

# A Greedy Heuristic using Redundancy for Data Layout on Cache Oblivious Mesh Layouts

Zachary DeStefano, Shan Jiang, Gopi Meenakshisundaram  
University of California, Irvine

August 23, 2014

## Abstract

In Computer Graphics, it is important to figure out how to lay out the data units for a cache oblivious mesh layout in such a way that minimizes seek time required. Finding a deterministic solution to the problem is NP-hard so various heuristics have been proposed. In this paper, we present a solution to the problem based on introducing redundancy to the layout. It will first use the idea to get an optimal layout without adding extra units. It will then try to reduce seek time while using the least amount of redundancy possible. This will prove to be a better heuristic than existing ones proposed both analytically and experimentally. The use of redundancy will provide better seek time in both cases than the optimal solution without redundancy.

## Introduction

The Data Layout Problem can be formulated as follows. The input is a linear sequence of data units. Each of the data units is assigned at least one color and many of them have multiple colors assigned. The length of a color is the distance from its first data unit to its last data unit. The data units can be rearranged as desired. The output we would like is the sequence of data units that will minimize the total length of all the colors.

In Yoon's paper [7], the Data Layout Problem is described as well as the metric that is motivating the above definition. We noticed that access patterns described in the paper can be very general and there could easily be data units that are far apart in the sequence but need to be accessed together. This led us to realize that if we copy data units and move them closer to other ones that share the same access pattern then we could save a lot of seek time without adding much storage space. The rest of this paper is about the algorithm developed to optimize seek time while minimizing the redundancy required to accomplish that.

## Related Work

### Data Layout

Sajadi et al [5] looked at how to lay out the geometric data on the disk. According to the paper, many of the previous attempts to speed up rendering of interactive walk-through had not considered the question of how the data layout on the hard disk itself. They proposed a paged based data structure where the geometric primitives were grouped together in the form of disk-pages. There is a hierarchy on the disk-pages that helps organize them for efficiency. This data structure proved to have many advantages over previous attempts. Because of these results, we explored the optimal way to organize uniform data units on the disk.

Sajadi et al [6] explored the data layout for Solid-State Drives (SSD) due to their advantages over Hard-Disk Drives (HDD) and the properties of how they read and write data. They explored the properties of existing layouts. They found that high dimensional spatial data does not render efficiently with HDDs. They also found that Cache-Aware Layouts and Cache-Oblivious Layouts have similar performances on an SSD however Cache-Aware Layouts have many key advantages. They then went into detail on how to construct the cache-aware layout. This eventually led them to develop an efficient out-of-core interactive walk-through rendering application. Despite these advantages of SSDs, due to cost and size limitations HDDs still have to be used for some applications. Through this exploration though it was found that if we reduce seek time on the HDD then the performance will be greatly improved. We thus need to introduce redundancy to reduce seek time. The idea is that with redundancy, the performance of the HDD will be close to the performance of an SSD. Redundancy is not too big of a deal with an HDD due to the inexpensiveness of the space.

Yoon et al [7] explored the problem of data layout on a HDD in such a way as to reduce the seek time required. They developed a cache-oblivious metric to estimate the seek time given the data layout and access requirements for the data. Reducing the seek time then became an optimization problem. Unfortunately the problem was trying to find an optimal permutation of up to millions of data units which proved to be NP-hard. Similar to other problems where an optimal permutation needs to be computed, they developed a multi-level minimization algorithm as a heuristic to get an approximate solution quickly. We decided to take the metric they use and see how much we can reduce seek time if we add redundancy after the heuristic for the optimal layout has been applied.

### Out-Of-Core Rendering

Out-Of-Core rendering has been explored as a method for rendering large geometric data sets when the PC has limited main memory. Coorea et al [3] came up with the iWalk system for rendering large models using this method. They put all the geometric data into an octree and then during runtime only the necessary nodes from the octree and put into the main memory. At the time, they were able to process a very large model on an inexpensive PC and have it render at 9 frames per second. To their knowledge, they were the first to render such a large model on a PC with such limited memory.

Varadhan et al [4] also explored out-of-core rendering but used a different graph approach to obtain what should be put into main memory at a given time. They first construct a scene graph using Level of Detail and Visibility between frames. During run-time they use a parallel processing approach to render the active part of the scene as well as fetching objects from the disk. They were able to use this approach on a variety of environments that were up to a few gigabytes in size and they rendered successfully. The algorithm scaled successfully and the main memory requirements were output sensitive.

## **Image-based Rendering**

Image-based rendering (IBR) techniques are commonly used to improve rendering. Zhang et al [9] presented a survey of image-based rendering techniques. This technique involves sampling the space using a plenoptic function in the first stage and then rendering the continuous plenoptic function in the next stage. It is similar to the technique in signal processing of sampling a set of numbers and then using those samples to get the continuous function and reconstructing the continuous function. The paper goes into detail on various applications of this technique. He also talked about various methods for improving the processing required to adopt this technique. This approach does help render scenes and can be used at the same time as our approach.

Li et al [8] explored various compression methods for image-based rendering. Specifically, they explored the benefits of 3 compression algorithms. In the end all three were able to achieve Just-in-time rendering of large models and they varied in the complexity of the decoding and the compression ratio they were able to achieve. These methods are good for improving the rendering time and can also be used alongside our approach.

## **Motivation**

Jiang et al [2] explored solutions to the data layout problem where the disk seeks are minimized. He was able to achieve a seek time of one between frames. This was accomplished by using redundancy in the data layout. This means that multiple copies of the same geometric primitive were put into the data layout so that the seek time between primitives who would be accessed together is reduced. The redundancy factor ended up being quite large for that case so the algorithm was not practical. The performance was better though so we decided to explore the trade-off between performance and redundancy.

Jiang et al [1] later explored the trade-off between performance and redundancy. Specifically, they looked at the question of given a seek time requirement, what is the minimum amount of redundancy that can be achieved. In this way you can figure out the least amount of space required to achieve a certain seek time in your application. It ended up being an integer linear programming algorithm. After implementing their algorithm and analyzing the results, the consistency and interactivity of the system was greatly improved. The amount of redundancy using this improved algorithm was significantly less than with the single-seek layout paper, making this algorithm much more practical. The algorithm however did not compute the actual

layout of the data that is optimal.

In summary, the page-based data structure gave us motivation to use uniform data units arranged linearly on a disk in order to store all the information for a walkthrough. The data management paper showed us that when we use redundancy to reduce seek time, it will improve the performance for a walkthrough stored on an HDD. The Single Seek Layout paper showed the result of using a great amount of redundancy. The optimization paper found the least redundancy factor required for a certain seek time but it did not compute the actual layout of the data. This paper computes the layout of the data and seeks to find the optimal seek time given an upper limit on redundancy.

## Greedy Redundancy-based Cache Oblivious Mesh Layout Algorithm

The main time when this problem arises is when walking through a large 3D environment. Since we cannot put all the geometric data into memory all at once, we want to make sure that as little time as possible is spent seeking out the new data units during walkthrough. Thus, we need to put the data units as close together as they can be. When they are close together, the walkthrough performance will be significantly better.

The algorithm in Yoon et al. computes a locally optimal solution. Our algorithm is meant to take over after a locally optimal solution has been found or approximated. The basic idea of the algorithm is to take data units and copy them to a place that reduces seek time. This way if a data unit should be in more than one place to optimize seek time then it will be. If the old data units end up not being used then we delete them. Because there are cases where we would delete old units, the first loop in the algorithm is to take care of those cases first as it would reduce seek time without adding any redundancy. In order to get this algorithm to work, there are a few important issues to consider. We need to know which data units should be copied, where it should be copied to, which data units should be used by each access requirement, and which how many data units should be copied.

### Which data unit

Since we only care about the length of each access requirement, we will only be copying the data units that are on the endpoints of an access requirement. The key to this algorithm is that we are shortening each access requirement at each step, so hopefully we can get to a point where they are all grouped together.

### Where to locate data unit

We want to copy the data units to somewhere between the one after the first one and the last one. That way we are guaranteed to reduce the seek time for the access requirement we care about. For our own access requirement, it won't matter where in that interval we place our data unit. However, for the other access requirements, we are adding one unit of seek time since that data unit gets inserted. Therefore, we want to find which place will interrupt the least number of access requirements. We then have to search through each unit to see where the least number of overlaps occurs. We can assume that we have already precomputed the number of overlaps at each unit. We now have a problem of given a dynamic set of intervals (access requirement ranges in our case), and an arbitrary range, what is the least number of overlapping intervals in that range?

This problem is just find the least number in an arbitrary range. For our purposes, we will do a linear search through each entry in the range in order to find the ideal entry. If  $k$  is the size of the access requirement, then this gives us  $O(k)$  query time. Updates will also be  $O(k)$  and construction will be  $O(N)$  with  $N$  being the number of data units. 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  $O(\log N)$  query time, but our construction time and storage would be  $O(N \log N)$ . Updating the data structure would take at a minimum  $O(k \log(N))$  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 however given our exceptionally large input, the construction, storage, and update bounds are too prohibitive.

### **Which data unit is used by each access requirement**

A given data unit will have a few access requirements attached to it. When you copy the data unit, it will move into a new position and benefit your current access requirement. It may benefit other ones too. For the other access requirements, if they will be shorter if they use this new copy then they should do that. Otherwise, they should stick to the old copy.

### **Number of data units to be copied**

If we want to get the best seek time possible with redundancy, we will copy each data unit as many times as it has access requirements. We will then group all the access requirements into their own blocks as they will now have their own copies of the data that they need. Unfortunately, given memory restrictions, this is not always possible. In practice, the redundancy factor was around 20 when we did this.

For this paper, we tackle the problem of how do we optimize seek time given some limit on our redundancy.

## **Run-time and Storage Analysis**

We now need to analyze the running time and storage requirements of our algorithm. For simplification, we will assume that the number of access requirements is a constant ratio of the number of data units. This is not a bad simplification as in many applications, the number of access requirements can easily be close to the number of data units. Thus, we will denote  $N$  as the number of data units and access requirements. We

will use  $k$  as the average length of a single access requirement. The variable  $Q$  will represent the number of runs of the redundancy loop, and  $R$  will be the number of runs of the new copy loop. For simplicity, we will also assume that any given data unit has up to a constant number of overlapping access requirements.

**\*TWO THINGS TO INCORPORATE:**

(1) The number of overall access requirements should be kept as its own parameter.

(2) The number of overlapping access requirements is proportional to the  $(\text{RedundancyFactor}-1) \cdot (\text{Number of Data Units})$  where RedundancyFactor is the factor required for a single seek layout.\*\*

The Initial Construction loop is run on all  $N$  access requirements. There are  $\log(N)$  operations to insert the data into the heap and a constant number of operations plus  $O(k)$  operations for the search of the AR to get the benefit information. Thus it takes a total of  $O(N \cdot k \cdot \log(N))$  operations to do the initial construction.

With the redundancy and new copy loop, there are  $Q + R$  runs of it. Since we are assuming a constant number of overlapping access requirements, there are a constant number of operations, except for reforming the heap and updating the nodes in the AR. With that, there are  $O(\log(N))$  operations done a constant number of times plus  $O(k)$  operations for updating the data nodes in the access requirement. Thus the redundancy and new copy loop takes  $O((Q + R)k \cdot \log(N))$  operations.

This means that in total, our algorithm takes  $O((N + Q + R) \cdot k \cdot \log(N))$  operations.

### **Theoretical Improvements over existing algorithms**

The first part of the algorithm will produce a better solution than proposed by Yoon without adding extra units, thus it is an improved solution to the Data Layout Problem. Here is an example layout produced by Yoon's algorithm:

**\*INSERT PICTURE OF LAYOUT PRODUCED BY YOON\*\***

With our algorithm we will produce the following layout:

**\*INSERT PICTURE OF OUR NEW LAYOUT\*\***

Existing algorithms do not consider redundancy. Even if we find a polynomial solution to the data layout problem, we can actually achieve a seek time better than the optimal one without redundancy. Here is a case where that happens:

**\*INSERT PICTURE OF DATA LAYOUT\*\***

Without redundancy, this is the optimal solution:

**\*INSERT PICTURE OF LAYOUT WITHOUT REDUNDANCY\*\***

With redundancy, this is the result

**\*INSERT PICTURE OF LAYOUT WITH REDUNDANCY\*\***



## **Experimental Results**

**\*\*INSERT THE INFO FROM SHAN'S THESIS\*\***

## **Conclusion and Future Work**

We have shown that we have a quadratic time algorithm with quadratic storage space for the Data Layout Problem. It achieves significant results analytically and experimentally. When walking through an extremely detailed 3D model, this algorithm can be used to ensure that the performance will not suffer. If we give the algorithm the proper access requirements with this 3D model, then the performance will be even better.

This leads to a logical extension of this work. Since we have a good algorithm that takes over once we know the access requirements, we should figure out how to ensure there are good access requirements to begin with. One idea on how to ensure this is to check the usage history of an application and group data units together if they are accessed together with high probability. This could even be done dynamically in the sense that after a certain amount of usage and repeating on a regular basis, you recompute the optimal access requirements and then use that to recompute the optimal layout.

## **Acknowledgments**

## References

- [1] Jiang S. Sajadi B. Ihler A. and Gopi M. Optimizing redundant-data clustering for interactive walk-through applications. *CGI 2014*.
- [2] Jiang S. Sajadi B. and Gopi M. Single-seek data layout for walkthrough applications.
- [3] Correa W. Klosowski J. Silva C. iwalk: Interactive out-of-core rendering of large models. *Technical Report, Princeton University*, 2002.
- [4] Varadhan G. Manocha D. Out-of-core rendering of massive geometric environments. *IEEE Visualization*, 2002.
- [5] Sajadi B. Huang Y. Diaz-Gutierrez P. Yoon S. and Gopi M. A novel page-based data structure for interactive walkthroughs. *ACM SIGGRAPH 2009*.
- [6] Sajadi B. Jiang S. Heo J. Yoon S. and Gopi M. Data management for ssds for large-scale interactive graphics applications. *ACM SIGGRAPH 2011*.
- [7] Yoon S. Lindstrom P. Pascucci V. and Manocha D. Cache oblivious mesh layouts. *ACM SIGGRAPH 2005*.
- [8] Li J. Sun H. Zhang Y. On the compression of image based rendering scene. *Image Processing*, 2:21–24, September 2000.
- [9] C. Zhang and T. Chen. A survey on image-based rendering - representation, sampling and compression. *EURASIP Signal Processing: Image Communication*, 19(1):1–28, January 2004.

## Appendix

### Algorithm Summary

```
Initialize AR heap and newCopy list
for each accessRequirement P's head node and tail node U:
    Set benefit to distance from U to next or previous node
    Let destination be spot with least number of overlapping access requirements
    For each access requirement T that also uses U
        See if T will be shorter by using new copy. Add T to oldCopyList if not.
        Add that to benefit if so
    Add benefit to heap
    If oldCopyList is empty then add U to newCopy list
while newCopy is not empty:
    take out random element and move the node
    Update AR heap and newCopy list
while there exists more space for redundancy:
    pop best element from heap
    copy the element U to its destination
    update affected access requirements
    update heap
```