Interactive Rendering for Large City Models

by

Ali Abbas Dawood Lakhia

B. S. (University of Texas at Austin) 1998

A thesis submitted in partial satisfaction of
the requirements for the degree of
Master of Science

in

Computer Science

in the

Graduate Division
of the
University of California at Berkeley

Committee in charge:

Professor Avideh Zakhor
Professor Carlo Séquin

Spring 2004

The thesis of Ali Abbas Dawood Lakhia is approved:

Chair	Date
Professor	Date

Interactive Rendering for Large City Models

by

Ali Abbas Dawood Lakhia

Abstract

Recent acquisition systems, such as the one developed at the Video and Image Processing Lab at Berkeley, are capable of collecting large, detailed, highly textured models that standard levels of detail (LOD) rendering techniques [18] cannot handle efficiently.

We propose an out-of-core rendering engine which applies the cost and benefit approach of the Adaptive Display algorithm by Funkhouser and Séquin [18] to Hierarchical Levels of Detail (HLODs). Unlike the Adaptive Display algorithm, we do not skip objects to maintain interactivity when many objects are visible.

Funkhouser and Séquin apply hysteresis by adding a penalty in the benefit heuristics to discourage disturbing visual effects due to fast switching of detail in the model. However, this penalty may not be sufficient if the user is moving around rapidly in the scene. Instead, we have developed a more robust temporal hysteresis by retaining how much detail is rendered over a time period.

We have implemented our rendering engine to run on a common personal computer with a standard graphics card. The engine is capable of visualizing, in both walk-through and fly-through mode, a detailed model of 25 city blocks comprised of 8.9 million triangles and 937 million color pixels. Our engine maintains a constant frame rate while limiting excessive flickering.

Contents

Chapter 1

Introduction

1.1 Motivation

Interactive visualization of 3D scenes is used in different contexts such as architectural evaluation of buildings, CAD applications, game playing and simulations. These visualizations require some models to be constructed manually, generated procedurally, or acquired from a physical model in the first place.

Models created manually tend to be inherently simple. Objects, such as buildings, are usually modeled with perfect primitives, and are often axis-aligned. Also, texture is usually tiled and the same texture is used for many objects.

Procedural generation, for instance, may use an extended L-system as well as user-specified parameters, such as density maps and building height maps, to create an entire city [28]. However, such systems only have a finite number of building classes. Each building class generally uses simple extrusions to create building modules.

Most acquisitions, in the past, have yielded simple models. For example, Jepson and his team at UCLA created models by using aerial images to identify features such as street widths and building foot prints [23]. They represented objects such as buildings, as simple rectilinear extrusions to the building height. Photographs were applied as texture to these highly simplified geometric shapes to generate the final models.

To summarize:

- Manually constructed or procedurally generated models consist of simple primitives and small textures that are tiled.
- Traditionally acquired models have been limited in complexity or size because:
  - the acquisition process is only semi-automated and requires manual intervention, which slows or hinders acquisition,
  - the acquisition takes place in a stop-and-go fashion and needs considerable setup time, and

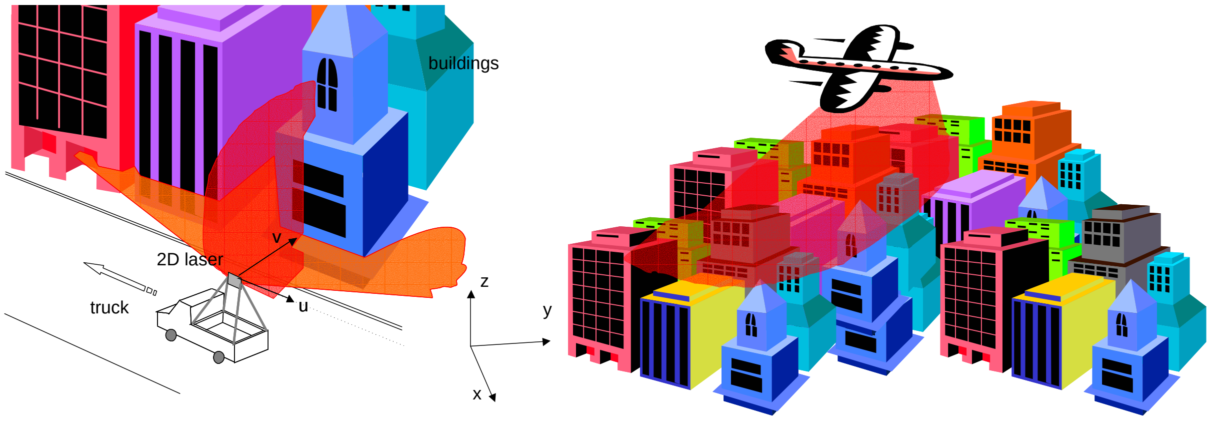


Figure 1.1: (a) The ground acquisition setup (b) Acquisition of aerial data

– the acquisition was not proven to scale up to larger models.

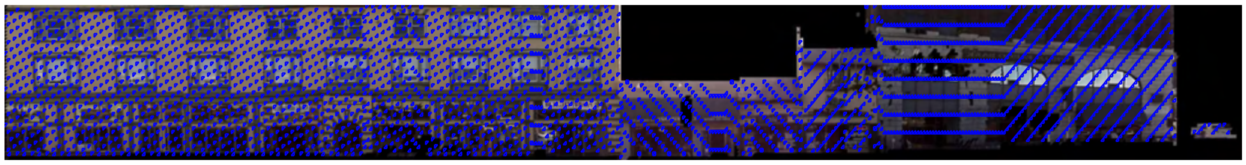
Consequently, rendering and interactive viewing of such limited models has been possible on today's powerful hardware without much difficulty.

Recent acquisition methods, however, require little human intervention and are capable of collecting large and detailed models. One example of this is an acquisition system developed at the Video and Image Processing Lab at Berkeley which is capable of rapidly acquiring large, detailed, highly textured models of urban environments from the ground level by using two 2D laser scanners and digital cameras [13, 14].

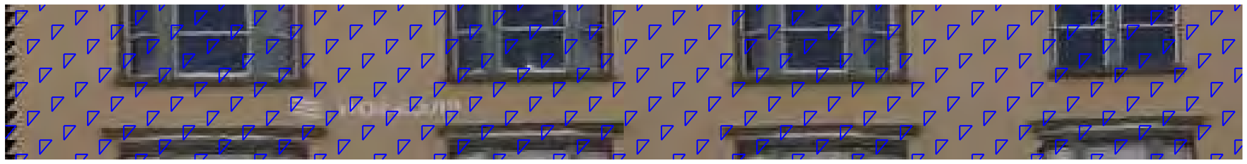
The ground acquisition system is mounted on a truck being driven on public roads as demonstrated in Fig. 1.1 (a). During motion, the vertical scanner captures the shape of the building facades, whereas the horizontal scanner is used for position estimation by matching successive horizontal scans. The spacing between vertical scans depends on the speed of the truck and traffic conditions and is typically about 10 cm. Data is acquired continuously, rather than in a stop-and-go fashion. Finally, the data is post-processed automatically without any manual intervention to produce triangular meshes with large texture maps [12].

As an example, a 20-minute acquisition, followed by automatic post-processing, yields a complex city model consisting of 25 complete blocks of building facades, where none of the buildings share the geometry or the texture with any other building. Specifically:

- Buildings that are "boxy" are represented by tens of thousands of triangles instead of a single cube
- Buildings that are of uniform color still carry the same amount of texture as a



(a)



(b)

Figure 1.2: (a) The highest level of detail texture map of a block in Berkeley with one in every ten triangles overlaid in blue on top of the texture. Note that the texture is not tiled and the triangles are not simplified or collapsed to form larger ones. (b) A closeup view of the texture map.

building with non-repetitive texture, since none of the texture is tiled

This can be seen in Figs. 1.2 (a) and (b).

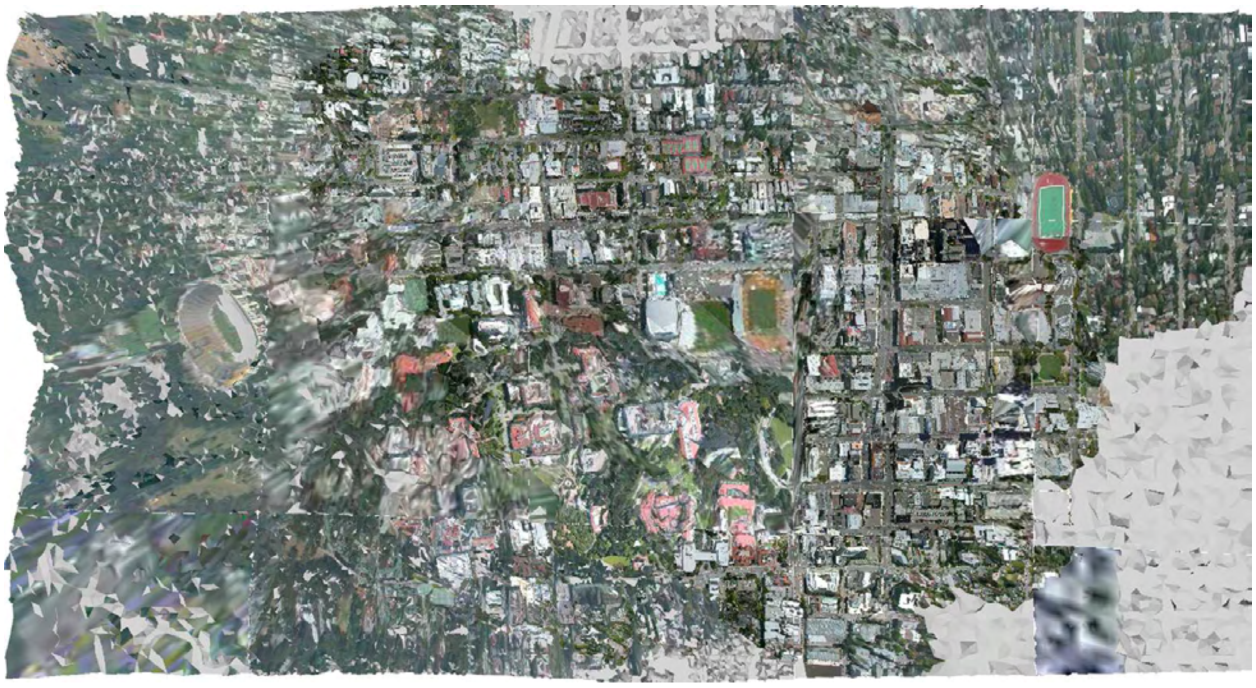
Far-range Digital Surface Map (DSM) data and aerial imagery, acquired as shown in Fig. 1.1 (b), are then registered with respect to the ground-based model and merged to create a single model [15]. This final city model has about 8,950,000 triangles. The texture consists of over 937,000,000 pixels in full color or about 2680 MB of uncompressed data. An overview of the entire model can be seen in Fig. 1.3 (a). A closeup in Fig. 1.3 (b) shows the detail in the model.

Although most rendering algorithms are theoretically capable of using arbitrary data, the complexity and quantity of data that can be viewed interactively is extremely limited. Rendering a large model, such as this data set, poses new and interesting challenges that have not been addressed by previous work. The primary goal of this work is to address the real-time interactive rendering of such large models.

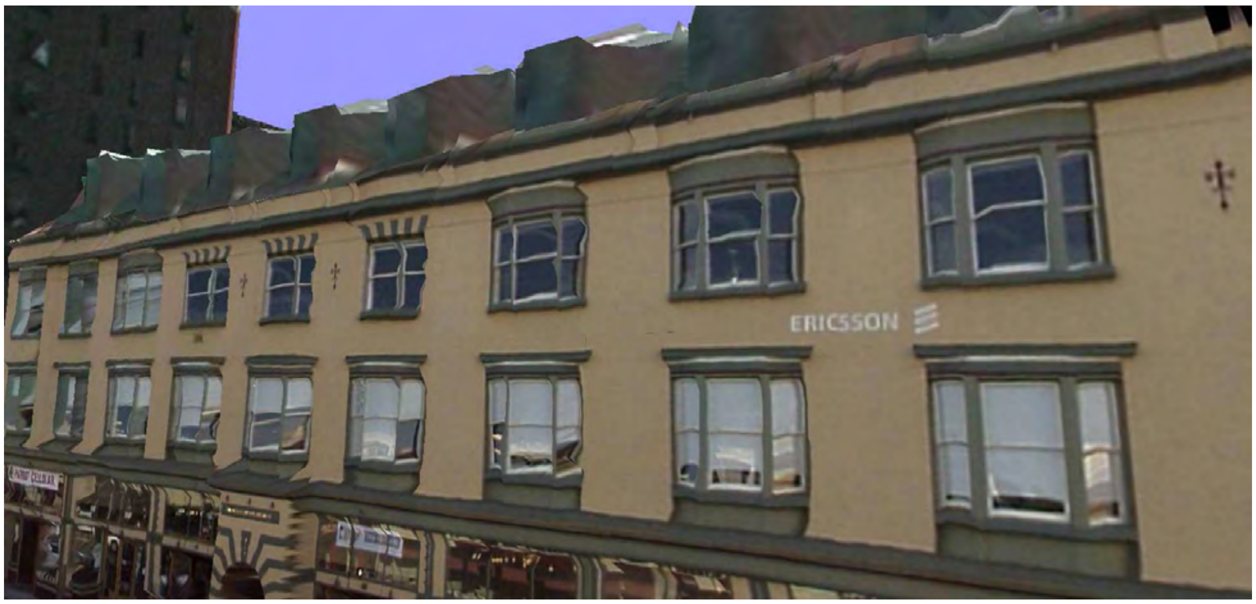
1.2 Goals

Hardware

Our goal is for our rendering system to run on a common personal computer with a standard graphics card and average amount of system memory.



(a)



(b)

Figure 1.3: (a) The entire city model, as seen through our rendering system, has about 25 blocks. The grey regions are triangles that do not have corresponding texture; (b) Closeup of one of the blocks in the city model.

Model

The rendering system should be independent of the size of the model. The model may not necessarily fit in system memory and dynamic loading and unloading of meshes must be performed to do out-of-core rendering.

One could reduce the size of the model to enable interactive rendering. For example, one could throw away a lot of triangles that approximately lie on the same plane. Similarly, one could remove repetitive texture and replace it with a titled texture. However, small changes in geometry and texture are still perceptible if seen from a certain view-point and at a certain distance. That is, rather than throwing away data, our goal is to manage the full details in the model to and solve the associated interactive rendering issues.

Performance

This rendering system should:

- maintain a specified frame rate to ensure an interactive experience for the user,
- utilize as much of the render time as possible by adding detail where it is most observable by the user,
- not skip rendering of objects that can be seen by the user, and
- preserve some coherency between frames and minimize switching between discrete levels of detail.

Interface

The rendering interface should:

- allow the user to stand still or move at arbitrary velocities,
- permit use of all 6 degrees of freedom so that the user may move about in an unrestricted manner,
- let the user go anywhere and not be confined to a pre-defined path.

1.3 Organization

The thesis is organized as follows. Chapter 2 reviews previous work in interactive visualization of large scenes. We also explain why these approaches were not suitable to meet our goals. In Chapter 3, we describe how we create the HLODs and show the pre-processing step. Chapter 4 elaborates on our rendering engine. We compare our approach in Chapter 5 to the Adaptive Display Algorithm and present overall results in Chapter 6.

Chapter 2

Previous Work

Significant progress has been made in the past few decades to allow ever more complex geometry and larger models to be viewed interactively.

2.1 Representation of Data

Polygon Representation

Polygon representations are most popular and their rendering is well optimized for most, if not all, graphic hardware architectures. Levels of detail (LODs), introduced by Clark in 1974, consist of a hierarchy of objects at ever simpler representations [4]. He used the appropriate representations to improve interactivity.

Rendering LODs of large objects, however, are less optimal. For example, consider a slanted view of a building facade in Fig. 2.1 (a). Note that a coarse representation of the object is ideal for the portion of the object that is far away from the camera but is too coarse near the camera. Similarly, a highly detailed LOD provides good detail near the camera but wastes too many triangles for detail that is not perceptible from that position.

One possibility to overcome this is to break large objects into smaller pieces. However, smaller pieces restrict simplification locally to that piece and yield substantially sub-optimal LODs at the coarsest levels. The use of a hierarchy of LODs, or HLODs, was proposed to overcome suboptimal use of LODs [10].

Progressive meshes are a flexible representation of polygon geometry that can be adaptively tailored to produce different LODs [21]. Such a mesh is simplified iteratively by collapsing some selected edges. If the edges to be collapsed are chosen carefully, annoying visual “popping” can be minimized.

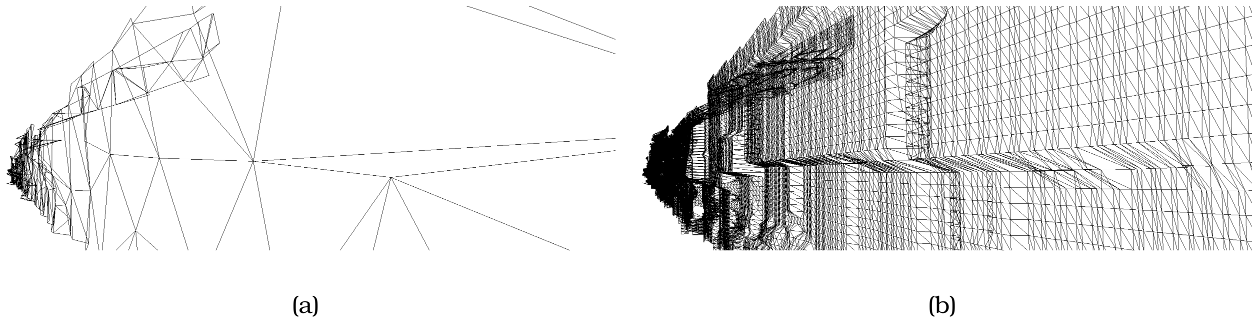


Figure 2.1: (a) LOD of large object is too coarse near the camera; (b) Using a higher detailed representation wastes detail that is far away from the camera.

Geometry that is stored as triangle strips provides a compact representation that saves memory, reduces bandwidth requirements, and takes less time to render [11]. This technique was improved to generate triangle strips for LOD meshes [2] and in real-time [32]. Nonetheless, such techniques do not make render time output-dependent and rendering of large models require some management to ensure interactiveness.

However, progressive meshes are not able to fully utilize the graphics hardware accelerations since the geometry changes frequently as the user moves around. In addition, texture mapping of progressive meshes is quite cumbersome [30].

Point-based Representation

Point-based rendering is another technique that scales well with complexity [24]. Rusinkiewicz and Levoy describe QSplat as a system that uses spheres of varying sizes to form the model. Detail is added by replacing larger spheres with smaller ones [29].

Unfortunately, the large flat surfaces and creases usually found in buildings typically require a very large number of points to maintain fidelity. Consequently, less detailed point-based representations would either have holes or be grossly inaccurate. Therefore, less detailed triangular representations are better suited than less detailed point-based representations.

Consequently, hybrid approaches were developed that use both polygons and points as rendering primitives [3, 5]. However, these algorithms have exploited only minimally the acceleration available from graphics hardware.

Dachsbacher et al. proposed a way to convert a hierarchy of points and polygons into a linear list that could then be rendered quickly by graphics hardware with minimal CPU load [7]. The trade-off with this approach is that hierarchical culling cannot be performed since parent-child relationship is lost in a sequential list.

Other Representations

Height fields are often used for terrain visualization [9, 25, 27, 33]. However, our city model has detailed building facades as well as aerial data. Thus, our model cannot easily be represented as height fields and these strategies are not applicable to our problem.

Maciel and Shirley introduced the use of image-based “impostors” to replace the underlying 3D geometry [26] and their idea was refined by others [8, 31]. Such an approach works well for highly detailed polygonal models with little or no texture since it trades off geometric complexity with texture management. However, this trade-off is less desirable for models, such as ours, with large texture maps.

2.2 Culling

Visibility culling rejects objects that are invisible and are guaranteed not to contribute any pixels to the screen. Three different culling techniques are in use. The first one is frustum culling, which is very effective and adds minimal overhead [1]. Back-face culling eliminates surface geometry that faces away from the viewer [22] and is used readily in scenes consisting of solid objects. Both of these culling methods are classical in the graphics community and various hierarchical and optimized algorithms exist.

Occlusion culling, on the other hand, requires significant processing. Recent variations and approximate algorithms, described in a recent survey paper [6], have improved the overhead to some extent. However, in many cases, like fly-throughs, the payoff for occlusion culling is very limited.

To summarize, culling can help reduce complexity of a large dataset. However, it is not effective enough to make the render-time output-dependent instead of input-dependent. Therefore, other techniques have to be used in tandem with culling algorithms.

2.3 Data Management

Memory management

Memory management ensures that memory usage does not exceed available capacity, and tries to prevent forced, in-opportune swapping of data by the operating system between physical memory and the hard disk.

Funkhouser prevents swapping by asynchronously prefetching data as needed [17]. However, the prefetching algorithm cannot always guarantee availability of objects in memory and objects may still pop into view after they are loaded.

Varadhan and Manocha implement an out-of-core rendering engine using two processes: one that renders the scene, and one that prefetches HLODs [34]. However, if the prefetching heuristics are miscalculated or the user moves unpredictably, the rendering engine may stall while that HLOD is loaded. That is, the render time is dependent on the loading time of an object.

Detail management

The details of the model must be managed to prevent overload of the graphics pipeline and to ensure interactivity. Specifically, rendering too much 3D geometry will increase render time because it is approximately proportional to the polygon count. Rendering may also stall if the texture is too large to fit in graphics memory.

Funkhouser and Séquin were the first to realize that levels of detail can be used not only to reduce the complexity of the scene but also to limit it. They call their approach the Adaptive Display algorithm, where they use a heuristic to determine the ratio of cost and benefit of each object at each of its LODs. Furthermore, they equate the graphics pipeline load management problem to the multiple choice knapsack problem and offer an approximate solution that is at least half as good as the optimal [18].

However, Funkhouser and Séquin employ a simple 2-level hierarchy and have a one-to-one correspondence between all the LODs of an object [18]. Large data sets, such as ours, require the model to be broken up into smaller pieces at different LODs to prevent wasted detail, and to coalesce smaller pieces to enable higher simplification. The Adaptive Display Algorithm cannot generalize to such a hierarchy of LODs.

Maciel and Shirley use a hierarchy of LODs and imposters. They traverse the hierarchy in a bottom-up fashion in the first pass. In the second pass, they refine their selection [26]. Since they visit all the nodes of the hierarchy, the complexity of their algorithm is $\mathcal{O}(N)$. This limits scalability.

Erikson et al. traverse the hierarchy top-down and use a screen-space error metric to choose which HLODs are refined [10]. However, the refinement process does not directly consider the cost of each refinement and can result in significantly non-optimal use of render times. The polygon budget is simply based on previous frame render times, which can lead to frequent switching of HLODs between successive frames.

2.4 Other Approaches

Wand et al. suggest a novel rendering algorithm that is output sensitive [35]. They use a randomized Z-buffering algorithm that chooses dynamically from a set of random surface sample points to render the scene. Their data set uses the same geometry repeated numerous times to demonstrate their approach with a high triangle count.

Chapter 3

Model Preparation

3.1 Overview

We choose to use hierarchical levels of detail or HLODs to represent our model since they are more efficient than LODs. Also, static HLOD nodes better utilize graphics hardware acceleration and optimization techniques such as display lists. HLODs also allow hierarchical culling, substantially reducing CPU and GPU load.

Since all the parts in our model are large, we create HLODs by segmenting our acquired data instead of merging adjacent parts. Each part of the model is segmented into pieces that are further broken up into smaller pieces as more detail is added. This yields a hierarchy[1] shown in Fig. 3.1.

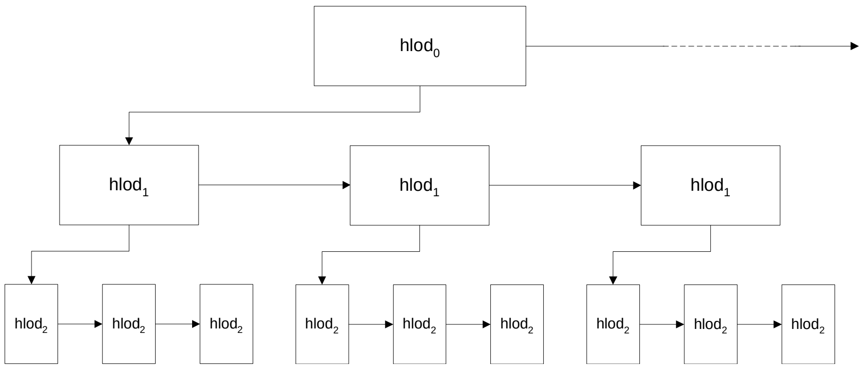


Figure 3.1: HLODs for a single block. The node $hlod_0$ is most coarse whereas $hlod_2$ is the most detailed. Note that $hlod_0$'s siblings are other blocks.

The last pre-processing step is to generate a scene description file that stores the HLOD hierarchy and caches computations that are used during rendering. Both of these steps are elaborated below.

[1]The subscript refers to the amount of detail in our HLODs.

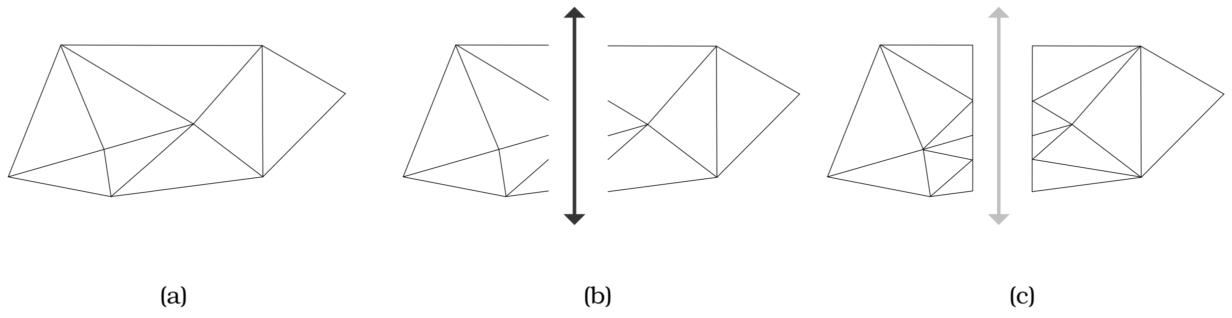


Figure 3.2: (a) A set of triangles before split; (b) Cutting plane is used to create two sets of triangles that are shown separated for clarity. New vertices are created on the plane; (c) The split triangles are re-triangulated.

3.2 HLOD Generation

Our city model is acquired[2] in a continuous fashion that is then split up into blocks. These blocks typically corresponds to one side of a city block.

Before we generate the HLODs, we define a constant that determines the desired number of triangles for all HLODs. We denote this constant as $c_{desiredTris}$[3]. We also define a ratio of number of triangles and size of texture between two successive levels of the hierarchy, $c_{levelFactor}$. These two constants are used for all the blocks.

For each block, b, we generate the least detailed HLOD, $hlod_0$, by simplifying it using Qslim Simplication Software [20] with a metric such that the number of triangles decrease by a factor defined below:

$$simplification_{hlod0} = \frac{numTriangles_b}{c_{desiredTris}}$$

Here, $numTriangles_b$ is the number of triangles in the block that is being processed. The original texture data is down-sampled by this simplification factor as well.

To create the next level in our hierarchy, $hlod_{d+1}$, we simplify the original block's triangle mesh and down-sample texture by a simplification factor computed from the previous factor:

$$simplification_{hlod_{d+1}} = \frac{simplification_{hlod_d}}{c_{levelFactor}}$$

We also create $c_{levelFactor} - 1$ cutting planes perpendicular to the longest dimension of the block that run from one end of the bounding box to the other end.

We apply each cutting plane to a block in order to separate triangles into two sets,

[2]For details on the acquisition process, refer to papers by Frueh and Zakhor [12–14].
[3]The list of all constants and their values are listed in Table 6.1

depending on which side of the plane the vertices fall. Vertices that form triangles across the cutting plane are split into 3 smaller triangles by introducing two vertices at the intersection of the two edges and the cutting plane. This technique, illustrated in Fig. 3.2, minimizes cracks that appear when HLODs at different levels are next to each other.

This division step yields $c_{levelFactor}$ pieces, where each piece corresponds to a node in the HLOD hierarchy. The division and simplification process is recursively repeated on each piece with a smaller simplification factor until the factor becomes less than 1.

The end result is a hierarchy where the top most node is most simplified and represents the entire block. The next level has $c_{levelFactor}$ pieces that collectively represent the parent. That is, each node in this level has more detail but represents an increasingly smaller portion of the entire block. This relationship holds all the way to the leaf nodes that are most detailed but contain the smallest piece of the block.

We repeat this procedure for all the blocks. One such block is shown in Fig. 3.1 where $c_{levelFactor}$ is 3. Our final data has a total of 114 $hlod_0$ nodes from 114 blocks. The total number of nodes in the hierarchy is 2296 and the maximum depth is 5.

Note that on average each node would have $c_{desiredTris}$ triangles. However, the actual number of triangles vary considerably due to the varying density of samples and varying effect of simplification along different sections of the block.

3.3 Other Pre-processing

After the hierarchy is created, the triangle geometry of each node is stored in a binary format on disk to facilitate fast loading. The textures are saved in compressed JPEG format for the same reason.

Lastly, the following hierarchy attributes for each node are saved on disk in a binary scene description file:

- first child,
- next sibling[4],
- unique ID,
- path to triangle geometry file,
- path to texture image file,
- bounding box,
- cost,
- static benefit, and
- average normal

[4]Instead of each parent linking to all the children, we create a linked list of siblings to allow arbitrary number of children.

The child and sibling information defines the parent-child relationship between all the nodes. The unique ID is used as a key for a hash table to quickly find a node in Section 4.3. Since each node's geometry and image are stored in a separate file for loading to be independent of other nodes, we must save the path to both files.

The last four attributes are pre-computed to prevent computing them each time the rendering engine loads a node or reloading them separately from disk. The bounding box is useful for culling in Section 4.2. The cost of a node, n, is approximately proportional to the time needed to render the node. It is the weighted sum of the number of textured and untextured triangles:

$$\begin{aligned} cost_n &= numTexturedTriangles_n \times c_{textured} + \\ &\quad numUnTexturedTriangles_n \times (1 - c_{textured}) \end{aligned} \tag{3.1}$$

The constant weight, $c_{textured}$, is calculated empirically by comparing the render time of a set of triangles both with texture and with flat shading.

The benefit of a node, n, is complex to measure and involves semantic and contextual information [18]. We compute a static approximation of the benefit in the preprocessing step as the geometric mean of the accuracy of its representation and its total surface area. Specifically, given that $triangles(n)$ is the set of triangles in n and $area_t$ is the area of triangle, t, we have:

$$staticBenefit_n = \sqrt{numTriangles_n \times \sum_{t \in triangles(n)} area_t}$$

That is, the static benefit of node, n, is calculated when n is 1 unit[5] away from the camera with no foreshortening[6]. This measurement is dynamically adjusted during the rendering phase as discussed in Section 4.2.

Since most of our nodes are facades of buildings, the orientation of most of the triangles is fairly uniform. Therefore, the average normal of a node serves as a good approximation to adjust for foreshortening during the rendering phase. The normal of n, $avgNormal_n$, is calculated by taking the average normal of each triangle, weighted by its area that is then normalized:

$$\overrightarrow{avgNormal_n} = \frac{\sum_{t \in triangles(n)} \left(\overrightarrow{normal_t} \times area_t\right)}{\| \sum_{t \in triangles(n)} \left(\overrightarrow{normal_t} \times area_t\right) \|}$$

Here, $\overrightarrow{normal_t}$ is the normalized normal of triangle, t.

[5]One unit is 1 meter or 39.37 inches for our implementation.

[6]Foreshortening is when an object appears smaller because of a slanted viewpoint.

Chapter 4

Rendering Algorithm

4.1 Overview

The overview of our rendering algorithm is shown in Fig. 4.1. Our implementation consists of two threads:

- the rendering thread, and
- the loading thread

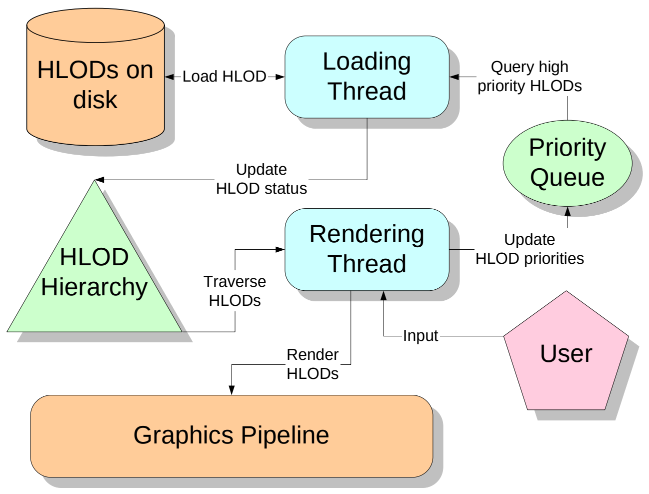


Figure 4.1: Overview of the rendering engine architecture.

The rendering thread traverses the HLOD hierarchy to render each frame within an allotted time by selecting nodes that are then sent to the graphics pipeline. After rendering each frame, the keyboard and mouse are queried for user input. With each traversal, the rendering thread computes and updates the priority of each node in a priority queue. The priority reflects how important the node is in the current state of the scene.

The loading thread, the other component of our implementation, queries this priority queue and asynchronously pre-fetches from the disk those nodes with the highest priority and unloads the nodes deemed least important from memory. Since the HLOD hierarchy is not completely resident in memory, each load and unload is reflected into the hierarchy by updating a status flag. The rendering thread uses this flag to determine availability in memory.

4.2 Rendering Thread

The primary contribution of this thesis is the node selection process. Node selection, as illustrated in Fig. 4.2, is dependent on the cost and benefit of a node, which determine its priority. The time slice is assigned by splitting the available render time among the node and its siblings based on the ratio of their priorities and their cumulative priority. This time slice is then recursively divided among each node's children.

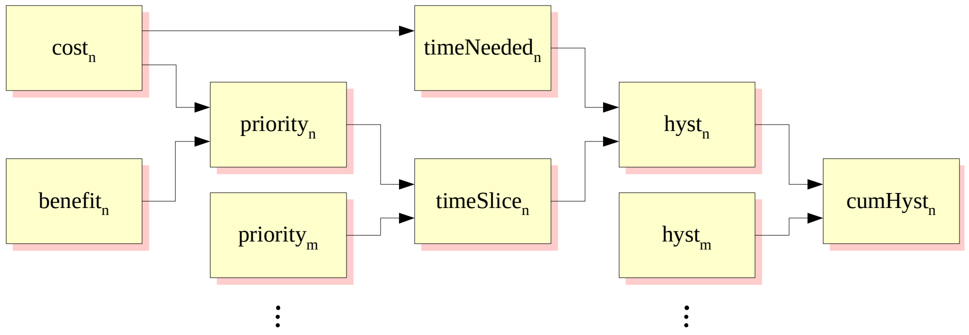


Figure 4.2: Dependency graph of each of the attributes of a node, n on other attributes of n and siblings m, ..., etc. The *cumHyst* attribute determines which nodes are selected and sent to the graphics card by the rendering thread.

Subsequently, the estimated time needed to render a node is compared with the time slice allocated to the node. This comparison is used to update a hysteresis counter for each node to limit excessive switching of HLODs. Finally, a cumulative hysteresis value is calculated for a set of siblings which determines if all the visible siblings should be rendered or not. These are all discussed in more detail below.

Initialization

Before rendering can begin, the rendering thread initializes by first loading the scene description file. This file provides scene data, such as a node's bounding box, and is kept in memory at all times to avoid recomputing it during run-time or reloading it from disk.

Next, we load all $hlod_0$ nodes into main memory. This ensures that a user traveling in an unpredictable fashion would at least see a simplified version of any node. Otherwise, such a node would not have been loaded, and the user would:

- lose interactivity while waiting for the node to get loaded, or
- see an empty space instead of the node[1].

We assume that all $hlod_0$ nodes take a small fraction of memory and, thus, not only can they all be kept in memory at all times, but that they also leave a large portion of memory unused for higher levels of detail. This storage penalty can be reduced by increasing the simplification factor between levels, $c_{levelFactor}$, or by increasing the depth of the hierarchy by decreasing $c_{desiredTris}$.

After the rendering thread has finished initialization, the loading thread is started at a low operating system priority.

Node Hierarchy Traversal

After initialization, the rendering thread recursively traverses nodes in the hierarchy in a top-down, breadth-first manner to render each frame. On traversal, each node's priority is calculated and maintained in a priority queue.

Since we need not visit all the nodes in the hierarchy, the running time is dependent on the target render time for one frame and the number of objects in the scene. Breadth-first traversal ensures that all siblings are queried before their children. If a node is visible and not loaded in memory, none of its siblings can be rendered either and the parent node must be rendered instead[2]. For this reason, it is preferable that each node not have too many children.

Priority Heuristics

Upon visiting each node, n, the rendering thread calculates its priority based on a benefit and cost ratio as done by Funkhouser and Séquin [18]:

$$priority_n = \frac{benefit_n}{cost_n}$$

Note that a higher priority indicates higher importance.

Recall that $staticBenefit_n$ is calculated offline for each node in Section 3.3. This benefit is measured as the approximate, unforeshortened screenspace of a node that is 1 unit away. We compute the benefit from the static benefit, by adjusting for distance, visibility

[1]Once the node finishes loading, it would then suddenly pop into view.

[2]Note that $hlod_0$ nodes are always kept loaded in memory.

and foreshortening:

$$benefit_n \quad = \quad \frac{staticBenefit_n \times vis_n \times foreshorten_n}{{distance_n}^2}$$

The distance is measured from n to the camera and is slightly adjusted based on the user's velocity to exploit temporal coherence.

We compute visibility, vis_n, by frustum culling. We do not perform occlusion culling because of its high overhead but it would be easy to incorporate it into our heuristics:

$$vis_n \quad = \quad 1 + (c_{vis} \times frustumVis_n)$$

Here, $frustumVis_n$ is 1 if n is in the view frustum. Otherwise, it linearly decreases to 0 based on how far outside of the view frustum n is. The constant, c_{vis}, weighs the importance of visibility.

Lastly, we adjust for foreshortening by multiplying with the dot product of the normal of the image plane, $\overrightarrow{view}$, and the average normal of n, $\overrightarrow{avgNormal_n}$. This is weighted by a constant, c_{fore}:

$$foreshorten_n \quad = \quad 1 + (\overrightarrow{view} \cdot \overrightarrow{avgNormal_n} \times c_{fore})$$

The cost of n, $cost_n$, is also calculated offline as shown in Equation 3.1. To summarize, the final priority for node n is:

$$priority_n \quad = \quad \frac{staticBenefit_n \times (1 + (c_{vis} \times frustumVis_n))}{cost_n \times {distance_n}^2} \times foreshorten_n$$

Time Slicing

We calculate $timeSlice_n$ or the amount of time that has been allocated to render n, in a top-down approach based on the overall time target, and n's relative priority with respect to the total priority of its siblings:

$$timeSlice_n \quad = \quad shareTime_n \times \frac{priority_n}{\sum_{i \in siblings(n)} priority_i} \tag{4.1}$$

Here, $siblings(n)$ is defined to be the set of all visible nodes that have the same parent as n. Thus, n is a member of this set if n is visible.

Also, $shareTime_n$ is the total time a node and all its siblings share. This is assigned a constant value, $c_{targetTime}$, for all n at level 0. Each node, n at $hlod_0$, recursively passes along its time slice, $timeSlice_n$, to its children to be shared among them based on

Equation 4.1. Thus, $shareTime_n$ is defined as:

$$shareTime_n \quad = \quad \begin{cases} timeSlice_{parent(n)} & \text{if } parent(n) \text{ exists} \\ c_{targetTime} & \text{otherwise} \end{cases}$$

Note that $parent(n)$ is the parent of n.

Next, we estimate the time required to render n, which is proportional to n's cost:

$$timeNeeded_n \quad = \quad v_{time} \times cost_n$$

Note that v_{time} is a variable that is adjusted in a feedback loop based on comparing actual time used to render a frame with the estimated render time needed for all the nodes rendered plus a fixed overhead[3]. The need for this feedback loop is due to the fact that most graphics hardware performance is based on factors besides number of triangles, such as the fill rate.

One possible implementation of the rendering algorithm is to keep recursing as long as:

$$shareTime_n \quad > \quad \sum_{i \in siblings(n)} timeNeeded_i$$

When the above condition does not hold for a set of siblings, their parent is selected to be rendered instead. This approach guarantees that the time slice of the parent is never exceeded by the children. Consequently, the total estimated render time will never exceed $c_{targetTime}$.

In practice, however, the feedback loop introduces excessive switching of the HLODs as v_{time} oscillates up and down. Therefore, we must extend the above approach to incorporate hysteresis.

Hysteresis

The Adaptive Display algorithm incorporates a hysteresis component as part of the benefit heuristics [18]. However, this approach can still cause switching of LODs as objects become visible or invisible [16]. Furthermore, we would like the loading to be independent of hysteresis.

Maciel and Shirley implement a counter that is incremented every time the algorithm wishes to switch from parent to children. The switching is allowed only if the counter exceeds a pre-fixed threshold [26]. However, this implementation is inflexible and does not account for nodes that need to be switched more urgently than others. Consequently, their frame rate exhibits dramatic variation.

[3]The fixed overhead cost is measured by disabling actual rendering.

We choose, instead, to increment or decrement the counter based on an urgency factor. Let us denote the counter by $hyst_n$ for node n. With each traversal of n, we update the counter:

$$hysteresis_n \longleftarrow hysteresis_n + \frac{timeSlice_n - timeNeeded_n}{timeNeeded_n}$$

We add a constraint to the above and restrict $hysteresis_n$ to lie in the range of $-c_{hystLimit}$ and $c_{hystLimit}$, where $c_{hystLimit}$ is a constant.

$$hysteresis_n \longleftarrow \begin{cases} c_{hystLimit} & \text{if } hysteresis_n > c_{hystLimit} \\ -c_{hystLimit} & \text{if } hysteresis_n < -c_{hystLimit} \end{cases}$$

That is, if the time slice of n is larger compared to the time needed to render n, then the hysteresis counter will approach positive $c_{hystLimit}$ over time. If both are almost the same, $hysteresis_n$ will remain near a 0 value. Otherwise, it will go towards negative $c_{hystLimit}$.

The approximate hysteresis counter value of 0 indicates that selecting that node to render will approximately use the render time allotted to the node. However, a node cannot be rendered without rendering its siblings. Therefore, we need a cumulative hysteresis value for all the siblings. Our approach is:

$$cumHyst_n = \sum_{i \in siblings(n)} trunc\left(\frac{hysteresis_n}{c_{hystSwitch}}\right)$$

Here, $trunc()$ truncates the floating point value to an integer by discarding the decimal value. This eliminates the least significant bits responsible for oscillations.

The constant, $c_{hystSwitch}$, determines the threshold for switching and should be between 1 and $c_{hystLimit}$. A higher value implies longer delay to switch HLODs but with a looser guarantee on how close the render time will be to $c_{targetTime}$. Conversely, a lower value implies that render times will be closer to $c_{targetTime}$ at the expense of more switching of HLODs.

Our rendering algorithm recurses breadth-first and updates the cumulative hysteresis value until we reach the last sibling. If:

$$cumHyst_n < 0$$

we render the parent of the siblings. Otherwise, we recursively visit the children of all the sibling nodes, again, in a breadth-first order.

Unlike Maciel and Shirley, we do not use the hysteresis counter to override switching of nodes [26]. Instead, our counter is sensitive to the urgency of a node needing to be switched. Consequently, we rely on the hysteresis counter alone to select the front.

Feedback Loop

Our approach can select nodes that are not loaded in memory. This leads to sub-optimal use of render time. Since availability of nodes in memory has high temporal coherence, we add a second feedback loop that gradually increases $shareTime$ for $hlod_0$ nodes if some nodes were not selected because their cumulative hysteresis value was below 0 and the actual total render time used was substantially below $c_{targetTime}$. Similarly, if the total render time substantially exceeds $c_{targetTime}$, we decrease $shareTime$ for $hlod_0$ nodes.

The feedback loop ensures that if there are nodes that can utilize the render time, they will have a chance to do so. Also, as nodes get loaded in memory, the actual render time will get larger. Consequently, the feedback loop will gradually cut back on the available render time to ensure consistent frame rate. This describes the final implementation of our rendering thread.

4.3 Loading Thread

Incremental Pre-fetching

The rendering thread starts rendering with only $hlod_0$ nodes in memory. The rendering thread relies on the loading thread to asynchronously query the priority queue and load high priority nodes that have not been loaded.

The loading is done incrementally by reading only a small portion of a node at a time. The priority queue is checked between each incremental load, and the previous loading is suspended and loading of a new node is started if the priority changes. This feature makes our loading more responsive to erratic movements by the user.

To limit memory usage and avoid swapping, we assume that memory usage for a node is proportional to its cost. Thus, memory usage is bounded by the set of nodes in memory, mem:

$$maxMemory \quad > \quad c_{memory} \times \sum_{i \in mem} cost_i$$

The loading thread prevents memory usage from substantially exceeding a fixed size, $maxMemory$, by unloading the nodes in the set mem with the smallest priority until the above condition is satisfied. The constant, c_{memory}, is established experimentally[4] and $maxMemory$ is based on the system resources.

[4]We cannot calculate storage requirements for display lists and find the constant experimentally.

Priority Queue Implementation

The loading thread uses a priority queue to maintain the list of nodes that are loaded and nodes that need to be loaded. Priority queues are traditionally implemented as heaps because they guarantee $\mathcal{O}(log(N))$ insertions and deletions and $\mathcal{O}(1)$ to query the highest priority node.

We chose to implement a different data structure: a doubly-linked list sorted in decreasing order of priority with a hash table to lookup any node in the list. This data structure works well for our needs because there is a high degree of temporal coherency between the updates that we make to the list of nodes.

For example, note that most new insertions of nodes would go at the end of the list because the priority of a node is very unlikely to become high too quickly to displace the most important nodes at the head of the list. Similarly, deletions of nodes will happen at the end of the list. In the worst case, we may need $\mathcal{O}(N)$ to add or delete a node. However, for the average case, our insertions and deletions would require $\mathcal{O}(1)$.

Priority updates for nodes in the list will also move the node a few places up or down the list between two successive frames. We can find a node in the list in $\mathcal{O}(1)$ time by using the unique ID as the key in our hash table. So, the overall priority update will also take an average time of $\mathcal{O}(1)$.

Lastly, a query for the most important node that has not been loaded must be performed. We avoid a $\mathcal{O}(N)$ search[5] for every query by caching the unique ID of the last node that is queried. We update the cache if an insert, delete or update causes:

- a node below the cached node to go above the cached node in the priority queue, or
- the cached node to move below an unloaded node in the priority queue,

This amortizes the $\mathcal{O}(N)$ cost for searching to $\mathcal{O}(1)$.

Consequently, our implementation of the priority queue provides better query, insertion, delete, and update times of nodes for the average cases compared to a heap data structure. It also scales well with the number of nodes in the hierarchy.

[5]A search with a heap would also take $\mathcal{O}(N)$ time.

Chapter 5

Discussion and Comparison

The nodes selected to render must provide maximum benefit but be within a specified cumulative cost. This cost is measured as the estimated render time for each frame. The optimal selection of such nodes is a known $\mathcal{NP}$-complete problem [19]. Therefore, the solution is approximated by using greedy heuristics [10, 18, 26].

In the following sections, we compare our heuristics with the Adaptive Display algorithm of Funkhouser and Séquin [16–18]. We also cover the strengths and weaknesses of both approaches.

5.1 Adaptive Display Algorithm

Funkhouser and Séquin utilize a simple hierarchy where each node has exactly one child or no children. Each node is assigned a “value” that is the ratio of the benefit and cost [18]. The selection of LODs to render is based on picking those nodes with the largest value that do not cause total estimated render time to exceed the target render time.

An analogous way to visualize this algorithm is illustrated in Fig. 5.1 where each representation of all the objects that are not culled away is plotted in two dimensions based on its benefit and cost. The LODs that represent the same object are linked with a line.

The selection process starts with a vertical line through the origin. This line is rotated clockwise and any node that goes through the line is selected as long as the total cost does not exceed the target render time. If a node is selected, other nodes connected to it are not considered for selection. The line continues to be swept until there are no more objects left to be selected.

This algorithm exhibits certain characteristics:

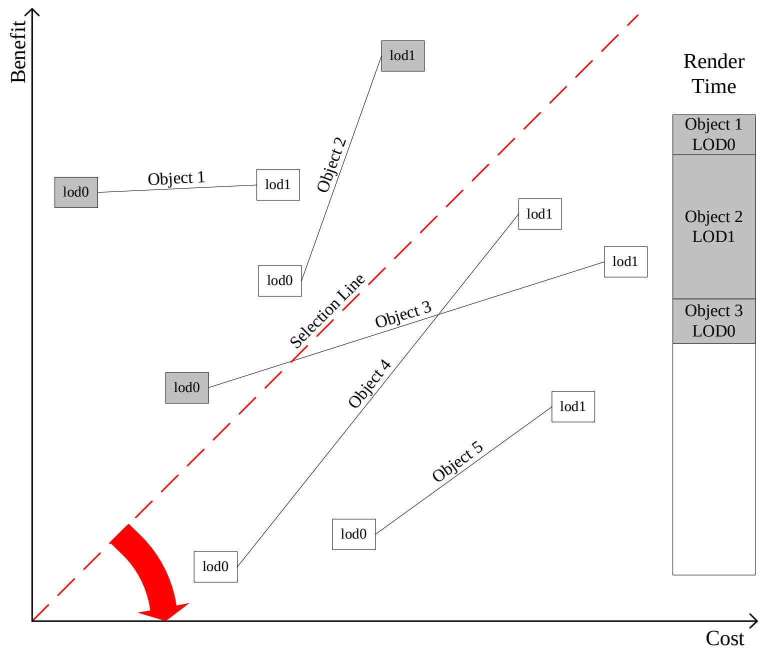


Figure 5.1: The Adaptive Display algorithm selects nodes by sweeping a selection line clockwise. The render time is shown on the right with selected nodes in grey.

Under-utilization of render time: Less detailed objects can be selected because they usually have very low cost. As shown in Fig. 5.2 (a), these nodes will be selected and some of the more detailed objects will not be selected in spite of having render time available for them.

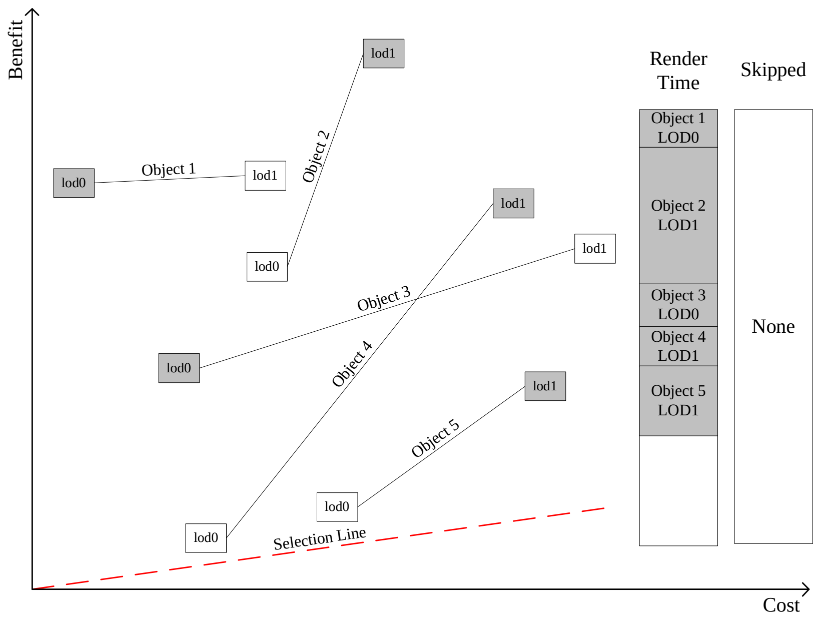
The Adaptive Display algorithm compensates for this with an incremental algorithm that updates the previous selection by iteratively replacing high value nodes with a node of higher accuracy [18]. Similarly, nodes of low value are replaced with a node of lower accuracy. The replacements continue until the same representation is upgraded and downgraded in the same iteration.

Over-utilization of render time: If the render time is small, or if very few nodes can be culled away, or some nodes use up a large portion of the render time, then the actual render time will exceed the target render time.

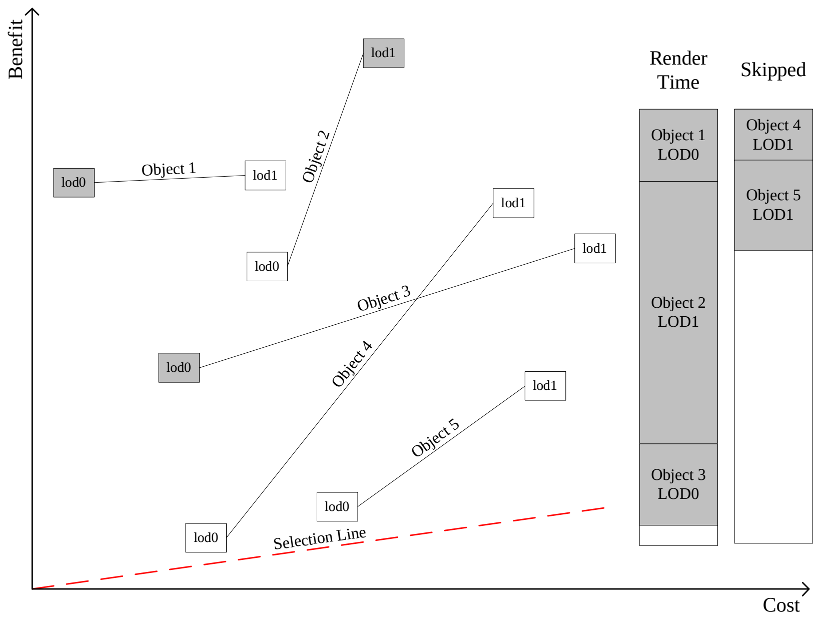
The Adaptive Display algorithm resolves this by skipping objects that, when rendered, would exceed the target render time [18]. This is illustrated in Fig. 5.2 (b). We prefer reduction in detail of some objects over skipping objects entirely.

Flickering from changing benefit: As the user moves around, some objects become more beneficial as they approach the camera. Other objects become less beneficial. One can usually find certain positions where a small movement will cause flickering of the LODs.

The Adaptive Display algorithm incorporates a hysteresis value into the benefit heuristic for each object. This value is proportional to the difference in the level of detail of a node from the one selected in the previous frame [18]. This scheme



(a)



(b)

Figure 5.2: (a) Render time is under-utilized because objects 1 and 3 could have been rendered with more detail; (b) Over-utilization of render time is avoided by skipping objects 4 and 5. However, rendering object 2 in less detail could have prevented this.

offers spatial hysteresis. That is, a node will toggle only when the user moves to compensate for the toggle penalty.

Flickering from changes in visibility: Again, one can find certain viewpoints where a slight movement or rotation can cause a substantial change in the number of objects in the view frustum. This affects how the render time is distributed and introduces flickering.

The Adaptive Display algorithm tries to use as much of the render time without exceeding it. This implies that the change in visibility of even a single object can cause flickering [16]. The Adaptive Display algorithm does not offer a solution to minimize this flickering.

Running Time Complexity: The Adaptive Display algorithm must sort each representation of all the visible objects by "value." This takes $\mathcal{O}(N\,log(N))$ time where N is the total number of representations. Subsequently, it takes $\mathcal{O}(N)$ time to traverse the entire sorted list to select representations.

The incremental algorithm improves the running time by using the previous selection as a starting solution. In this case, the algorithm must update the "value" of each representation in $\mathcal{O}(N)$ time. The nearly sorted list, on average, is resorted in $\mathcal{O}(N)$ time or $\mathcal{O}(N log(N))$ in the worst case. The number of replacement iterations are usually a few in number so the total running time is $\mathcal{O}(N)$ on average and $\mathcal{O}(N\,log(N))$ in the worst case.

5.2 Proposed Algorithm

The rendering thread selects a front[1] in the node hierarchy by distributing the render time in proportion to the importance of a node compared to its siblings. The normalized difference between the time slice and estimated render time is used to update a hysteresis counter. Those nodes are selected to be rendered whose cumulative hysteresis counter is non-negative and closest to zero.

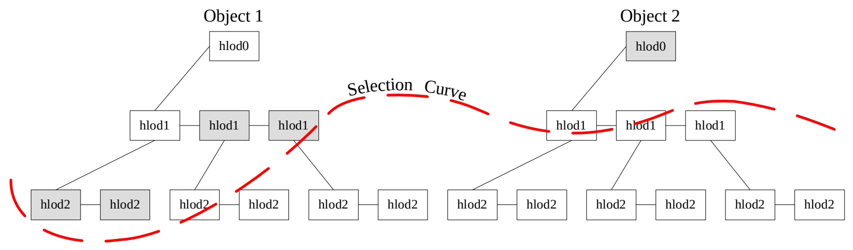


Figure 5.3: Our algorithm selects nodes by tracing a curve across the hierarchy. Nodes selected are shown in grey.

[1]A front is the set of nodes such that all traversals from the root to a leaf node go through exactly one front node.

Note that the truncated hysteresis counter for each node is any arbitrary integer. It is likely to be positive near the root and negative towards the leaves. Perhaps, somewhere between the path from the root to a leaf, one of the node's hysteresis counter will be zero. If not, we interpolate the positive and negative values to find the zero crossing between a parent and its children.

We define the selection curve such that it crosses the entire hierarchy where the truncated hysteresis counter is zero. One such curve is illustrated in Fig. 5.3. Nodes are selected such that the sum of the signed distances between the curve and a set of siblings is non-negative. This is equivalent to summing up the truncated hysteresis counter and selecting the bottom-most set for which the sum is greater than or equal to zero.

Subsequent selection curves are drawn by moving the previous frame's selection curve up and down along each node. The amount of movement depends on how quickly the node changes in importance.

Our algorithm exhibits certain characteristics:

Under-utilization of render time: Although the curve can be drawn arbitrarily, discrete nodes must be chosen for rendering. Also, one node cannot be chosen unless the siblings are selected as well. Lastly, if a node is not loaded in memory, the parent must be selected instead. These three issues lead to some wastage of the render time. An example is shown in Fig. 5.4 (a).

We compensate for this with the feedback loop mentioned in Section 4.2. This adjusts the target render time passed to the root of the HLOD hierarchy if the actual render time is used inefficiently and loaded nodes were rejected.

Over-utilization of render time: The curve can be drawn such that $hlod_0$ nodes would be skipped. Since we do not skip $hlod_0$ nodes, this can exceed the target render time. This is demonstrated in Fig. 5.4 (b). We can also over-utilize the render time if the data sent to the graphics pipeline exceeds the graphics card memory.

The feedback loop discussed in Section 4.2 also decreases the target render time given to $hlod_0$ nodes if the actual render time is much larger than the target.

Flickering from changing benefit: Since we move the selection curve up and down from the previous location, our algorithm is not subject to immediate flickering over slight movements back and forth. Furthermore, the amount of time between each switch can be controlled by the hysteresis parameter, $c_{hystSwitch}$.

Recall that the Adaptive Display Algorithm's hysteresis is spatial in nature and large movements can compensate for the benefit penalty imposed by hysteresis. Our hysteresis implementation is temporal since our hysteresis counters retain memory over time. The amount of memory is determined by $c_{hystLimit}$.

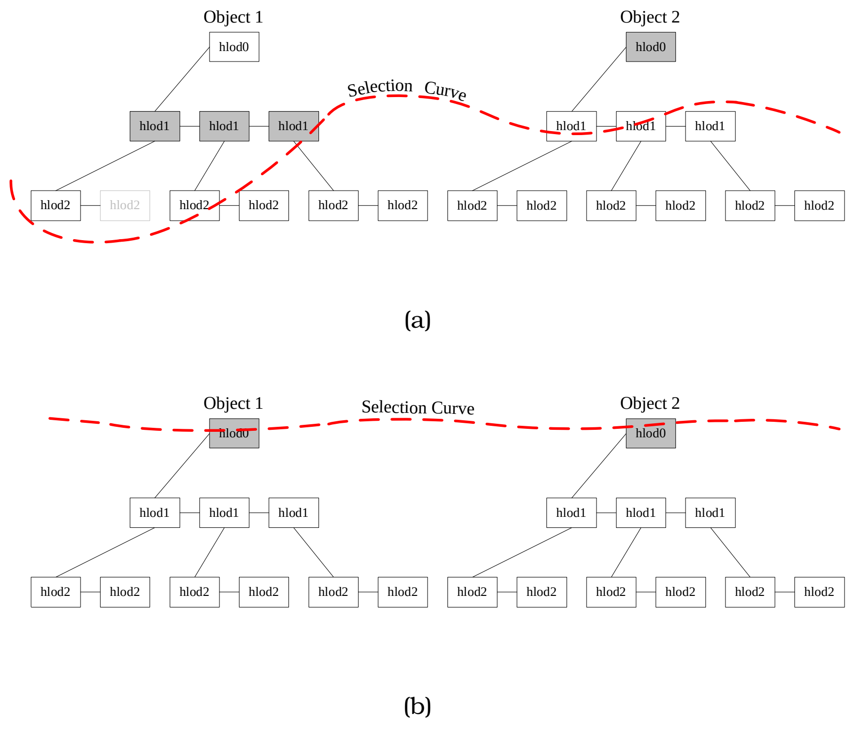


Figure 5.4: (a) Render time is under-utilized because one of objects 1's $hlod_2$ is not available in memory. Also, there is enough time to render object 2's first $hlod_1$, but not enough to render all siblings; (b) When the target render time is reduced, the selection curve moves upward. Consequently, we over-utilize render time by not skipping Object 1 and 2.

Flickering from changes in visibility: Our hysteresis solution limits immediate switching when the number of objects in the view frustum change. The delay in switching depends on the number of objects that are added or removed by culling. The delay may also be influenced by the hysteresis parameters.

Flickering from unavailability of culled nodes: When a node is not visible and not available in memory, one can render the sibling nodes anyway. However, if the user moves in such a manner that the unavailable node becomes visible, the siblings cannot be rendered and the parent must be selected. If the user moves back to the original position, the sibling nodes will be displayed in more detail again. This flickering[2] can be bothersome for the user.

We have implemented an option for the user to require that a switch from parent to children only be made if all children are available in memory. Although higher detailed HLODs take longer to toggle, they switch to less detail only if one of the nodes gets unloaded[3] or if the hysteresis value decreases substantially.

Running Time Complexity: Our rendering selection need not visit each node in the HLOD hierarchy. Note that when the $cumHyst$ counter is below zero for any set of siblings, we select the parent and do not visit any of their children. Recall from

[2]The Adaptive Display algorithm does not have sibling nodes and is not subject to this.
[3]Nodes usually get unloaded only if the priority decreases substantially because the user moved far away.

Fig. 4.2 that $cumHyst$ depends on the time slice and the render time. Let us look at both of these below:

- The time slice is dependent on the relative cost and benefit compared to other siblings and the target render time. However, if one node's benefit and cost changes, other nodes will roughly relinquish some time slices or use up any extra time slices. That is, the selection curve will change but the number of nodes visited will approximately be the same. Therefore, the complexity to render one frame depends on the number of triangles that can consume $c_{targetTime}$.
- The time needed to render all nodes must approximately sum up to the target render time. Again, the constant, $c_{targetTime}$, determines the running time complexity to render a frame.

For any given selection curve, we not only visit the nodes near it but also above it. If we visit v nodes, then we also visit $\lceil \frac{v}{c_{levelFactor}} \rceil$ parents and so on. This defines a geometric series that is in $\mathcal{O}(c_{targetTime})$.

One last observation is that we must visit all the $hlod_0$ nodes and render at least that. The number of $hlod_0$ nodes corresponds to the number of objects in the scene and is independent of the number of nodes in the HLOD hierarchy. Therefore, the total running time to render a frame is $\mathcal{O}(c_{targetTime} + numObjects)$.

5.3 Summary

One major difference between the Adaptive Display algorithm and our algorithm is that we generalized our algorithm to work with HLODs rather than LODs. Secondly, we do not skip any objects. Instead, we remove detail from the high priority nodes to maintain a consistent frame rate.

Also, our implementation of hysteresis does not guarantee an absolute frame rate. Instead, we put more emphasis on decreasing flickering of nodes at the expense of sometimes slightly going over the alloted frame render time. We argue that a slight change in frame render time is less noticeable than drastic flickering of nodes. Furthermore, the trade off between frame rate consistency and flickering can be controlled by the hysteresis parameters.

Chapter 6

Results and Remarks

We have implemented our rendering engine on a Windows architecture using the Visual C++ language. Our tests are run on a Windows XP PC with a 2.0 GHz Intel Pentium IV CPU, 1024 MB of system RAM, and a Nvidia GeForce4 Ti 4600 graphics card with 128 MB of RAM.

Our implementation allows the user to move around in all 6 degrees of freedom without any restrictions. The user has the choice of standing still or navigating at arbitrary speeds[1].

Parameter	**Description**	**Assignment**	**Value**
$c_{desiredTris}$	Desired number of triangles for each HLOD node	Guessed, based on model size	900
$c_{levelFactor}$	Simplification factor between two successive levels of the hierarchy	Guessed, 2 yielded too many HLOD levels	4
$c_{textured}$	Weight of number of textured triangles to estimate render time	Measured render time of textured and untextured triangles	0.127
c_{vis}	Importance of visibility for the priority heuristics	Guessed	1.275
c_{fore}	Importance of foreshortening for the priority heuristics	Guessed	0.450
$c_{targetTime}$	Target render time for each frame	User preference	30 ms
$c_{hystSwitch}$	Hysteresis threshold for switching a node	User preference	1500
$c_{hystLimit}$	Range of values for the hysteresis counter	User preference	4510
c_{memory}	Proportionality constant between cost and memory usage	Measured by monitoring memory usage	935 bytes

Table 6.1: List of parameters used in our implementation.

In the following sections, we discuss some of the tests we ran, the results we acquired

[1]In practice, we are limited by the precision of the floating point operations.

from these tests, and the conclusions we made based on the results. For all the tests, we set our parameters as shown in Table 6.1. The values for these parameters were chosen based on:

- user preference,
- experimental measurements, or
- some guesswork along with trial and error.

6.1 Memory Management Ability

We would like to demonstrate that our memory management algorithm, described in Section 4.3, prevents memory usage from exceeding a specified threshold. To do so, we conduct a test where we run our rendering engine under standard conditions and monitor the memory utilization five times every second.

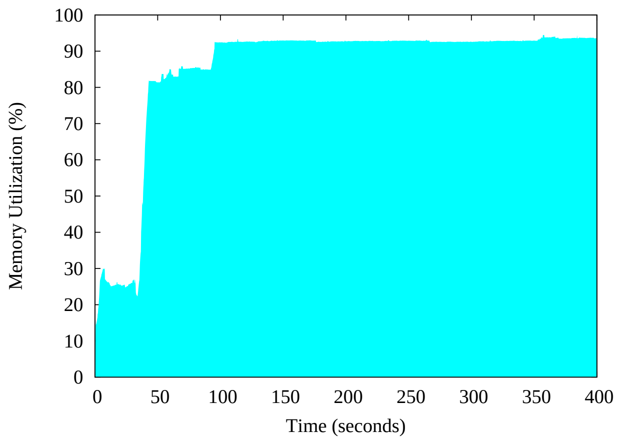


Figure 6.1: This graph shows the memory utilization for the first 400 seconds.

Observe in Fig. 6.1 that the rendering engine consumes 16 percent of memory immediately after launch. After that, the loading thread asynchronously loads and unloads nodes as dictated by the priority queue. Peak memory utilization of about 93 percent is reached at the 95th second. Memory usage, for the remainder of the experiment, remains relatively flat.

6.2 Detail Management Effectiveness

To show that our rendering engine adapts to varying rendering loads by varying the amount of detail in the scene, we did two tests. In both tests, we examine each rendered node's position in the hierarchy to determine the amount of detail that was rendered.

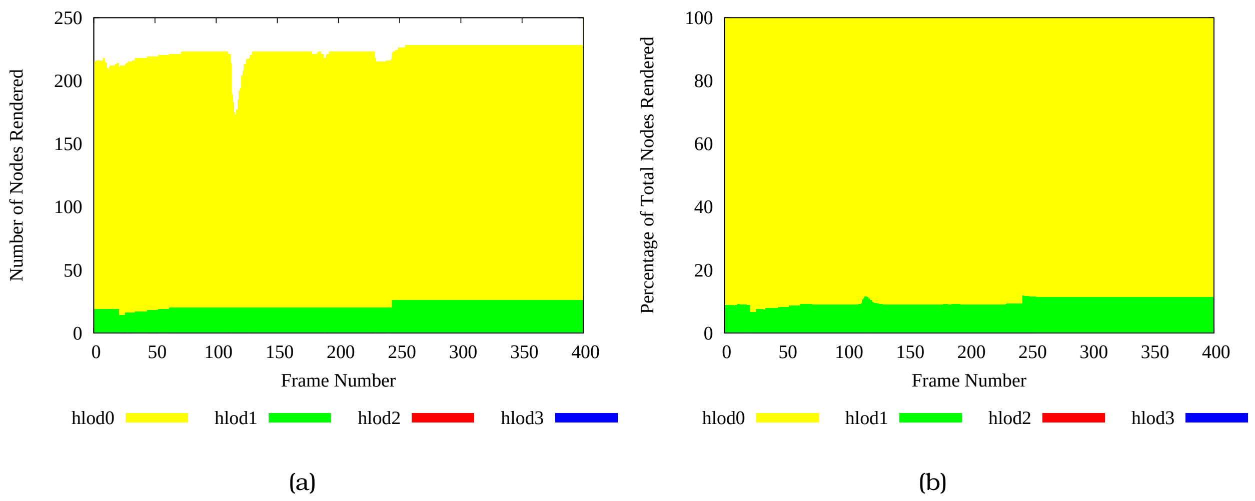


Figure 6.2: (a) Fly-through mode renders only $hlod_0$ and $hlod_1$ nodes; (b) Least detailed $hlod_0$ nodes constitute 80 to 90 percent of total nodes rendered

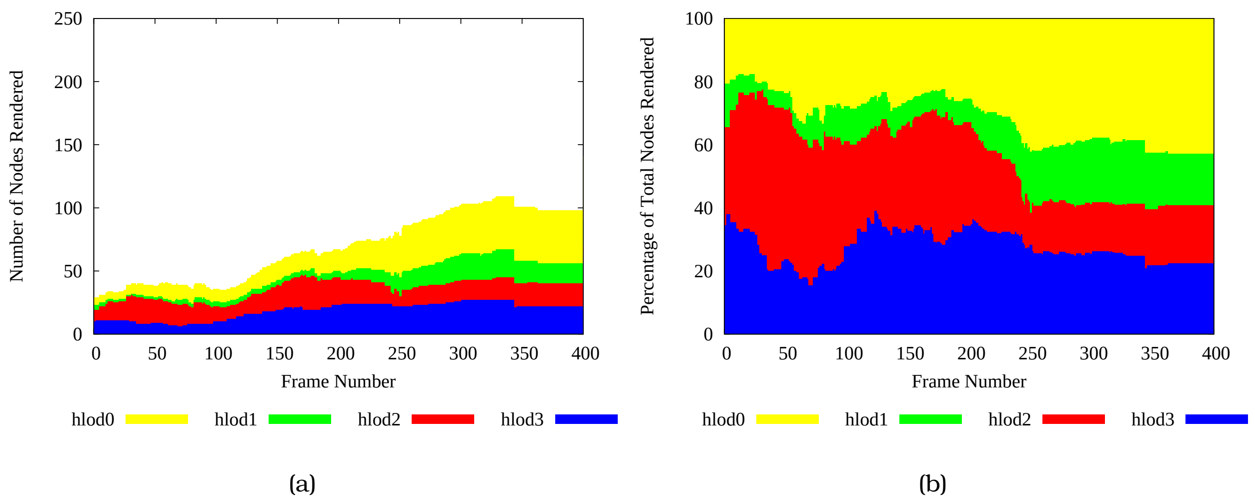


Figure 6.3: (a) Walk-through at ground level renders fewer total HLODs; (b) A larger percentage of more detailed HLODs are selected to be rendered

In the first test, we render the scene in a fly-through mode such that the entire model is visible. Figs. 6.2 (a) and (b) show that very few highly detailed HLODs were selected to be rendered. That is, only a small percentage of nodes are shown in high detail.

The second test was done at ground level where many nodes could be culled away. Figs. 6.3 (a) and (b) show that many highly detailed HLODs were rendered. Also, $hlod_0$ nodes were only 20 to 45 percent of total nodes that were rendered.

This confirms that our rendering engine effectively manages detail based on the complexity of the scene that is to be rendered.

6.3 Accuracy of Estimated Render Time

Our rendering algorithm maintains a consistent frame rate with the use of time slices. These time slices are compared with estimated render times. Consequently, our ability to estimate the render time is crucial to making any frame rate guarantees.

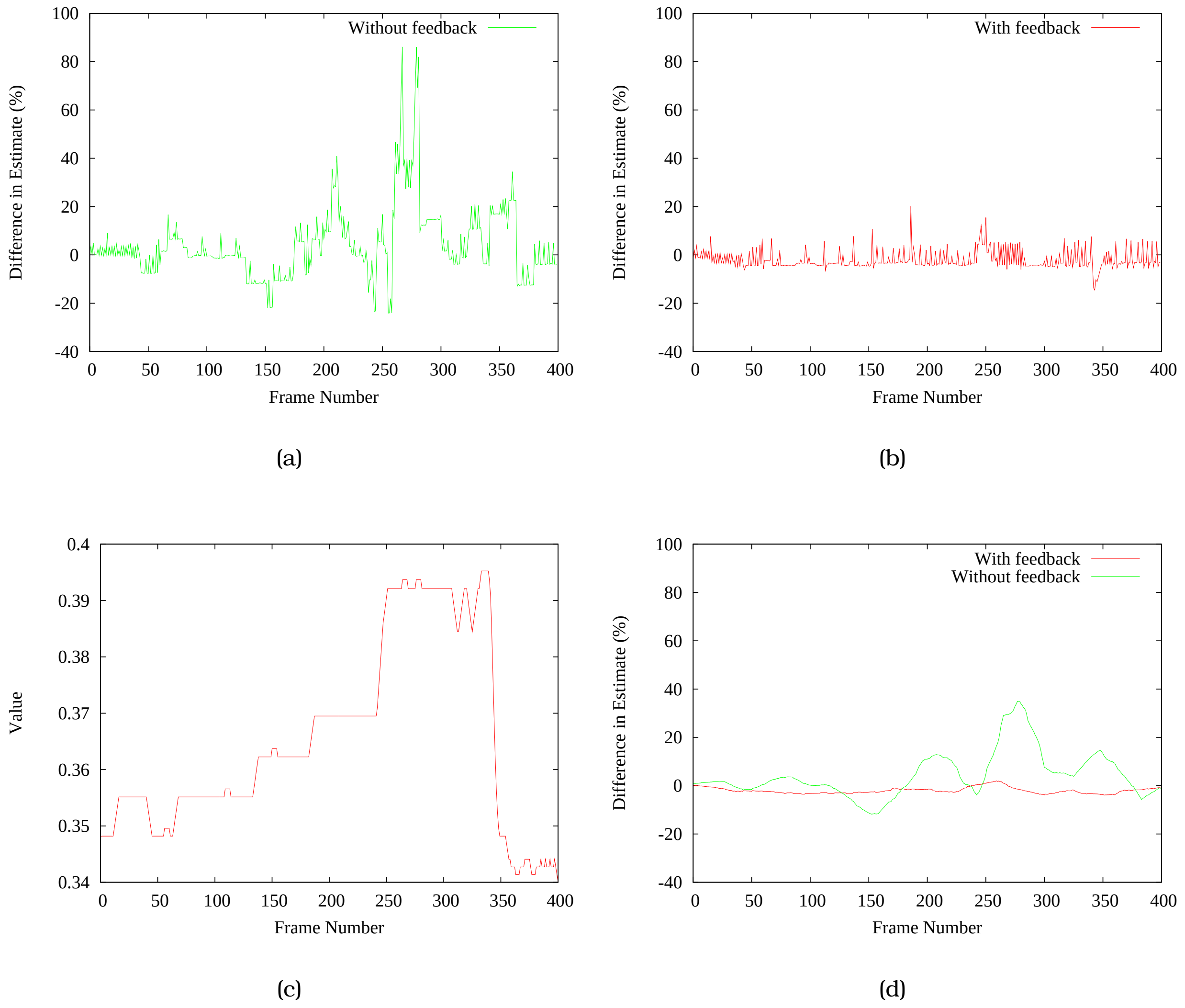


Figure 6.4: (a) Percentage difference between the estimated render time and the actual render time without a feedback loop; (b) Percentage difference where the estimated render time is updated with a feedback loop; (c) The feedback loop variable, v_{time}, is shown; (d) Moving average of 35 frames contrasts the difference with and without the feedback loop.

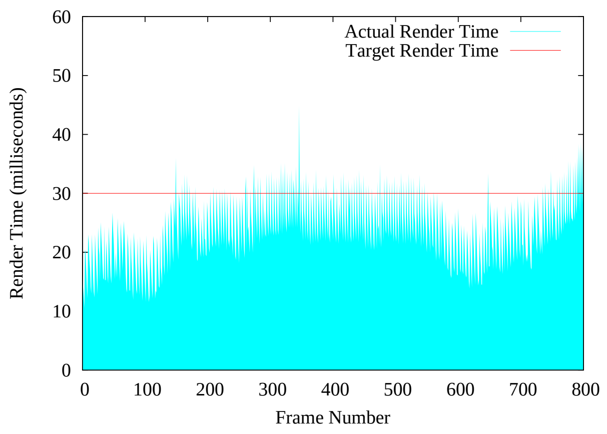
We conduct a walk through and measure the actual time to render each frame. We also estimate the render time of each frame with and without a feedback loop. Fig. 6.4 (a) shows the difference in percentage between the render time estimated directly from the total cost and the actual render time. The average difference is 4.39% and the standard deviation is 15.19%.

When we incorporate a feedback loop into the estimated render time, the average difference improves to -2.01%[2]. The standard deviation shows the most notable improvement by dropping to only 3.89%. In Fig. 6.4 (b), we see the percentage difference when the estimated render time is augmented with a feedback loop.

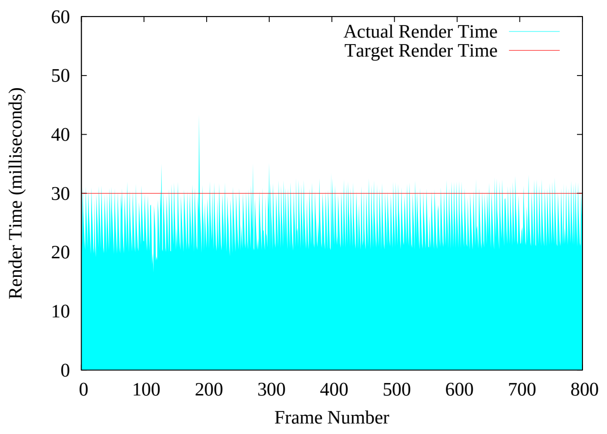
Since the render times are subject to some noise, we compare a 35-frame moving average of the percentage difference for both types of estimates in Fig. 6.4 (d). Note that the feedback loop ensures a more accurate estimate of render time and justifies its need.

6.4 Frame Rate Consistency

We measure the actual render time for the two tests that we conducted in Section 6.2. These are shown in Figs. 6.5 (a) and (b) for walk-through and fly-through respectively.



(a)



(b)

Figure 6.5: Actual render time compared to target render time of 30 milliseconds for (a) walk-through and (b) fly-through.

As can be seen from these figures, the walk-through mode culls away a lot of the $hlod_0$ nodes and, thus, the time utilization depends on the loading of more detailed HLODs. The fly-through render times are much closer to the target render time since a large

[2] A negative percentage difference implies a conservative time estimate.

majority of nodes that are selected to be rendered are $hlod_0$ nodes that are resident in memory.

Note that occasionally, the actual render time slightly exceeds the target render time. These are largely due to the noise in render times mentioned in the above Section. Some occasions may also be attributed to the hysteresis implementation allowing more time to be consumed to render a frame. On the whole, our algorithm utilizes the alloted render time well and prevents the actual render time from exceeding the target render time by excessive amounts.

6.5 Flicker Prevention

In order to measure the flickering, we count each time a nodes gets upgraded to its children[3] between successive frames. Similarly, we tally each downgrade from sibling nodes to a parent node. We do not count nodes that are culled away between frames.

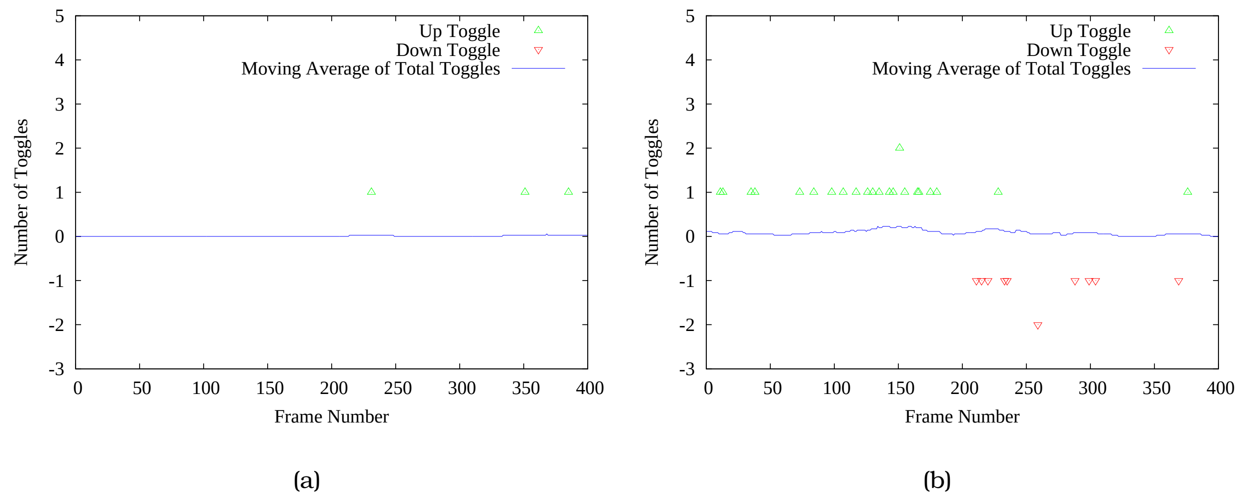


Figure 6.6: Flickering measured during (a) walk-through, and (b) drive-through. The moving average is taken over 35 frames.

First, we performed a walk-through test. We show 400 frames from this walk-through in Fig. 6.6 (a)[4]. We performed 3 toggles in total. This averages to 0.008 toggles per frame or 1 toggle for every 133 frames.

The second test was conducted at approximately 10 times the velocity of the walk-through. The toggles and the moving average are shown in Fig. 6.6 (b). Note that in

[3]This is counted as one node toggle, regardless of the number of children.
[4]A down toggle is shown in the negative axis for clarity only.

drive-through mode, the toggles are more frequent since the user is moving around the scene more rapidly. Consequently, we see one toggle every 11 frames on average.

Overall, the two tests show that we do not get successive up and down toggles associated with oscillations from the feedback loops. Therefore, the overall flicker is minimized and user experience with our rendering engine is positive.

6.6 Future Work

Our cost heuristics do not take into account the texture size because we have established that increasing the amount of texture does not increase the render time. The exception to this occurs when the texture size exceeds the texture memory on the graphics card. In this case, the render times change substantially. Although our feedback loop reduces allocated render time to compensate for this situation, it would be interesting to directly address this.

Another improvement would to be to accumulate unused time slices and give it to other nodes that could make use of it. The challenge is to do this without introducing flickering. Our current approach indirectly solves this with the feedback loop for the allocated render time.

The cost heuristics can also be improved to better model fill rate and other intrinsic properties of a graphics card. Lastly, the foreshortening in the benefit heuristics could be improved by clustering similarly oriented triangles together in one node.

6.7 Conclusion

We have presented an algorithm that uses hierarchical levels of detail to efficiently render large, detailed models for walk-through and fly-through modes of interaction. Our implementation runs on a common PC with a moderate graphics card and system memory.

Our rendering engine limits memory usage, maintains a specified frame rate by managing detail, and incorporates hysteresis into a simple unified approach. Furthermore, our pre-fetching scheme does not skip objects that are visible or delay rendering to load objects.

Lastly, our implementation scales well with increasing data size and degenerates gracefully to the case where it does not render any of the more detailed HLODs.

Acknowledgment

We would like to thank John Flynn, who developed an earlier version of our rendering engine. Chris Frueh provided some code to generate HLODs. Lu Yi also contributed to the implementation. We are grateful to Avideh Zakhor and Carlo Séquin for their valuable insights. This research was funded by Army Research Office MURI contract #DAAD19-00-1-0352.

Bibliography

[1] U. Assarsson and T. Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools: JGT*, 5(1):9–22, 2000.

[2] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, C. Rebollo, and M. Fernandez. Multiresolution triangle strips. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing (VIIP 2001)*, pages 182–187, 2001.

[3] B. Chen and M. X. Nguyen. Pop: a hybrid point and polygon rendering system for large data. In *Proceedings of the conference on Visualization '01*, pages 45–52. IEEE Computer Society, 2001.

[4] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[5] J. D. Cohen, D. G. Aliaga, and W. Zhang. Hybrid simplification: combining multi-resolution polygon and point rendering. In *Proceedings of the conference on Visualization '01*, pages 37–44. IEEE Computer Society, 2001.

[6] D. Cohen-Or, Y. Chrysanthou, and C. T. Silva. A survey of visibility for walkthrough applications. *Proc. of EUROGRAPHICS course notes*, 2000.

[7] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. In *SIGGRAPH 2003, Computer Graphics Proceedings*, pages 657–662. ACM Press / ACM SIGGRAPH, 2003.

[8] X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, 1999.

[9] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[10] C. Erikson, D. Manocha, and W. V. Baxter, III. HLODs for faster display of large static and dynamic environments. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 111–120. ACM Press, 2001.

[11] F. Evans, S. S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In R. Yagel and G. M. Nielson, editors, *IEEE Visualization '96*, pages 319–326, 1996.

[12] C. Frueh and A. Zakhor. 3D model generation for cities using aerial photographs and ground level laser scans. In *IEEE Computer Vision and Pattern Recognition Proceedings*, volume 2.2, pages II – 31–38, 2001.

[13] C. Frueh and A. Zakhor. Fast 3D model generation in urban environments. In *International Conference on Multisensor Fusion and Integration for Intelligent Systems*, volume 2.2, pages 165–170. The University of California at Berkeley, 2001.

[14] C. Frueh and A. Zakhor. Data processing algorithms for generating textured 3D building façade meshes. In *3D Data Processing, Visualization and Transmission*, pages 834–847, 2002.

[15] C. Frueh and A. Zakhor. Constructing 3D city models by merging ground-based and airborne views. *IEEE Computer Graphics and Applications*, 23(6):52–61, 2003.

[16] T. A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, The University of California at Berkeley, 1993.

[17] T. A. Funkhouser. Database management for interactive display of large architectural models. In W. A. Davis and R. Bartels, editors, *Graphics Interface '96*, pages 1–8. Canadian Human-Computer Communications Society, 1996.

[18] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics*, 27(Annual Conference Series):247–254, 1993.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[20] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[21] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108. Microsoft Research, ACM Press, 1996.

[22] S. F. J. Foley, A. van Dam and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 1993.

[23] W. Jepson, R. Liggett, and S. Friedman. An environment for real-time urban simulation. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 165–166. ACM Press, 1995.

[24] M. Levoy and T. Whitted. The use of points as a display primitive. Technical report, Computer Science Department, University of North Carolina at Chapel Hill, January 1985. TR 85-022.

[25] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH'96*, pages 109–118, 1996.

[26] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics*, pages 95–102, 211, 1995.

[27] R. Pajarola, M. Antonijuan, and R. Lario. Quadtin: Quadtree based triangulated irregular networks. In *Proceedings of IEEE Visualization*, pages 395–, 2002.

[28] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM Press, 2001.

[29] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[30] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture mapping progressive meshes. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 409–416. ACM Press / ACM SIGGRAPH, 2001.

[31] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Computer Graphics*, 30(Annual Conference Series):75–82, 1996.

[32] M. Shafae and R. Pajarola. DStrips: Dynamic triangle strips for real-time mesh simplification and rendering. In *Proceedings Pacific Graphics 2003*, pages 271–280. IEEE, Computer Society Press, 2003.

[33] R. Toledo, M. Gattass, and L. Velho. Qlod: A data structure for interative terrain visualization. Technical report, VISGRAF Laboratory, 2001. TR-01-13.

[34] G. Varadhan and D. Manocha. Out-of-core rendering of massive geometric datasets. In *Proceedings of the conference on Visualization*, pages 69–76. IEEE Computer Society, 2002.

[35] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 361–370. ACM Press / ACM SIGGRAPH, 2001.