Performance Driven Redundancy Optimization of Data Layouts for Walkthrough Applications

Abstract

Performance of interactive graphics walkthrough systems depend on the time taken to fetch the required data from the secondary storage to the main memory. It has been earlier established that a large fraction of this fetch time is spent on seeking the data on the hard disk. In order to reduce this seek time, redundant data storage has been proposed in the literature, but the redundancy factors of those layouts are prohibitively high. In this paper, we develop a cost model for the seek time of a layout. Based on this cost model, we propose an algorithm that computes a redundant data layout with the redundancy factor that is within the user specified bounds, while maximizing the performance of the system.

Keywords: Data Layout Problem, Out-Of-Core Rendering, Cache Oblivious Mesh Layout, Redundant Data Layout, Walkthrough Application

1 Introduction

In typical walkthrough systems, data sets consisting of hundreds of millions of triangles and many gigabytes of associated data (e.g. walking through a virtual city) are quite common. Rendering such massive amounts of data requires out-of-core rendering algorithms that bring only the required data for rendering into main memory from secondary storage. In this process, in addition to the rendering speed, the data fetch speed also becomes critical for achieving interactivity, especially when we handle large-scale data. In general, data fetch speed depends on data seek time and data transfer time. Transfer time depends only on the amount of data that is transferred. Seek time is the time taken to locate the beginning of the required data in the storage device and depends on different factors depending on the storage medium.

For a hard disk drive (HDD), its seek time depends on the speed of rotating the disk, and the relative placement of the data units with respect to each other, also called the data layout [Rizvi and Chung 2010]. For a solid state drive (SSD), this seek time is usually a small constant and is independent of the location of the data with respect to each other [Agrawal et al. 2008]. An earlier work utilized this difference between SSD and HDD, and designed a data layout tailored for using SSDs with the walkthrough application [Sajadi et al. 2011]. There have been many other techniques utilizing SSDs for various applications [Saxena and Swift 2009]. SSD, unfortunately, is not the perfect data storage and has its own technical problems, including limited number of data overwrites allowed, high cost, and limited capacity [Rizvi and Chung 2010].

On the other hand, the HDD technology – including disk technologies such as CDs, DVDs, and Blu-ray discs – has become quite reliable and inexpensive thanks to their extensive verifications and testing, and is thus in widespread use. Even for massive data sets HDDs are still and will be the preferred medium of storage for the foreseeable future [Rizvi and Chung 2010], mainly because of its stability and low cost per unit. As an example, according to [Domingo 2014], as of 2014, an HDD can cost $0.08 per GB, while an SDD can cost $0.60 per GB. As a result, optimizing components of walkthrough systems with HDDs is critical. In particular, addressing the seek time, the main bottleneck of accessing data from HDDs, remains the main challenge for interactive rendering of massive data sets.

In this paper, we leverage the inexpensive nature of HDDs to store redundant copies of data in order to reduce the seek time. Adding redundancy in order to improve the data access time is a classic approach, e.g., RAID [Patterson et al. 1988]. We are also not the first to consider redundancy for walkthrough applications. Redundancy based data layouts to reduce the seek time were introduced in a recent work [Jiang et al. 2013], in which the number of seeks for every access was reduced to at most one unit. However, in order to achieve this property, the redundancy factor – the ratio between the size of the data after using redundancy to the original size of the data – was prohibitively high around 80.

Another recent work [Jiang et al. 2014] took the data transfer time, seek time, and redundancy, and proposed a linear programming approach to optimize the data transfer and seek time in order to satisfy the total data fetch time constraint. In the process, redundancy was a hidden variable that was minimized. Unfortunately, this approach does not directly model redundancy or seek time, and thus can have unnecessary data blocks and unrealistic seek times.

Main contributions: In this paper, we propose a model for seek time based on the actual number of units between the data blocks in the linear data layout. Using this model, and given the spatial proximity of the data set for a walkthrough application, we develop an algorithm to duplicate data blocks strategically to maximize the reduction in the seek time, while keeping the redundancy factor within the user defined bound. We will show that our greedy solution can generate both the extreme cases of data layout with redundancy, namely the no redundancy case (a simple cache oblivious mesh layout with a potentially high seek time) and the maximum redundancy case (a layout where seek time is at most one), as well as reasonable solutions for redundancy factor constraints in between the extremes. We show that the implementation of our algorithm significantly reduces average delay between frames and noticeably improves the consistency of performance and interactivity.

2 Related Work

Massive model rendering is a well studied problem in computer graphics. Most of the early works focused on increasing the rendering efficiency. At that time the fundamental problem was not fitting the model into the main memory but the speed of the graphics cards. Hence these works provided solutions to reduce the number of primitives to be rendered while maintaining the visual fidelity. These solutions included level-of-detail for geometric models [Luebke et al. 2002], progressive level of detail [Hoppe 1998; Hoppe 1997; Hoppe 1996; Shaffer and Garland 2001], and

image based simplification [Aliaga et al. 1999]. Soon thereafter the size of the main memory became the bottleneck in handling ever increasing sizes of the model. Hence memory-less simplification techniques [Lindstrom and Turk 1999] and other out-of-core rendering systems [Silva et al. 2002; Varadhan and Manocha 2002] emerged in which just the limited amount of required data that needs to be processed and rendered was brought from the secondary storage to the main memory.

The speed at which this data could be brought from the secondary to the main memory in these out-of-core algorithms is limited by the data bus speed, disk seek time, and data transfer time. These limitations could be ameliorated to some extent by better cache utilization that would increase the utilization of data that is brought to the main memory and thus reduce the number of times the disk read is initiated. This meant that subsequent works focused on cache aware [Sajadi et al. 2011] and cache oblivious data layouts [Yoon et al. 2005; Yoon and Lindstrom 2006] on the disk to reduce the data fetch bottleneck. Our work falls under this class of algorithms that reduces the data fetch time.

Redundancy based data layouts were mentioned in [Patterson et al. 1988; Jiang et al. 2013; Jiang et al. 2014] as potential solutions to this problem of reducing seek time. In particular [Jiang et al. 2014] presented an algorithm that limits the amount of redundancy required but there were major drawbacks. First, it provides a grouping of data units for each seek but it does not provide a data layout. This is because it does not relate one data group with another and it does not consider their relative layout. Such an approach could easily result in unnecessary data block duplications. The redundancy minimization is thus not modeled after physical representation of the data layout on the disk. The second major drawback is that the model for seek time is also not based on physical reality. Typically, seek time depends on the relative distance on the disk between the last data unit accessed and the data unit currently being requested. However, in [Jiang et al. 2014], seek time is simplistically modeled as counting the number of seeks, independent of the number of data units between them. This means that irrespective of whether the requested data blocks are adjacent to each other or far apart, this model would assign the same cost for both layouts. Our approach aims to address these issues.

3 Redundancy-based Cache Oblivious Data Layout Algorithm

3.1 Definitions

Let us assume that the walkthrough scene data, including all the levels of details of the model, are partitioned into equal sized data blocks (say 4KB) called data units. This is the atomic unit of data that is accessed and fetched from the disk. Typically, vertices and triangles that are spatially together (and belong to the same level of detail), have high chances of being rendered together, and hence can be grouped together in a data unit. All the data units required to render a scene from a viewpoint is labeled as an *access requirement*. In order to minimize the number of access requirements, the navigation space in the walkthrough scene, which defines the space of all possible view points, is partitioned into grids and all the viewpoints within each grid is grouped together to define one access requirement. Thus the number of grid partitions define the number of access requirements. Primitives in a data unit can be visible from many viewpoints, and hence that data unit will be part of many access requirements.

That was one example of data units and their access requirements. In general, the access requirements are determined by the application and are meant to be sets of data units that are likely to be accessed together.

Given a linear ordering of data units that may eventually be the order in which they are stored in the hard drive, for an access requirement A, the total span of A is the total number of data units between the first and last data units that use A. If a data unit is not required by A but lies between the first and last unit of A then it is still counted in the span of A. Figure 1 shows a linear order of data units and three different access requirements shown by solid, double-dashed and dotted lines. The span of an access requirement is the number of blocks between the first and the last data unit that use that access requirement. For example, for the access requirement shown with the solid line, the span is 11; the double-dashed line one has span 12, and the dotted line one has span 11. A data unit can be part of many access requirements. In the example shown in Figure 1, data units 1, 4 and 12 are part of two access requirements and data unit 9 is part of all three.

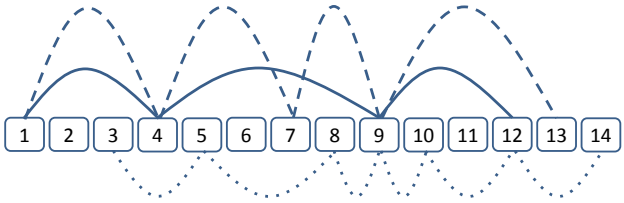

Figure 1: *Illustration of linear order of data units and three example access requirements. The lines connect data blocks that belong to the same access requirement and represent parts of the span of an access requirement.*

3.2 Seek Time Measure

Given a linear order of data units and the access requirements, and assuming that each access requirement is equally likely to be used, we would like to estimate the seek time for that application. For each access requirement the read head of the hard disk has to move from the first data block to the last irrespective of whether the intermediate blocks are read or skipped. Hence the span of an access requirement is a measure of seek time - time taken to seek the last data unit starting from the first data unit. Let I be the set of access requirements and A_i represent the span of the access requirement i. Then estimated total seek time EST is given by

$$EST = \sum_{i \in I} A_i$$

It is interesting to note that [Yoon et al. 2005] used span to measure the expected number of cache misses. Typically, with every cache miss, the missing data will be sought in the disk and fetched, thus adding to the seek time. Hence using span to measure the seek time is justified.

3.3 Algorithm Overview

In [Yoon et al. 2005], the only allowed operation on the data units is the move operation and the optimal solution is computed using only that operation. For our purposes, we are allowed

to copy data units, move them, and delete them if they are not used. Using these operations, our goal is to minimize EST while keeping the number of redundant copies as low as possible. After constructing a cache oblivious layout of the data set to get an initial ordering of data units, we copy one data unit to another location, and reassign one or more of the access requirements that uses the old copy of the data unit to the new copy, such that the EST is reduced. If all the access requirements that used the old copy, now use the new copy of the data unit, then the old copy is deleted. We repeat this copying and possible deletion of individual data units until our redundancy limit has been reached.

Blocks to Copy: Note that the span of an access requirement does not change by moving an interior data unit to another interior location. Cost can be reduced only by moving the data units that are at the either ends of the access requirement. This observation greatly reduces the search space of data units to consider for copying. Additionally, for the sake of simplicity of the algorithm, we operate on only one data unit at a time.

Location to Copy: Based on the above observation, given an access requirement, we can possibly move the beginning or the end data units of an access requirement to its interior. This will reduce its span, thus reducing the EST for the layout. However, if the new location of the data unit is in the span of other access requirements, such as location 11 in Figure 1, it increases the span of each of those accesses by one unit. Let j be the new location for the start or end data unit of an access requirement i. Let ΔA_i denote the change in the span of the access requirement i by performing this copying operation. Let k_j denote the number of access requirements whose span overlaps at location j. The reduction in EST by performing this copying operation is given by

$$\Delta EST_C(i,j) = \Delta A_i - k_j$$

where C denotes *copying* the data unit for access requirement i to the location j. We find the location j where the start or end data unit of the access requirement i needs to be copied using a simple linear search through the span of i as

$$argmax_j(\Delta EST_C(i,j))$$

Assignment of Copies to Access Requirements: The above operation would result in two copies of the same data unit, say d_{old} and d_{new}. Clearly the new copy d_{new} in location j will be used by the access requirement i. But d_{old} could be accessed by multiple access requirements. All other access requirements that accesses d_{old} can either continue to use d_{old} or use d_{new} depending on the overall effect on their span. Let S be the set of access requirements whose span does not increase by using d_{new} instead of d_{old}. Now the total benefit by copying the data unit d_{old} of the access requirement i to the new location j is

$$\Delta EST_C(i,j) = \Delta A_i - k_j + \sum_{s \in S} \Delta A_s. \quad (1)$$

Moving versus Copying: Let T be the set of access requirements whose span will increase by accessing d_{new} instead of d_{old}. Further, let k_{old} be the number of access requirements in whose span d_{old} is. If we force all the access requirements that uses d_{old} to use d_{new} and then delete d_{old} – in other words, if we move d instead of copying – then the benefit of this move would be given by

$$\Delta EST_M(i,j) = \Delta A_i - k_j + \sum_{s \in S} \Delta A_s + \sum_{t \in T} \Delta A_t + k_{old}$$

$$\Delta EST_M(i,j) = \Delta EST_C(i,j) + \sum_{t \in T} \Delta A_t + k_{old}$$

where $\Delta EST_M(i,j)$ gives the benefit of *moving* a start or end data unit of the access requirement i to position j. Note that each of ΔA_t is negative. Hence the benefit of moving might be more or less than the benefit of copying depending on the relative values of $\sum_{t \in T} \Delta A_t$ and k_{old}. But the main advantage of moving instead of copying is that this operation does not increase the redundancy thus it keeps the storage requirement the same. So we perform moving instead of copying as long as $\Delta EST_M(i,j)$ is positive.

Data Unit processing order: We now need to figure out how to use this information to decide in what order the copying and moving should be done. We will make two heaps: E_M and E_C. The E_M heap will organize the move operations and consist of the values of $\Delta EST_M(i,j)$ for the start and end data units for all access requirements i where the units are put in their optimal location j. The E_C heap will be the same thing except it will organize the copy operations and consist of the values of $\Delta EST_C(i,j)$.

We process the E_M heap first as long as the top of the heap is positive and effect the move of the data unit at the top of the heap. After each removal and processing, ΔEST_M and ΔEST_C of the affected access requirements and the corresponding heaps are updated. If there are no more data units where ΔEST_M is positive, then one element from the top of the heap E_C is processed. After processing and copying a data unit from the top of heap E_C, the heaps E_C and E_M are again updated with new values for the affected access requirements. If this introduces an element in the top of E_M heap with positive values, the E_M heap is processed again. This process gets repeated until the user defined bound on redundancy factor is reached. As a summary, the psuedo-code of this algorithm is shown as algorithm 1.

```
Input: Data units and their access requirements (AR) ;
for start and end unit of each AR i do
    Find optimal location j for copy;
    Calculate ΔEST_M(i,j) and insert into E_M ;
    Calculate ΔEST_C(i,j) and insert into E_C ;
end
while true do
    while top element of E_M is positive do
        Pop top element and move the data unit to its destination ;
        Update E_M and E_C ;
    end
    if there is more space for redundancy then
        Pop top element and copy the data unit to its destination ;
        Update E_M and E_C ;
    else
        break
    end
end
```

Algorithm 1: Pseudo-code for our algorithm

4 Complexity Analysis

We now analyze the running time and storage requirements of our algorithm. Let N be the number of data units and A be the number of access requirements. We will use k as the average span of a single access requirement. The variable Q will represent the number of data units that can be copied as specified by the user. Let r be the amount of redundancy if we have a single-seek layout [Jiang et al. 2013]. The average number of overlapping spans for a single data unit ends up being $O(rN)$.

Time Complexity: The construction of the heaps E_M and E_C involve computing the benefit information for all A access

requirements and inserting each one into the heap. Computing the benefit information of moving or copying a data unit involves scanning the span of the access requirement. We justify this approach which takes $O(k)$ operations in Appendix A. Inserting this benefit information into the heap takes $O(log(A))$ operations. Thus it takes a total of $O(A(k+logA))$ operations to do the initial construction of the heaps.

After the initial construction, in every iteration, an element from the top of the heap is removed and processed, the benefit function is recalculated for affected access requirements, and the heap is updated. For each data unit move or copy, $O(rN)$ overlapping spans are affected, and for each of these access requirements $O(k + log(A))$ is required to recalculate the benefit data and update the heap. Thus each iteration takes $O(rN(k + logA))$ operations.

For simplicity we will assume that the loop where we move the data units instead of copying is run $O(N)$ times total. This comes from the fact that the cache oblivious layout [Yoon et al. 2005] should be a good approximation so the number of moves that would still be useful should be limited. We defined Q as the number of redundant data units added. We thus can assert that there are $O(Q + N)$ iterations of the move or copy then update operation. This means that the moving and copying loops will take a total of $O((QrN + rN^2)(k + logA))$ operations.

In total, our algorithm takes $O((QrN + rN^2 + A)(k + logA))$ operations. In practice, the optimal redundancy factors we found were greater than 2 thus we can say that $Q > N$ meaning that $QrN > rN^2$. Additionally, since $r \geq 1$, we know that $rN >> A$. Thus we can simplify the expression to say that our algorithm takes $O(QrN(k + logA))$ operations.

Space Complexity: During the run of the algorithm, we have to store the number of overlapping spans at each data unit, which will require $O(N)$ storage. We will also have to store a heap of access requirements, which can be stored using $O(A)$ space. We also have a list of access requirements and that information will take up $O(A)$ space. In total we thus have $O(A + N)$ storage space used during the run of the algorithm.

5 Experimental Results

Figure 2: *City model: 110 million triangles, 6 GB*

Experiment context: In order to implement our algorithm, we used a workstation that is a Dell T5400 PC with Intel (R) Core (TM) 2 Quad and $8GB$ main memory. The hard drive is a 1TB Seagate Barracuda with 7200 RPM and the graphics card is an nVIDIA Geforce GTX 260 with 896 MB GPU memory. The data rate of the hard drive is 120 MB/s and the seek time is a minimum of 2 ms per disk seek.

Benchmarks: We use three models to perform our experiments, each model represents a use case or scenario. The City model (Figure 2) is a regular model that can be used in a navigation

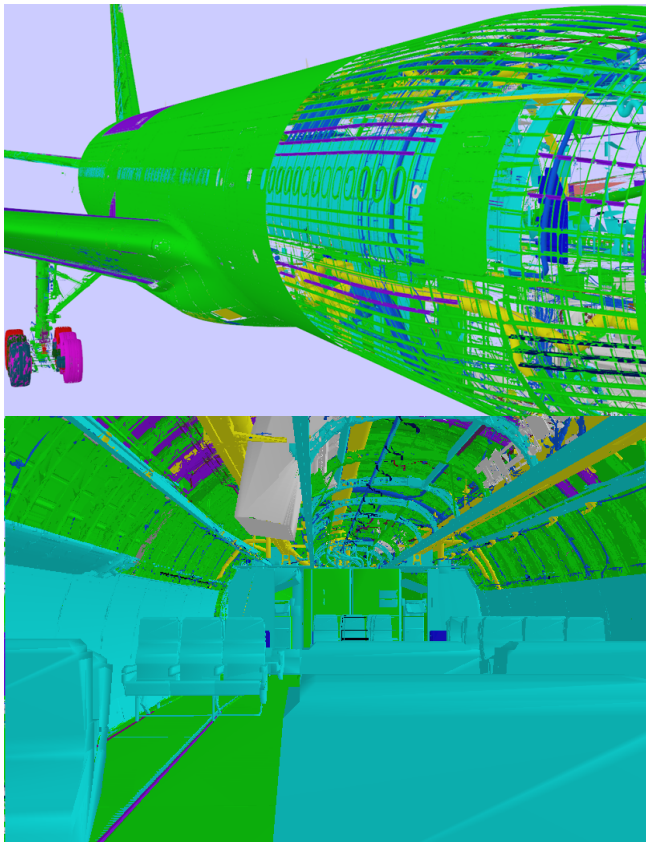

Figure 3: *Boeing model: 350 million triangles, 20 GB. Overview of model (top) and model detail (bottom)*

Figure 4: *Urban model: 100 million triangles, 12 GB*

simulation application or virtual reality walkthough. The Boeing model (Figure 3), on the other hand, represents scientific or engineering visualization applications. The Urban model (Figure 4) has texture attached to it, which is commonly used in games. By comparing performance of cache-oblivious layout without redundancy [Yoon et al. 2005] to our method using redundancy on these three models, our goal is to show that the redundancy based approach can achieve more stable and generally better performance on different real time applications.

To apply our method on these large-scale models, we had to find a proper set of access requirements. In general, that question is deep enough that it can be discussed as a separate research topic. Here, however, we had good performance using only simple schemes for creating access requirements. Each model ended up having a separate scheme. Nonetheless, an access requirement represents a set of data that is highly likely to be accessed together.

For the City model, a 2D grid is used to divide the space into square cells. For each pair of adjacent cells, the difference of the data is considered as an access requirement. By propagating this rule, access requirements are created, and the number of them is determined by the resolution of the grid [Jiang et al. 2013].

For the Boeing model, the predefined objects are used as a conceptual level to create access requirements. Samples of view positions are distributed across the model. For each sample, four fixed directions and four random directions are considered. Objects visible from this position in any one of these eight directions are added to access requirement for this specific sample. The density of these samples depends on the complexity of local occulders to reduce load of each access requirement, i.e. more samples are distributed to places with more complex geometry.

The Urban model is different from the previous two in a way that it involves textures. Building heavy redundancy of textures increases the total size of the dataset significantly, while keeping textures away from redundancy leads to inevitable long seek time, which is completely against the philosophy of this work. To solve this problem, we applied a spatial Lloyds clustering [Lloyd 1982] on objects. By moving centers of clusters, we look for a solution such that each cluster involves almost same amount of texture data. Between clusters, textures can be redundantly stored, but within each cluster, texture data are stored uniquely. In this way, each cluster is used as an access requirement.

The computation time to create redundancy layout is generally linearly correlated to the final redundancy factor for each model. For the examples we used in Figure 5, it took 16 minutes to create the redundancy layout from the cache-oblivious layout without redundancy for the City model. This number is 80 minutes and 38 minutes for the Boeing model and the Urban model respectively.

Results: Figure 5 shows the results of delays caused by fetching data on the experimental models we used. We compare the results of a cache-oblivious layout without redundancy and one with redundancy. For the layout with redundancy, we set the redundancy factor equal to 4.2. This factor was chosen because as can be seen in Figure 6, it had considerably better performance than lower redundancy factors and did not have significantly worse performance than higher redundancy factors. A factor of 4.2 is also still practical, as the largest model we tested, the Boeing model, becomes 84 GB which is still acceptable given the large capacity of modern secondary storage devices.

It is clear that the performance of the layout with redundancy has generally shorter delays than the cache-oblivious layout without redundancy. As can be observed from the results, although the layout with redundancy does not eliminate delays for most of sample points on the walkthrough path, it reduces delays to a small range and keeps the performance more consistent. This is the benefit we get from using our algorithm which adds redundancy. Since the algorithm tends to eliminate seeks with longer seek time first, in practice the larger delays are avoided.

When we compare our method to the one in [Jiang et al. 2014], there is a performance benefit that can be observed in Figure 7. Additionally in [Jiang et al. 2014], the user does not have any control over the final redundancy factor however each time we duplicate one data unit, we can halt it if the redundancy factor reaches a certain threshold. This helps us create data layouts with arbitrary redundancy factors. We use this fact to test different redundancy factors and see their results. In Figure 6, we show the results of using layouts with redundancy factors that range from 1.0 to 10.0.

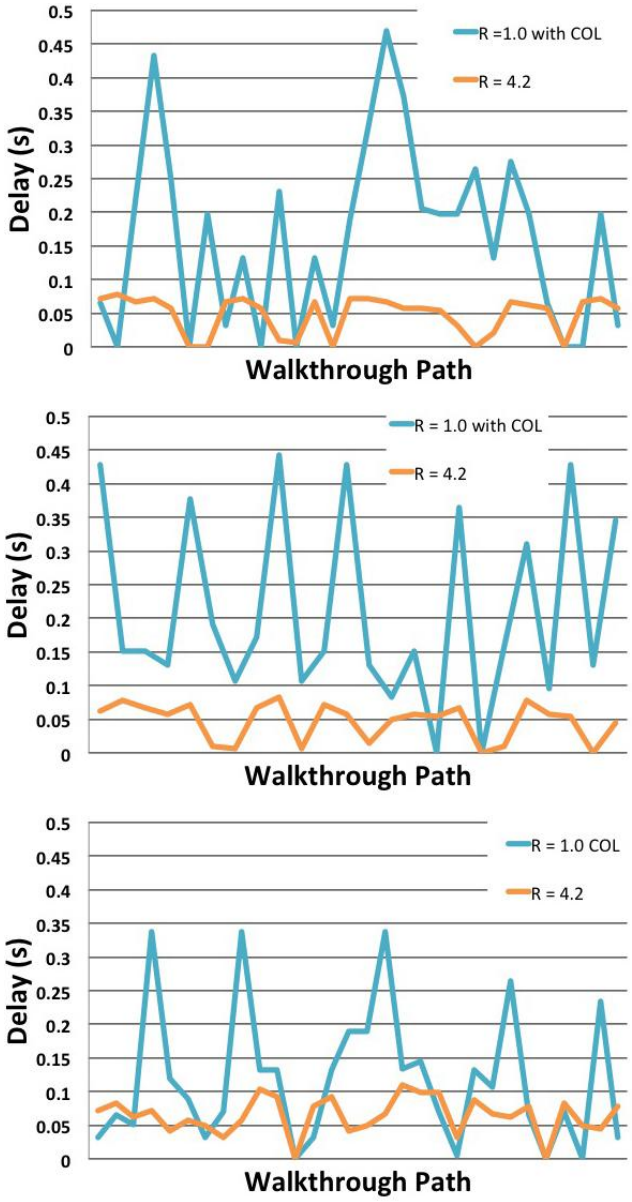

Figure 5: *Statistics of delays caused by the fetching processes for the City model (top), the Boeing model (center), and the Urban model (bottom), with and without redundancy.*

The y-axis in this figure is the ratio of the estimated seek time (EST) of the layout with redundancy over the EST of the layout without redundancy. This value starts at 1.0 where redundancy factor is 1.0, meaning no redundancy, and decreases as redundancy factor goes larger. We can see that the rate of this decrement is not constant, and the benefits we gain at beginning are larger than the ones we get later. This implies that most of the performance improvement resides at the earlier phase of raising redundancy factor. This implies that it is worthwhile to limit the redundancy factor used because after a certain point you are using much more secondary storage space without improving seek time by much. It also implies that our algorithm dramatically reduces seek time in practice by using only small redundancy factors.

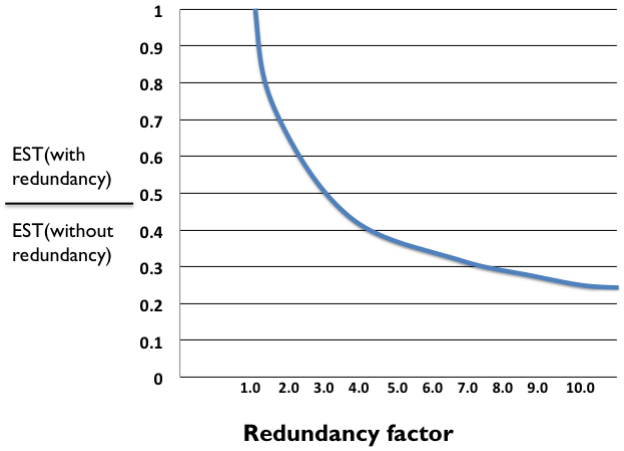

Figure 6: *Plot of the ratio of the EST of layout with redundancy over the EST of cache-oblivious mesh layout without redundancy.*

6 Analysis and Comparisons over Prior methods

In the algorithm, we make a list of data units that will reduce seek time by just moving instead of copying them. We perform these moves first before working with data units that need copying. This initial step will produce a better solution than proposed by [Yoon et al. 2005] without adding extra units. This is because our algorithm will consider cases where data units are close to each other but in hierarchically different blocks in [Yoon et al. 2005]. To show this, consider a case where we have two access requirements of 5 data units each. Figure 8 shows an example of that kind of layout. In the middle of that figure is the result of using the cache oblivious layout. Because hierarchically it would arrange the units in each block and then arrange the blocks, it does not detect that the units with the black access requirement can be grouped together. On the other hand, the algorithm we propose would shorten the black access requirements without adding redundancy, as shown in the bottom of that figure.

The algorithm in [Yoon et al. 2005] did not necessarily produce the best cache oblivious mesh layout. However, even if we had the best layout without redundancy, we would actually achieve a better seek time than it using redundancy. We have such an example with Figure 9. As can be seen in the figure, the total seek time is 7 units which turns out to be the minimum possible seek time without redundancy, as found through a brute-force search. With redundancy, the total seek time is the minimum required which is 6 units. While a reduction from 7 to 6 units may not seem dramatic, when this result is scaled up to the hundreds of millions, this makes a big difference in seek time, which we saw in practice.

7 Conclusion and Future Work

Given the data units, access requirements, and the desired upper bound on redundancy factor, we have proposed an algorithm that would create a cache oblivious layout with the primary goal of reducing the seek time through duplicating the data units. We proposed a cost model for estimating the seek time, and in our algorithm we choose and copy data units in appropriate locations such that it reduces the estimated seek time. We have shown that such a layout significantly improves both the performance and consistency of interactivity in massive model walkthrough

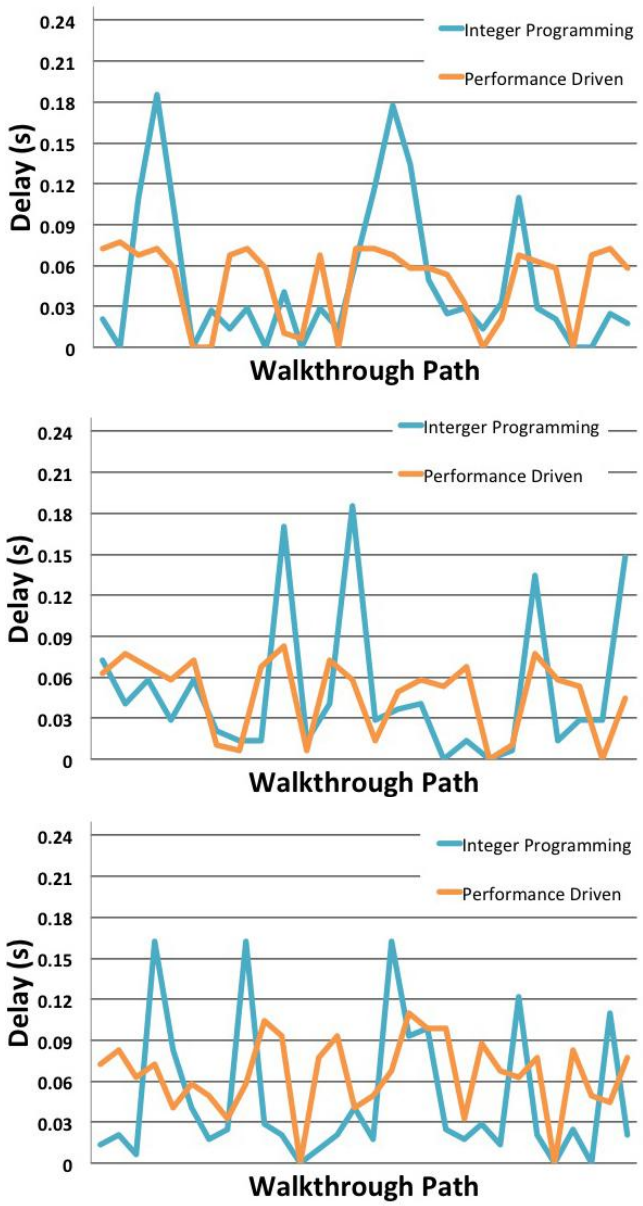

Figure 7: *Statistics of delays caused by the fetching processes for the City model (top), the Boeing model (center), and the Urban model (bottom), using integer programming and our method*

applications.

Our proposed redundant storage of data may limit editing and modification of data because the data has to be modified at all copies. However, we foresee no problem in recomputing and updating the layout due to this modification using our algorithm since every iteration in our algorithm just assumes a layout and improves on it. After data modification, we can delete/modify the relevant data units, update the access pattern and run a few iterations of our algorithm to get a better layout. In other words, our algorithm is incremental and can be used for dynamic data sets also which might be a result of scene editing and modification.

Our model for estimated seek time assumes equal usage of

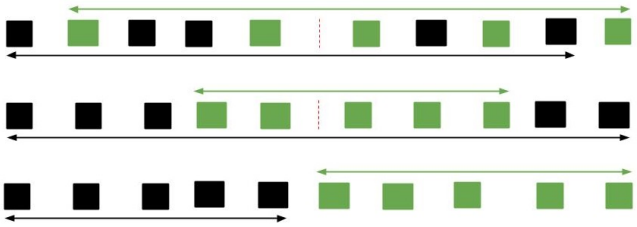

Figure 8: *Example of two access requirements of 5 data units each. The red line represents the boundary between blocks in the cache oblivious layout hierarchy. The original layout (top), cache-oblvious layout (middle), as well as the layout after running our algorithm (bottom) is shown.*

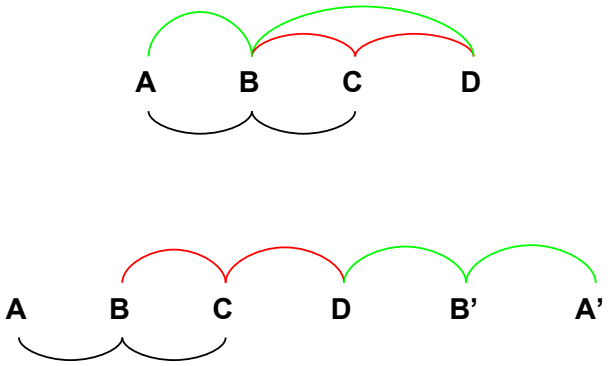

Figure 9: *Data Units with varying access requirements on the top. It is laid out in its optimal layout without redundancy there. Its optimal layout with redundancy is shown at the bottom*

all access requirements. However, if we are given the information about usage statistics of each of the access requirements, the model can prioritize reduction of seek time of frequently used access requirements. Further, the set of data units that belong to an access requirement can also be refined to get better results. This can be done by checking the usage history of an application and group data units together if they are accessed together with high probability.

References

AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. 2008. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, 57–70.

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. 1999. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 199–206.

DOMINGO, J. S. 2014. Ssd vs. hdd: What's the difference. *PC Magazine* (February).

HOPPE, H. 1996. Progressive meshes. In *Proceedings SIGGRAPH*, 99–108.

HOPPE, H. 1997. View-dependent refinement of progressive meshes. In *SIGGRAPH*, 189–198.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization*, 35–42.

JIANG, S., SAJADI, B., , AND GOPI, M. 2013. Single-seek data layout for walkthrough applications. *SIBGRAPI 2013*.

JIANG, S., SAJADI, B., IHLER, A., AND GOPI, M. 2014. Optimizing redundant-data clustering for interactive walkthrough applications. *CGI 2014*.

LINDSTROM, P., AND TURK, G. 1999. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics 5*, 2 (April-June), 98–115.

LLOYD, S. 1982. Least squares quantization in pcm. *Information Theory, IEEE Transactions on 28*, 2 (Mar), 129–137.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2002. *Level of Detail for 3D Graphics*. Morgan-Kaufmann.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ACM, SIGMOD '88, 109–116.

RIZVI, S., AND CHUNG, T.-S. 2010. Flash ssd vs hdd: High performance oriented modern embedded and multimedia storage systems. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, vol. 7, V7–297–V7–299.

SAJADI, B., JIANG, S., HEO, J., YOON, S., AND GOPI, M. 2011. Data management for ssds for large-scale interactive graphics applications. In *I3D '11 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, 175–182.

SAXENA, M., AND SWIFT, M. M. 2009. Flashvm: Revisiting the virtual memory hierarchy. In *Proc. of USENIX HotOS-XII*.

SHAFFER, E., AND GARLAND, M. 2001. Efficient adaptive simplification of massive meshes. In *Proc. IEEE Visualization*, Computer Society Press, 127–134.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization Course Notes*.

VARADHAN, G., AND MANOCHA, D. 2002. Out-of-core rendering of massive geometric datasets. In *Proceedings IEEE Visualization 2002*, Computer Society Press, 69–76.

YOON, S.-E., AND LINDSTROM, P. 2006. Mesh layouts for block-based caches. *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization) 12*, 5, 1213–1220.

YOON, S., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache oblivious mesh layouts. *ACM SIGGGRAPH 2005*.

A Linear Search Justification

In order to find the best place to copy a data unit, we perform a linear search within the span of the access requirement for location with the largest benefit. If k is the span of the access requirement, then the linear search takes $O(k)$ query time. Updates will also be

$O(k)$ time. Construction of the list of data units where each data unit stores the number of overlapping access requirements will be $O(N)$. There are other approaches, such as a range tree or dynamic programming, that may produce better query times, but their construction and update times will be worse as well as their storage.

With dynamic programming, we would have to maintain a matrix where an entry (i, j) would contain the minimum value in that range. This would give us a $O(1)$ query time but the construction and storage would be $O(N^2)$ where N is the number of data units. The update time would be $O(N)$ when we add a data unit. Since the N for this problem domain is in the hundreds of millions, that is an unacceptable storage bound. The construction run time would also be prohibitive given the magnitude of our input.

We could use a range tree. The initial binary search tree would be sorted by index and at each entry would be a pointer to a binary search tree sorted by value. If we put the min value at each of the nodes of the initial tree, we can speed up our queries. We would get a $O(logN)$ query time, but our construction time and storage would be $O(NlogN)$. Updating the data structure would take at a minimum $O(klog(N))$ time if we do careful indexing and only update the nodes that need to be updated. If we have a large access requirement, then this would represent a significant improvement in query time. Given our exceptionally large input, however, the construction, storage, and update bounds are too prohibitive.

As it turns out, the common data structures that would be used for the finding the minimum value in an arbitrary part of a list are not practical for our purposes. Thus, while a simple linear search may seem inefficient at first, as it turns out it is the best option given our constraints.