its own strengths and weaknesses as described in section 2.1.2. For the MMDr proof of concept, it was decided that the resulting spatial hierarchy should have a minimal memory footprint. In order to accomplish this goal, an octree was selected as the primary storage data structure since its subdivision parameters are implicitly encoded in the data structure. While this provides for a relatively small size, the tradeoff is the inability to precisely configure the spatial hierarchy for complex regions. In other words, the octree allows for the size of the tree to be kept at a minimum at the potential cost of having reduced rendering performance for non uniformly spaced scenes as shown by Meißner [38].

3.4.1 Basic Approach

In MMDr, the octree data structure is stored as a simple vector of cells that is always allocated in blocks of 8. Having a block size of 8 is important since it allows for the children of a cell to easily be referenced by an offset into the vector. Having pointers is important since it allows for empty cells to remain unallocated and easily identified. In order to handle the common case of a cell not having any child cells the 0 block is always pre-allocated and guaranteed to contain NULL pointers.

3.4.2 Memory Conservation

Each cell within the octree is defined such that it contains the minimum amount of information required. For cells that do not contain any heavyweight geometric data, this is

limited to a 32-bit state field, an integer offset to its children, and an integer reference count used by the auto pointers. On 64-bit builds, this will result in a base size of 24 bytes per empty cell. For cells that contain heavy geometric data, there is an additional disposable auto pointer data member that brings the size up to 40 bytes per shape cell. Further, all cells are stored as pointers and therefore incur an additional 8 byte penalty that brings the grand total to 48 bytes for shape cells and 32 for empty cells.

To put the size consideration into perspective, imagine an octree containing one billion triangles. Assuming on average there are a thousand triangles per cell, the tree would therefore contain one million cells and require approximately 46 MB of memory. Adding something as simple a bounding box would increase the size of each cell by only 24 bytes, however this would result in the same tree now requiring approximately 69 MB of memory. Of course, both of these numbers are miniscule when considering that the geometric data if fully loaded would, at a minimum, require over 27 GB of memory. For the size of the octree to reach even 1.0% of the total amount of required memory, only 1.4% of the model can be loaded, and with the addition of a bounding box it increases to only 1.7% of the model. This implies that while maintaining a small tree size is important, it is easily dwarfed by the heavyweight geometric data and therefore may not have been a good metric to base design decisions around.

3.4.3 Bounding Box Calculation

One key advantage of the octree data structure is that the bounding box of any cell is easily calculated from the path taken to reach the cell. This means that it is unnecessary to store the bounding box of each cell, since it can be easily calculated while traversing the tree. In MMDr, this accomplished by storing the current path within the octree iterator and then

exploiting the IEEE 754 floating-point number format to efficiently derive the bounding box of the current cell from the bounding box of the whole tree.

For each sub-division of the octree there are 8 potential children. Each child can be identified by whether the octree iterator remains at its current position at the "minimum" corner of the node, or steps along each of the axes. As it continues to traverse down the graph, the path to any cell is therefore identified by whether or not, at each level, the iterator stepped along an axis. This is easily encoded into a group of three-bit fields, one for each of the three primary coordinate axes. When traversing to a child cell each of the bitfields are shifted to the left and further for each axis in which the minimum value changes a 1 is logically ORed into the path.

These resulting bit fields are powerful because they can be used to calculate the bounding box for any cell using 9 multiplies, 6 shifts, 6 additions, and 4 subtracts. In other words operations like loops, conditional statements, and divides, that have an order-of-magnitude longer execution time, are easily avoided. This is accomplished by exploiting floating point storage and multiplication.

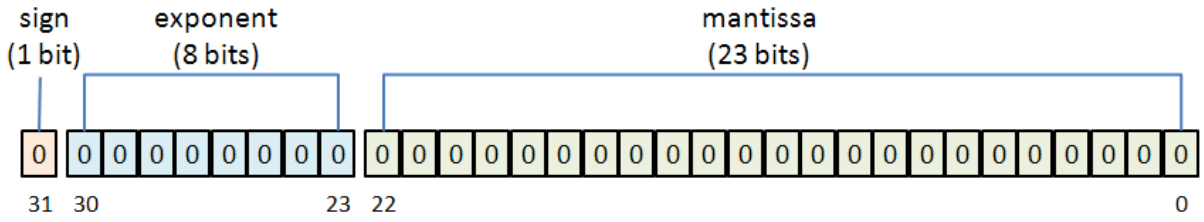

Figure 8: IEEE 754 single precision float bit layout

The basic principle is that the path along each axis can be used as the upper most bits of the mantissa of a denormalized float. This float can then be multiplied by 2^{127} and the length of a bounding box along an axis, which is then added to the axis-minimum value in

order to obtain the current minimum value. Adding one to the path and performing the same operations results in the maximum value for the same axis. Combining the results from all axes will give the bounding box for the current cell.

$$IntFloatUnion\ fu : fu.ui = pat$$

4 RESULTS AND DISCUSSION

In order to validate the complete system, the performance characteristic of each individual component was evaluated independently. A variety of machines were used when applicable during the measurement process in order to ensure that the performance characteristics of the individual machines do not bias the results.

Table 1: Set of test machines

Nickname	Bensley	i7	Nehalem
Processor	Intel Xeon X5355	Intel i7 920	Intel Xeon W5880
Number	2	1	2
Speed	2.66 GHz	2.66 GHz	3.2 GHz
Ram	8 GB (DDR2)	12 GB (DDR3)	24 GB (DDR3)
GPU	Quadro FX 5600	Quadro FX 4800 Geforce GTX 295	Quadro FX 4800
OS	Win64	Win64	Win64

To analyze the performance of the system various data sets were used. For general testing this included everything from a fishing reel (15K triangles) to more complex models such as a formula car (860K triangles), cotton picker (16M triangles), and even a corporate jet (44.5M triangles). The models used for general testing were selected from those sets available at Siemens that had either been generated for marketing purposes or provided by customers for testing of advanced features. The point of this testing was demonstrate that the system performance is not drastically influenced by or limited to a specific type of input data which is often the case in proposed systems. For more advanced testing that required larger data sets multiple instances of the corporate jet were used. While not necessarily ideal, this provided a way of collecting data within the broad gap between the largest and second largest test models. Finally, in order to truly test the massive model visualization capabilities of the MMDr proof of concept the Boeing 777 (437M triangles) was used since it is one of the commonly used models for demonstrating interactive frame rate for arbitrarily large datasets.

4.1 Partitioning

Partitioning of a product structure into a spatial hierarchy is one of the key components required for making the visualization of arbitrarily large models a reality. The process should be optimized to be as fast as possible and grow at most linearly as the model size increases. It should make full use of the available hardware resources, multi-threading where ever possible, and properly manage memory so as to ensure the available memory is not exceeded.

4.1.1 Spatial Hierarchy Generation

One of the key abilities of the MMDr proof of concept is its ability to translate a logically defined product structure into a spatial hierarchy to achieve a higher level of rendering performance. This process needs to be computationally efficient, regardless of the layout or size of the dataset, if it is to be incorporated into modern visualization software. In order to show that MMDr achieves a high level of performance, the extraction time was measured for a variety of datasets of various sizes. Further, these time measurements were broken out into the two main components, spatial hierarchy extraction and serialization to disk, in order to identify if either component acts as a bottleneck for the total performance.

The extraction performance was measured on a variety of data sets ranging from very small containing only 200K triangles to fairly large containing over 44M triangles. As shown in Figure 10, plotting the number of triangles against the total extraction it easy to see the amount of time required to extract a spatial hierarchy grows linearly as the data set size increases. Further, a comparison of the extraction time verses the time required to serialize

the data to disk is shown in Figure 11. On average, an equal amount of time is spent during each of the stages of the extractions time.

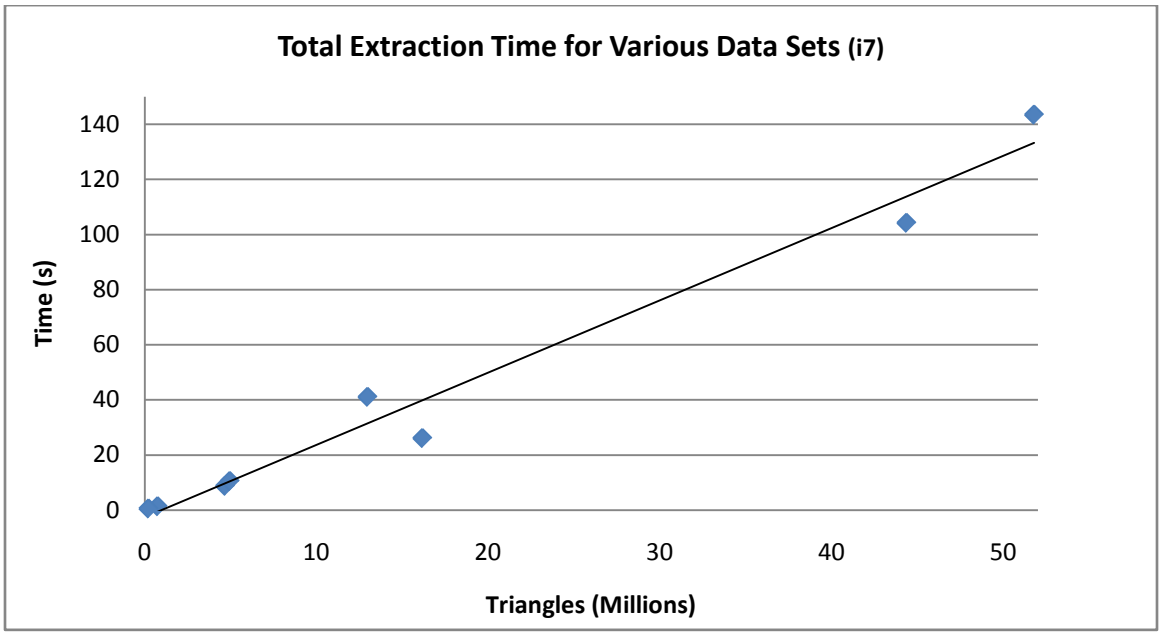

Figure 10: Total extraction time for various data sets

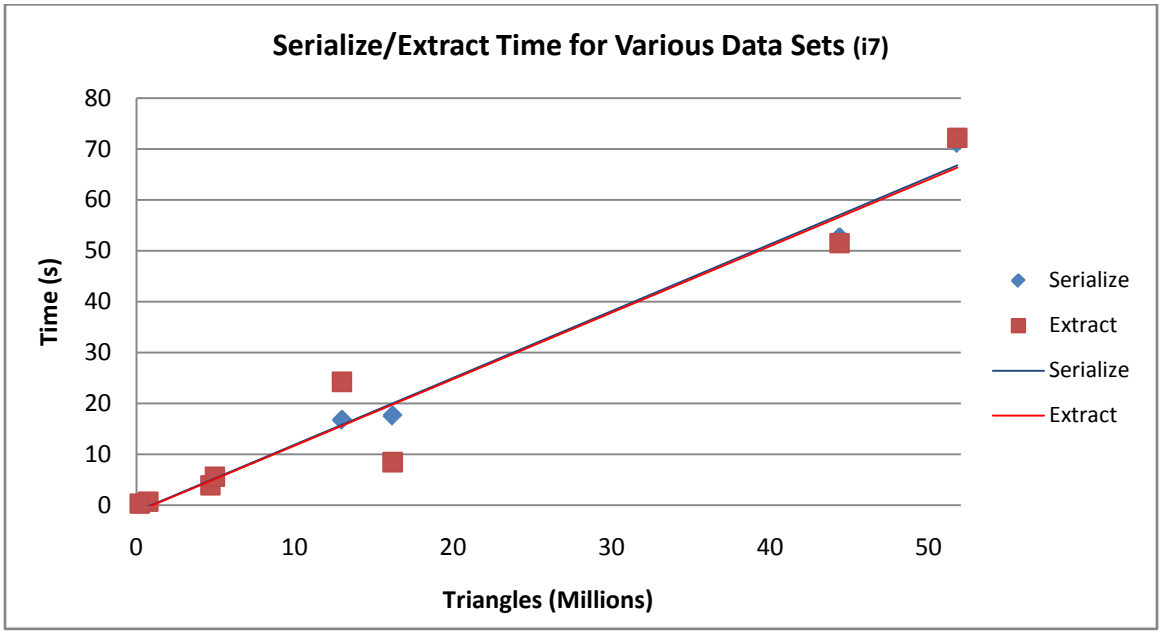

Figure 11: Serialize versus extract times for various data sets on the i7

The initial measurements while informative do not fully stress the system since none of the data sets are individually large enough to cause a low memory condition to be encountered in the test. To address this concern, another set of runs where executed on multiple instances of the largest model, a corporate jet, arranged spatially in a non uniform manner. Under these conditions a different pattern starts to emerge. As shown in Figure 12 the same linear trend is maintained for up to five instances of the model, however after the number of models increases beyond this point the slope of the trend increases dramatically. This is not unexpected since at six instances and above extraction requires so much memory that it actually causes the machine to start using swap space.

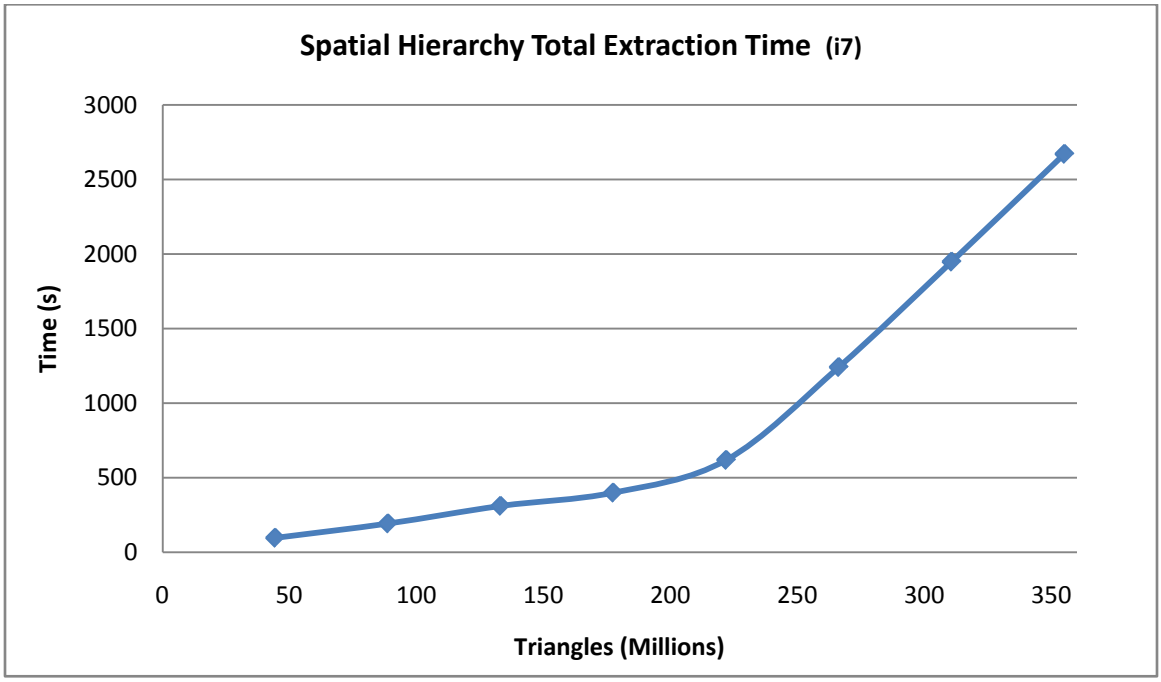

Figure 12: Total spatial extraction time based on number of triangles

What is interesting about this scenario is that the serialization and extracting phases no longer continue to run in parallel when this starts to occur. In fact, as shown in Figure 13, the serialization time continues along the same trend line regardless of whether or not swap

space is being used, and the increase is actually limited to only the extracting phase. This means that under the current approach, arbitrarily large models will eventually be bound solely by the extraction time and not by the serialization of the resulting data.

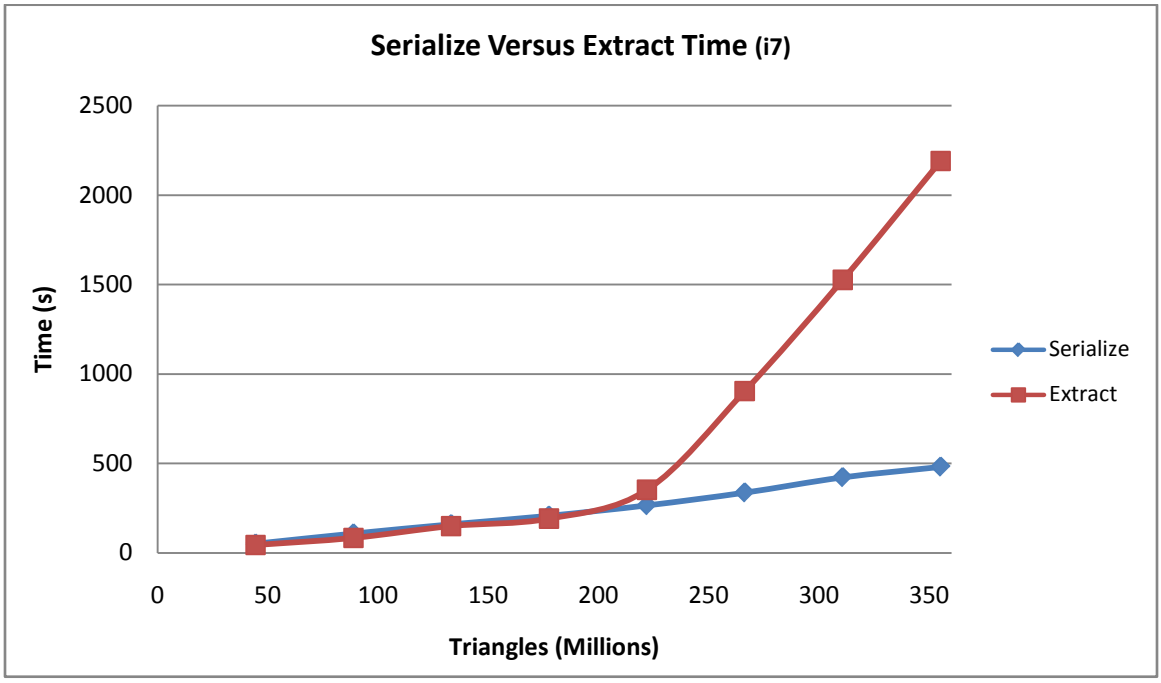

Figure 13: Comparison of serialize versus extract time based on number of triangles

4.1.2 Multi-threaded Extraction time

Extraction time versus data size is not the only interesting phenomenon. In order to make use of the of growing trend towards multi-core and multi-processor systems the extraction process should support running on as many threads as possible. Thus it is usefully to explore how the number of threads used influences the extraction time. Figure 14 plots the effects of multiple threads compared to the amount of time it takes to fully extract the corporate jet. This graph shows that there is a discernable performance enhancement up to six threads; however the benefit begins to tail off rather quickly. Figure 15 translates the time for extraction into an efficiency value when running on a variable number of threads.

This shows the benefit of adding more threads quickly decreases such that after 14 threads performance actually begins to degrade.

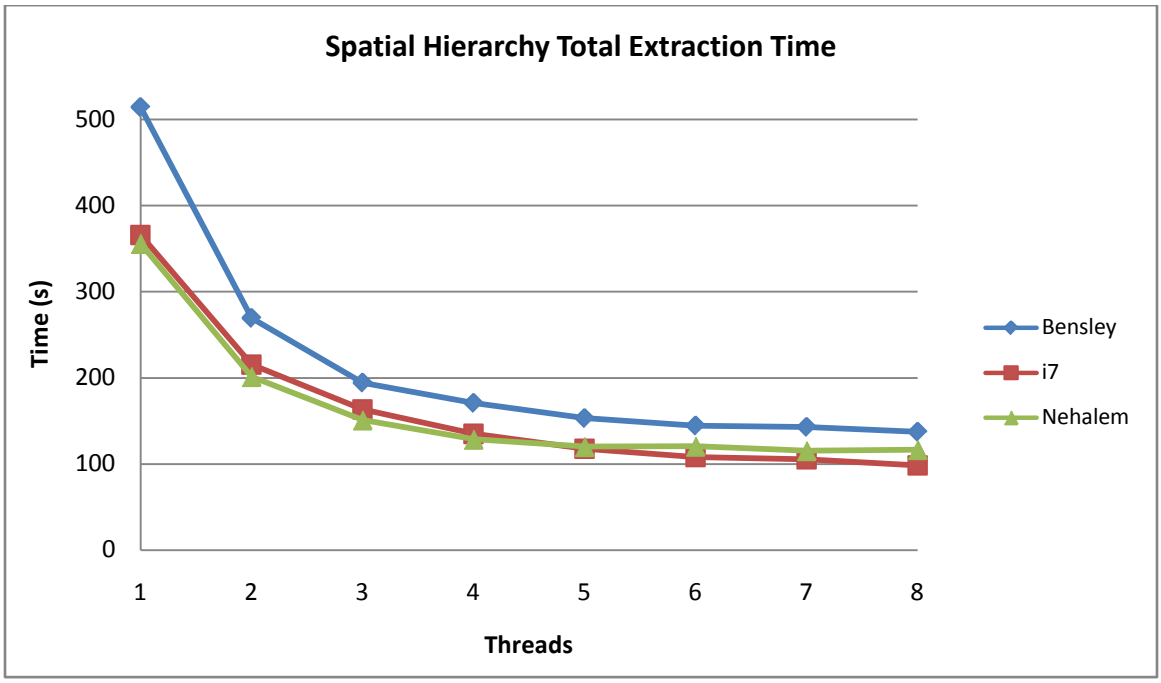

Figure 14: Time to extract on n threads

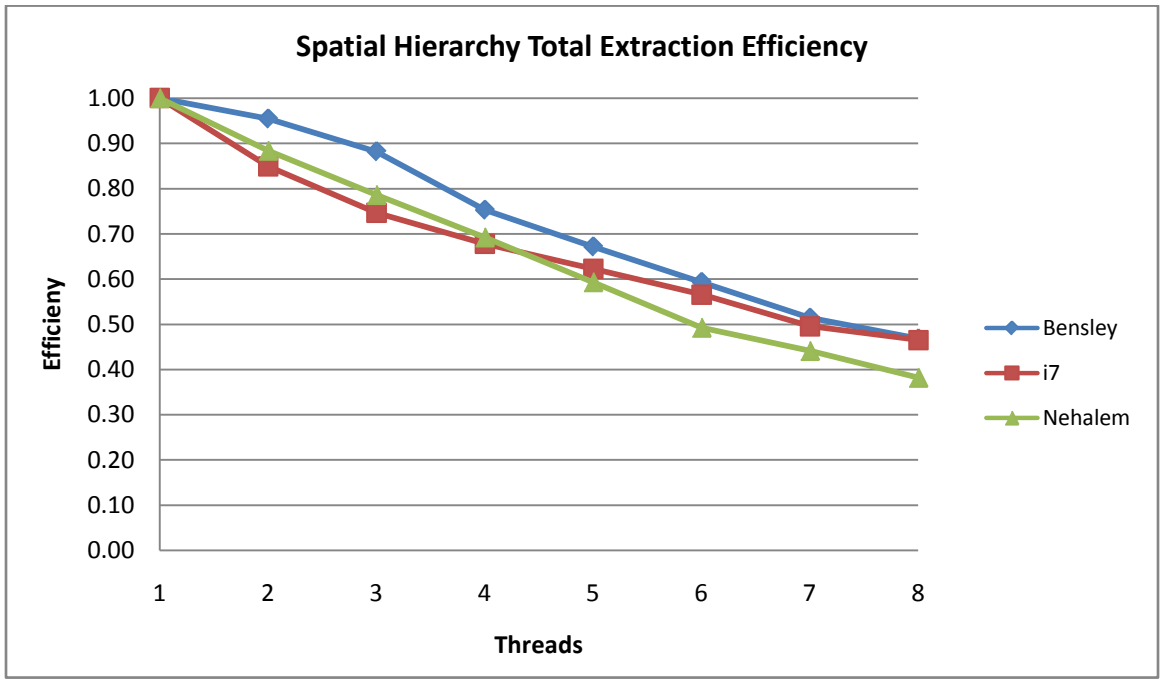

Figure 15: Efficiency of extracting on n threads

Splitting the analysis so that extraction and serialization are plotted independently of each other as in Figure 16 shows that the performance gains in extraction tail off faster than the gains in serialization. This shows that any performance gains to be had in terms of thread performance are in the extraction logic.

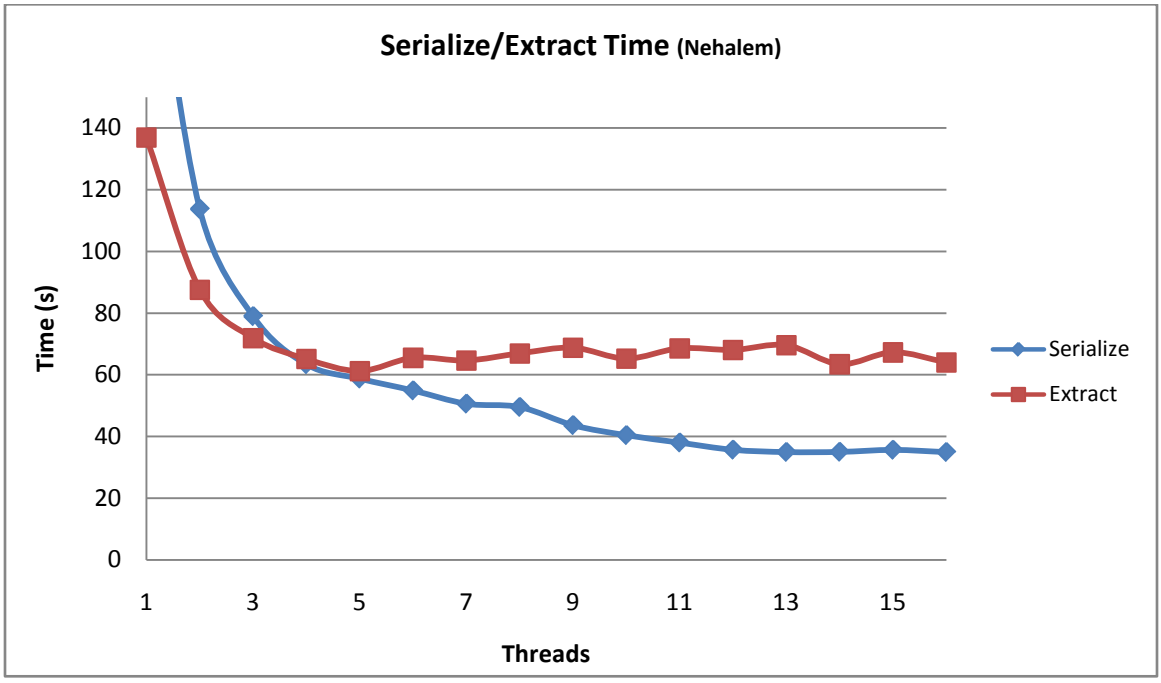

Figure 16: Write versus extraction time

4.1.3 Boeing 777 Extraction time

The first implementation of the MMDr proof of concept was only able to extract the 777 model on the Nehalem machine. While disappointing, since the low memory management scheme should have allowed it to extract on any machine, it does provide a data point for comparison against the 1920 seconds extraction time reported by Intervis3D for the same model on a 2.0 GHz Core 2 Duo with only 2 GB ram [1]. At 5143 seconds using 8 threads on a significantly higher end machine MMDr appears to be completely missing the mark. Investigation of why this is the case is still ongoing, however the initial assessment is

pointing a finger at memory thrash and the high cost of using topological compression during serialization.

4.2 Rendering

A key objective of the MMDr proof of concept is to demonstrate that interactive frame rates can be achieved on arbitrarily large data sets through the use of a spatial hierarchy and occlusion query-based rendering techniques. This is not something that can easily be shown. What can be shown is that the rendering performance of a spatial hierarchy based approach is either equal to or substantially greater than that of a highly optimized product structure approach.

4.2.1 Rendering Performance

In order to compare the rendering performance of a nominal product structure-based rendering algorithm and MMDr's spatial hierarchy enhanced technique, the frame rate was measured on a series of data sets that contain an increasing number of instances of the corporate jet.

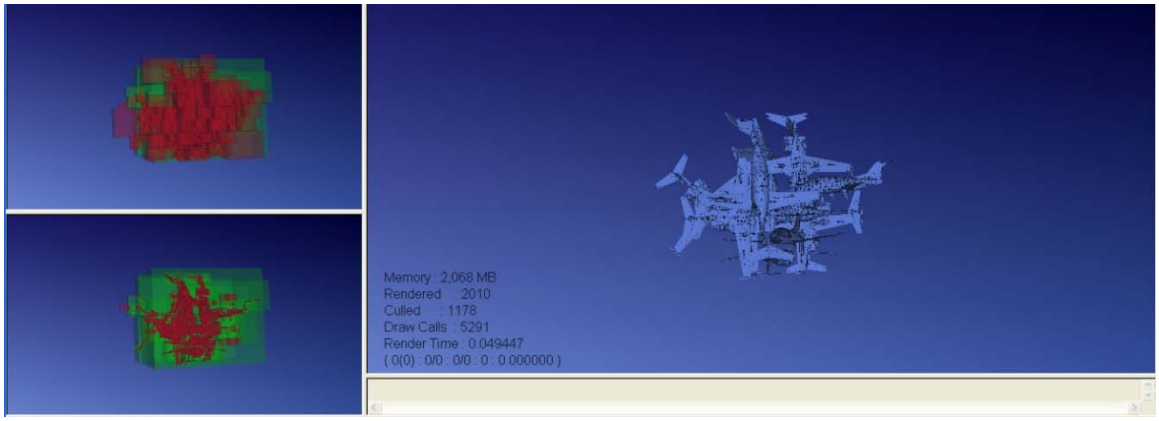

Figure 17: Rendering 11 instances of the corporate jet

The basic approach was to open the data set, ensure only the required data is loaded, and then measure the average frame rate achieved when rotating the model 360 degrees at 1

degree increments. For the product structure-based method measurements the DirectModel framework test viewer GDr was used on the original JT files. For the spatial hierarchy measurement, MMDr was use on the spatial JT files that were generated from the original JT files with a max cell complexity of 5000 triangles. Since the spatial hierarchy based renderer does not support materials they were disabled when rendering with GDr. This left the only difference between MMDr and GDr to GDr using a separate strategy thread for shape culling and making use of the level of details that are present within the product structure and MMDr making use of hardware assisted occlusion culling. A full list of the features for each of the viewers is shown in Table 2.

Table 2: Comparison of viewer features

	GDr	MMDr
Window Size	953x446	953x446
Rendered Shapes	The shapes that are to be rendered are determined in a separate strategy pass that executed over the product structure on its own thread.	The shapes that are to be rendered are determined during the actual render pass.
Level of Detail	LOD selection is allowed to occur with the strategy pass.	LOD was not implemented with the spatial hierarchy.
Screen Coverage Culling	Shapes that do not meet the specific screen coverage value are culled as part of the strategy pass.	Cells that do not meet the specified screen coverage value are culled as part of the render pass.
Hardware Assisted Occlusion Culling	Not used since the cost of making the queries exceeds the savings from not rendering the culled shapes.	Makes use of the hardware assisted occlusion algorithm as defined in GPU Gems 2.
Materials	Shapes' materials are overridden to be the same color in order to ensure a fair comparison against the spatial hierarchy.	Shape materials are currently not persisted into the spatial hierarchy.

Figure 18 demonstrates the difference in performance when a minimum amount of screen coverage culling is applied. When running with a screen coverage value of 1e-5, GDr and MMDr cull any shapes or cells whose bounding box denotes it would contribute to 4 pixels or fewer on the screen. When this value is reduced to 1e-6, the culled shapes and cells are reduced to only those whose bounding box denotes it would contribute to 1 pixel or fewer

on the screen. The data shows that a spatial hierarchy will almost always achieve a higher frame rate than the nominal product structure approach. The performance difference is most pronounced for small shapes; however at around 270 million triangles the spatial hierarchy shows a higher capacity for maintaining its frame rate even if briefly. Of further is the fact that that there is minimal performance difference between culling objects that contribute up to 1 pixel and objects that contribute up to 4 pixels when rendering over 340 million triangles.

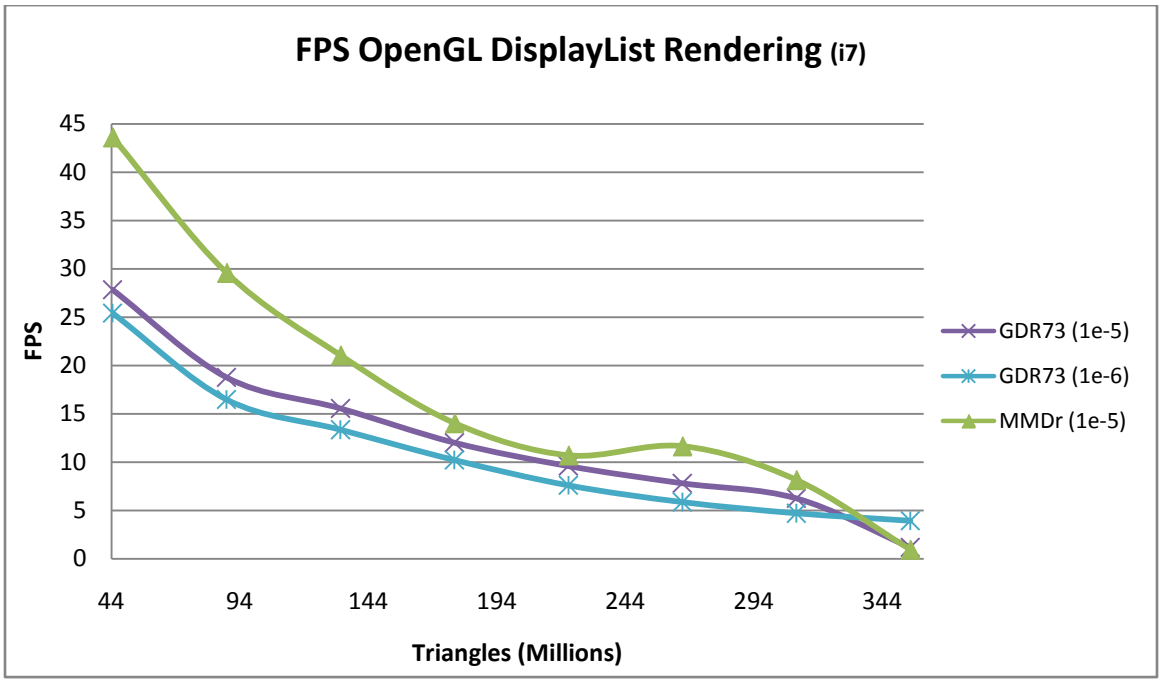

Figure 18: Render performance with a minimum amount of screen coverage culling

Figure 19 plots the difference in performance when a moderate amount of screen coverage is applied. For both of the runs, any shape or cell whose bounding box denotes it would contribute to 42 pixels or fewer on the screen is culled. The data clearly shows that the spatial hierarchy is capable of achieving a significantly higher frame rate at this level of screen coverage culling. This is most likely due to the existence of many more small cells

within the spatial hierarchy than there are small shapes within the product structure. Note also that the spatial hierarchy frame rate seems to stabilize between roughly 180 million triangles and 270 million triangles at a respectable 35 fps. This is promising since it hints that a stable frame rate might be possible on arbitrarily large models, and it will be well above the values that are often considered interactive.

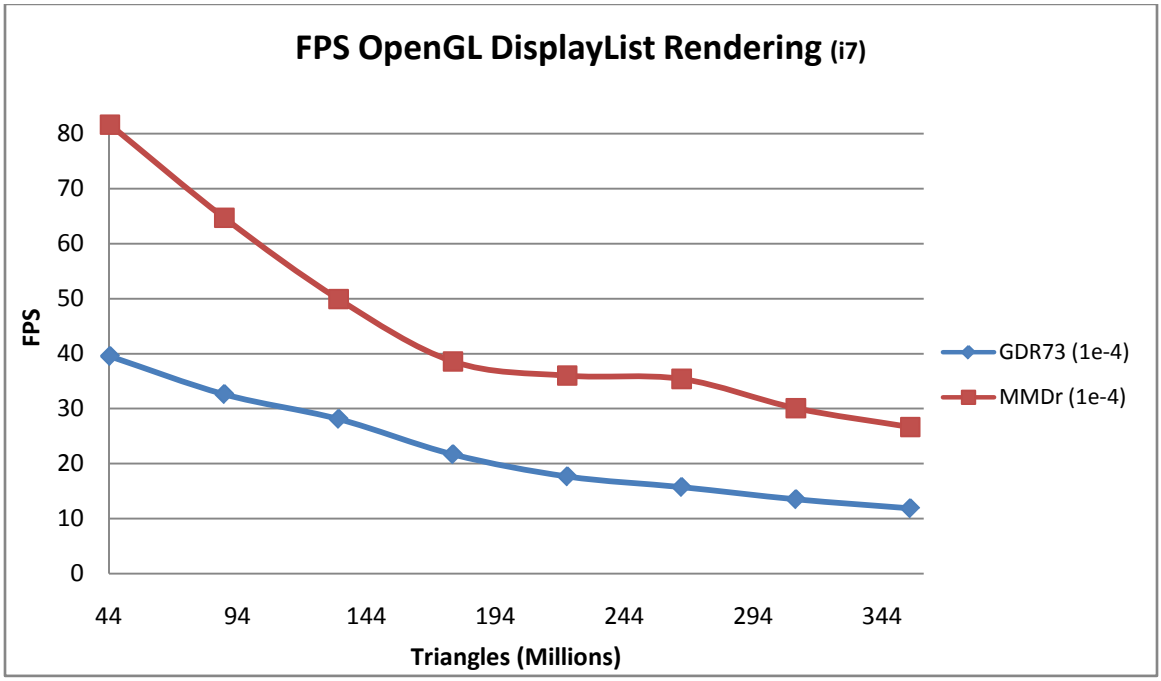

Figure 19: Render performance with a moderate amount of screen coverage culling

Figure 20 demonstrates the difference between rendering with minimal and moderate screen coverage culling when using a spatial hierarchy. This comparison is important because it shows that same curve is followed regardless of the amount of screen coverage that is used. In other words, screen coverage only seems to influence the maximum frame rate achieved and not how the performance will behave when increasing the number of triangles.

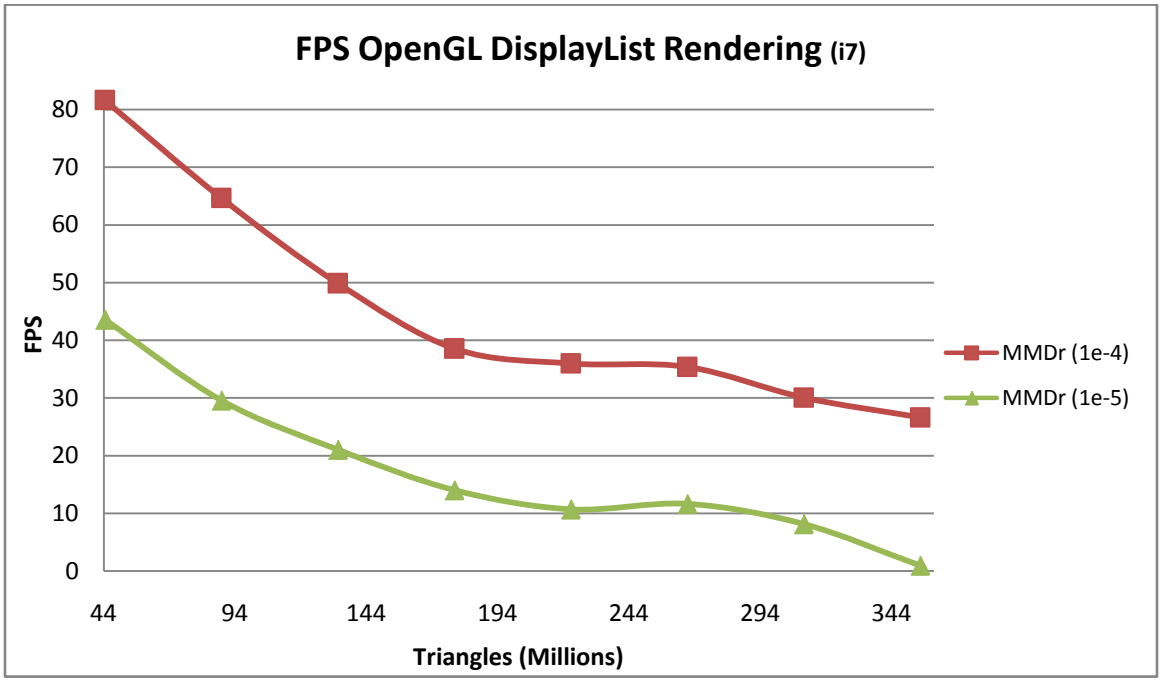

Figure 20: Comparison of rendering when using minimal and moderate screen coverage culling with a spatial hierarchy

Overall, rendering using a spatial hierarchy seems to show promise, especially when compared to rendering using a product structure alone. While the data does not show that interactive frame rate will necessarily be maintained as the data size increases, it does show promise towards actually being able to reach that goal.

4.3 Memory Management

In order to visualize arbitrarily large data sets at interactive frame rates, the amount memory required for partitioning and rendering must be kept to a minimum and safeguards put in place to effectively handle the condition where more memory is required than is currently available.

4.3.1 Partitioning Memory

The MMDr proof-of-concept does not report the maximum amount of memory used when partitioning a dataset. This was an oversight in implementing data statistics. MMDr was designed to ensure that the total amount of memory used does not result in swapping. It is intended to cap its memory usage just below the amount of memory available in the machine; however this was not working correctly at the time of testing. This was most evident when trying to extract the Boeing 777 model on the i7. Memory would quickly approach almost twice the amount available in the machine before MMDr would crash. In order to work around this issue the Nehalem with 24GB of memory had to be used, which is not practical for most users.

4.3.2 Rendering Memory

The amount of memory used by the spatial hierarchy and product structure was also recorded when collecting the frame rate information for a variety of data sets. Therefore, it is also possible to look at the amount of memory required as the number of triangles in the dataset increases.

Figure 21 shows the amount memory used for rendering when a minimal amount of screen coverage culling is applied. The use of the spatial hierarchy does not seem to fare well under this scenario, since it practically always uses a significantly greater amount of memory than the product structure alone. For some reason, it seems to recover at 350 million triangles, however, this may just be a fluke in the data collection since there does not appear to be a good reason for this to occur. One encouraging observation can be made from this data is that even though the rendering performance as shown in Figure 18 did not increase

much between 1 and 4 pixels, the amount of memory used actually decreases substantially as a large number of triangles are rendered.

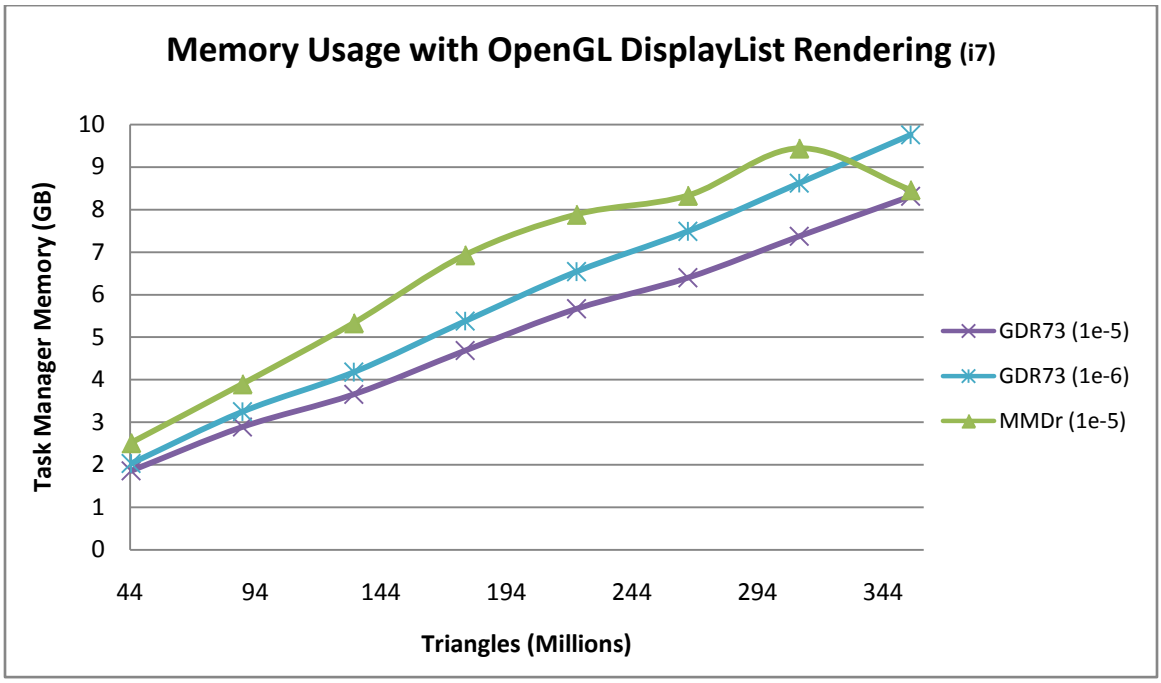

Figure 21: Memory usage when rendering with minimal screen coverage culling

Figure 22 shows the amount memory used for rendering with a moderate amount of screen coverage culling. The spatial hierarchy in MMDr seems to fare better under this scenario. Up until about 210 million triangles it seems to match the performance of the product hierarchy at which point it drastically shifts towards using much less memory, but then still continues to increase at about the same rate as it was previously. While not necessarily desirable, it is worth pointing out that the amount of memory used increases linearly as the number of triangles increases. This is not totally unexpected, considering that the data set uses instances of the same dataset, rotated across 360 degrees. In other words, each new data point should result in roughly the same amount of geometry being loaded.

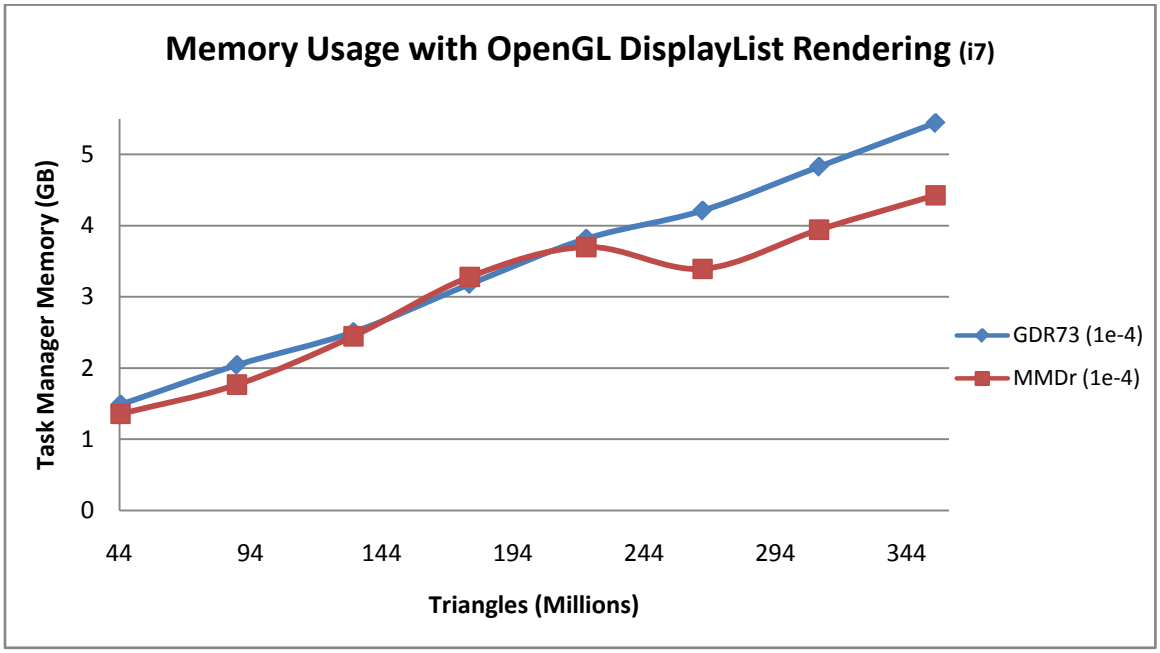

Figure 22: Memory usage when rendering with moderate screen coverage culling

Figure 23 shows that moderate screen coverage culling results a significant reduction in the amount of memory used for rendering when compared to minimal screen coverage culling especially as the number of triangles increases.

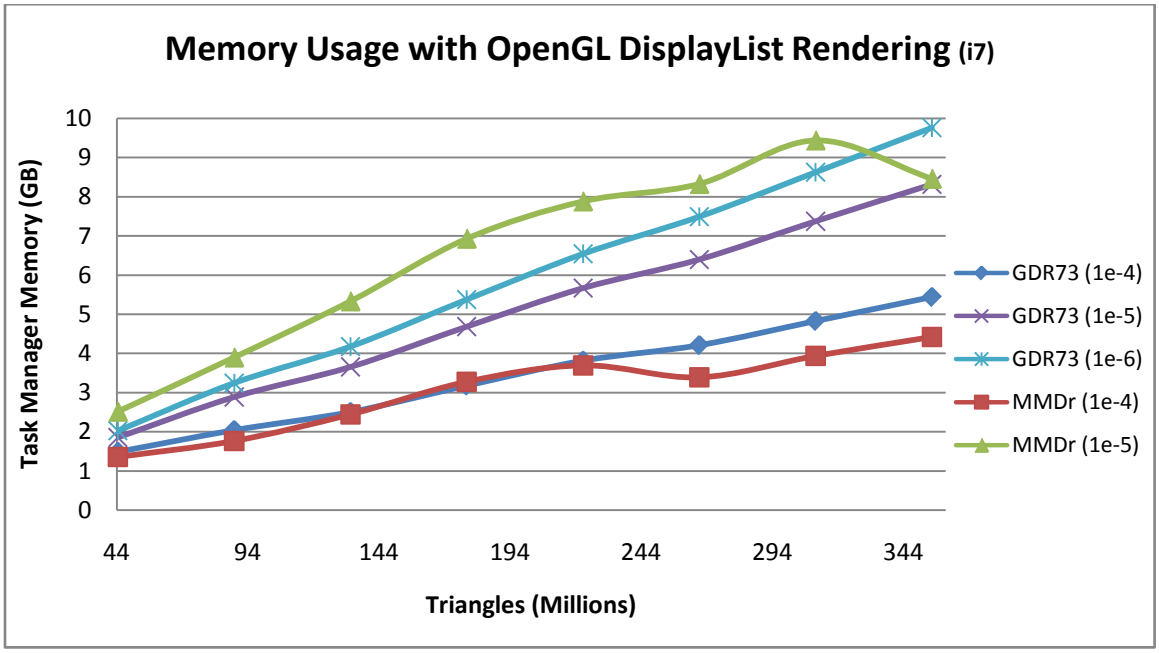

Figure 23: Memory usage when rendering

Overall, the amount of memory used by the spatial hierarchy based viewer was disappointing. The biggest issue with arbitrarily large data sets is that it is impossible to load them completely in memory. The linear increase in memory usage means that eventually a point will be reached at which rendering performance will fall off precipitously due to required memory exceeding that which is physically present on the machine in question. This effect could probably be demonstrated on the current data if run on a machine with less memory than the i7.

5 CONCLUSION AND FUTURE WORK

5.1 Summary and Conclusions

The MMDr proof-of-concept shows that there are gains to be had even with a minimal transition from current brute force rendering methods to those that make use of spatial hierarchies. Furthermore, a simple three-component system is all that is needed to achieve interactive frame rates on relatively large models, and can easily be expanded to work on arbitrarily large models.

Whereas most Massive Model Visualization approaches utilize complicated algorithms and highly specialized data structures to achieve interactive frame rates, MMDr shows that similar results can be achieved by far simpler means which are far more conducive to being able to achieve a complete visualization solution.

5.1.1 Partitioning

Partitioning a product structure into a spatial hierarchy can be achieved in a reasonable amount of time. While it cannot be done on the fly, especially for arbitrarily large models, it is well within the reasonable limits of what an average user would be willing to tolerate.

A general algorithm for sub-dividing an arbitrarily large dataset into any one of many different types of spatial hierarchies has been shown. While there is definitely room for improvement, this algorithm shows that managing memory consumption while being able to fully utilize all available threads is realizable.

5.1.2 Rendering

The use of nothing more than a simple spatial hierarchy combined with occlusion queries is shown to far surpass the rendering performance of an ordinary product structure renderer. This was the case even when comparing against the highly optimized rendering pipeline of the well established and commercially available DirectModel toolkit

5.1.3 Memory Management

Lazy memory management is sufficient to load and unload heavyweight geometric data; however it is adequate at best. By only loading data whenever it is needed, there is a greater chance that lag or artifacts will be introduced into the system while waiting for the data to load. This is a tradeoff since processing activities will stall while waiting on the data to load, whereas rendering may contain artifacts in areas for the frames rendered between the time when the data is first requested and the frame in which it is finally loaded.

5.2 Future Work

There are many paths that are open for pursuit in this area of research especially when it comes to enhancing the capabilities of the MMDr proof of concept.

5.2.1 Partitioning

Partitioning has been shown to utilize the capabilities of multi-core and multi-processor systems, however the performance gains quickly tail off as the number of threads increase. It is suspected that this is due to a large amount of memory allocation contention between the threads as well as significant amount of serial preprocessing that must occur in order to ensure the proper balancing of work between threads.

The memory contention issue can best be handled by further isolating memory allocations and limiting the frequency at which they occur. A good place to start would be eliminating unnecessary object creation and tear down that can occur on a per-cell basis, however this alone is probably not enough. Memory analysis tools will have to be applied to identify, analyze, and resolve the bottlenecks.

Implementing a multi-threaded method for splitting of single cells will optimize the serial preprocessing. While in the long run it is more efficient to process cells independently of one another, a level of parallelization can still be achieved on the early-generated cells due to the large number of shapes that will need to be processed. This would allow the initial processing of multi-thread single cell extraction until there are sufficient cells waiting to be handled so that the processing of individual cells can be efficiently farmed out to all of the threads.

While spatial partitioning has been shown to work on some impressively large data sets, it was barely capable of handling arbitrarily large data sets. The problem stems from the partitioner not correctly handling the scenario in which the input data set is so large that it causes the amount of available memory to be exceeded. Logic was inserted to preprocess cells until it was possible to handle groups of the remaining cells in main memory; however this appears to not to work as intended. The issue here is twofold. The prediction algorithm for the amount memory that is going to be required does not accurately predict the maximum amount of use. Second, analyzing the resulting data after performing an extraction hints at a severe memory leak that is causing the amount of memory use to be significantly larger than it should be. Effort needs to be expended to first track down the memory leak and then

memory usage analysis must be conducted in order to come up with a more accurate prediction algorithm.

The current partitioning scheme sub-divides any cell whose complexity, or number of triangles, exceeds the maximum complexity number set on the underlying spatial tree. While this approach is efficient for ensuring that the spatial hierarchy is sufficiently spread out, it can easily result in an arbitrary number of small cells being created. This not only increases the overall memory size, but also has the potential for causing a decrease in performance due to needing to render extra cells. An extra step should be added to the partitioning to automatically merge relatively small child cells with their parent cell. Performance measurements will need to be gathered before and after merging small child cells in order to determine if this approach improves memory usage and performance and if so, conduct further tests on a variety of models to determine when this approach is appropriate.

5.2.2 Rendering

One of the biggest shortfalls of the MMDr rendering is that it currently does not maintain the material colors from the original data set. Future versions should be enhanced to include this. This will have impact on both the partitioning and rendering logic. Partitioning will need to be enhanced to support attaching a material to triangles associated with each part contained within the cell. Further the triangles will need to be sorted so that triangles with the same material are next to one another. This is needed to ensure that the number of state changes when rendering the triangles of each cell is kept to a minimum. Testing within the DirectModel rendering system has shown that forgoing this optimization can easily result in over a 20 percent reduction in rendering performance. The shape

rendering logic will need to be enhanced to support rendering multiple sets of triangles with a different material being applied to each.

Rendering performance for the spatial hierarchy is quite impressive, however it can optimized further. Currently, it does not scale as well as expected when the size of the data set increases. This is believed to be an artifact of far too much geometric data being rendered, such that even with occlusion query and screen coverage culling it cannot keep up, therefore the addition of further culling methods will need to be considered.

One option is to consider enforcing a fixed frame rate, which is nothing more than providing a means for the rendering loop to terminate early in order to ensure that each frame does not exceed a maximum allocated time. Usually this involves calculating the duration of the last frame rendered and its associated number of triangles, then extrapolating the number triangles that can be rendered in the next frame while still rendering at the desired frame rate. Since cells are always rendered front to back the cells that are culled by this method will almost always be far away from the user and therefore may not contribute much to the quality of the frame.

5.2.3 Memory Management

Memory management is currently the weakest link in the MMDr proof of concept. The lazy loading of data often results in artifacts during rendering due to the necessary geometric data for a cell being available. A more active approach would predict the cells that will be needed next, and ensure that they are loaded before they are actually needed. Further, this approach can be adapted to remove cells that will no longer be necessary.

Another deficiency is the current mechanism for setting a cell's loading priority based solely upon its area. This is based upon the misguided premise that a larger cell will always contribute more to the screen that a smaller cell. This is not case, especially when the larger cell is significantly farther away from the view location than a smaller cell. The loading priority of each cell therefore needs to be adjusted based upon a combination of its size and distance.

5.2.4 Massive Model Visualization

In order to fully enable massive model visualization, there are key technologies that need to be developed and integrated in the MMDr proof of concept.

The spatial partitioner and its underlying data structure need to be enhanced to support the dynamic updating of the spatial hierarchy based upon changes in either the original product structure or spatial hierarchy. Intervis3D showed that is much more efficient to update only a small portion of tree than having to constantly update the whole tree. Further, this capability is required in order to support advance features such as manipulating parts on the fly.

The spatial tree also needs to be enhanced to support picking. Further picks against the tree should automatically be translated into picks against the parts in the original product structure such that the user feels as though they are working against the product structure.

A key component to any visualization application is the ability to manipulate the location and orientation of any part within the product structure. In order for any massive model visualization solution to be considered complete it needs to be able to support this feature. The MMDr proof of concept was designed with this in mind and therefore has

already started to lay the ground work needed to make this possible. Further enhancements will need to occur including those listed above, however unlike previous solutions, being able to support manipulation is definitely within the realm of possibilities for proposed solution.

5.3 Acknowledgments

The work contained within would not be possible without the countless support and involvement of many individuals. My gratitude and thanks go to the many individuals who have made this possible.

I would like to recognize my committee, Dr. James Oliver, Dr. Yan-Bin Jia, and Dr. Eliot Winer. Thank you for understanding that I am both a graduate student and a full-time engineer and being so supportive of me doing my own independent research. I would especially like to thank my advisor Dr. Oliver for all the guidance he has provided over the years. I have learned a lot from him as undergraduate research assistant, as an engineer, and as graduate student and I suspect I will continue to learn a lot more. When it came time for me to go back to graduate school, there was no doubt about who I wanted to be my major professor.

I would thank Dr. Adrian Sannier. It takes an amazing professor to put up with a student asking every class for an entire semester to be hired as undergraduate research assistant. I am forever grateful that he was willing to take chance on me and even more so for the countless opportunities that has presented to me.

I would like to thank the many group members that I have had the opportunity to work with throughout my classes. An extra handful of gratitude goes Adam Faeth, Mike

Oren, and Eric Marsh for all the hard work they put into our project. I would especially like to thank Adam for his friendship and the countless hours he has spent with me working on course work and for not giving up when given the task of proof reading my thesis. I would also especially thank Adam's wife for putting up with me for constantly stealing her husband for hours on end.

I would like to thank my countless colleagues at Siemens PLM software. Thank you so very much for putting up with both my crazy schedule and me being an Oscar the Grouch from being tired all the time. I would like recognize Tony DeLuca, Brett Harper, Andreas Johannsen, and Mike Carter for their bountiful support as went I went back to graduate school. I would like to thank the members DirectModel team Sashank Ganti, Jianbing Huang, and Bo Xu, who bore the brunt and had to pick up the slack when I could not. I would like to especially thank Mike Carter for being my manager and my friend. This would not have been possible without his full and undying support. I am forever grateful for his willingness to put up with my crazy schedule, the countless discussion we have had about Massive Model Visualization, and the endless hours he spent proofreading this thesis.

I would like to thank Samuel Gateau from NVIDIA and David Kasik from Boeing. This would not have possible without their willingness to provide technical expertise and data.

I would like to give my express gratitude to Mike Brockert. For being my coach, my teacher, and my first true mentor especially in the area of CAD. I wouldn't be where I am now if wasn't for all the things that he has taught me.

I would also like to thank my family and friends for their endless support and for putting up with being so unresponsive at times do to the craziness of my schedule. A person is only as strong as the people standing behind them.

REFERENCES

[1] D. Kasik, "Visibility-guided rendering to accelerate 3D graphics hardware performance," , San Diego, California, 2007.

[2] Ulf Assarsson and Tomas Moller, "Optmized View Frustum Culling Algorithms," Technical Report 99-3, 1999.

[3] J.H. Clark, "Hierarchical geometric models for visible surface algorithms," vol. 19, no. 10, pp. 547-554, 1976.

[4] Ulf Assarsson and Tomas Moller, "Optmized View Frustum Culling Algorithm for Bounding Boxes," *Journal of Graphics Tools*, vol. 5, no. 1, pp. 9-22, September 2000.

[5] M. Meibner, and T. Huttner. D. Bartz, "Opengl assisted occlusion culling for large polygonal models.," vol. 23, no. 3, pp. 667-669, 1999.

[6] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E Hoff III, "Visibility Culling using Hierarchical Occlusion Maps," , Nice, 1997.

[7] Naga K Govindaraju, Avneesh Sud, Sung-Eui Yoon, and Dinesh Manocha, "Interactive visibility culling in complex environments with occlusion-switches," 2003.

[8] T Hudson et al., "Accelerated Occlusion Culling using Shadow Frusta," , 1997, pp. 1-10.

[9] Michael Wimmer and Jiri Bittner, "Hardware Occlusion Queries Made Useful," in *GPU Gems 2*.: Pearson Education, March 2005.

[10] Hanan Samet, *The Design and Analysis of Spatial Data Structures*.: Addison-Wesley, 1989.

[11] Hanan Samet, *Foundations of Multidimensional and Metric Data Structures*.: Morgan Kaufmann, 2006.

[12] B., DeRose, T., Lischinski, D., Salesing, D., Snyder, J. Chamberlain, "Fast rendering of complex environments using a spatial hierarchy.," pp. 132-141, 1996.

[13] Enrico Gobetti, David Kasik, and Sung-eui Yoon, "Technical Strategies for Massive Model Visualization," in *Proceedings of the 2008 ACM symposium on Solid and physical modeling* , New York, 2008, pp. 405-415.

[14] J Goldsmith and J Salmon, "Automatic cration of object hierarchies for ray tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14-20, 1987.

[15] J D MacDonald and K S Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 6, pp. 153-165, 1990.

[16] V Harvan, "Analysis of cache sensitive representations for binary space partitioning trees," *Informatica*, vol. 29, no. 3, pp. 203-210, 1999.

[17] Kimberly Weaver, "Design and evaluation of a perceptually adaptive rendering system for immersive virtual reality envrionments," Ames, 2007.

[18] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman, *Real-Time Rendering.*: AK Peters, 2008.

[19] Daniel Aliaga et al., "A framework for real-time walkthroughs of massive models.," 1998.

[20] Daniel Aliaga et al., "MMR: an interactive massive model rendering system using geometric and image-based acceleration.," , 1999, pp. 199-206.

[21] Enrinco Gobbetti and Fabio Marton, "Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms.," , vol. 24, 2005, pp. 878-885.

[22] Hugues Hoppe, "View-Dependent Refinement of Progressive Meshes," , 1997, pp. 189-198.

[23] Dirk Bartz et al., "Jupiter: A Toolkit for Interactive Large Model Visualization," in *Proceeding of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, Piscataway, NJ, 2001, pp. 129-134.

[24] Paolo Cignoni et al., "Adaptive tertrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models," , New York, 2004, pp. 796-803.

[25] Beat Bruderlin, Mathias Heyer, and Sebastian Pfutzner, "Interviews3D: A Platform for Interactive Handling of Massive Data Sets," vol. 27, no. 6, pp. 48-59, 2007.

[26] Wagner T Correa, James T Klosowski, and Claudio T Silva, "Visibility-Based Prefetching for Interactive Out-of-Core Rendering," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, Washington, DC,

2003, p. 2.

[27] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha, "Cache-Oblivious Mesh Layouts," , 2005.

[28] Perdro V. Sander, Diego Nehab, and Joshua Barczak, "Fast triangle reordering for vertex locality and reduced overdraw," , New York, 2007, p. 89.

[29] The Walkthrough Group. (2001, March) The Warlkthru Project. [Online]. http://www.cs.unc.edu/~walk/

[30] L Darsa, B Costa, and A Varshney, "Navigating Static Environments Using Image-Space Simplification and Morphing," , Providence, 1997, pp. 25-34.

[31] Francois Sillio, George Drettakis, and Benoit Bodelet, "Efficient Imposter Manipulation for Real-Time Visualization of Urban Scenery," vol. 16, no. 3, pp. 207-218, 1997.

[32] M Garlan and P Heckbert, "Surface Simplification using Quadratic Error Bounds," 1997.

[33] C Erikson and D Manocha, "GAPS: General and Automatic Polygonal Simplication," 1998.

[34] 3Dinteractive. (2009) 3Dinteractive. [Online]. http://www.3dinteractive.de/products/products.html

[35] (2009) JT Open. [Online]. http://www.jtopen.com/

[36] "Jt File Format Reference Version 8.1 Rev-B," Plano, 2008.

[37] Michael B Carter, Andreas Johannsen, Michael C. McCarty, Jeremy S. Bennett, and Bo Xu, "SYSTEM AND METHOD FOR LIGHTWEIGHT POLYGONAL TOPOLOGY REPRESENTATION," Software 11/837,236, August 10, 2007.

[38] M., Bartz, D., Hüttner, T., Müller, G., Einighammer, J. Meißner, "Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models," , 1999.