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 however given our exceptionally large input, the construction, storage, and update bounds are too prohibitive.

5 Experimental Results

Figure 4: *City model: 110 million triangle, 6 GB*

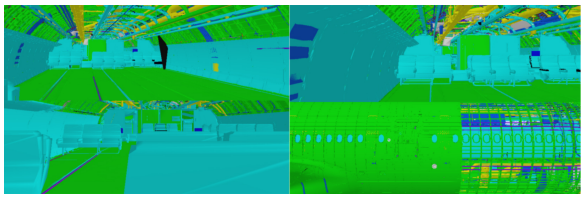

Figure 5: *Boeing model: 350 million triangle, 20 GB*

Figure 6: *Urban model: 100 million triangle, 12 GB*

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

Benchmarks: We use three models to perform our experiments, each model represents a use case or scenario. The City model (Figure 4) is a regular model that can be used in a navigation simulation application or ~~visual~~ reality walkthough. The Boeing model (Figure 5), on the other hand, represents scientific or engineering visualization applications. The Urban model (6) has texture attached to it, which is commonly used in games. By comparing performance of cache-oblivious layout without redundancy to our method using redundancy on these three models, our goal is to show the redundancy based approach can achieve more stable and generally better performance on different real time applications.

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

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

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

The Urban model is different from the previous two in a way that it involves textures. Building heavy redundancy of textures increases the total size of the dataset significantly, while keeping textures away from redundancy leads to inevitable long seek time, which is completely against the philosophy of this work. To solve this problem, we applied a spatial Lloyds clustering 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.

Results: Figure 7 shows the results of delays caused by fetching data on the experimental models we used. We compare the results of a cache-oblivious layout without redundancy and one with redundancy. For the layout with redundancy, we set the redundancy factor equal to 4.2. This factor was chosen because as can be seen in Figure 8, it had considerably better performance than lower redundancy factors and did not have significantly worse performance than higher redundancy factors. A factor of 4.2 is also still practical, as the largest model we tested, the Boeing model, becomes 84 GB which is still acceptable given the large capacity of modern secondary storage devices.

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

There is another major benefit to our approach. Since each time we duplicate one data unit, we can halt it when the redundancy fac-