Vis Comput
DOI 10.1007/s00371-015-1165-0

ORIGINAL ARTICLE

Optimally Redundant, Seek-Time Minimizing Data Layout for Interactive Rendering

Jia Chen[1] · **Shan Jiang**[1] · **Zachary Destefano**[1] · **Sungeui Yoon**[2] · **M. Gopi**[1]

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. By using a set of training access requirements and a set of validation access requirements, our proposed method is able to automatically maximize system performance with an optimal redundancy factor. Experimental results show that 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
jiac5@uci.edu

[1] University of California, Irvine, USA

[2] Korea Advanced Institute of Science and Technology, Daejeon, South Korea

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 various 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 in the walkthrough applications [16]. There have been many other techniques utilizing SSDs for various applications [14]. SSD, unfortunately, is not a 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 in the foreseeable future [13], mainly because of its stability and

Published online: 05 November 2015

low cost per unit. As an example, according to [3], as of 2014, an HDD can cost $0.08 per GB, while an SDD costs $0.60 per GB. As most of walkthrough applications are still using HDDs, optimizing components of walkthrough systems with HDDs is critical. In particular, addressing the seek time, the main bottleneck of accessing data from HDDs remains a 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 with 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 [11]. Redundancy-based data layouts for walkthrough applications to reduce the seek time 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. Nevertheless, it gave an upper bound on performance improvement using redundancy.

Another recent work of ours [8], again for the walkthrough application, 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. The advantage of this approach was to directly model the final performance goal (frames per second), and implicitly derive the redundancy factor from this model. 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 distance-based model for measuring expected seek time. Using this model, we adopt an idea from Monte Carlo simulation: the performance on a large number of random walkthrough access requirements reflects the average performance for all possible walkthrough access requirements, and thus optimizing performance on random walkthrough access requirement also improves overall performance on all possible walkthrough operations. Based on this idea, the original problem is converted into "given a number of access requirements, optimize the data layout to achieve optimal performance on these access requirements". Thus, we develop an algorithm to duplicate data units strategically to maximize the reduction in the expected seek time, while keeping the redundancy factor within the user defined bound. We will show that our greedy solution can generate a series of data layouts between the two extreme cases of data layout with redundancy, namely the maximum redundancy case (a layout where expected seek time is at most one) and the no-redundancy case (a simple cache oblivious mesh layout with a potentially high seek time). To avoid the generated data layout being overfitted on the given training access requirement set, another set of randomly generated access requirements play a role as validation set to determine a proper redundancy ratio. Our experiment results 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 computing capacity of the graphics cards. Hence these works focused on providing solutions to reduce the number of primitives to be rendered while maintaining the visual fidelity. These solutions included level-of-detail for geometric models [10], progressive level of detail [4–6,12,15], and image-based simplification [2]. Soon thereafter the size of main memory became the bottleneck in handling ever increasing sizes of the models. Hence memory-less simplification techniques [9] and other out-of-core rendering systems [17,18] 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 [16] and cache oblivious data layouts [19,20] 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 [7,8,11] 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.

3 System overview

3.1 Definitions

Data units 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 4 KB) 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. Therefore, all of our operations will be on the data unit level, rather than on each byte.

Data layout Data layout is a specific sequence of data units. In this paper, it is also used to refer to a map between the original positions of data units in 3D model file and their positions after processing. As original model file may be extremely large, it is impossible to frequently modify the original file. Therefore, all the operations introduced in the later parts of this paper are made on this map, which not only allows fast operation without modifying the original file, but also allows our algorithm to store its intermediate results with a small cost of storage. The original file will be modified only in the final step after optimal mapping is found.

Access requirements When rendering a frame, the data units required to fetch from disk is defined as an *access requirement*. In other words, an access requirement is the set of additional data units, which are required by the current frame but not by the previous frame depending on the motion of the user's viewpoint through the scene. The relative ordering of data units within an access requirement or the order in which those data units are fetched does not matter since all the data units that are used by an access requirement are going to be eventually fetched.

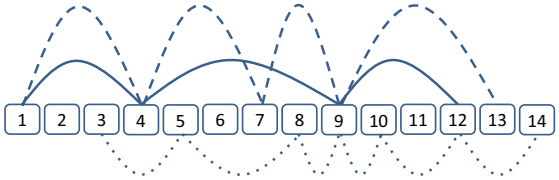


Fig. 1 Illustration of a linear order of data units and three example access requirements. *Lines* connect data blocks that belong to the same access requirement. The span of the access requirement shown in the *solid line* is 11

Walkthrough path A walkthrough path is the path along which camera travels in the walkthrough scene. The cost of rendering each walkthrough path is composed of a series of continuous access requirements. The performance of rendering a walkthrough path depends on the performance of rendering each of its access requirements, so in this paper, rather than investigating the performance of walkthrough paths, we focus on optimizing the performance on access requirement level, which will subsequently improve the overall performance of walkthrough paths.

Span of an access requirement 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 1 shows a linear order of data units and three different access requirements shown by solid, double-dashed and dotted lines. 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 Fig. 1, data units 1, 4, and 12 are part of two access requirements and data unit 9 is part of all three.

3.2 Training access requirement generation

Access requirements are determined by the combinations of viewpoint locations, view direction, and user operations, and these can be used as parameters to represent access requirement. For example, $(X, Y, Z, \omega, \eta, \theta, \phi)$ denotes the current location (X, Y, Z), moving direction (ω, η), and the new

viewing direction (θ, ϕ). The new position can be calculated by moving a unit distance in the direction of (ω, η) from (X, Y, Z). In two of city model data sets, the view point moves on a fixed plane parallel to the XY plane. Hence the parameter η is not used. For the Boeing model, the viewpoint moves in 3D space. In practice, the moving and viewing direction are discretized, and in the process, viewpoints are also discretized in space.

Camera locations Camera locations are randomly generated. In walkthrough applications, it is not uncommon that geometry primitives are non-uniformly distributed in the scene space: some part of the scene has more geometry primitives, while some other part has less. In a densely filled space, small movement of viewpoint would bring in big changes to the rendered scene. So we first subdivide the whole space into regular cells, and depending on the density of primitives, we generate a random number of viewpoints within that cell for each viewing point we calculate a number of access requirements. Higher the density of the cell, more viewpoints will be generated.

Viewing directions View directions are simply randomly generated using random number generator. The only constraint is that in some of the walkthrough applications, the camera is limited to only a range of directions. For example, in some city navigation systems, the camera is only allowed to be parallel to the horizon plane, and in such case, the access requirements are generated in consistent with the walkthrough system's constraints.

Moving directions Although smooth view direction motion can be achieved in the walkthrough system, for data representation, the directions are discretized and for each direction, an access requirement is calculated and stored. For a given viewing direction, the union of the data units of all the closest directions is fetched. Typically, in our implementation, we use eight directions with 90° field of view. Any request of in-between viewing direction is serviced by fetching the data corresponding to all the nearby represented viewing directions.

3.3 System overview

In a walkthrough application, the scene may be very large, and the number of possible access requirements is prohibitively high. It is impossible to iterate all the possible access requirements. We borrow the idea from Monte Carlo simulation that the performance on a large number of random inputs, is a good representation of average performance for all possible inputs. Thus, we randomly generate a large number of training access requirements, and optimize the data layout based on its performance on the training access requirements. Then we iteratively apply a redundancy-based cache oblivious data layout optimization algorithm to optimize the training access requirements.

```
Initialize training access requirements;
Initialize validation access requirements;
while
redundancy_factor < MAX_REDUNDANCY_FACTOR
do
    Run Cache Oblivious Optimization algorithm;
    Calculate and record average estimated seek time on
    validation access requirements;
    Increase redundancy_factor by a predefined step;
end
Apply data layout with least estimated seek time on validation
access requirements
```

Algorithm 1: Pseudo-code for the system

Similar to the "overfitting" phenomenon in machine learning problems, the data layout will "memorize" the training access requirements, and thus may be over-optimized after a certain stage. In such cases, the generated data layout will maximize the performance for training access requirement set, but may have a poor performance on other access requirement sets. To improve the overall performance rather than just the performance on training data, we adopt the idea of "validation" from machine learning domain. While continuously optimizing the data layout using training data set, we get a series of data layout with increasing redundancy factor, and we use a small set of validation access requirements to determine which data layout in this series is expected to have best result in real use (Algorithm 1).

4 Redundancy-based cache oblivious data layout optimization

4.1 Seek time measure

Given a linear order of data units and a set of 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. 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:

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

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 to the following for our purposes.

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

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 [7,8] 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.

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.

4.2 Algorithm overview

Given the access requirements and the data units, the goal of our algorithm is to compute a data layout that reduces 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 minimize EST while also 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. 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.

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. 1, it increases the span of each of those accesses (all those three access requirements in Fig. 1) 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

$$\Delta\text{EST}_\text{C}(i, j) = \Delta A_i - k_j,$$

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

$$\text{argmax}_j(\Delta\text{EST}_\text{C}(i, j)).$$

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

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

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

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

where $\Delta\text{EST}_{\text{M}}(i, j)$ 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 $\Delta\text{EST}_{\text{M}}(i, j)$ 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 $\Delta\text{EST}_{\text{M}}(i, j)$ 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 $\Delta\text{EST}_{\text{C}}(i, j)$.

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 that is at the top of the heap. After each removal and processing, $\Delta\text{EST}_{\text{M}}$ and $\Delta\text{EST}_{\text{C}}$ of the affected access requirements and the corresponding heaps are updated. If there are no more data units where $\Delta\text{EST}_{\text{M}}$ 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 2.

```
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 2: Pseudo-code for our algorithm

5 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 $O(m + A + log A)$ 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. This comes from the fact that the cache oblivious layout [20] should be a good approximation so the number of moves that would be useful should be limited. 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.

6 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 8 GB 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 (Figs. 2, 3, 4). The City model (Fig. 3) is a regular model that can be used in a navigation simulation application or virtual reality walkthrough. The Boeing model (Fig. 4), on the other hand, represents scientific or engineering visualization applications. The Urban model (Fig. 2) 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

Fig. 2 *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

Fig. 3 *City model* 110 million triangles, 6 GB

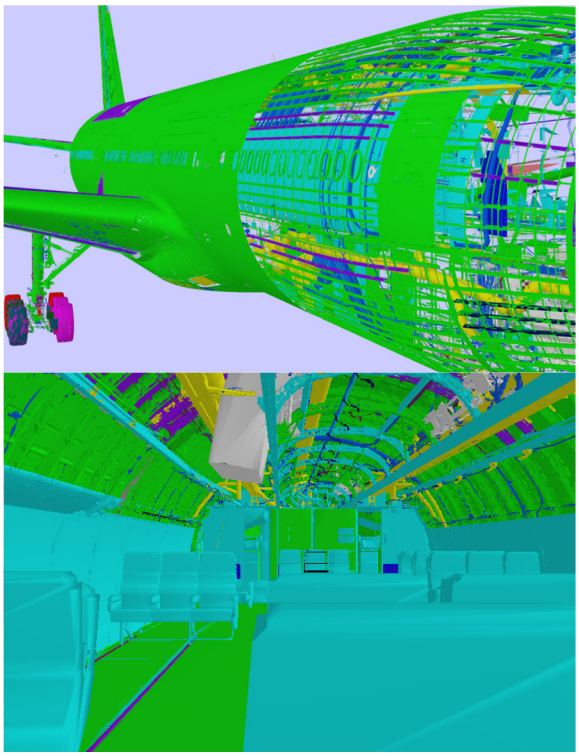


Fig. 4 *Boeing model* 350 million *triangles*, 20 GB. Overview of model (*top*) and model detail (*bottom*)

is to show that the redundancy-based approach can achieve more stable and generally better performance on different real time applications.

The number of training access requirements we generate is in linear with the size of models. We generate 300,000 access requirements for Boeing model, which has 350 million triangles, 20GB. We generate 100,000 access requirements for Urban model, which has 100 million triangles, 12GB. And we generate 100,000 access requirements for City model, which has 110 million triangles, 6GB. And for all three models, 3000 validation access requirements are generated.

And 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 Fig. 6, it took 16 min to create the redundancy layout from the cache-oblivious layout without redundancy for the City model. This number is 80 and 38 min for the Boeing model and the Urban model, respectively.

6.1 Results and comparison with prior methods

In Fig. 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. For training access requirements, this value

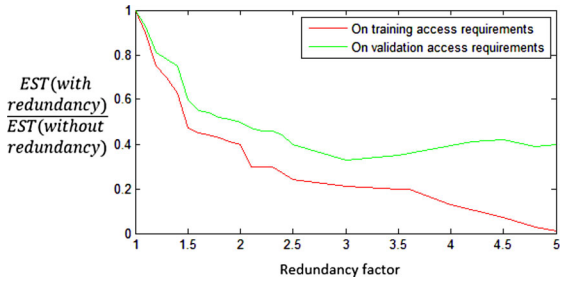


Fig. 5 Plot of the ratio of the EST of the layout with redundancy over the EST of the cache-oblivious mesh layout without redundancy for the City model. For training access requirements, the ratio continuously decreases along with the increment of redundancy factor, while for the validation access requirements, the ratio achieves best result when redundancy is around 2.5–3.5. Final redundancy factors for the other models are determined in the same way

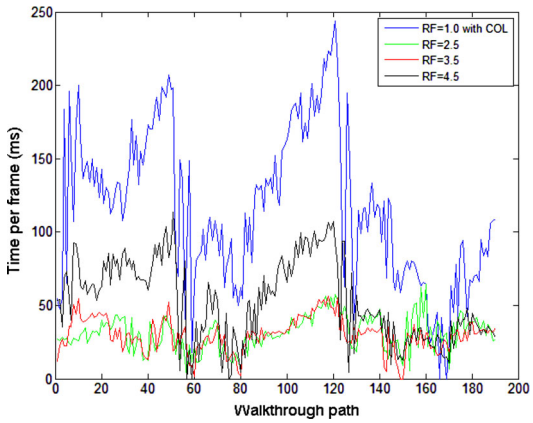


Fig. 6 Statistics of time per frame (averaged per 10 frames) for the city model, with and without redundancy. COL indicates a cache-oblivious layout that does not use any redundancy. RF indicates the redundancy factor

starts at 1.0 where redundancy factor is 1.0, meaning no redundancy, and continuously decreases as redundancy factor goes larger. But for validation access requirements, this value decreases at first, and achieve the minimal value at around 3.0, and then increases along with the further increase of redundancy factor.

The same phenomena can also be observed in random walkthrough test shown in Fig. 6 . Given the same walkthrough path, Fig. 6 compares the time per frame statistics of a cache-oblivious layout 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 redundancy factor 2.5 and 3.5 has a better performance compared to redundancy factor 4.5. Therefore, we can use the performance on validation access requirements to determine which data layout we should choose.

Figure 7 shows the comparison of our method with the one proposed in [8]. When we do the comparison there is again a

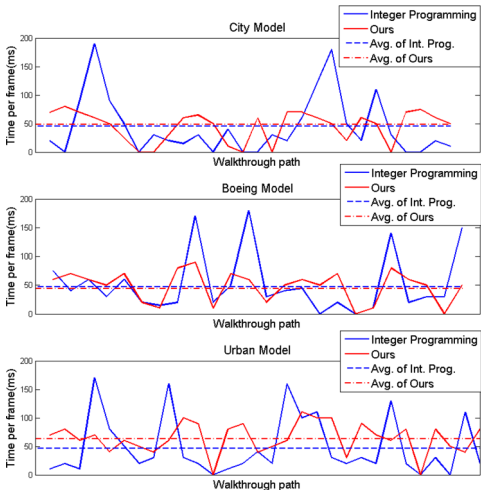


Fig. 7 Statistics of time per frame (averaged per 200 frames) for the City model (*top*), the Boeing model (*center*), and the Urban model (*bottom*), using integer programming (redundancy factor 8.3) and our method (redundancy factor <3.0)

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 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.

7 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 five 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

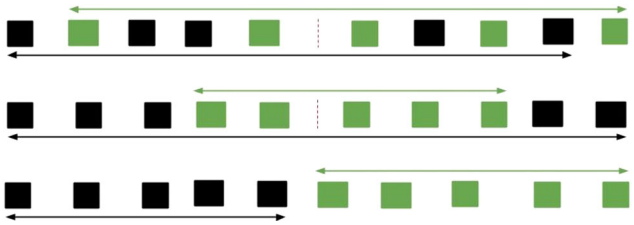


Fig. 8 Example of two access requirements of 5 data units each. The *red line* represents the boundary between blocks in the cache oblivious layout hierarchy. The original layout (*top*), cache-oblvious layout (*middle*), as well as the layout after running our algorithm (*bottom*) is shown

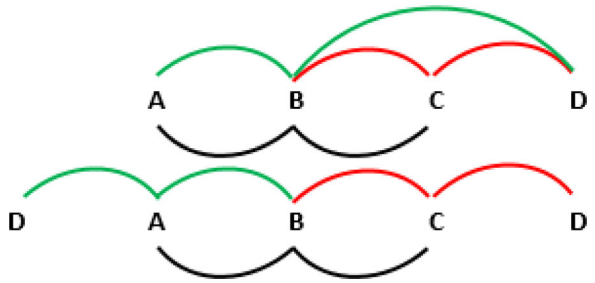


Fig. 9 Data Units with varying access requirements on the *top*. The *letters* represent data units and each *color* represents a different access requirement. It is laid out in its optimal layout without redundancy on *top*. Its optimal layout with redundancy is shown at the *bottom*

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 Fig. 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.

8 Limitations

Our proposed redundant storage of data may limit editing and modification of data because each data unit 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 requirement 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 algorithm assumes that all the access requirements have the same possibility to used. However, in many 3D scenes, there are parts that users are more interested in, thus their corresponding access requirements actually have a higher possibility. If we are given information as to which access requirements are more likely to be used, we would assign more weights to the more frequently used access requirements, and achieve a better performance.

9 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.

Appendix: 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. The main reason why we use the linear search over other alternatives is its cheap update cost. 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

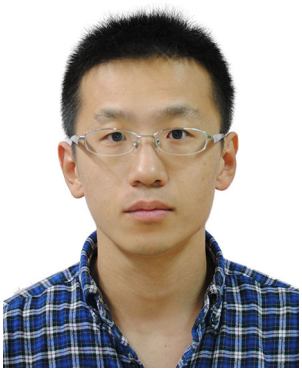
the construction and storage would be $O(N^2)$ where N is the number of data units. The update time would be $O(N)$ when we add a data unit. Since the N for this problem domain is in the hundreds of millions, that is an unacceptable storage bound. The construction run time would also be prohibitive given 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.

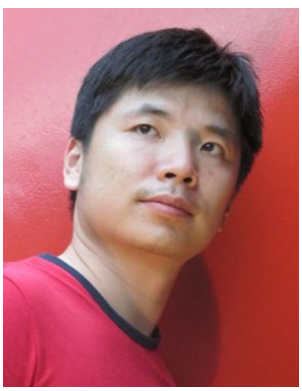
As it turns out, the common data structures that would be used for the finding the minimum value in an arbitrary part of a list are not practical for our purposes. Thus, while a simple linear search may seem inefficient at first, as it turns out it is the best option given our constraints.

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). http://www.pcmag.com/article2/0,2817,2404260,00.asp
4. Diaz-Gutierrez, P., Bhushan, A., Gopi, M., Pajarola, R.: Constrained strip generation and management for efficient interactive 3D rendering. Comput. Graph. Int. **2005**, 115–121 (2005). doi:10.1109/CGI.2005.1500388
5. Hoppe, H.: Progressive meshes. In: Proceedings SIGGRAPH, pp. 99–108 (1996)
6. Hoppe, H.: View-dependent refinement of progressive meshes. In: SIGGRAPH, pp. 189–198 (1997)
7. Jiang, S., Sajadi, B., Gopi, M.: Single-seek data layout for walkthrough applications. In: SIBGRAPI 2013 (2013)
8. Jiang, S., Sajadi, B., Ihler, A., Gopi, M.: Optimizing redundant-data clustering for interactive walkthrough applications. In: CGI 2014 (2014)
9. Lindstrom, P., Turk, G.: Evaluation of memoryless simplification. IEEE Trans. Vis. Comput. Graph. **5**(2), 98–115 (1999)
10. Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., Huebner, R.: Level of Detail for 3D Graphics. Morgan-Kaufmann, San Fancisco (2002)
11. 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)
12. Peng, J., Huang, Y., Kuo, C.C.J., Eckstein, I., Gopi, M.: Feature oriented progressive lossless mesh coding. Comput. Graph. Forum **29**(7), 2029–2038 (2010). doi:10.1111/j.1467-8659.2010.01789.x
13. Rizvi, S., Chung, T.S.: Flash SSD vs HDD: high performance oriented modern embedded and multimedia storage systems. In: 2nd International Conference on Computer Engineering and Technology (ICCET), 2010, vol. 7, pp. V7–297–V7–299 (2010)
14. Saxena, M., Swift, M.M.: Flashvm: revisiting the virtual memory hierarchy. In: Proceedings of USENIX HotOS-XII (2009)
15. Shaffer, E., Garland, M.: Efficient adaptive simplification of massive meshes. In: Proceedings of IEEE Visualization, Computer Society Press, pp. 127–134. (2001)
16. 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)
17. 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)
18. Varadhan, G., Manocha, D.: Out-of-core rendering of massive geometric datasets. In: Proceedings IEEE Visualization 2002, Computer Society Press, pp. 69–76. (2002)
19. Yoon, S.E., Lindstrom, P.: Mesh layouts for block-based caches. IEEE Trans. Visual. Comput. Graph. **12**(5), 1213–1220 (2006)
20. Yoon, S., Lindstrom, P., Pascucci, V., Manocha, D.: Cache oblivious mesh layouts. In: ACM SIGGGRAPH 2005 (2005)

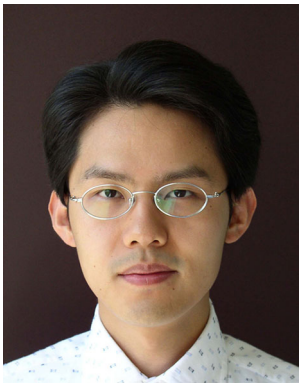


Jia Chen received B.S degree from Beihang University, China and MS degree from Chinese Academy of Sciences. He is a Ph.D student in the School of Information and Computer Science, University of California, Irvine, CA. His research interests include large-scale model rendering, geometry processing, and other geometric problems.



Shan Jiang received the Bachelor's of Science degree from Cyprus College in 2007, and the PhD degree in computer science from the University of California, Irvine, in 2013. He is a software engineer in Altair Engineering Inc. His main responsibility is researching and developing modern graphics technology for the HyperView(R) product. Interactive visualization, massive model rendering, and post-processing on CAD and CAE models are the primary fields he works on.

Zachary Destefano received B.A. degree from New York University. He is a Ph.D student in the School of Information and Computer Science, University of California, Irvine, CA. His research interests broadly include computer vision, machine learning and geometry processing.



Sungeui Yoon received B.S. and M.S. degrees from Department of Computer Science, Seoul National University and Ph.D degree from Department of Computer Science, University of North Carolina at Chapel Hill. He is an associate professor at KAIST (Korea Advanced Institute of Sci. and Tech.). His research interests include scalable graphics/geometric algorithms, large-scale model rendering/visualization, cache-coherent algorithms, collision detection, and other geometric problems.

M. Gopi received BE degree from Thiagarajar College of Engineering, Madurai, MS degree from the Indian Institute of Science, Bangalore, and PhD degree from the University of North Carolina at Chapel Hill. He is a professor of computer science in the Department of Computer Science, University of California, Irvine. His research interests include geometry and topology in computer graphics, massive geometry data management for interactive rendering, and biomedical sensors, data processing, and visualization. His work on representation of manifolds using single triangle strip, hierarchyless simplification of triangulated manifolds, use of redundant representation for big data for interactive rendering, and biomedical image processing have received critical acclaim including best paper awards in two Eurographics conferences and in ICVGIP. He served as the program co-chair and papers co-chair of ACM Interactive 3D Graphics conference in 2012 and 2013, respectively, area chair for ICVGIP in 2010 and 2012, a program co-chair for International Symposium on Visual Computing 2006, and an associate editor of the Journal of Graphical Models. He is a gold medalist for academic excellence at Thiagarajar College of Engineering. He received the Excellence in Teaching Award at UCI and a Link Foundation Fellow. He is a member of the IEEE and ACM.