CGI2015 manuscript No.
(will be inserted by the editor)

Performance Driven Redundancy Optimization of Data Layouts for Walkthrough Applications

Jia Chen · Shan Jiang · Zachary Destefano · Sungeui Yoon · M. Gopi

Abstract Performance of interactive graphics walkthrough systems depends on the time taken to fetch the required data from the secondary storage to 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 elegant 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. Unlike most existing methods, our data layout method can work with models with textures. The interactive rendering speed of the walkthrough system was improved by a factor of 2-4 by using our data layout method when compared to existing methods with or without redundancy.

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

Jia Chen
University of California, Irvine

Shan Jiang
Altair Engineering, Inc.

Zachary Destefano
University of California, Irvine

Sungeui Yoon
Korea Advanced Institute of Sci. and Tech.

M. Gopi
University of California, Irvine

Fig. 1 Urban model: 100 million triangles, 12 GB with textures. Using our redundancy based data layout method the walkthrough rendering speed for this model was improved by a factor of 2 over existing methods.

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 [13]. 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 [1]. An earlier work utilized this difference between SSD and HDD, and designed a data layout tailored for using SSDs with the walkthrough application [15]. There have been many other techniques utilizing SSDs for various applications [16]. 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 [13]. 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 [13], mainly because of its stability and low cost per unit. As an example, according to [3], 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.

There are generally two types of disk-based secondary storage devices. For devices with constant linear velocity (CLV), for example, Blu-ray, the seek speed is linearly dependent on the seek distance, the physical distance between data units. For devices with constant angular velocity (CAV), such as modern CDs and DVDs, most of the data is stored along the rim to enable faster seek time, so we can assume the seek speed is almost linear which respect to seek distance. In both cases, minimizing seek distance generally produces a data layout that will minimize seek time.

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 [12]. Redundancy based data layouts to reduce the seek time in walkthrough applications were introduced in a recent work [7], in which the number of seeks for every access was reduced to at most one unit. However, in order to achieve this nice 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 [8] 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. For example, accessing two adjacent data blocks is considered as two different access, and the information about the intersection data between two different access patterns is not used in any way to minimize redundancy.

Main contributions: In this paper, we propose a cost model for seek time based on the actual number of data units between the requested data units in the linear data layout. Using this model, and given the data access requirements for a walkthrough application, we develop an algorithm to duplicate data units 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 maximum redundancy case (a layout where seek time is at most one) and the no-redundancy case (a simple cache oblivious mesh layout with a potentially high seek time), and in practical applications, a reasonable redundancy factor can be determined adaptively to achieve high performance improvement with low redundancy cost. Although [7] can also generate the single seek layout, our single seek layout has a substantially lower redundancy factor than that of [7] because, while [7] just repeats the data required for different access requirements, we also consider the common blocks between the access patterns and place them strategically without duplication, thus we can achieve a single seek layout with much lower redundancy. We show that the implementation of our algorithm significantly reduces average delay and the maximum 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 main memory, but fully utilizing 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 [11], progressive level of detail [6,5,4,17], and image based simplification [2]. Soon thereafter the size of main memory became the bottleneck in handling ever increasing sizes of the model. Hence memory-less simplification techniques [9] and other out-of-core rendering systems [18,19,14] emerged in which just the limited amount of required data that needs to be processed and rendered was brought from the secondary storage to main memory.

The speed at which this data could be brought from the secondary to 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 main memory and thus reduce the number of times the disk read is initiated. This meant that subsequent works focused on cache aware [15] and cache oblivious data layouts [20,21] 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 [12,7,8] as potential solutions to the problem of reducing seek time. In particular [8] presented a data layout algorithm based on integer programming specifically useful in walkthrough applications that models the seek time as the number of seeks to the beginning of different data groups. These data groups are the ones to be fetched to render one frame. However, there were major drawbacks. First, although it provides the data units to be grouped and considered as one seek, for each seek it does not provide a data layout. This is because it does not relate one data group with another. Such an approach could easily result in unnecessary data block duplications since groups of data units can overlap with each other and only one copy of the common data unit may be required. There is no mechanism in the integer programming solver to detect whether this redundancy is necessary because of some scene context or simply created blindly due to local optimization. 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 [8], seek time is simplistically modeled, as number of data groups accessed for each fetch independent of the number of data units between these data groups. 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.

Fig. 2 City model: 110 million triangles, 6 GB

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 located spatially closely (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.*

There can be many different ways of defining access requirements and data units. One simple choice is to introduce a concept of navigation space for the walkthrough application. The navigation space in the walkthrough scene, which defines the space of all possible view points, can be partitioned into cells, and all the data units required by each of the viewpoints within a cell is grouped together to define one access requirement. Thus the number of cell 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.

Suppose that we have a linear ordering of data units that may eventually be the order in which they are stored in the hard drive. Given an access requirement A, the total span of A is the total number of data units between the first and last data units that are used by 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 3 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

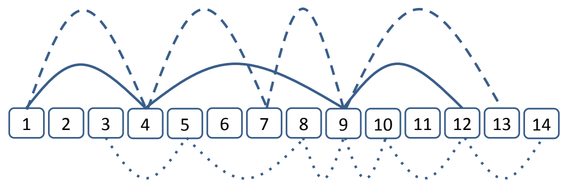

Fig. 3 Illustration of a linear order of data units and three example access requirements. Lines connect data blocks that belong to the same access requirement. The span of the access requirement shown in the solid line is 11.

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 3, data units 1, 4 and 12 are part of two access requirements and data unit 9 is part of all three.

3.2 Seek Time Measure

Given a linear order of data units and the access requirements, 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 can be used as a measure of seek time - time taken to seek the last data unit starting from the first data unit. It is interesting to note that [20] 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. In this aspect, using the span to measure the seek time is justified too. In the following measure of total seek time, we use a relative probabilistic measure to include the frequency of use of each access requirement. Let I be the set of access requirements and A_i represent the span of the access requirement i. Let p_i be the probability that A_i will be used during rendering. We now define Estimated Seek Time (EST) as:

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

In this paper, we assume all access requirements are equally likely to be used thus all p_i values will be the same. We will use this to simplify the above equation, by ignoring the common scale factor, to the following for our purposes.

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

It is important to note that the same measure can be used to describe the data transfer time. As mentioned earlier, whether the data between two required data units is read or skipped, the time taken to go from the first to the last required data unit is a measure of the delay caused by the disk. If all the intermediate data in the span is read, this time will be a measure of data transfer time, and if it is skipped, it is a measure of seek time. In other words, this measure also defines very well the total data fetch time, which is the sum of data seek and transfer times. However, in this paper, we assume that only the required data is read and use this measure to quantify seek time.

The seek time is also measured in other works [8,7] as number of seeks and not parameterized using the distance between the required data units. In this work, we model seek time as the distance between the data units and optimize this measure. Using this measure, we show better performance than earlier works.

If we reduce the total EST in our optimization, then the average estimated seek time will be reduced. During optimization, we first choose and process the access requirement with the maximum span. As a result, we not only reduce the average span, but also the maximum span, and hence the standard deviation in spans. This will in turn have an effect of providing consistent rendering performance with low data fetch delays as well as consistently small variation between such delays during rendering.

3.3 Algorithm Overview

Given the access requirements and the data units, the goal of our algorithm is to compute a data layout that reduces the EST. In [20], the only allowed operation on the data units is the move operation and the optimal layout 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, we want to minimize EST while also keeping the number of redundant copies as low as possible. Given an initial ordering of data units, we copy one data unit to another location. We reassign one or more of the access requirements that use the old copy of the data unit to the new copy making sure 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. In our implementation, we consider a group of data units as one data unit if they are adjacent to each other in the current layout and are always accessed together by any potential access requirement.

Location to Copy: Based on the above observation, given an access requirement, we can possibly move the beginning or the end data units to a position that will reduce the span of the access requirement. This operation will reduce the span of a specific access requirement, however, if the new location of the data unit is in the span of other access requirements, such as location 11 in Fig. 3, it increases the span of each of those accesses (all those three access requirements in Fig. 3) 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. If copying of the data block is the chosen operation, 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 other 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

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

If copying of the data block is the chosen operation, 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

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

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 used 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

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

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. If moving of the data block is the chosen operation, we find the location j where the start or end data unit of the access requirement i needs to be moved using a simple linear search through the span of i as

$$argmax_j(\Delta EST_M(i,j)).$$

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 except it will organize the copy operations and consist of the values of $\Delta EST_C(i,j)$.

```
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 $\Delta EST_M(i,j)$ and insert into $E_M$ ;
    Calculate $\Delta EST_C(i,j)$ 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

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 that is 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 pseudo-code of this algorithm is shown in Algorithm 1.

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 m as the average span of a single access requirement. Let r be the redundancy factor limit specified by the user so that $O(rN)$ units can be copied. For the sake of analysis each data unit will be used by $O(A)$ access requirements and at each location there will be $O(A)$ access requirements whose span overlaps it.

Time Complexity: The construction of the heaps E_M and E_C involves computing the benefit information for all A access requirements and inserting each one into the heap. For a single access requirement, computing the benefit information of moving or copying one of its data units involves scanning each data unit in its span. This approach takes $O(m)$ operations. Calculating $\sum_{s\in S}\Delta A_s$ and $\sum_{t\in T}\Delta A_t$ will take $O(A)$ operations since there are $O(A)$ access requirements to potentially have to sum over. Inserting this benefit information into the heap takes $O(log(A))$ operations. In total then it takes $O(m+A+logA)$ or $O(m+A)$ operations per access requirement to get the benefit information. The initial construction thus takes $O(A(m+A))$ operations.

After the initial construction, the move and copy loops are executed. In every iteration of move or copy, 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. There are potentially $O(A)$ overlapping access requirements whose benefit information needs to be recalculated. As shown above, for each of these access requirements $O(m+A)$ operations are required to perform the recalculation and update the heap. Each iteration of move or copy thus takes a total of $O(A(m+A))$ operations.

For simplicity we will assume that the move loop runs $O(N)$ times total. There are $O(rN)$ copies made so there are that many iterations of the copy loop. We thus can assert that there are $O(rN+N)$ iterations of the move or copy loops. We can simplify this to $O(rN)$ operations since $r\geq 1$. In total then the moving and copying loops will take $O(rNA(m+A))$ operations, which is also the running time for the whole algorithm.

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

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.

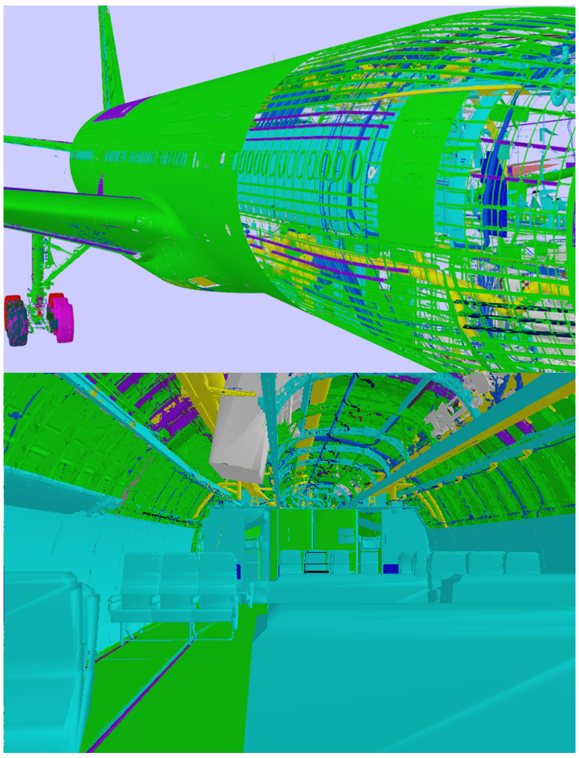

Fig. 4 Boeing model: 350 million triangles, 20 GB. Overview of model (top) and model detail (bottom).

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 simulation application or virtual reality walkthrough. The Boeing model (Figure 4), on the other hand, represents scientific or engineering visualization applications. The Urban model (Figure 1) has texture attached to it, which is commonly used in games. By comparing performance of cache-oblivious layout without redundancy [20] 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. An access requirement represents a set of data that is highly likely to be accessed together. 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.

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 the access requirement for this specific sample. The density of these samples depends on the complexity of local occluders to reduce load of each access requirement, i.e. more samples are distributed to places with more complex geometry.

For the City model, a 2D grid is used to divide the space into square cells. The difference of the data between each pair of adjacent cells 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 [7].

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 [10] 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.

5.1 Results and Comparison with Prior Methods

In Figure 5, we show the results of using layouts with redundancy factors that range from 1.0 to 5.0. 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. The rate of decrease is exponential with larger benefits in the beginning meaning increasing the redundancy factor there reduces the seek time much more significantly than increasing the redundancy factor later. In other words, it is worthwhile to limit the redundancy factor used because after a certain point the cost of using redundancy is very high – much more secondary storage space will be used without any significant improvement in seek time. It also implies that our algorithm dramatically reduces seek time in practice by using only small redundancy factors. The same phenomena can be observed in Figure 6 . Figure 6 compares the time per frame statistics of a cache-oblivious layout

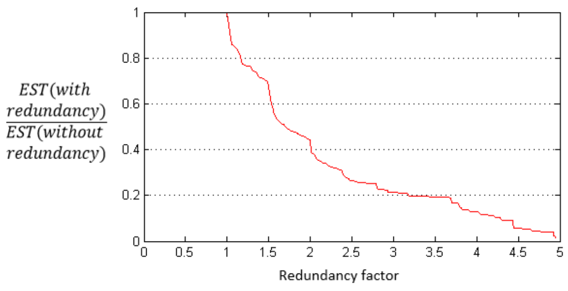

Fig. 5 Plot of the ratio of the EST of the layout with redundancy over the EST of the cache-oblivious mesh layout without redundancy for the City model. The ratio decreases exponentially along with the increment of redundancy factor. Based on the stopping threshold we are using in the experiments, the algorithm stops at R=2.5. Redundancy factors for the other models are determined in the same way.

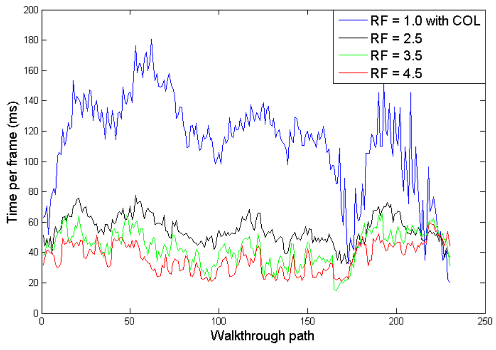

Fig. 6 Statistics of time per frame (averaged per 10 frames) for the city model, with and without redundancy. COL indicates a Cache-Oblivious Layout that does not use any redundancy. RF indicates the redundancy factor.

without redundancy [20] and data layouts computed using the proposed method with redundancy factor 2.5, 3.5, 4.5 respectively. From the figure, the overall rendering performance is improved by a factor between 2 to 4, and apparently the performance difference between redundancy factor 1.0 and 2.5 is much larger than the difference between 2.5 and 4.5. Based on the observations above, we adopt an adaptive method to determine the final redundancy factor in our experiment: if there is no upper bound specified by user, the algorithm will stop optimizing when the decrease speed of ETS is lower than a predefined threshold.

Figure 7 shows the comparison of our method with the one proposed in [8]. When we do the comparison there is again a performance benefit and less redundancy required. While the redundancy factor used for the linear programming method was 8.3 which was the redundancy that produced the best performance with that method, the final redundancy factors in our method are all less than 3.0. The graphs clearly show that our

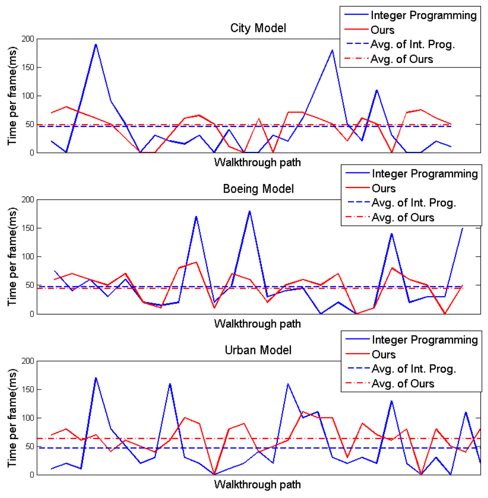

Fig. 7 Statistics of time per frame (averaged per 200 frames) for the City model (top), the Boeing model (center), and the Urban model (bottom), using integer programming (redundancy factor = 8.3) and our method (redundancy factor < 3.0).

method significantly reduces the maximum delay with at most a third of the redundancy factor when compared to [8] for all the three models. Reduction of maximum delay is the key for consistency in interactivity. Additionally in [8], the user does not have any control over the final redundancy factor however in our proposed method, each time we duplicate one data unit, we can halt it if the redundancy factor reaches a certain threshold. This helps us to create data layouts with arbitrary redundancy factors.

6 Cache Oblivious Layout With and Without Redundancy

In the algorithm, we make a heap 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 [20] without adding redundant units. This result is possible mainly because our optimization algorithm searches wider sets of potential locations for moving cases in an efficient manner. 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 it hierarchically constructs blocks and arranges the units in each block, 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 [20] did not necessarily produce the best cache oblivious layout. However, even if we had the best layout without redundancy, we would actually achieve a better seek time using redundancy. We show 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.

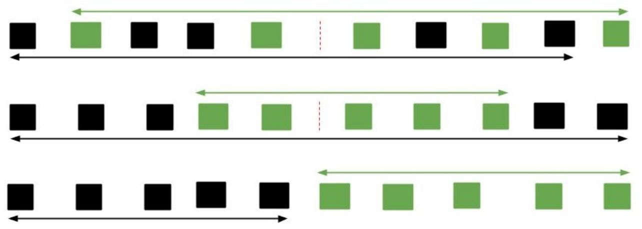

Fig. 8 Example of two access requirements of 5 data units each. The red line represents the boundary between blocks in the cache oblivious layout hierarchy. The original layout (top), cache-oblvious layout (middle), as well as the layout after running our algorithm (bottom) is shown.

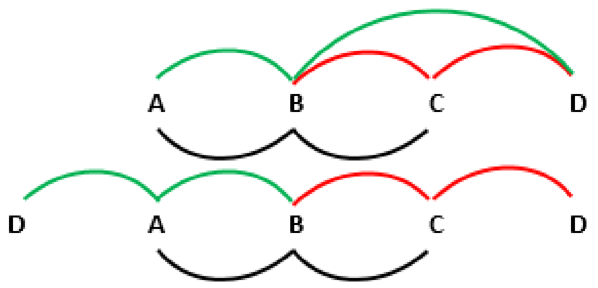

Fig. 9 Data Units with varying access requirements on the top. The letters represent data units and each color represents a different access requirement. It is laid out in its optimal layout without redundancy on top. Its optimal layout with redundancy is shown at the bottom.

7 Limitations

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, which also might be a result of scene editing and modification.

Our cost model does not consider distance between access requirements. We only take into account distance between data units in the same access requirement and we do not consider the seek time between access requirements. If we do take this account in our model, then if we are given information as to which access requirement is more likely to be used before or after another access requirement we would have an even more accurate model for seek time.

8 Conclusion

We have proposed an algorithm that creates a cache oblivious layout with the primary goal of reducing the seek time through duplicating some of the data units. We proposed a cost model for estimating the seek time of a data layout, and we move or copy data units to appropriate locations such that it reduces the estimated seek time. Given an arbitrary data layout, our algorithm can generate a family of data layouts, which covers data layouts between the maximum redundancy case and no-redundancy case. By considering the data units shared by access requirements, our algorithm achieves single seek layout with about one third of the redundancy factor in [7], and due to the efficient data structure we applied, preprocessing time for our algorithm is significantly less than previous methods. Unlike [8] and [7], our algorithm enables direct control on the redundancy factor: the redundancy factor can either be constrained by user specified bounds or determined adaptively, to achieve high performance improvement with low redundancy cost.

Acknowledgements

This paper is partly supported by NSF CAREER IIS-0846144. And Yoon is partly supported by the StarLab project of MSIP/IITP and NRF-2013R1A1A2058052.

References

1. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M., Panigrahy, R.: Design tradeoffs for ssd performance. In: ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 57–70. USENIX Association, Berkeley, CA, USA (2008)
2. Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., Manocha, D.: MMR: An interactive massive model rendering system using geometric and image-based acceleration. In: Proceedings Symposium on Interactive 3D Graphics, pp. 199–206. ACM SIGGRAPH (1999)
3. Domingo, J.S.: Ssd vs. hdd: What's the difference. PC Magazine (2014). URL http://www.pcmag.com/article2/0,2817,2404260,00.asp
4. Hoppe, H.: Progressive meshes. In: Proceedings SIGGRAPH, pp. 99–108 (1996)
5. Hoppe, H.: View-dependent refinement of progressive meshes. In: SIGGRAPH, pp. 189–198 (1997)
6. Hoppe, H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In: Proceedings IEEE Visualization, pp. 35–42 (1998)
7. Jiang, S., Sajadi, B., , Gopi, M.: Single-seek data layout for walkthrough applications. SIBGRAPI 2013 (2013)
8. Jiang, S., Sajadi, B., Ihler, A., Gopi, M.: Optimizing redundant-data clustering for interactive walkthrough applications. CGI 2014 (2014)
9. Lindstrom, P., Turk, G.: Evaluation of memoryless simplification. IEEE Transactions on Visualization and Computer Graphics **5**(2), 98–115 (1999)
10. Lloyd, S.: Least squares quantization in pcm. Information Theory, IEEE Transactions on **28**(2), 129–137 (1982). DOI 10.1109/TIT.1982.1056489
11. Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., Huebner, R.: Level of Detail for 3D Graphics. Morgan-Kaufmann (2002)
12. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, SIGMOD '88, pp. 109–116. ACM (1988)
13. Rizvi, S., Chung, T.S.: 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, pp. V7–297–V7–299 (2010)
14. Sajadi, B., Huang, Y., Diaz-Gutierrez, P., Yoon, S.E., Gopi, M.: A novel page-based data structure for interactive walkthroughs. In: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, I3D '09, pp. 23–29. ACM, New York, NY, USA (2009)
15. Sajadi, B., Jiang, S., Heo, J., Yoon, S., Gopi, M.: Data management for ssds for large-scale interactive graphics applications. In: I3D '11 Symposium on Interactive 3D Graphics and Games, pp. 175–182. ACM, New York, NY (2011)
16. Saxena, M., Swift, M.M.: Flashvm: Revisiting the virtual memory hierarchy. In: Proc. of USENIX HotOS-XII (2009)
17. Shaffer, E., Garland, M.: Efficient adaptive simplification of massive meshes. In: Proc. IEEE Visualization, pp. 127–134. Computer Society Press (2001)
18. Silva, C., Chiang, Y.J., Correa, W., El-Sana, J., Lindstrom, P.: Out-of-core algorithms for scientific visualization and computer graphics. In: IEEE Visualization Course Notes (2002)
19. Varadhan, G., Manocha, D.: Out-of-core rendering of massive geometric datasets. In: Proceedings IEEE Visualization 2002, pp. 69–76. Computer Society Press (2002)
20. Yoon, S., Lindstrom, P., Pascucci, V., Manocha, D.: Cache oblivious mesh layouts. ACM SIGGGRAPH 2005 (2005)
21. Yoon, S.E., Lindstrom, P.: Mesh layouts for block-based caches. IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization) **12**(5), 1213–1220 (2006)