A Scalable Visualization System for Improving Space Situational Awareness

Ming Jiang
Lawrence Livermore National Laboratory
Michael Andereck
The Ohio State University
Alexander J. Pertica, Scot S. Olivier
Lawrence Livermore National Laboratory

ABSTRACT

Visualization plays a crucial role in Space Situational Awareness, because it has the expressive power to provide insights to support the assessment and planning process of SSA. As computers become more powerful, SSA modeling and simulation in testbed environments are producing massive amounts of data, including more debris particles at higher fidelity and longer orbital propagations with uncertainty quantification. What is lacking in current visualization systems for SSA is the ability to visualize and exploit these massive amounts of modeling and simulation data in real-time. The main reason for this lack of capability is the underlying assumption that the entire data set can be completely loaded into the main memory before any processing or visualization, which is certainly not the case anymore. One effective strategy for dealing with massive amounts of data that cannot fit in main memory is to use out-of-core, or external memory, algorithms, which operate on smaller subsets of the data, while keeping in memory only as much of the data as needed by the algorithms. In this paper, we present a scalable visualization system for SSA simulation data that operates in an out-of-core fashion for the data access and the rendering process. We utilize an out-of-core octree to spatially partition simulation data points, and we use a real-time rendering approach that incorporates techniques for view frustum culling and discrete level-of-detail. We provide experimental results to demonstrate the efficacy of our proposed visualization system.

1. INTRODUCTION

Satellites orbiting the Earth are critical components of a rapidly growing worldwide information infrastructure supporting modern capabilities for a wide range of services, including communications, navigation and research. Space Situational Awareness (SSA) supports understanding the activities and processes that could affect space operations by providing a fundamental basis for the protection of space assets and helping maintain the delivery of their important services. Visualization plays a crucial role in SSA, because it has the expressive power to provide insights to support the assessment and planning process of SSA. As computers become more powerful, SSA modeling and simulation in testbed environments are producing massive amounts of data, including more debris particles at higher fidelity and longer orbital propagations with uncertainty quantification. What is lacking in current visualization system for SSA is the ability to visualize and exploit these massive amounts of modeling and simulation data in real-time. The main reason for this lack of capability is the underlying assumption that the entire data set can be completely loaded into the main memory before any processing or visualization, which is certainly not the case anymore.

One effective strategy for dealing with massive amounts of data that cannot fit in main memory is to use out-of-core, or external memory, algorithms, which operate on smaller subsets of the data, while keeping in memory only as much of the data as needed by the algorithms. To minimize the I/O bottleneck that can result from communicating between fast internal memory and slow external disk, out-of-core algorithms typically provide a data layout scheme along with a redesigned algorithm to enable data access patterns with minimal performance degradation. Existing tools, such as Satellite Tool Kit (STK) from Analytical Graphics, Inc. (AGI), Google Earth from Google Inc., and World Wind from National Aeronautics and Space Administration (NASA), are great at fusing many different types of astrophysical and geospatial data into a coherent visualization. However, they lack adequate capabilities to handle large-scale data that are beyond the range of typical main memory. In particular, they lack sufficient capabilities to manage data in an out-of-core fashion or render geometry in a progressive manner. This inability creates problems for analysts who must explore and analyze SSA data in real-time in order to carry out their mission.

At Lawrence Livermore National Laboratory (LLNL), we have developed a Testbed Environment for SSA (TESSA) that provides a scalable system for visualizing and exploiting massive amounts of SSA data generated from a variety of modeling and simulation tools [10]. Our visualization system is comprised of advanced data structures and algorithms that are needed to address the fundamental scalability issues associated with large-scale modeling and simulation. Through our interactions with SSA analysts, scientists and engineers, we have identified three main visualization needs for SSA:

1. Browse and navigate large amounts of SSA simulation and modeling data
2. Provide real-time progressive rendering for a variety of SSA-related data
3. Allow interactive queries for SSA analysts and simulation feedbacks

Towards addressing these needs, we have developed a scalable visualization system using advanced data layout and real-time rendering techniques. In particular, we have developed an out-of-core data layout scheme for efficient representation of time-dependent, unstructured data, such as debris particles and orbital tracks. In order to visualize the massive amounts of data in this new representation scheme, we have developed a real-time rendering approach that can operate on each subset of the data independently in real-time to iteratively produce a final visualization of the entire data set. Our approach incorporates techniques for view frustum culling and discrete level-of-detail. Together, they enable our system to interactively visualize modeling and simulation data sets that are orders of magnitude larger than what current SSA visualization software can handle.

The rest of this paper is organized as follows. First, we motivate the large-scale data problem that is facing SSA. Next, we describe our proposed visualization system. It includes an out-of-core data layout components and a real-time rendering framework that is designed to operate in an out-of-core fashion. Then, we describe some advanced rendering techniques that are based on this type of data layout and provide examples and results to demonstrate its efficacy. Finally, we summarize our contributions and outline directions for future research.

2. LARGE-SCALE DATA PROBLEM

According to [9], there are approximately 19,000 objects in the near-Earth space environment larger than 10 cm in diameter that are known to exist. These objects, which can be tracked by the U. S. Space Surveillance Network (SSN), include active satellites, defunct satellites and man-made debris (i.e., space junk). For particles that are between 1 cm and 10 cm in diameter, which cannot be tracked in the current SSN, that number is estimated to be around 500,000 [9]. For particles that are smaller than 1 cm in diameter, that estimate easily exceeds tens of millions. At hypervelocity speeds (~ 10 km/s), even these small debris particles can cause serious damage during a collision due to the energy involved.

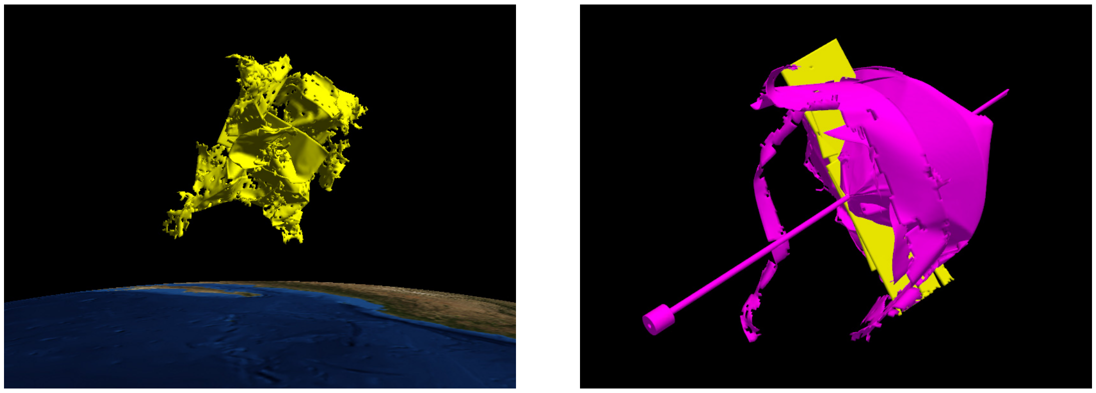


Fig. 1. High-resolution debris geometry generated from modeling hypervelocity impact inside TESSA using ParaDyn. (left) shows debris pieces generated from a "box" satellite collision that produced 2,000 particles with 79,849 vertices in the geometry. (right) shows debris pieces generated from the Cosmos 2251 – Iridium 33 collision simulation that produced 10,799 particles with 546,617 vertices in the geometry.

It is a challenge just to visualize these existing objects and particles flying through the near-Earth environment. This is especially the case when one considers that even with a simple satellite model the geometry involved can easily contain hundreds of vertices. However, the problem becomes compounded by the fact that many of the SSA simulation and modeling efforts are geared towards studying potential orbital collisions between existing objects that can produce tens of thousands debris pieces with high-resolution geometry that can involve hundreds of thousands of vertices. The danger from orbital collisions is the production of new orbital debris pieces, which spread out in phase space and increase the risk of subsequent collisions for more orbiting objects. If these new debris pieces collide with another object before re-entering, they can produce a new cloud of debris, thus increasing the collision probability even more and potentially creating a cascade of collisions. This *collisional cascading* problem, also known as the Kessler Syndrome, was initially defined in [7], as the cascading of particles from random collisions between objects in low Earth orbit (LEO) breaking up other objects at an ever increasing rate. One implication of the Kessler Syndrome is that if no strategy is implemented to de-orbit space junk, then at some point in the future the entire LEO environment can be densely filled with debris, making it too hazardous to continue space operations.

In addition to the massive amounts of particles that can result from SSA simulations, the amount of high-resolution geometry generated from modeling hypervelocity impact can also be overwhelming. Modeling hypervelocity impact for debris generating involves many complex phenomena that span several time and length-scales. Advanced hydrodynamic codes capable of handling large deformations and fracture, statistical material failure models and detailed satellite mesh models enable debris generation modeling. In order to simulate hypervelocity collision event, an explicit hydrodynamics code (ParaDyn) is used along with smooth particle hydrodynamics (SPH). An anisotropic, statistical fracture model (MOSSFRAC) is used to model the impact-induced fragmentation of the satellites, from which high-resolution detailed geometry of debris particles is extracted [10]. Fig. 1 shows two examples of high-resolution debris geometry generated from modeling hypervelocity impact inside TESSA using ParaDyn. Fig. 1(left) shows debris pieces generated from a "box" satellite collision that produced 2,000 particles with 79,849 vertices in the geometry, and Fig. 1(right) shows debris pieces generated from the Cosmos 2251 – Iridium 33 collision simulation that produced 10,799 particles with 546,617 vertices in the geometry.

An added dimension to the large-scale data problem facing SSA is the consideration of time. Once debris particles are generated, they are propagated through space based on the location of impact, the orientation of the satellites and the debris velocity vector. Using either the low-accuracy simplified general perturbation (SGP4) propagator or the high-accuracy force model propagator [10], the position for each particle can be determined for a desired number of time steps. Although the geometry remains invariant during orbital propagation, the amount of positional information is essentially amplified by the number of time steps. Once an orbit is determined for a debris particle, a further consideration in the data problem is quantifying the uncertainty of the determined orbit. Uncertainty data, in the form of covariance matrices for position and velocity, are calculated at each time step throughout the entire orbital propagation. Hence, the amount of uncertainty data can easily surpass the entire orbit propagation data.

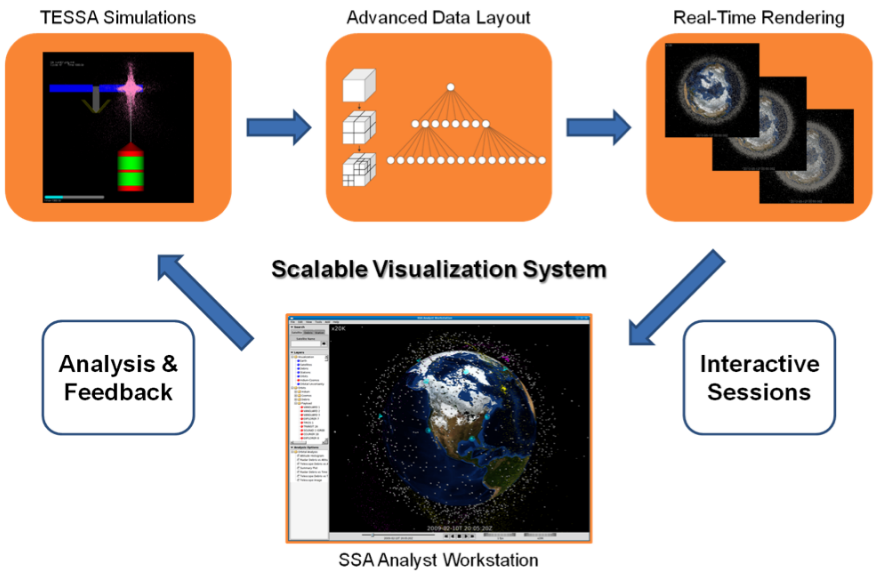


Fig. 2. Diagram illustrating the data flow and components for the proposed scalable visualization system.

3. SCALABLE VISUALIZATION SYSTEM

In order to address the large-scale data problem facing SSA, we propose a scalable visualization system that meets the visualization needs of SSA. It incorporates an advanced data layout scheme using out-of-core octrees and a real-time rendering process with view frustum culling and discrete level-of-detail. Fig. 2 illustrates the data flow of our proposed visualization system. Starting with TESSA simulations, the resulting data is passed through the advanced data layout component that stores the data in an out-of-core format. Next, the out-of-core representation is passed through the real-time rendering component that can render the incoming data in an out-of-core fashion. The third component is an SSA analyst workstation that can be used to interact with the rendered data in real-time for analysis and simulation feedback. What makes our visualization system scalable is that it can handle massive amounts of data in a predictable fashion. In other words, unlike existing visualization systems, such as STK, Google Earth and NASA World Wind, the performance of our system does not suffer dramatically as the data size increase past main memory size. This enables us to explore large-scale SSA simulations that were not possible before.

3.1 Out-of-Core Data Layout

One effective strategy to deal with large-scale data is to use out-of-core, or external memory, algorithms [16], which operate on smaller subsets of the data, while keeping in memory only as much of the data as needed by the algorithm. To minimize the I/O bottleneck that can result from communicating between fast internal memory and slow external disk, out-of-core algorithms typically provide a data layout scheme along with a redesigned algorithm to enable data access patterns with minimal performance degradation. In recent years, as data sizes grow exponentially, a wide variety of computer graphics and visualization techniques have focused on developing efficient out-of-core algorithms to handle these data. For an excellent survey of these techniques, please see [13]. The out-of-core approach is used in other fields as well, such as scientific computing. Toledo [14] provides an excellent survey of methods for solving large-scale linear systems in an out-of-core fashion.

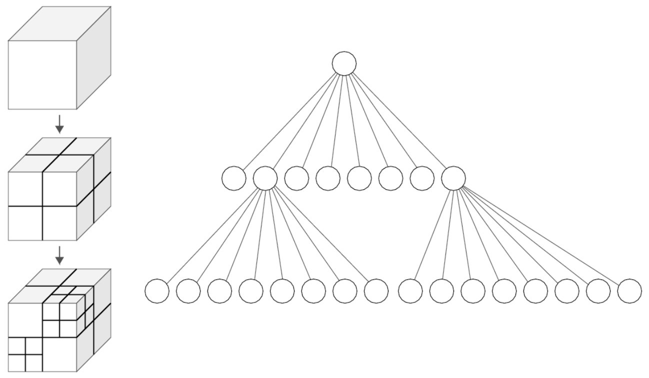


Fig. 3. Recursive spatial partition of the physical domain and the corresponding octree structure.

For our out-of-core approach, we start with a data partitioning scheme using an octree [12]. Octrees are trees in which every node has exactly eight children. They are often used to partition a 3D physical domain by recursively subdividing it into eight octants. An octant with no children is called a leaf, and a leaf node is where data are typically stored. Fig. 3 illustrates how the spatial partitioning of the physical is carried out recursively using an octree and what the corresponding tree structure looks like. Out-of-core octrees have been used in various areas of computer graphics and visualization. Examples include: streamline tracing [15], interactive walkthroughs [5], simplification of huge meshes [3], and surface reconstruction [2]. We choose this approach due to its simplicity and effectiveness.

For a given time step from an SSA simulation, our approach constructs an out-of-core octree for all the positional data of the objects and particles in the simulation. (For now, we are only focusing on the positional data and the geometry data is stored separately.) We treat each positional data as a point in 3D space, and we recursively partition the near-Earth environment using the octree by inserting these points into their respective octants. Our approach for constructing the out-of-core octree is similar to the one described in [5]. Essentially, we construct the out-of-core octree as a preprocessing step to rendering and at run-time we load on demand only the octants that are visible to the rendering. To store the octree on disk, our algorithm saves the positional data of each octree node in a separate file. We create an octree structure file that contains information about the spatial relationship of all the nodes in the octree hierarchy. For each node, the octree structure file contains an entry with the following auxiliary data needed for fast access and rendering efficiency: 1) node id, 2) bounding box, 3) hierarchy level, 4) total points, 5) visited flag, and 6) children's ids if they exist. Given a file path prefix, the name of the file containing the leaf node data can be constructed uniquely by using its id. One key assumption we make is that the octree structure file fits in memory, which is reasonable even for very large data sets given today's main memory size.

Our out-of-core octree construction algorithm works in an incremental fashion. We assume that the original simulation data, 3D points representing the positions of particles, is broken up into manageable chunks that are individually saved into separate files. Starting with a single leaf node whose bounding box contains the physical domain of interest, we insert all the data points within a single chunk, one file at a time. This ensures that we never read in more data than the system memory can handle. If the number of inserted data points within a leaf node exceeds a predefined threshold, then eight children are created under that leaf node and all the data points are redistributed among them. As each new leaf node is created, the octree structure file is updated accordingly.

Once the out-of-core octree is created, inserting a new chunk of points into the octree can be accomplished as follows. First, read into memory the octree structure file. At this point, we do not have to read into memory any of the leaf node files. As we start inserting points from the new chunk, the inserted points are appended to the end of their respective leaf node files. Since there is a limit as to the maximum number of data points a leaf node can contain, at some point during the insertion process, leaf nodes may become non-leaf nodes and new leaf nodes are created. For each leaf node that becomes a non-leaf node, we read its node file into memory and redistribute its data points into its children's node files. Because the leaf nodes have limited size, this redistribution process has a small memory footprint.

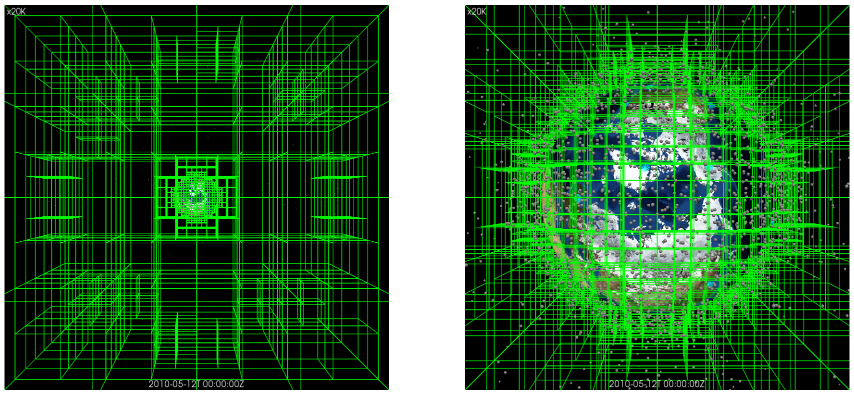


Fig. 4. The green lines represent the octree structure that was created for one of the simulation data sets. For this octree partitioning, the maximum leaf node threshold was 256. (left) is a zoomed out view showing the entire octree. (right) is a zoomed in view showing the octree structure around LEO.

Octrees are typically axis-aligned for efficiency reasons. In order to fully capture the near-Earth environment using the octree and to maintain the axis-aligned property, we map all the data points into the rendering coordinate system. In this system, the center of the Earth is the original; the z-axis is aligned with the North-South pole, and the x-axis is aligned with the Prime Meridian. Each unit in this coordinate system represents 1,000 km – the Earth is modeled using a sphere of size 6.378135 radius. In order to capture more of the near-Earth environment of interest, we set the bounding box of the root node to be from (-50,-50,-50) to (50,50,50) . Fig. 4 shows the resulting octree for one of the simulation data sets. The green lines represent the octree structure. For this octree partitioning, the maximum leaf node threshold was 256. Fig. 4 (left) shows a zoomed out view of the entire octree and Fig. 4 (right) shows a zoomed in view around LEO.

3.2 Real-Time Rendering

In the traditional rendering approach, data access and rendering are carried out on the same thread in a sequential manner. As data becomes larger and the rendering algorithm more sophisticated, this approach becomes less viable for a real-time system. Either the system runs out of memory before anything can be even rendered, or the rendering become inactive while waiting for the entire data set to load and fully rendered. In an out-of-core rendering approach, the idea is to decouple the data access from the rendering and execute them on separate threads. The entire process becomes asynchronous. As the data access thread reads in data, it signals the rendering thread when a sufficient amount of data is ready for rendering. Likewise, the data access thread can be interrupted by the rendering thread when a change in view occurs and a new scene needs to be rendered. Care must be taken in order to maintain the previously rendered data within the same view. A simple and effective way to accomplish this is to maintain the OpenGL depth buffer (i.e., do not clear it) throughout all the renderings within the same view.

In this fashion, the incremental rendering converges in an iterative fashion towards the final rendering with accuracy and correctness ensured. Hence, the entire rendering process remains interactive and scalable for large-scale data sets. Note that in this approach the I/O and rendering computation can be readily overlapped, which is an advantage in today's heterogeneous computing environments (e.g., multi-core and GPU). One disadvantage of this approach is what is known as the *popping effect*, which occurs when one switches from a final rendering to an initial rendering after a change in view. One way to reduce this effect is to keep a set of data resident in main memory so that it can always be rendered as part of the initial rendering. Another disadvantage of this approach is that as the data size increases, so does the number of iterations to converge to a final rendering. One way to overcome this issue is to introduce a notion of multi-resolution representation: coarse levels can be used to render an overview and finer levels can be used for specific detailed views.

Our out-of-core rendering approach is designed to work with any type of out-of-core data format; in particular, it is not limited to the out-of-core octree structure that was described in the previous section. In fact, it can even interactively visualize the original data chunks that are stored in individual files. This approach also works well with traditional time-dependent data, where each time step is stored in an individual file. One interesting application of our rendering approach is to visualize the orbital propagation data in a single image, as opposed to a sequence of images. Our visualization system is implemented in C++ using the following software libraries: Qt for graphical user interface, VTK and OpenGL for rendering, and pthreads for multi-threading. In order to modify the traditional rendering framework of VTK, we utilized VTK's multi-pass rendering capabilities.

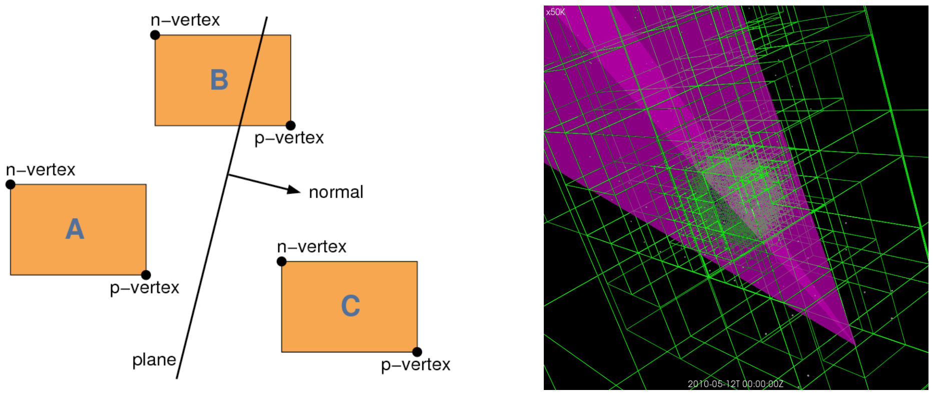


Fig. 5. (left) illustrates how to determine the p-vertex and n-vertex given a plane and intersection scenarios. (right) shows the rendering of a view frustum (magenta) and the octree nodes that it intersected for a simulation data.

3.2.1 View Frustum Culling

The view frustum encompasses the region in space that is within the field of view of the rendering camera. The shape of this region is a rectangular pyramid whose apex is at the position of the camera. The frustum shape is determined by the *near* and *far* planes of the camera that are perpendicular to the view direction. When rendering, only objects that are within the view frustum are visible on the screen. The idea behind view frustum culling is to remove objects that are completely outside the view frustum from the rendering process. When dealing with massive amounts of data points, individually testing each data point for intersection with the view frustum may not be an efficient approach. Instead, we utilize the out-of-core octree structure for speeding up the culling process. What makes octrees efficient for the culling process is the fact that if a non-leaf node is determined to be outside the view frustum, then all the descendants of that non-leaf node must also be outside the view frustum. Hence, those descendants do not even have to be tested for intersection with the view frustum. The goal is to minimize the intersection testing time so that it does not incur too much cost for the overall rendering process. Depending on the scene, this type of hierarchical culling process can very quickly remove data points from consideration.

Given a bounding box of a node in the octree and the six bounding planes of the view frustum, an exact intersection test between the box and the frustum may be too expensive to compute. A naïve approach is to test the eight vertices of the box and consider it to be outside the frustum if all the vertices are outside the frustum. However, this may not always be the case. A conservative approach would be to reject a box if, and only if, all vertices are on the wrong side of the same plane. The side effect of this approach is that more boxes may be tested than necessary. Box testing can be optimized, up to a certain extent, by testing only two of its vertices, namely those that form the diagonal that is most closely aligned with the plane's normal and that passes through the box center. These points are called the n-vertex and p-vertex, where the p-vertex has a greater signed distance from the plane than the n-vertex.

Our intersection testing procedure is based on [1] and proceeds as follows. For each plane of the frustum, if the p-vertex is outside the planes, then the box is outside the frustum, and the procedure terminates immediately. Else if the n-vertex is outside the plane, then the box may intersect the frustum, and the procedure continues onto the next plane. For axis-aligned boxes, computing the p-vertex and n-vertex for each plane is straightforward, and it only involves checking the sign of the components of the plane normal. Fig. 5 (left) illustrates how to determine the p-vertex and n-vertex given a plane and the possible intersection scenarios. Fig. 5 (right) shows the rendering of a view frustum (magenta) and the octree nodes that it intersected for a simulation data.

3.2.2 Level-of-Detail

Level-of-detail (LOD) is an important tool for maintaining interactivity during the rendering process. When geometric data sets become too complex to render at interactive rates, simplifying the polygonal geometry of small or distant objects can increase interactivity. LOD techniques increase the efficiency of rendering by decreasing the workload on graphics pipeline stages by reducing the number of vertex transformations. The idea is to tradeoff fidelity with performance. The reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or moving fast. By controlling the amount of details, one can avoid unnecessary computations and still deliver adequate visual quality. In a complex environment, the amount of information presented about the various objects in the environment varies according to the fraction of the field of view occupied by those objects. In SSA simulations, the amount of objects and particles distributed throughout the near-Earth environment can be tremendous. Navigating through such an environment creates a natural opportunity for exploiting LOD techniques to increase interactivity.

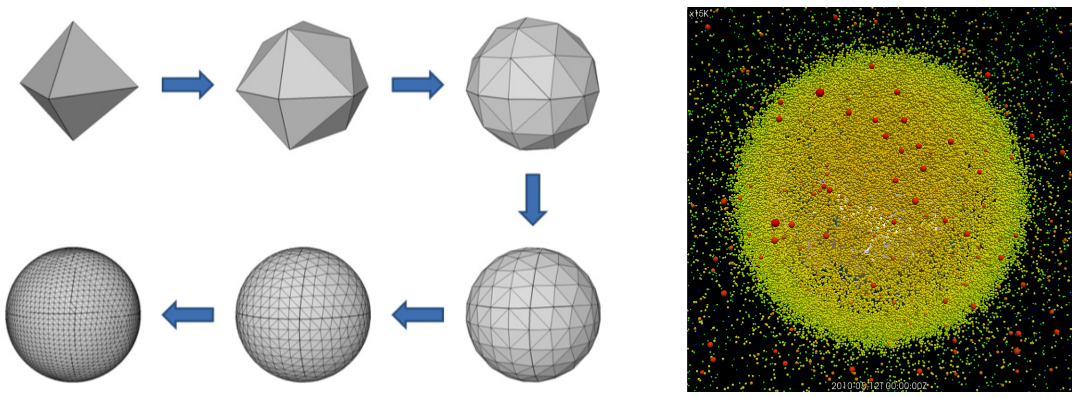


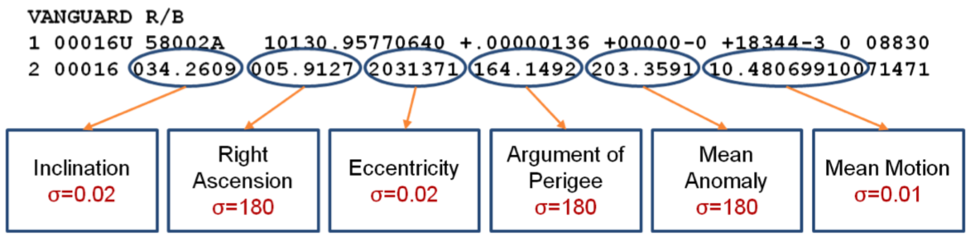
Fig. 6. (left) illustrates six levels of the octahedral subdivision scheme for approximating the sphere. (right) shows the distance of objects in a simulation colored using the standard color map: red means near and blue means far.

The traditional approach to LOD is known as discrete LOD, which creates LODs for each object separately in a pre-process, and then at run-time pick each object's LOD based on distance [4]. Discrete LOD is the simplest programming model, because it decouples the simplification step from the rendering step. For problems that involve drastic simplifications, such as subdividing large objects or combining small objects, continuous LOD is more ideal [8]. By building data structures for the desired level of detail at run-time, one can achieve better granularity that results in higher fidelity and smoother transitions between levels. For view-dependent LOD, the idea is to allocate polygons where they are most needed, such as nearby compared to faraway, and silhouette regions of an object compared to interior regions [6]. Essentially, the idea is to use higher resolution where the user is looking rather than the user's peripheral vision.

The LOD technique that we utilized is the discrete LOD. For objects that do not have a geometry associated with it, which is most of them, we use a sphere to visualize the position of the object. There are many ways to achieve an LOD approximation for spheres [11]. We chose the octahedral subdivision scheme for its simplicity. Fig. 6 (left) illustrates six levels of the octahedral subdivision scheme for approximating the sphere. In our scheme, rather than computing the distance from the camera for all data points, we utilize the octree structure and only compute the distance from the center of the leaf node bounding box. This approach results in a large amount of computational savings and the discrepancies in this distance approximation is bounded by the size of the bounding box. Fig. 6 (right) shows the distance of objects from the camera colored using the standard color map: red means near and blue means far.

4. EXPERIMENTAL RESULTS

We performed a series of experiments to test the effectiveness of our scalable visualization system. Currently, the amount of tracked objects in the near-Earth environment is only around 19,000 [9]. The orbital (Keplerian) elements, which are the parameters required to uniquely identify a specific orbit, for each object are stored as a Two-Line Element (TLE) developed by North American Aerospace Defense Command (NORAD). In order to conduct our scalability tests, we decided to simulate new objects whose TLEs are sampled from the existing NORAD catalog. In this fashion, we can stochastically generate large amounts of TLEs that have similar orbital parameter distributions as the actual NORAD catalog.



```
VANGUARD R/B
1 00016U 58002A   10130.95770640 +.00000136 +00000-0 +18344-3 0 08830
2 00016 034.2609 005.9127 2031371 164.1492 203.3591 10.48069910071471
```

Fig. 7. Gaussian sampling of six parameters of an existing TLE for VANGUARD R/B.

The procedure for simulating TLEs proceeds as follows. For each new TLE that we want to generate, we randomly select an existing TLE from the NORAD catalog. We use a Gaussian sampling strategy for six of its TLE parameters that determines the position and velocity of the object. These parameters are: orbital inclination, right ascension of the ascending node, orbital eccentricity, argument of perigee, mean anomaly, and mean motion. Fig. 7 shows the sigmas for all six parameters used for in Gaussian sampling. Using this approach, we generated four data sets of simulated TLEs of sizes: 16,384, 65,536, 262,144 and 1,048,576.

datasets	# of nodes	# of leaves	timing	# of objects	# of polygons	reduction
16,384	201	164	0.313s	14,168	691,240	91.76%
65,536	393	328	0.641s	56,844	2,641,440	92.13%
262,144	1,257	1,081	3.672s	227,130	11,127,840	91.71%
1,048,576	5,193	4,383	15.250s	887,725	40,585,760	92.44%

Fig. 8. Using a maximum of 1,024 elements per leaf, we generated an out-of-core octree for each data set. # of nodes refers to the total number; # of leaves refers to nodes that contain data; timing is the initial construction time. For a zoomed in view, we calculated # of objects intersecting the view frustum, # of polygons based on the discrete LOD, and the percentage of reduction compared to rendering all the points at full resolution.

Fig. 8 provides a table listing the results from constructing and using our out-of-core octree for each data set. To ensure that not too many files are generated during the construction process, we chose a maximum leaf node size of 1,024. In the table, # of nodes refers to the total number in the tree, # of leaves refers to nodes that contain data, and the timing refers to the initial construction time for the octree, measured on a Dell XPS M1710 laptop with Windows XP, 2 GHz processor and 2 GB RAM. For a zoomed in view, we calculated # of objects intersecting the view frustum, # of polygons based on the discrete LOD, and the percentage of reduction compared to rendering all the points at full resolution. In this case, we used the first four levels of the octahedral subdivision, which contains 8, 32, 128 and 512 polygons. Note that we were able to consistently achieve more than 90% reduction rate using our approach. The simulated TLEs were successfully rendered using our scalable visualization system and shown in Fig. 9. Each column represents a data set starting with 16,384 on the left. The top row shows the zoomed in view and the bottom row shows the zoomed out view. In Fig. 10, we compare the visual quality of using the full resolution (left) for all objects at 512 polygons against using the discrete LOD (middle) with four levels of octahedral subdivision, and provide the resulting grayscale difference image (right) to highlight the small differences.

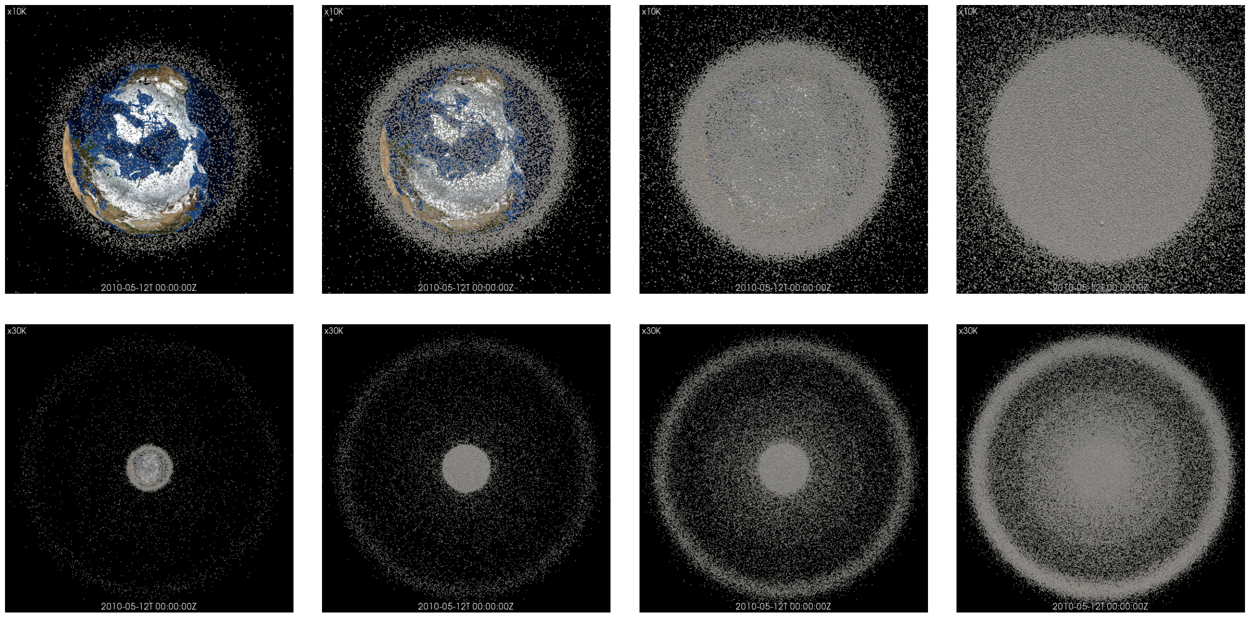


Fig. 9. Four data sets: 16,384, 65,536, 262,144 and 1,048,576, of simulated TLEs successfully rendered using our scalable visualization system. Top row shows the zoomed in view and the bottom row shows the zoomed out view.

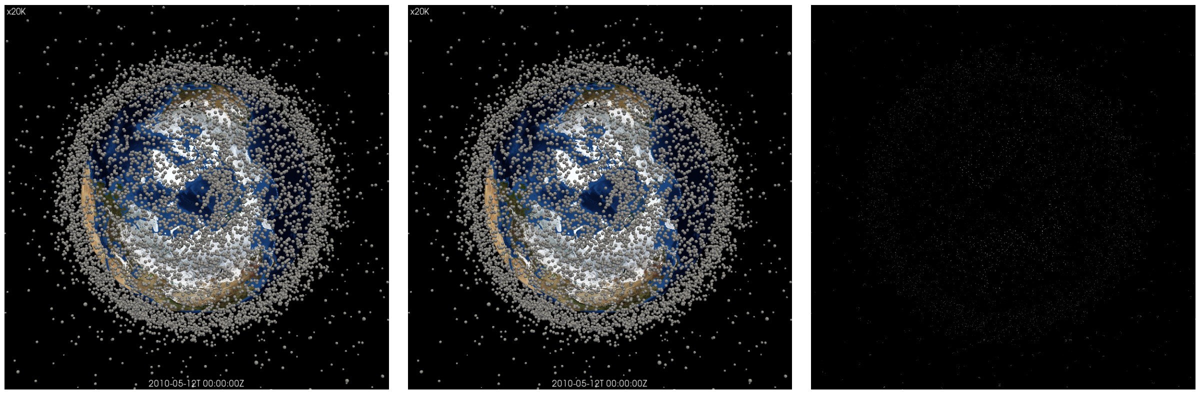


Fig. 10. Comparing the visual quality of using the full resolution (left) against using discrete LOD (middle) and the resulting difference image (right) in grayscale.

5. CONCLUSIONS

We have presented a scalable visualization system that can help the SSA community deal with the looming large-scale data problem. As computers become more powerful, SSA simulation and modeling capabilities will produce a tremendous amount of data that current SSA visualization and analysis systems cannot efficiently handle. One effective technique for dealing with the large data issue is to employ out-of-core techniques that can effectively operate on subsets of the data. In our system, we developed an out-of-core data representation scheme for object and particle positions based on the concept of octrees that spatially partition the near-Earth environment in a recursive fashion. We also developed an out-of-core rendering approach that decouples the data access from the rendering process using multithreading. The rendering process produces intermediate results that iteratively converge to the final rendering while ensuring accuracy and correctness. To further speedup the rendering process, we incorporated a view frustum culling technique that can efficiently utilize the octree to remove objects that are outside the field of view of the rendering camera. We also utilize discrete LOD based on the octahedral subdivision for objects that are too small or distant from the rendering camera. To test the scalability of our visualization system, we simulated four data sets of increasing sizes using the existing NORAD catalog. Based on the results, we were able to successfully visualize all four data sets interactively by reducing the workload on the rendering process by more than 90%.

For future work, we plan to investigate out-of-core multi-resolution representation for the data layout. As data size increases, the number of iterations for the rendering process increases as well. With a multi-resolution scheme, we can judiciously choose the appropriate range of resolutions to render. We also plan to investigate other spatial partitioning schemes based on the sphere. For Earth-centered visualizations, a spherical partition can prioritize based on the distance from the center of the Earth to enable more advanced rendering techniques.

6. ACKNOWLEDGEMENTS

We are grateful to Sergei Nikolaev for providing the material on the Kessler Syndrome and to Wim De Vries for providing the source code for simulating TLEs. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-455073.

7. REFERENCES

1. Assarsson, U. and Möller, T., Optimized View Frustum Culling Algorithms for Bounding Boxes, *Journal of Graphics Tools*, 5, pp. 9-22, 2000.
2. Bolitho, M., Kazhdan, M., Burns, R., and Hoppe, H., Multilevel Streaming for Out-of-Core Surface Reconstruction, *Eurographics Symposium on Geometry Processing*, pp. 69-78, 2007.
3. Cignoni, P., Montain, C., Rocchini, C., and Scopigno, R., External Memory Management and Simplification of Huge Meshes, *IEEE Transactions on Visualization and Computer Graphics*, 9(4), pp. 525-537, 2003.
4. Clark, H. J., Hierarchical Geometric Models for Visible Surface Algorithms, *Communications of the ACM*, 19(10), pp. 547-554, 1976.
5. Corrêa, W. T., Klosowski, J. T., and Silva, C. T., Visibility-Based Prefetching for Interactive Out-of-Core Rendering, *IEEE Parallel and Large-Data Visualization and Graphics Symposium*, 2003.
6. Hoppe, H., Smooth View-Dependent Level-of-Detail Control and Its Applications to Terrain Rendering, *IEEE Visualization Conference*, pp. 35-42, 1998.
7. Kessler, D. J. and Dour-Palais, B. G., Collision Frequency of Artificial Satellites: The Creation of a Debris Belt, *Journal of Geophysical Research*, 83(A6), pp. 2637-2646, 1978.
8. Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G. A., Real-Time, Continuous Level of Detail Rendering of Height Fields, *ACM SIGGRAPH Conference*, pp. 109-118, 1996.
9. NASA Orbital Debris FAQs, http://orbitaldebris.jsc.nasa.gov/faqs.html, 2009.
10. Olivier, S. S., Cook, K., Fasenfest, B., Jefferson, D., Jiang, M., Leek, J., Levatin, J., Nikolaev, S., Pertica, A., Phillion, D., Springer, K., and De Vries, W., High-Performance Computer Modeling of the Cosmos-Iridium Collision, *Advanced Maui Optical and Space Surveillance Technologies Conference*, pp. 720-731, 2009.
11. Praun, E. and Hoppe, H., Spherical Parametrization and Remeshing, *ACM SIGGRAPH Conference*, pp. 340-349, 2003.
12. Samet, H., The Design and Analysis of Spatial Data Structures, *Addison-Wesley*, 1990.
13. Silva, C. T., Chiang, Y.-J., El-Sana, J. and Lindstrom, P., Out-of-Core Algorithms for Scientific Visualization and Computer Graphics, *IEEE Visualization Conference Course Notes*, 2002.
14. Toledo, S., A Survey of Out-of-Core Algorithms in Numerical Linear Algebra, *External Memory Algorithms*, pp. 161-179, 1999.
15. Ueng, S.-K., Sikorski, C., and Ma, K.-L., Out-of-Core Streamline Visualization on Large Unstructured Meshes, *IEEE Transactions on Visualization and Computer Graphics*, 3(4), pp. 370-380, 1997.
16. Vitter, J. S., External Memory Algorithms and Data Structures: Dealing with Massive Data, *ACM Computing* Surveys, 33(2), pp. 209-271, 2001.