Single-Seek Data Layout for Walkthrough Applications

Shan Jiang, Behzad Sajadi, M. Gopi
University of California, Irvine Irvine, USA
Web page: graphics.ics.uci.edu

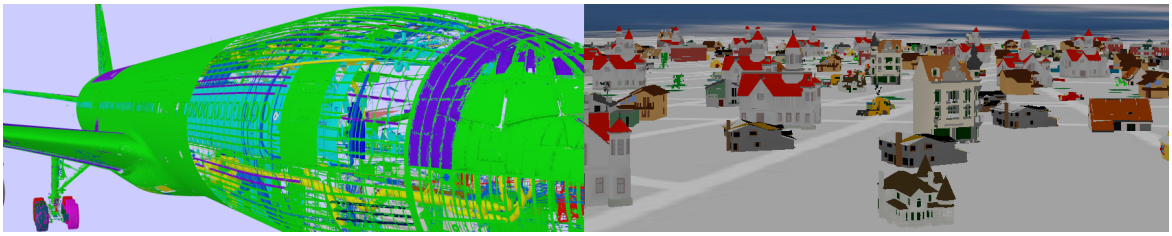

Fig. 1. Images rendered in real-time. Left: a Boeing 777 airplane model with over 350 million triangles, which occupies $20GB$ of secondary storage space; Right: a city model with 110 million triangles occupying $6GB$ of secondary storage space.

***Abstract*—With increasing speed of graphics rendering, the bottleneck in walkthrough applications has shifted to data transfer from secondary storage device to main memory. While techniques designed to reduce the data transfer volume and amortize the transfer cost are well-studied, the disk seek time, which is one of the most important components of the total rendering cost is not reduced explicitly.**

In this work, we propose an orthogonal approach to address the disk seek time bottleneck, namely single-seek data layouts. This is a solution in one end of the spectrum of solutions that guarantee an upper bound on the number of disk seeks. Using this approach, we can reduce the number of disk seeks required to load the data for any viewpoint in the scene to no more than one. We achieve this single seek layout using data redundancy. We provide a theoretical proof on the upper-bound of this redundancy factor, and analyze its trade-off with the rendering performance through an implementation that uses this data layout for walkthrough applications of datasets with hundreds of millions of triangles.

***Keywords*-Out-Of-Core algorithms; Data Layouts;**

I. Introduction

With increasing data sizes and need for more rendering effects in large datasets, real time rendering is still challenging. Interactive rendering of large datasets involves optimizing different stages of data-handling during its journey from the secondary storage device to pixels on the display. These include *disk seek time, data transfer time, online data processing time*, and finally *rendering time*. Together they constitute the processing time for each rendered frame.

In the context of ever increasing capabilities and speed of CPUs and GPUs, both the quality and the speed of the online data processing and rendering have been improved significantly. The fundamental bottleneck in real-time rendering is in transferring the required large amount of data from the secondary storage to the main memory. In order to minimize the effects of this bottleneck, usually the data brought once to the main memory is reused to render as many consecutive frames as possible. Such reusability requires collective use of various techniques including effective cache-management and cache-coherent data layouts on the secondary storage. These layouts reduce the total cost of data transfer time. Furthermore, many predictive data prefetching methods are used to reduce the perceived latency in the data transfer. These techniques can amortize seek and data transfer times over a few consecutive frames at the cost of loading possibly unnecessary data. Finally, cache oblivious layouts try to reduce the number of disk seeks that are needed to fetch the required data to some extent. However, even for a well-arranged cache oblivious layout, there is no guarantee that for each viewpoint the required data blocks will be coherent on the hard drive [24].

Seek time reduction is important when a consistent frame rate is required *even in the worst case*, since each disk seek takes 10 ms in average for commodity hard drives, and can even be slower in other slow secondary storage devices. Assuming the goal is to render 30 frames per second, in the worst case, there would be frames in which the seek time takes nearly one-third of the time-budget for rendering those frames. Since the layouts are optimized for efficient cache utilization but not to reduce the seek time, even the best disk layouts can cause disturbing delays between two consecutive frames in the worst case when frame-to-frame coherence based prediction is not effective. Furthermore, Sajadi et al. [24] showed that any data layout of a geometric dataset that provides a linear ordering of the data over the disk results in frequent disk seeks and some of these seeks will be inevitably long and therefore cannot be skipped. This is usually where the worst case happens for all the layouts discussed above. One solution to this problem is to use Solid State Drive (SSD) as secondary storage device which significantly reduces the bottlenecks led

by data seek time [24]. However, regardless of the fact that SSDs have become more popular during the last few years, because of their limitations such as restrictive number of write operations and relative cost, many applications still rely on slow secondary storage devices which are cheaper and with much larger capacity, such as hard drives, DVDs, and Blu-rays. For these devices, bounding number of disk seeks remains an open problem.

Main Contributions: In this work, we explore one of the extreme cases in the spectrum of the solution space of the data layout problem that has an upper bound on the number of disk seeks to fetch the required data. We present a novel data layout to limit the number of disk seeks per frame to one. The key observation that enables bounded number of seeks is to store the data redundantly at appropriate places in order to fetch all the required data in fewer seeks. We call this a non-linear data layout scheme since there is no linear ordering of the data. In other words, in a linear layout, each geometric primitive (except for the first and last ones) appears after one primitive and before another primitive; however, in our non-linear layout, multiple copies of each primitive can appear before and after different primitives. We also show that many online processing techniques can be incorporated in our layout computation as preprocessing stages. Thus, non-linear data layout not only reduces the disk seek time, but also reduces the online processing time. Thus, our achievements are:

- We introduce a single-seek data layout scheme which can be built in to common large 3D dataset walkthrough applications to improve the worst case performance.
- We derive the relationship between the object space subdivision for navigation, redundancy factor, and the average transfer volume and hence the frame rate. This enables the user to choose an appropriate level of redundancy and still have a single disk seek layout.
- Through a set of experiments with two large models with different data density properties, we demonstrate how different data processing operations including simplification, visibility culling, and back-face culling can be included as preprocessing stages in our data layout and achieve much better worst case performance than current rendering systems.
- In practice, we explore in which ways our layout can improve rendering performance of existing orthogonal techniques like using predictive pre-fetching of data.

Our generic formulation of this problem of data layouts has, depending on the application, three different parameters to be traded-off – maximum number of disk seeks, maximum data transfer volume, and maximum redundancy factor. This work is the first step to explore this spectrum of data layout solutions by providing a solution for one extreme instance of this problem, in which the upper bound on disk seeks is set to one. In our solution, keeping this constraint on the number of seeks, we can trade-off transfer volume to redundancy while designing the layout. A generic solution for optimizing the layout for different bounds on the above three parameters is left open for future investigation.

II. Related Work

Interactive massive model rendering has a long history and a large body of literature focussing on various stages of the pipeline including fetch time reduction, online processing, and rendering. We only briefly discuss this literature as our current work is almost orthogonal and complementary to all the other relevant works, and can be used along with these works.

Data Transfer Time Optimization: Data transfer time or fetch time is the time spent on loading the data required for rendering from the secondary storage devices to the main memory. Although reducing the amount of data that needs to be fetched (e.g. through operations such as simplification) should be the primary focus at this step, since the problem is that of massive model visualization, usually the amount of data to fetch cannot be reduced beyond a certain level. Therefore, the focus has shifted from reducing the amount of data directly, to improving the use of the data that is already fetched by utilizing frame-to-frame consistency. In other words if the cache coherency is improved, fetch time is amortized over multiple frames. Many algorithms use space-filling curves [22] to compute cache-friendly layouts of grids. These layouts have been widely used to improve the performance of image processing [27] and terrain visualization [15], [20]. Isenburg et al. [14] proposed processing sequences as a generalization of the rendering sequences for various kinds of large-data processing. This processing sequence can be stored as an indexed mesh, called streaming mesh. Streaming meshes are represented as interleaved triangles and vertices that can be streamed through a small buffer [13]. These representations are particularly useful for offline applications like compression, which can adapt their runtime computations to a fixed ordering. Yoon et al. [28] proposed a generalized cache-oblivious layout of a mesh or a graph for efficient rendering and processing of massive models. Diaz-Gutierrez et al. [6], [7] presented a graph based algorithm for generalized cache-oblivious layouts of triangles for rendering and geometry processing. Sajadi et al. [23] proposed a graph-based cache-oblivious data layout scheme called the 2-factor layout and showed its benefits in a rendering method.

Online Processing Time Optimization: Usually, the number of primitives directly determines the online processing time required by the application. Various multi-resolution hierarchies have been designed to reduce the number of primitives and thus improve the performance of rendering of massive models [18]. There are a few approaches to use a dual hierarchy that can serve as a spatial hierarchy and multi-resolution hierarchy [19]. Primitive culling techniques, including back-face and view-frustum culling, are also effective ways to reduce the number of primitives and to improve the performance of rendering [29]. For models with high depth complexity, visibility culling accelerated by graphics hardware are proposed [3]. Other techniques on this domain use image-based [2], [16], [11] or geometry-based multiresolution representations [8], [4] to achieve interactive performance.

These methods also use explicit out-of-core data management techniques [26] and pre-fetching mechanisms [5] to efficiently deal with the massive models.

Rendering Time Optimization: The most popular approach to reduce the rendering time is to take advantage of the architecture of the GPUs more effectively. Hoppe [12] cast the computation of rendering sequences as a discrete optimization with a cost function derived for a specific vertex buffer size used in a GPU. Sander et al. [25] designed a fast triangle reordering method to compute a cache-aware layout for a vertex cache size depending on a GPU. Parallelism in GPU is exploited by Eilemann and Pajarola [9] through a framework for parallel rendering. There is a large body of work on using GPU features to reduce the rendering time and at the same time increase the realism of the rendered image. Since rendering time optimization is not the focus of the current work, we refer to [10] for further details on this topic. Peng and Cao [21] combined mesh simplification algorithm towards GPU architecture and GPU out-of-core approach to utilize the power of high performance GPU. We have a close comparison with this approach in Section V-C.

Our Approach: Single-seek layout is complementary to any of the previous approaches, and is designed to bound the number of disk seeks to be one at each time data is loaded from secondary storage devices. This is motivated by the fact that a large chunk of total rendering time is spent on seeking over the disk. We guarantee that the number of disk seeks required for each frame is limited to one. Such ideas have already been used and ad hoc solutions that are highly implementation dependent are employed by a few game companies to reduce the large latency due to disk seek time of Blu-ray drives on PlayStation 3 machines. Our work, in contrast, presents a formal method for general walkthrough applications along with theoretical analysis. Our method can work along with all the primitives reduction and rendering optimization methods described above. Specifically, our data layout can work with multiple levels of detail (LOD) and still retrieve all the data required to render each frame with a single disk seek. Other online processing steps, like primitive culling, can also be done off-line and stored using our layout. The proposed single seek data layout *significantly improves the worst case performance of the rendering system without compromising on the best case performance*.

III. Single-Seek Data Layout

Let us consider a data-request state v as a state in which a bounded size of data is requested by its application. In other words, for the same state v, same bounded amount of data is required by the application, independently of its access pattern. Also, assume that there is a finite number of data-request states. We define a single-seek data layout as a function $S(v) = (p, b)$ where p is the starting position on the disk from which all the data required for the state v is stored, and b is the contiguous number of bytes that is to be read to have all the data required for state v in the memory. We call b the transfer volume for state v. Assuming the number of states v and the amount of data b for each v are bounded, the total amount of storage required by single-seek data layout is bounded as well. However, since part of the data requested by one state can be requested by other states as well, this common data can be repeated on the disk at the segments corresponding to each of those different states. Thus single-seek layout is achieved at the cost of redundancy in the storage. We will see in the rest of the section that a larger redundancy can be traded off with a smaller average transfer volume.

In a walkthrough application of static and bounded data, the viewpoint and viewing direction may define a data-request state v. If this user navigation space of viewpoints is discretized, then we have a finite number of data-request states and further, since the visualized data is static and bounded, this application is suitable for a single-seek data layout.

We observe that it is better to parameterize the data-request state v using the transition between viewpoints instead of the viewpoint itself. In other words, it can be parameterized as $v_1 \rightarrow v_2$, for which the requested data is the additional data required when the viewpoint moves from v_1 to v_2. Such a parametrization makes better use of the caching properties of the system than just the viewpoint parametrization. In the rest of this section, we present our method for single-seek data layout for walkthrough applications.

We have three stages of offline preprocessing to create a single-seek layout: discretization of the navigation space, primitives reduction, and data layout computation. We also describe the online data management and rendering process.

A. Discretization of the Navigation Space

As discussed above, in order to have a finite number of data-request states v to enable a single-seek data layout, we divide the navigation space into small contiguous regions called *cells*.

Any data that is visible from any viewpoint within a cell but not already in the main memory is loaded once when the user enters that cell. As a practical approach, we sample five discrete viewpoints and eight discrete view directions within a cell, collect the union of all the primitives visible from each viewpoint and view direction combination, and consider it as the data visible from that cell. View direction culling is processed while rendering and is discussed in Section V-B. In the single-seek layout parameterization, the state of the system is given by (c_1, c_2) where the view point is crossing the boundary from cell c_1 to cell c_2. The data that is visible from c_2 and not from c_1 is laid out contiguously on the disk, and hence is fetched in a single seek. In other out-of-core walkthrough algorithms that use this cell-based approach, this difference data may not be organized and can be anywhere on the disk.

Note that such cell-based navigation space discretization has been used before for LOD management [17] and texture-substitution based simplification methods [1] for interactive rendering. Our method is complementary to these preprocessing methods and can work together with them. We can store, using our single-seek data layout algorithm, the entire

geometry or simplified LOD models and/or texture bill-boards built for each cell, depending on the application requirements.

B. Primitives Reduction

Theoretically, when the entire geometry in its full resolution (with no simplification) can be seen from every cell, and occlusion culling cannot be applied, the entire original model must be treated as the data required for each cell. In this extreme case, differences among cells become empty sets, with the (initial) transfer volume equals to the data size, and the redundancy factor is just 1.0 (because there is no repetition of data). The entire model is fetched to the main memory in one seek, and the rendering process becomes in-core rendering method similar to [21]. In other words, the single-seek layout solution space spans both in-core rendering and out-of-core rendering techniques. Thus from the perspective of out-of-core rendering, single-seek data layout, and in fact interactive rendering of large data sets, becomes interesting only with different levels of detail representation, and there are cells in the navigation space with different visible primitive sets.

In our experiments, for LOD creation, we use [17] in which the object space is subdivided using octree, and different LODs are created by collapsing the vertices within the same octree node at different levels in the hierarchy. Note that one can substitute this method with any other simplification method that is suitable for a specific application. To ensure rendering quality, user can specify a screen space projection error tolerance for coarser LODs. From a given viewpoint, if the number of pixels spanned by the edges of an octree node is less than this tolerance, then that node is collapsed. The LOD of an object required by a cell is given by the finest LOD of that object required by any sample viewpoint within that cell. The total data required by that cell is the union of (required) LODs of all objects of the scene visible from it.

Moreover, similar to simplification, other online processing methods that are dependent on viewpoint (not necessarily on viewing direction) such as visibility culling can also be precomputed and stored in our data layout. These operations reduce both the data transfer requirements and the online processing time during rendering.

C. Data Layout

Page Unit of Data: Given the above data required by (visible and rendered from) a cell, this data is partitioned into different clusters such that all the primitives in the same cluster have the same property that is required for processing, so that accessing and processing can be done at the resolution of a cluster. In our system, we define pages of fixed size, i.e. 16KB, such that all the primitives (usually vertices and triangles) in the same page are spatially close to one another and have similar normal directions. A similar approach has been used by Sajadi et al. in [23]. Using a fixed size for the pages helps us in data management during the rendering (Section III-D) and the spatial and normal direction coherency allow us to perform efficient view-frustum and back-face culling on the bounding box and normal cone of the disk pages instead of individual primitives. Furthermore, two triangles are added to the same page only if they belong to the same LOD and have at least two of their vertices in the same cell. These constraints ensure that two triangles representing the same object but belonging to different levels of detail are not in the same page, and hence are not rendered together at the same time.

Final Data Layout: Each seek in the layout is indexed by (c_i, c_j), where c_i is the current cell in which the view point exists, and c_j is the edge-adjacent cell to which the viewpoint is moving. At a steady state, the pages required by c_i are already in the main memory, and for (c_i, c_j), all the pages of data required by c_j but not by c_i are stored. In addition, the IDs of the pages that were used by c_i, but are no longer required by c_j is also stored. The data pages that need to be loaded to or removed from the memory are illustrated in Fig. 2 for a pair of adjacent cells in 2D.

D. Data Management

The previous three subsections described the off-line processing steps. In this section, we detail the online processing steps for interactive rendering – data fetching, online data processing, and rendering. Assuming that all the data required for the viewpoint in the current cell is present in the main memory, when the viewpoint moves to an adjacent cell, the difference data from the current to the new cell, which is stored contiguously in a linear order in the disk as described above, is fetched with a single seek. The pages in the main memory that are not required for the new cell are tagged dirty. The new data pages that are loaded, first replace the dirty pages, and to accommodate additional pages, if any, the main memory buffer size is expanded.

Since we push online processing methods to preprocessing as much as possible, the actual implementation using our method can end up with very limited online processing requirements. However, we can still perform some online processing such as view-frustum culling (at the page level using the bounding box information of the primitives in that page) and back face culling, to further improve the performance.

Finally, all the triangles in the remaining pages are rendered. Note that, while model simplification is important and required, other page culling techniques are optional. For example, if the entire model has to be rendered in transparency mode, performing back-face culling will decrease the image fidelity, and hence should be avoided. Single-seek disk layout replaces a major part of tranditional online processing with preprocessing, resulting in the availability of much more time for the final rendering step. Furthermore, note that, with our technique, we have all the required geometry at every frame even for faraway objects, and we do not use texture subtitution techniques to achieve interactive rendering. With additional time in hand, we can use the available geometry to achieve sophisticated rendering effects such as transparency, shadows, and realistic lighting and shading.

IV. Analysis of the Performance Parameters

Data layout can be optimized to the applications requirements through three parameters – average number of disk

seeks, redundancy factor, and average data transfer volume. We set the number of data seeks to be one here, and find a layout that optimizes the other two parameters to meet the applications requirements. Redundancy affects the storage requirements, and the data transfer volume affects the time (frame rate) requirements. In this section we show that these two parameters, are in, turn related to one single design parameter, namely the cell size.

Let us assume the given model is referred as level 0 and is simplified to m levels of details where level m is the coarsest one. We assume this simplification is processed via an octree-based technique, and the edge length of the octree nodes that are collapsed in the $(i+1)$th LOD is twice the edge length of the octree nodes collapsed in the ith LOD. For the purpose of this analysis, we assume the navigation space of the user is a 2D grid of cells each of size $a \times a$ units. For each cell the primitives located within a distance of approximately $a/\sqrt{2}+r$ from the center of the cell are rendered in the original resolution (0th LOD), and the primitives located between distances $a/\sqrt{2}+2^{i-1}r$ and $a/\sqrt{2}+2^{i}r$ are rendered in the ith LOD as illustrated in Fig. 3(Left). The value of r is determined by the size of the leaf nodes and the screen spaces pixel error tolerance requirements.

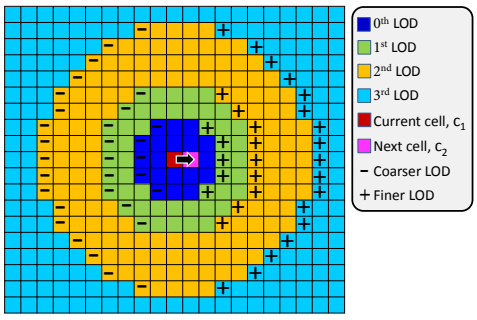

Fig. 2. Illustration of the data fetching. LODs are shown in different colors. The cells marked with '+' are loaded at a finer LOD and the cells marked with '-' are loaded at a coarser LOD.

Redundancy Factor: We define the average number of times the primitives in the ith LOD are stored in the disk as the redundancy factor R_i. The expected redundancy factor for the ith LOD, $E(R_i)$ is derived below.

Lemma 1. *Assuming the primitives are very small compared to the cell size,* $E(R_0) = 4\lceil\sqrt{2}+2r/a\rceil$, $E(R_i) = 4\lceil\sqrt{2}+2^{i+1}r/a\rceil + 4\lceil\sqrt{2}+2^{i}r/a\rceil$, $0 < i < m$, *and* $E(R_m) = 4\lceil\sqrt{2}+2^{m}r/a\rceil$, *where* E *denotes the expected value.*

Proof: Let us assume that we are navigating from cell c_1 to an adjacent cell c_2. Let us assume that cell c' is within distance $r_i^{'} = a/\sqrt{2}+2^{i}r, i < m$ from the center of c_2 but outside distance $r_i^{'}$ from the center of c_1. We call this a forward navigation towards c'. When navigating from c_1 to c_2 we need to load the content of the ith LOD of c'. For each such transition, the ith LOD of c' needs to be stored on the disk once.

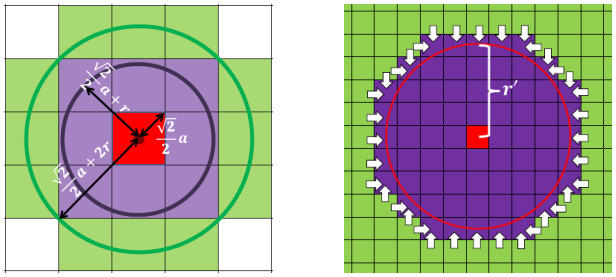

Fig. 3. Left: Illustration of the radii of the 0th and first LODs for a viewpoint within the red cell, $a/\sqrt{2}+r$ and $a/\sqrt{2}+2r$ respectively. Right: The green cells demonstrate the cells for which the red cell needs to be rendered at the ith LOD. The blue cells demonstrate the cells for which the red cell needs to be rendered at an LOD finer than the ith LOD. Each arrow shows a transition for which we need to load the ith LOD of the red cell. This equals to the number of cells covered in a Manhattan traversal around a circle of radius $r_i^{'}$ without getting inside any cell that has an overlap with the circle.

The number of such transitions is equal to the number of cell edges that form the outline of the circle with radius $r_i^{'}$, which is $8r_i^{'}/a$, after cell quantization (Fig. 3(Right)). Thus, the number of times we need to store c' in the ith LOD due to forward navigation is $F_i = 8\lceil\sqrt{2}/2 + 2^{i}r/a\rceil = 4\lceil\sqrt{2}+2^{i+1}r/a\rceil$. However, we may also navigate from c_2 to c_1 in which case we need to load the $(i+1)$th LOD of c' (the coarser representation of c') from the disk. We call this a backward navigation away from c'. We denote the number of times the $(i+1)$th LOD of c' needs to be stored due to a backward navigation B_{i+1} which is equal to F_i. Moreover, we know that we always load the 0th LOD of a cell either in a forward navigation towards it and the last LOD or in a backward navigation away from it. The other LODs can be loaded in either of forward or backward navigation. Therefore, we can compute the average number of times a cell has to be stored (redundancy factor) as follows:

$$R_0 = F_0 = 4\lceil\sqrt{2}+2r/a\rceil. \quad (1)$$

$$R_i = F_i + B_i = 4\lceil\sqrt{2}+2^{i+1}r/a\rceil + 4\lceil\sqrt{2}+2^{i}r/a\rceil, \quad (2)$$

$$0 < i < m.$$

$$R_m = B_m = 4\lceil\sqrt{2}+2^{m}r/a\rceil. \quad (3)$$

■

Let us assume the number of primitives in the ith LOD without any redundancy is equal to N_i. We define the total redundancy factor as

$$R = \frac{\Sigma_{i=1}^{m} R_i N_i}{\Sigma_{i=1}^{m} N_i}. \quad (4)$$

Transfer Volume: The average transfer volume is the average amount of data that needs to be loaded to the main memory when going from one cell to another. We define it as V. The expected transfer volume $E(V)$ is derived below.

Lemma 2. $E(V) = (\lceil(\sqrt{2}+2r/a\rceil+1)N_0 + \Sigma_{i=1}^{m-1}(\lceil\sqrt{2}+2^{i+1}r/a\rceil + \lceil\sqrt{2}+2^{i}r/a\rceil + 2)N_i + \lceil(\sqrt{2}+2^{m}r/a\rceil + 1)N_m)a^2/A$, *where* E *denotes the expected value.*

Proof: When entering cell c_2, the data of the finer LOD of all the cells that are marked by $'+'$ in Fig. 2 are loaded. The number of such cells is bounded by $2r_i^{'}/a + 1 = \lceil \sqrt{2} + 2^{i+1}r/a \rceil + 1$ for the ith LOD. Based on the similar theory from Lemma 1, while leaving from cell c_1, the data of the coarser LOD of the cells with $'-'$ signs in Fig. 2 are loaded as well. This number is bounded by $2r_{i-1}^{'}/a + 1 = \lceil \sqrt{2} + 2^i r/a \rceil + 1$ for the ith LOD.

Let us assume the total area of the model is A units and the density of the triangles over the whole model is uniform. Then the triangle density of the ith LOD is N_i/A. Furthermore, we know that the area of each cell is a^2. Let $C = a^2/A$, using these we can approximate the total transfer volume as:

$$V_0 = \lceil (\sqrt{2} + 2r/a \rceil + 1) N_0 C. \tag{5}$$

$$V_i = (\lceil \sqrt{2} + 2^{i+1} r/a \rceil + \lceil \sqrt{2} + 2^i r/a \rceil + 2) N_i C, \tag{6}$$

$$0 < i < m.$$

$$V_m = \lceil (\sqrt{2} + 2^m r/a \rceil + 1) N_m C. \tag{7}$$

$$V = V_0 + \Sigma_{i=1}^{m-1} V_i + V_m. \tag{8}$$

■

In summary, Equations 2 and 8 show that *increasing the cell size reduces the redundancy factor, but quadratically increases the transfer volume, which in turn reduces the frame rate*. Therefore, we need to find the appropriate cell size to avoid a very large redundancy factor and at the same time bound the transfer volume.

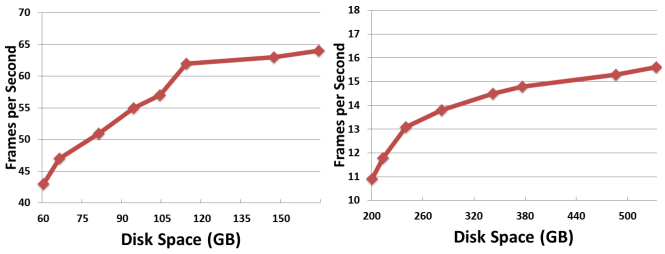

Fig. 4. Trade-off between the number of frames per second for different redundancy factors. Left: the City model; Right: the Boeing model. Different data point in the graph is collected at different cell sizes. The cell size corresponding to the knee of the graph is chosen as the optimum operating point beyond which there are diminishing returns for increased redundancy.

Computing the Cell Size: Since in Equations 1, 2, and 3 the ratio of r/a determines the redundancy factor, we set the cell size a proportional to r rather than absolute values in our experiments. In order to determine a proper cell size a, experiments with different cell sizes were performed, and we monitored the trade-off between the redundancy factor and the frame rate (which is inversely proportional to the transfer volume). Fig. 4 is such a graph for the City model. In general, we can see that decreasing the cell size increases the

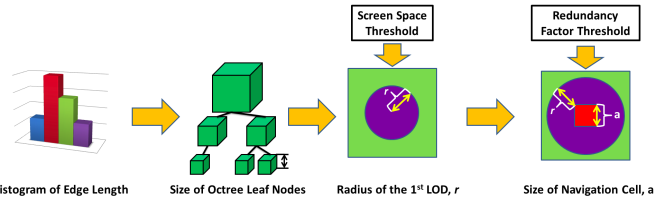

Fig. 5. The different steps through which we decide on the proper size of the leaf nodes and also the cell size, a. During this process, the radius of the zeroth LOD, r, is automatically determined based on the size of the leaf node, the desired screen resolution, and the pixel error threshold.

redundancy factor, reduces the transfer volume and increases the frame rate. In the initial stages, when the redundancy factor is low, and the transfer volume is still high, any reduction in the cell size and hence the transfer volume directly translates to improvement in the frame rate. After the knee of the graph, the cell sizes are small enough that for the same navigation distance and speed, the cell boundaries are crossed more often. Although the number of seeks for every boundary crossing is just one, the total number of disk seeks for the same distance navigation is large enough to counter the speed up gained by smaller transfer volume. Any further reduction in the cell size and hence the transfer volume has only marginal effect on the frame rate. This shows the significance of number of disk-seeks on the frame rate. In our implementation, we choose the cell size to be the knee of the graph, although any other cell size that satisfies an application specified bound on the disk space usage can be used.

V. Implementation and Results

We used two models in our experiments. The first one is a *Boeing 777* airplane model, which is partially displayed in Fig.1 (left). It includes 350 million triangles, with total $11.8GB$ storage space in binary uncompressed format with one LOD and $20GB$ with 4 LODs. For this model, in addition to visibility and back-face culling (Section III-B) we also store an ordering of the disk pages for each cell based on the distance of the pages from the center of the cell. Using this, we can allow the user to select any object in the scene and make it transparent with a small online processing overhead (Section V-B). Fig. 6 shows the Boeing model with the transparency effect. The entire preprocessing takes 336 minutes.

The second one is a city with 110 million triangles shown in Fig.1 (right), with total storage space of $3.7GB$ in binary uncompressed format with one LOD and $6.0GB$ with 3 LODs. No preprocessing other than the simplification was performed to this model. The preprocessing time for this model is 34 minutes.

For our experiments with use a Dell T5400 Workstation with Intel(R) Core (TM) 2 Quad with $4GB$ of Main memory, we used a 1 TB Seagate Barracuda hard drive with 7200 RPM and an nVIDIA Geforce GTX 260 graphics card with 896 MB dedicated memory in our system. Finally, all the statistics are gathered for rendering at 600×600 image resolution.

A. Experimental Parameters

In the experiment setup, all the parameters that are required for single seek data layout are computed. The computation pipeline is shown in Fig. 5. For octree construction, we used 15 bits to represent each of the x and z coordinates and 8 bits to represent the y values since the model is relatively flat. The number of levels in the octree, or, in other words, the size of the leaf node of the octree is determined by the average of edge length of triangles in the model.

Next, the value of r in Fig. 5, Equations 1, 2, and 3 is computed as the smallest distance from the cell at which the leaf node of the octree can be collapsed and still maintain the visual fidelity of the rendering up to the user specified screen space threshold. Given the redundancy factor threshold, cell size a is calculated based on Equation 4 and then applied to the layout.

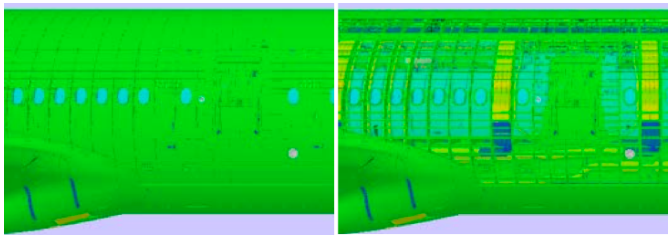

Fig. 6. Rendering of the Boeing model before (left) and after (right) making part of the body transparent. The user can interactively choose any part of the model and make it transparent.Discarding the visibility flags combined with the extra overhead of sorting the close by objects reduced the average frame rate of our system from $13fps$ to $4fps$ when allowing only one pixel error in the screen space.

B. Rendering

During rendering, we first load the required data and the visibility flag table we created during pre-processing. We preserve a 32 KB array buffer for marking dirty pages. The visibility flag table consumes approximately 6 MB memory space. Based on our experiments, the visibility culling for the Boeing model can cull-out 82.4% of the primitives in average. On the remaining data, we perform a low cost view frustum culling on the bounding box of each disk page. This online view-frustum culling for a $90°$ field-of-view culls-out around 75% of processed primitives, and back face culling culls roughly 40% of the remaining primitives. The actual memory usage of the Boeing model is 2.8 GB.

For rendering with the transparency effect, we order the pages based on following three rules before sending them to the renderer: first, coarser LODs are rendered earlier than finer LODs; second, within each LOD, the pages are rendered based on the pre-computed order of page IDs stored for each cell; finally, the primitives that are rendered in their finest LOD are sorted online before rendering. Also we do not use the visibility flags in the transparency mode. This does not lead to changes in the redundancy factor or transfer volume. However, since back face culling and visibility culling are not available any more, and view frustum culling takes more time, frame rate drops when compared to rendering without transparency. In our experiment, the frame rate in tranparency mode is about $1/9$ of opaque mode for the Boeing model.

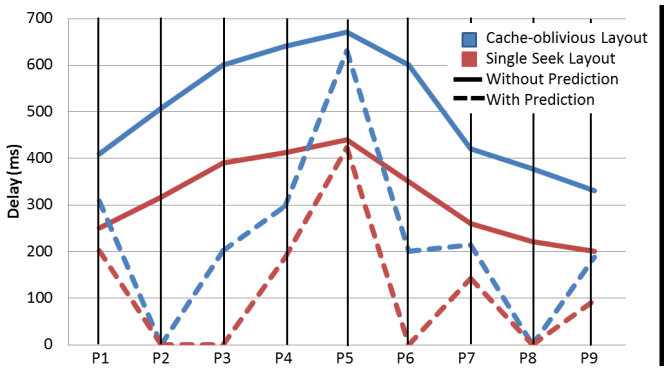

Fig. 7. Comparison of maximum delay between two consecutive frames when data loading happens for single-seek layout and cache-oblivious layout. Solid lines are the results without applying prediction, and dash lines are the results where prediction is available. All experiments are done on the Boeing model and in the same navigation path.

C. Comparison

In order to show the advantage of the single seek time achieved by our method we build another version of the City and the Boeing models with a cache-oblivious layout [28]. The data required for each cell is computed in the exact same way as the single-seek data layout. The only difference is that the data is not stored with redundancy. Instead, for each cell, there is a list of page IDs that need to be loaded from the disk. Thus, this data may be loaded with multiple seeks. For the City model, the worst case frames (plural) per second (FPS) is improved from 4.3 (cache-oblivious layout) to 7.0 (single-seek layout), which is an improvement of 62.8%. The average FPS in this case is improved from 48.5 (cache-oblivious layout) to 51.0 (single-seek layout). For the Boeing model, the worst case performance is improved from 1.5 fps (cache-oblivious layout) to 2.4 fps (single-seek layout), which is an improvement of 60.0%. The average FPS in this case is improved from 10.8 (cache-oblivious layout) to 13.1 (single-seek layout). These statistics are collected under 1 pixel error threshold, and the cell size a is measured as $1/4$ with respect to r. This result shows that the single-seek layout significantly improves the worst case frame rates, and also improves average frame rate. In Fig. 7, solid lines illustrate the maximum delays for the single-seek data layout and the cache-oblivious data layout for the Boeing model with one pixel error for same navigation sequence. Since the amounts of data to be loaded and rendered at each point P are the same, the lower delays are clearly due to the guaranteed single-seek.

The single-seek data layout is orthogonal to existing data prefetching techniques. We did the same experiment with prediction, which requires one more thread and a second buffer in the main memory. Dash lines in Fig. 7 demonstrate

the result. There are three different cases: (1) the predictions are correct and the data fetching finishes before reaching the destination cell: data transferring is completely avoided at cell boundaries, hence delays are eliminated, e.g. P2, P8; (2) the predictions are perfect but the data fetching cannot be done before reaching cell boundaries: single-seek data layout can improve the frame rates, e.g. P4, and is even possible to completely remove delays, e.g. P3, P6; (3) the predictions are proved to be incorrect: data fetching is still required at cell boundaries, and prefetching can only reduce delays in a portion, which makes single-seek data layout is useful in this worst case scenario, e.g.P1, P5, P7, P9.

Comparing our method with the GPU based approach introduced by Peng and Cao [21], in their work, they use an Intel Core i7 2.67GHz PC with 12 GB of RAM, and a Nvidia Quadro 5000 graphics card with 2.5 GB of GDDR5 device memory, which is a high performance computer and usually not available for common use. They also tested their algorithm on the same Boeing model that we use. By utilizing the large size of the main memory and the power of the GPU, they can load the entire Boeing model into the main memory and render it with only simplification rather than culling on an average of 9.8 fps. It is reasonable to say that this performance is the best current technology can accomplish without redundancy. In contrast, we use a relatively low cost PC and by employing the idea of preprocessing and single seek layout, we can achieve on an average 13.1 fps without GPU based acceleration. Since our method is complementary to other GPU based methods, there is further room for acceleration.

VI. Conclusion

In summary, we showed that the seek time for slow secondary storage devices is a major performance bottleneck for out-of-core interactive applications and can directly affect the frame rate and more importantly the maximum delay between two consecutive frames. Therefore, we proposed a data layout that utilizes redundant storage of the data to prevent more than one disk seek for fetching the data required for each frame. We have provided an analysis of the amount of redundancy resulting from our approach. Furthermore, we showed that a higher redundancy can be traded off with smaller fetch time (lower data transfer volume) and therefore higher frame rate in the worst cases. Using this analysis and a series of experiments, we showed how the user can choose the best redundancy factor based on the applications requirements and the available resources. Finally, we demonstrated how view-dependent pre-processing stages can be incorporated in our data layout to allow interesting rendering effects such as transparency with a small overhead on the system.

The main concern of single-seek data layout is the large storage space it consumes. Furthermore, it is not flexible enough when users are willingly to sacrifice performance slightly to save storage space requirement partially. There is a spectrum of data layout solution for different trade offs between number of disk seeks, amount of data transfer volume, and the redundancy factor. Single-seek layout is only the first work in this direction we have explored. In the future we would like to explore the generic problem of varying caps on number of disk seeks.

References

[1] *Automatic image placement to provide a guaranteed frame rate*. Aliaga, D., and Lastra, A.:ACM , 1999.

[2] *An interactive massive model rendering system using geometric and image-based acceleration*. Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., and Manocha, D.:ACM , 1999.

[3] *Coherent hierarchical culling: Hardware occlusion queries made useful*. Bittner, J., Wimmer, M., Piringer, H., and Purgathofer, W.:Eurographics , 2004.

[4] *TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models*. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R., Adaptive:ACM , 2004.

[5] *Visibility-based prefetching for interactive out-of-core rendering*. Corrêa, W. T., Klosowski, J. T., and Silva, C. T.:IEEE , 2003.

[6] *Constrained Strip Generation and Management for Efficient Interactive 3D Rendering*. Diaz-Gutierrez, P., Bhushan, A., Gopi, M., and Pajarola, R.:Computer Graphics International Conference , 2005.

[7] *Single Strips for Fast Interactive Rendering*. Diaz-Gutierrez, P., Bhushan, A., Gopi, M., and Pajarola, R.:The Visual Computer , 2006.

[8] *Roaming terrain: Real-time optimally adapting meshes*. Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B.:IEEE , 1997.

[9] *The equalizer parallel rendering framework*. Eilemann, S., and Pajarola, R.:Technical Report , 2007.

[10] *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Fernando, R.:Pearson Higher Education , 2004.

[11] *Far voxels: A multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms*. Gobbetti, E., and Marton, F.:ACM , 2005.

[12] *Optimization of mesh locality for transparent vertex caching*. Hoppe, H.:ACM , 1999.

[13] *Streaming meshes*. Isenburg, M., and Lindstrom, P.:IEEE , 2005.

[14] *Large mesh simplification using processing sequences*. Isenburg, M., Lindstrom, P., Gumhold, S., and Snoeyink, J.:IEEE , 2003.

[15] *Visualization of large terrains made easy*. Lindstrom, P., and Pascucci, V.:IEEE , 2001.

[16] *Geometry clipmaps: Terrain rendering using nested regular grids*. Losasso, F., and Hoppe, H.:ACM , 2004.

[17] *View-dependent simplification of arbitrary polygonal environments*. Luebke, D., and Erikson, C.:ACM , 1997.

[18] *Level of Detail for 3D Graphics*. Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., and Huebner, R.:Morgan-Kaufmann , 2002.

[19] *CLODs: Dual hierarchies for multiresolution collision detection*. Otaduy, M. A., and Lin, M. C.:Eurographics , 2003.

[20] *Global static indexing for real-time exploration of very large regular grids*. Pascucci, V., and Frank, R. J.:ACM/IEEE , 2001.

[21] *A GPU-based approach for massive model rendering with frame-to-frame coherence*. Peng, C., and Cao, Y.:Computer Graphics Forum , 2012.

[22] *Space-Filling Curves*. Sagan, H.:Springer-Verlag , 1994.

[23] *A novel page-based data structure for interactive walkthroughs*. Sajadi, B., Huang, Y., Diaz-Gutierrez, P., Yoon, S.-E., and Gopi, M.:ACM , 2009.

[24] *Data management for SSDs for large-scale interactive graphics applications*. Sajadi, B., Jiang, S., Gopi, M., Heo, J.-P., and Yoon, S.-E.:ACM , 2011.

[25] *Fast triangle reordering for vertex locality and reduced overdraw*. Sander, P. V., Nehab, D., and Barczak, J.:ACM , 2007.

[26] *Out-of-core rendering of massive geometric datasets*. Varadhan, G., and Manocha, D.:IEEE , 2002.

[27] *Digital halftoning with space filling curves*. Velho, L., and de Miranda Gomes, J.:ACM , 1991.

[28] *Cache-oblivious mesh layouts*. Yoon, S.-E., Lindstrom, P., Pascucci, V., and Manocha, D.:ACM , 2005.

[29] *Real-Time Massive Model Rendering*. Yoon, S., Gobbetti, E., Kasik, D., and Manocha, D.:Morgan & Claypool Publisher , 2008.