Efficient Interactive Rendering of Detailed Models with Hierarchical Levels of Detail

Ali Lakhia
The University of California at Berkeley
Email: `lakhia@eecs.berkeley.edu`

***Abstract*— Recent acquisition systems, such as the one developed at the University of California at Berkeley, are capable of collecting large, detailed, highly textured models that standard levels of detail (LOD) rendering techniques [15] 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 [15] to Hierarchical Levels of Detail (HLODs) [8]. 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 the amount of detail that 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 114 city blocks comprised of 7 million triangles and 720 million color pixels. Our engine maintains a constant frame rate and limits excessive flickering simultaneously.

I. Introduction

Recently, an acquisition system has been developed at the Video and Image Processing Lab at Berkeley which is capable of rapidly acquiring large, detailed, 3D textured models of urban environments from the ground level by using two 2D laser scanners and digital cameras [10], [11]. Far-range Digital Surface Map (DSM) data and aerial imagery is then registered with respect to the ground-based model and merged to create a single model [12].

The final city model has over 114 complete blocks of building facades, where none of the buildings share the geometry or the texture with any other building. The total number of triangles are about 7 million. The texture consists of over 720,000,000 pixels or about 2160 MB of uncompressed data. An overview of the entire model can be seen in Fig. 2 (a). A closeup in Fig. 2 (b) shows the details in the model.

A. Goals

The primary goal of this work is to address the rendering of such large models on a common personal computer with a standard graphics card and average amount of system memory.

The rendering system should scale with the size of the model without discarding any of the collected data. We argue that small changes in geometry and texture are perceptible if seen from a certain view-point and at a certain distance. That is, rather than throwing away data, we should manage the details in the model to solve interactive rendering issues.

Lastly, the rendering system should maintain a specified frame rate while maintaining some coherency between frames and minimizing toggles between discrete levels of detail.

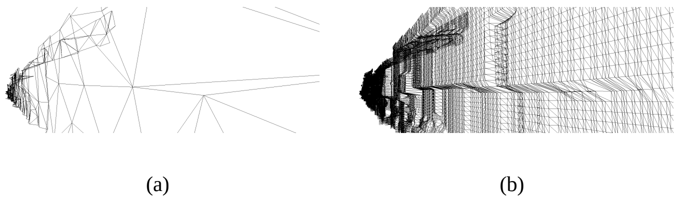


Fig. 1. (a) LOD of large object is too coarse near the camera; (b) Using a higher detailed representation wastes detail that is far away from the camera.

B. Organization

The paper is organized as follows. Section II reviews previous work in interactive visualization of large scenes. We also explain why these approaches are not suitable to meet our goals. In Section III, we elaborate on our approach. We present overall results in Section IV.

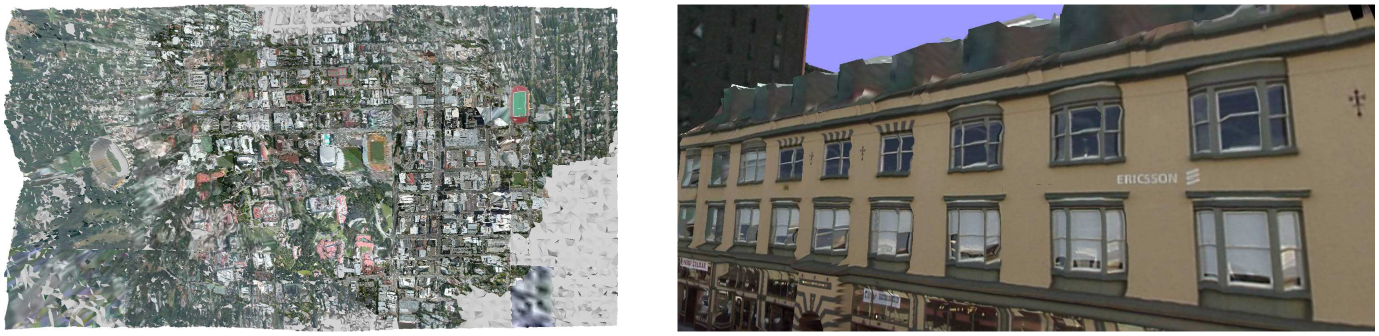
II. Previous Work

A. Representation of Data

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 [3]. He used the appropriate representations to improve interactivity.

Rendering discrete LODs of large objects, however, are less optimal. For example, consider a slanted view of a building facade in Fig. 1. 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



(a) (b)

Fig. 2. (a) The entire city model, as seen through our rendering system, has about 114 blocks. The grey regions are triangles that do not have corresponding texture; (b) Closeup of one of the blocks in the city model.

LODs at the coarsest levels. The use of a hierarchy of LODs, or HLODs, was proposed to overcome suboptimal use of LODs [8].

Progressive meshes are a flexible representation of polygon geometry that can be adaptively tailored to produce different LODs [17], [19], [25]. These LODs may also be made view-dependent such that more detail is presented where it is most observable by the user [18]. However, progressive meshes are not able to fully utilize the graphics hardware accelerations since the geometry changes frequently.

Geometry that is stored as triangle strips provides a compact representation that saves memory, reduces bandwidth, and takes less time to render [9]. This technique was improved to generate triangle strips for LOD meshes [1] and in real-time [27]. Nonetheless, such techniques do not make render time output-dependent and rendering of large models require some management to ensure interactiveness.

Point-based rendering is yet another technique that scales well with complexity [20], [24]. Hybrid approaches have been developed that use both polygons and points as rendering primitives [2], [4] to efficiently use large flat surfaces and creases. However, these algorithms exploit minimal graphics hardware acceleration.

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 [5]. The trade-off with this approach is that hierarchical culling cannot be performed since parent-child relationship is lost in a sequential list.

Height fields are often used for terrain visualization [7], [21], [23], [28]. 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 [22] and their idea was refined by others [6], [26]. 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.

B. Data Management

Memory usage must be managed to prevent swapping of data between memory and the hard disk. Funkhouser prevents swapping by asynchronously prefetching data as needed [14]. However, the prefetching algorithm cannot guarantee availability of objects in memory and objects may 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 [29]. 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.

The details of the model must also be managed to limit the load of the graphics pipeline and to ensure interactivity. Funkhouser and Séquin use a heuristic to determine the ratio of cost and benefit of each object at each of its LODs [15]. They equate the graphics pipeline load management problem to the multiple choice knapsack problem, and offer an approximation to the optimal solution [15]. However, they employ a simple 2-level LOD hierarchy that is inefficient for large, detailed objects.

Maciel and Shirley use a hierarchy of LODs and imposters. They traverse the hierarchy in a bottom-up fashion [22], and thus, the complexity of their algorithm is $\mathcal{O}(N)$ with respect to the total number of nodes. This limits scalability. Lastly, their hysteresis implementation worsens their frame rate dramatically.

Erikson et al. traverse the hierarchy top-down and use a screen-space error metric to choose which HLODs are refined [8]. However, the refinement process does not directly consider the cost of each refinement and can result in significantly non-optimal use of render times[1]. The polygon budget is simply based on previous frame render times, which can lead to frequent switching of HLODs between successive frames.

C. Other Approaches

Wand et al. suggest a novel rendering algorithm that is output sensitive [30]. 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.

III. Proposed Approach

Our experimental tests show that texture size is largely non-linear in relationship with the time taken to render an object. Therefore, our approach is to manage texture indirectly by efficiently managing the geometry that is rendered in each frame.

We choose to use HLODs to represent our model because, as mentioned previously, they are more efficient than discrete LODs. Also, discrete 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.

Our approach is to generate an HLOD hierarchy from the model, followed by pre-processing to reduce computations during rendering. As shown in Fig. 3, our rendering engine consists of a rendering and a loading thread. The rendering thread traverses the HLOD hierarchy to render each frame by selecting a front that is then sent to the graphics pipeline. On traversal, each node's priority is calculated and maintained in a priority queue. The loading thread queries the priority queue and asynchronously pre-fetches those nodes with the highest priority and unloads the nodes deemed least important.

These are discussed in more detail in the sections below.

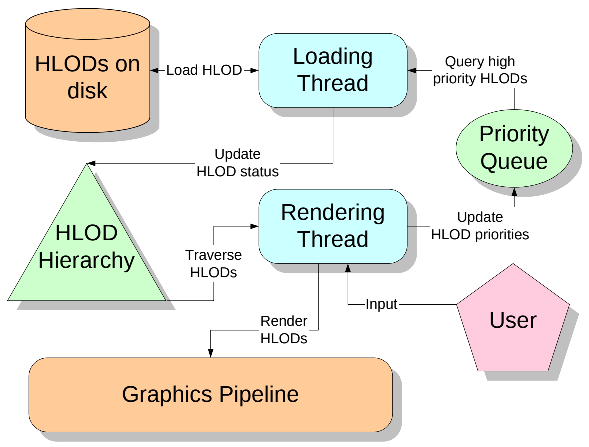


Fig. 3. Overview of the rendering engine architecture.

[1]Consider an example where 4 candidates for refinement are available and one of them has a slightly higher screen-space error. If this refined candidate consumes the entire triangle budget while using the same number of triangles as the other 3 replacements combined, then we get a sub-optimal solution.

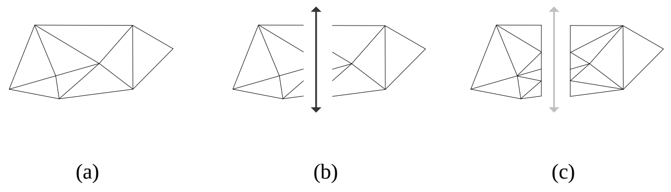


Fig. 4. (a) A set of triangles before split; (b) cutting plane is used to create two sets of triangles that are shown separated for clarity. New vertices are created on the plane; (c) The split triangles are re-triangulated.

A. Model Preparation

The city model is is split up into blocks that typically corresponds to one side of a city block. For each block, b, we generate the least detailed HLOD, $hlod_0$, by simplifying it using Qslim Simplication Software [16] so that the block has $c_{desiredTri}$ triangles. Therefore, the simplification factor is:

$$simplification_{hlod_0} = \frac{numTri_b}{c_{desiredTri}}$$

Here, $numTri_b$ 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.

Next, we define $c_{levelFactor}$ as the ratio of the number of triangles and size of texture between two successive levels of the hierarchy. 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 its triangles into two sets, 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. 4, 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.

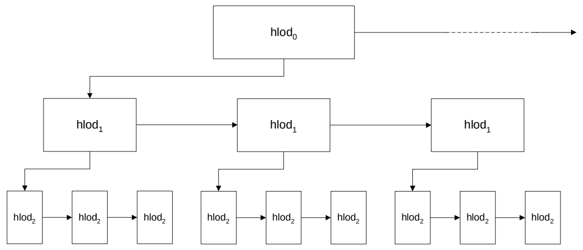


Fig. 5. HLODs for a single block. The node $hlod_0$ is most coarse whereas $hlod_2$ is the most detailed. Note that $hlod_0$'s siblings are other blocks.

We repeat this procedure for all the blocks in our model. Our final data has a total of 3028 nodes from 207 blocks. The maximum depth of the hierarchy is 5. One such block is shown in Fig. 5 where $c_{levelFactor}$ is shown to be 3.

On average we expect each node to have $c_{desiredTri}$ triangles. However, the actual number of triangles vary considerably due to the varying density of samples, and varying effect of simplification along different subsections of the block.

B. Other Pre-processing

We pre-compute and save each node's 1) bounding box for culling purposes, 2) cost that measures the estimated render time, 3) static benefit that is adjusted during run-time, and 4) average normal to calculate foreshortening.

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:

$$\begin{aligned} cost_n \; = \; & numTexTri_n \times c_{tex} + \\ & numUnTexTri_n \times (1 - c_{tex}) \end{aligned} \tag{1}$$

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

We compute a static approximation of the benefit, which is then dynamically adjusted during the rendering phase, as the geometric mean of the accuracy of its representation and its total surface area. Specifically, given that $tri(n)$ is the set of triangles in node n and $area_t$ is the area of triangle, t, we have:

$$staticBenefit_n \; = \; \sqrt{numTri_n \times \sum_{t \in tri(n)} area_t} \tag{2}$$

Since most of our nodes are facades of buildings, the orientation of most of the triangles is fairly uniform. Therefore, we compute the normal of a node by taking the average normal of each triangle, $\overrightarrow{normal_t}$, weighted by its area and then normalized:

$$\overrightarrow{avgNormal_n} \; = \; \frac{\sum_{t \in tri(n)} (\overrightarrow{normal_t} \times area_t)}{\| \sum_{t \in tri(n)} (\overrightarrow{normal_t} \times area_t) \|} \tag{3}$$

C. Rendering Thread

The rendering thread recursively traverses nodes in the hierarchy in a top-down, breadth-first manner to render each frame. 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].

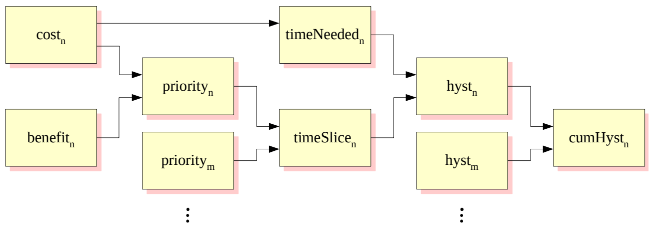


Fig. 6. Dependency graph of front selection in the rendering thread

Node selection, as illustrated in Fig. 6, 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.

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.

Priority Heuristics: Upon visiting each node, n, the rendering thread calculates its priority based on a benefit and cost ratio so that a higher priority indicates higher importance:

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

Recall that $staticBenefit_n$ is calculated offline for each node in (2). 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 and foreshortening:

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

The distance is measured from n to the camera and is adjusted by adding a portion of 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:

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

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

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

Lastly, we adjust the static benefit for foreshortening by multiplying with the dot product of the normal of the image plane, $\overrightarrow{view}$, and the average normal of n, $\overrightarrow{avgNormal_n}$ from (3). This is weighted by a constant, c_{fore}:

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

The cost of n, $cost_n$, is also calculated offline as shown in (1).

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:

$$timeSlice_n = shareTime_n \times \frac{priority_n}{\sum_{i \in sib(n)} priority_i} \tag{4}$$

Here, $sib(n)$ is defined to be the set of all visible nodes that have the same parent as n. 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 (4). Thus, $shareTime_n$ is defined as:

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

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

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

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. 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:

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

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 of Funkhouser and Séquin incorporates a hysteresis component as part of the benefit heuristics [15]. However, this approach can still cause switching of LODs as objects become visible or invisible [13]. 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 [22]. 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.

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:

$$hyst_n \leftarrow hyst_n + \frac{timeSlice_n - timeNeeded_n}{timeNeeded_n}$$

We add a constraint to the above and restrict $hyst_n$ to lie in the range of $-c_{hystLimit}$ and $c_{hystLimit}$. 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, $hyst_n$ remains near a 0 value. Otherwise, it tends to 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:

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

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 it reaches the last sibling. If $cumHyst_n$ is less than zero, we render the parent of the siblings. Otherwise, we recursively visit the children of all the sibling nodes, again, in a breadth-first order.

This describes the implementation of our rendering thread. Unlike Maciel and Shirley, we do not use the hysteresis counter to override switching of nodes [22]. 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.

D. Loading Thread

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:

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

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[3] and $maxMemory$ is based on the system resources.

IV. Results

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.

A. Detail Management Effectiveness

We show that our rendering engine adapts to varying rendering loads by varying the amount of detail in the scene. We conduct a walk-through at ground level, with many nodes culled away, that gradually becomes a fly-through such that the entire model is visible.

Fig. 7 shows the number of each type of HLOD that is rendered over time. For the path described above, initially many highly detailed HLODs are rendered. However, as more objects become visible, fewer highly detailed HLODs are selected to be rendered. That is, as we add more data to be rendered, our engine decreases detail, and degenerates gracefully to the case where no detail is rendered.

B. Flicker Prevention

In order to measure the flickering, we count each time a node gets upgraded to its children between successive frames. Similarly, we tally each downgrade from sibling nodes to a parent node.

[3]We cannot precisely calculate storage requirements for display lists.

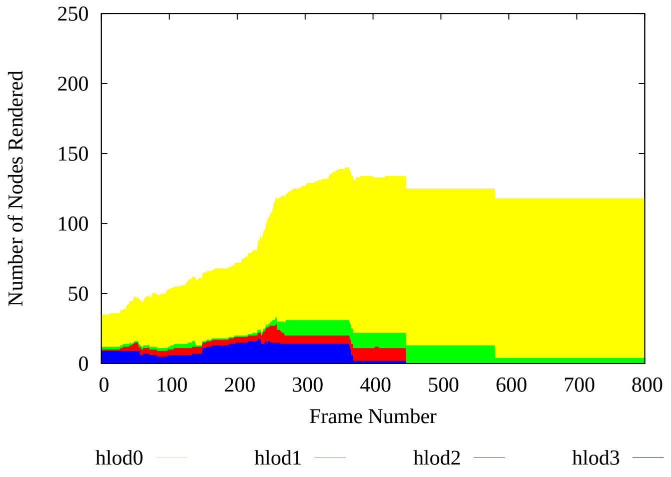


Fig. 7. Walk-through at ground level renders fewer total HLODs thus more detailed HLODs are selected. During fly-through, we render only $hlod_0$ and $hlod_1$ nodes.

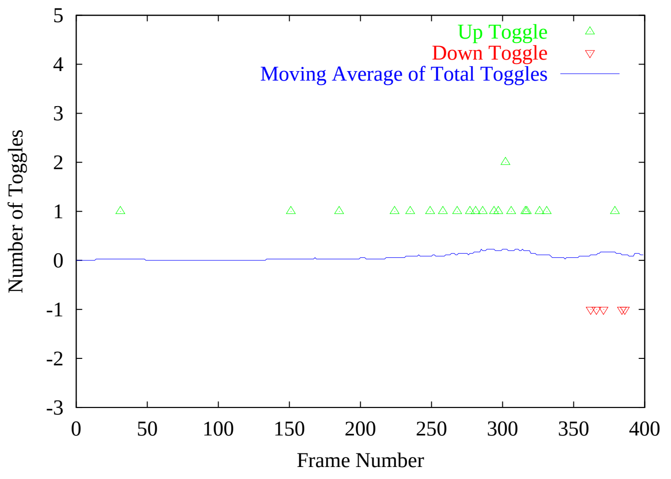


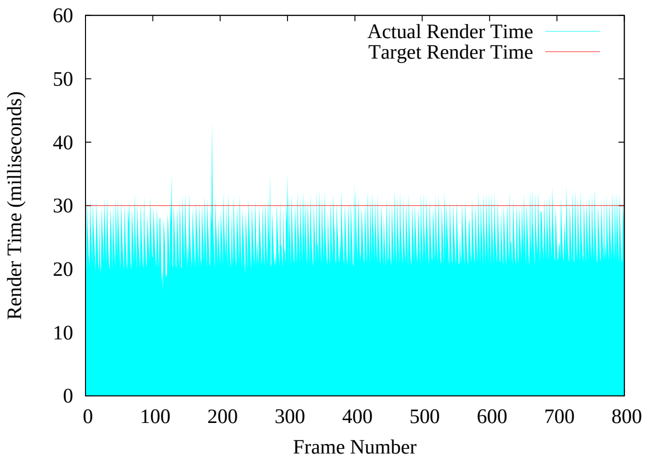
Fig. 8. Flickering measured during walk-through where the moving average is taken over 35 frames. A down toggle is shown in the negative axis for clarity.

We show the upgrade and downgrade toggles for 200 frames corresponding to a walk-through, followed by 200 frames in drive-through mode in Fig. 8. Note that in drive-through mode, the toggles are more frequent since the user is moving around the scene about 10 times faster. Overall, the result shows that we do not get successive up and down toggles associated with oscillations from the feedback loop.

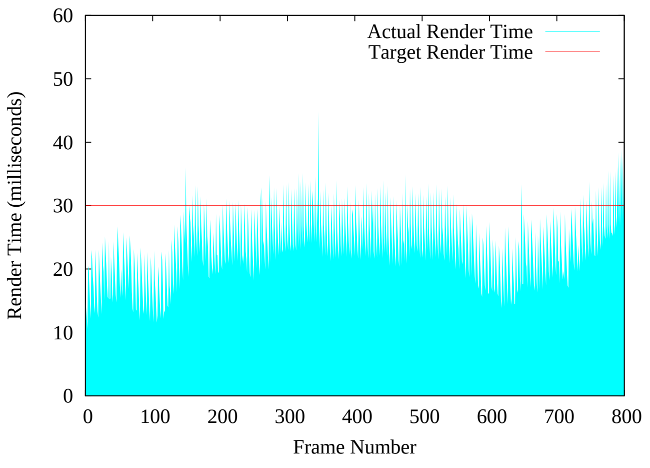
C. Frame Rate Consistency

We measure the actual render time and compare it to the the target render time during a fly-through and a walk-through. These results are shown in Figs. 9 (a) and (b) respectively.

As can be seen from these figures, the fly-through render times are much closer to the target render time since a large majority of nodes that are selected to be rendered are $hlod_0$ nodes that are resident in memory. The walk-through mode culls away several $hlod_0$ nodes and, thus, the time utilization



(a)



(b)

Fig. 9. Actual render time compared to target render time of 30 milliseconds for (a) fly-through and (b) walk-through.

depends on the loading of more detailed HLODs.

On average, our algorithm well-utilizes the alloted render time, and prevents the actual render time from exceeding the target render time.

V. 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.

VI. Future Improvements

Our cost heuristics currently do not account for the texture size because increasing the texture does not generally 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 compensates for this situation, a future improvement would be to directly address this. Lastly, the foreshortening in the benefit heuristics could be improved by clustering similarly oriented triangles together in one node.

Acknowledgment

We would like to thank John Flynn, Chris Frueh and Lu Yi for their contributions to the implementation. We are grateful to Avideh Zakhor and Carlo Séquin for his valuable insights. This research was funded by Army Research Office MURI contract #DAAD19-00-1-0352.

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] H. Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[19] H. Hoppe. Efficient implementation of progressive meshes. *Computers and Graphics*, 22(1):27–36, 1998.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.