Performance Driven Redundancy Optimization of Data Layouts for Walkthrough Applications

Figure 1: *Urban model: 100 million triangles, 12 GB. Using our method the walkthrough rendering ~~performance~~ for this model was ~~significantly~~ improved ~~over~~ existing methods.*

Abstract

Performance of interactive graphics walkthrough systems depend 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 algorithm that computes a redundant data layout with the redundancy factor that is within the user specified bounds, while maximizing the performance of the system.

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

1 Introduction

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

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

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

In this paper, we leverage the inexpensive nature of HDDs to store redundant copies of data in order to reduce the seek time. Adding redundancy in order to improve the data access time is a classic approach, e.g., RAID [Patterson et al. 1988]. We are also not the first to consider redundancy for walkthrough applications. Redundancy based data layouts to reduce the seek time were introduced in a recent work [Jiang et al. 2013], in which the number of seeks for every access was reduced to at most one unit. However, in order to achieve this 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 [Jiang et al. 2014] took the data transfer time, seek time, and redundancy, and proposed a linear programming approach to optimize the data transfer and seek time in order to satisfy the total data fetch time constraint. In the process, redundancy was a hidden variable that was minimized. Unfortunately, this approach does not directly model redundancy or seek time, and thus can have unnecessary data blocks and unrealistic seek times.

Main contributions: In this paper, we propose a model for seek time based on the actual number of units between the data ~~blocks~~ in the linear data layout. Using this model, and given the ~~spatial proximity of the data set~~ for a walkthrough application, we develop an algorithm to duplicate data ~~blocks~~ strategically to maximize the reduction in the seek time, while keeping the redundancy factor within the user defined bound. We will show that our greedy solution can generate both the extreme cases of data layout with redundancy, namely the 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), as well as

reasonable solutions for redundancy factor constraints in between the extremes. We show that the implementation of our algorithm significantly reduces average delay between frames and noticeably improves the consistency of performance and interactivity.

2 Related Work

Massive model rendering is a well studied problem in computer graphics. Most of the early works focused on increasing the rendering efficiency. At that time the fundamental problem was not fitting the model into 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 [Luebke et al. 2002], progressive level of detail [Hoppe 1998; Hoppe 1997; Hoppe 1996; Shaffer and Garland 2001], and image based simplification [Aliaga et al. 1999]. Soon thereafter the size of main memory became the bottleneck in handling ever increasing sizes of the model. Hence memory-less simplification techniques [Lindstrom and Turk 1999] and other out-of-core rendering systems [Silva et al. 2002; Varadhan and Manocha 2002] emerged in which just the limited amount of required data that needs to be processed and rendered was brought from the secondary storage to 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 [Sajadi et al. 2011] and cache oblivious data layouts [Yoon et al. 2005; Yoon and Lindstrom 2006] on the disk to reduce the data fetch bottleneck. Our work falls under this class of algorithms that reduces the data fetch time.

Redundancy based data layouts were mentioned in [Patterson et al. 1988; Jiang et al. 2013; Jiang et al. 2014] as potential solutions to ~~this~~ problem of reducing seek time. In particular [Jiang et al. 2014] presented ~~an~~ algorithm that ~~limits the amount of redundancy required;~~ but there were major drawbacks. First, it provides ~~a grouping of~~ data units for each seek, ~~but~~ it does not provide a data layout. This is because it does not relate one data group with another. Such an approach could easily result in unnecessary data block duplications ~~**YOON's comment: Unclear on how its problem results in unnecessary data duplications**~~ . The redundancy minimization is thus not modeled after physical representation of the data layout on the disk. The second major drawback is that the model for seek time is also not based on physical reality. Typically, seek time depends on the relative distance on the disk between the last data unit accessed and the data unit currently being requested. However, in [Jiang et al. 2014], seek time is simplistically modeled independent of the number of data units between ~~them~~. For example, 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.

Figure 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 ~~grids~~; and all the ~~viewpoints~~ within each ~~grid~~ is grouped together to define one access requirement. Thus the number of ~~grid~~ partitions define the number of access requirements. Primitives in a data unit can be visible from many viewpoints, and hence that data unit will be part of many access requirements.

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

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 ~~use~~ A. If a data unit is not required by A, but lies between the first and last unit of A, then it is still counted in the span of A. Figure 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 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