Real-time physics data-visualization system using Performer

Chris Mitchell[a] and Walter Gekelman[b]
Physics Department, University of California, Los Angeles, California 90095

(Received 22 September 1997; accepted 8 May 1998)

A data visualization system geared toward large (gigabyte or greater-than-gigabyte) time-varying scientific data sets has been written. The system is a distributed multiprocessed application that employs techniques used both in the video-game industry and military flight simulators to achieve real-time frame rates. On the computational side, the system calculates vector fields, streamlines, isosurfaces, and cutplanes in real time. The graphics component employs strategic visual database decomposition and levels of detail to maintain a fast frame rate. This allows one to navigate through the data sets as they evolve temporally. © *1998 American Institute of Physics.* [S0894-1866(98)01804-5]

INTRODUCTION

Advances in data acquisition have allowed unprecedented detail in the measurements of processes in plasmas. This is due to several innovations. First of all, industry has provided fast data buses (e.g., Camac and VXI technology), faster workstations with more storage, a variety of analog-to-digital converters, digital oscilloscopes, and function generators, all of which can be interfaced to computers. Probes attached to computer-controlled stepping motors have allowed automatic collection of samples at thousands of spatial locations. At present, gigabyte data sets are becoming common. The use of nanoelectronics and nanoprobe technology is expected to bring data sets to the terabyte size—far too large to visualize on an ordinary graphics workstation. Finally, computer simulations routinely disgorge gigabyte data sets with the same structures (scalars such as density, vectors such as electric and magnetic fields) that experimentalists measure.

We report on the design and development of a real-time data-storage server and real-time rendering client capable of handling ''arbitrarily'' large data sets, i.e., data sets that are limited in size only by secondary storage capability. The server has an asynchronous-transfer-mode (ATM) network connection to the client capable of delivering over 150 Mbits of data per second. In addition, software has been written that allows the server to calculate graphical objects such as vector fields, isosurfaces, streamlines, etc. in real time and pass them along to the client. The client employs military flight-simulation technology to effect fast rendering. During data analysis, the server updates the data (loads in a new time step or changes isosurface values) at approximately 2 Hz, while the client allows the user to ''surf'' through the data in real time by maintaining a frame rate ranging from 12 to 20 Hz. In this article we outline how the computer code can manage a real-time environment (a very different paradigm from graphics packages ordinarily used).

The Large Plasma Device (LAPD)[1] at the University of California, Los Angeles (UCLA), generates a quiescent and highly reproducible plasma, which lends itself to high-resolution measurements and large data sets. By interfacing probes to stepper motors, automated data acquisition[2] over many spatial positions at a single time step can yield megabytes of information. Thanks to the gigahertz sampling rates in today's fast analog-to-digital converters, data sets can reach well into the gigabytes.

While storing such large data sets is possible with optical disks and conventional hard drives, an effective real-time graphics package designed to visualize such data sets is not currently available. Much of the previous research in this area has focused on parallel computers. An example of this is parallel Visual 3, which places partitions of data on individual workstations where streamlines are individually calculated.[3] A recursive technique that maps data to a single pixel is capable of visualizing large amounts of data, but is not suitable for our data sets.[4] There are also numerous efforts employing supercomputers.[5] Typical graphics packages used to visualize physics data, such as AVS and NCAR graphics, slow to unacceptable levels when presented with large data sets.

These visualization systems fail to handle large data sets in part because they do not employ real-time graphics techniques. For example, they do not offer a simple way to ''fly through'' the data. This is because these packages assume that the entire space of data for each time step (and in some cases for all time steps) is contained within the viewable area, so that no fly-through is necessary. This seemingly innocuous assumption precludes many of the modern graphics techniques used by visualizers: clipping (in the case of AVS, a clipping plane is defined, but clipping geometry from the scene has no positive effects on performance), graphical level of detail (LOD), and efficient visual-database decomposition.

The techniques we employ will be introduced and discussed mostly in the context of the standalone prototype.

[a]E-mail: chrism@ucla.edu
[b]E-mail: gekelman@physics.ucla.edu

The current state of (and future development plans for) the distributed version will be discussed near the end of the article. The aim here is to simplify the framework of the discussion so that the problems and trade-offs encountered—which are common to both the standalone and distributed versions—can be explained as clearly as possible. Toward this goal, the article can be divided into four main topics: how these strategies and others have allowed us to manage large data sets without buying a supercomputer, how our standalone prototype is designed, the work in progress on the distributed system, and the data we are visualizing with the prototype.

I. OVERVIEW OF THE LAPD DATA SERVER

The current prototype is a multiprocessed application written in C++ using Performer[6] as the graphical library for data display. This technology has been employed by a group in UCLA's Department of Architecture and Urban Planning to create a sophisticated program that allows users to drive through photorealistic cities.[7] We have shared some implementation schemes with this group, but since the visual databases differ so drastically between our projects (dynamic isosurface generation versus temporally invariant skyscrapers in a virtual Los Angeles), the data structures used to organize these databases have forced us to write our own code. We employ Motif for window management and the Visualization ToolKit (VTK)[8] for numerical processing and data file format. Our standalone prototype runs on an SGI Indigo2 with an R4400 processor and 128 Mbytes of random access memory (RAM). Each time step of data is stored on a 30-Gbyte external hard-drive tower connected over a SCSI2 bus. The distributed prototype comprises a Sun Ultra-Sparc 2 with two 100-MHz RISC processors as the server and the same SGI as client. The current data set is a little over 2 Gbytes.

The visualization system's task is twofold: maximize the frame rate (number of times the viewport is redrawn per second) and maintain a visual-database update rate (time evolution of data) between 1 and 2 Hz. A faster frame rate allows smoother ''surfing'' through the data. Database updates are kept in the 1- to 2-Hz range since ''real time'' is not desirable for our data sets, considering that typical intervals between samples can be on the order of nanoseconds. It is currently capable of rendering approximately 17,000 triangles (with texture!) at a frame rate of just over 12 Hz. The time-evolution rate is a little over 2 Hz.

II. ACHIEVING REAL-TIME FRAME RATES

To reach real-time frame rates, the system must provide fast rendering and rapid database updates with a minimum number of searches. In general, data sets can grow to sizes that preclude any workstation from either rendering them or even fitting them in main memory. After deciding on an efficient rendering method, we must deal with the problem of data management. Both issues are discussed in Sec. II.

There are two main rendering categories: immediate mode and display-listable mode. In immediate mode, the CPU is continually feeding the graphics pipeline with graphics commands (vertex information, etc.). In display-listable mode, a list of commands is first compiled into a data structure that is designed for efficient transfer to the graphics subsystem.

For visualizing large amounts of time-dependent data, display-listable mode rendering is inferior. While the only inherent overhead in this rendering choice is in jumping to that memory address at which the list of display commands begins, it suffers from other inefficiencies in practice.

First, display lists are compiled. Compilation takes time and generally does not lend itself to visualizing temporally evolving data.

Second, for maximum efficiency, once a display list is compiled and ready for transfer to the graphics hardware, it is not readable. Display lists are ''closed'' data structures. This means that for visual-database queries which require a read operation, such as ''how much of the data set at this time step intersects the viewing area,'' the graphics data must be duplicated in memory in a readable form, searched, and updated. This means that display lists can incur large memory hits.[9]

In addition to efficient rendering, real-time graphics requires quick access to the models in the visual database. One strategy used to increase frame rate, called a level-of-detail switch (mentioned previously), is to redraw an object with fewer polygons if it is ''far'' from the near clipping plane. To accomplish this without incurring prohibitive overhead in determining which objects in the database are far enough away, the program must be able to ''find'' these objects quickly. To accomplish this, the LAPD data server (LAPDDS) stores the visual database in a tree structure.

Performer allows the programmer to easily design and organize the database into this tree structure. This tree, which is technically a directed acyclic graph,[10] is called a ''scene graph.'' It allows the quick finding capability mentioned above. The scene graph is a collection of intermediate nodes that provide functionality and leaf nodes that define geometry. A subset of these nodes is discussed in this section in the context of rendering cutplane data.

In order to render the scene, the Performer scene graph must be traversed by the program counter three times per frame in real time. The first traversal of the tree is responsible for updates: rotations, translations, color changes, etc. The second traversal removes polygons, and the third draws.

Each intermediate node on the scene graph provides a functionality that is applied to all its child nodes (other intermediates and leaves). For example, a ''level of detail'' node selects the appropriate child node (or subgraph comprising multiple child nodes) based on the distance from the near clipping plane to the LOD node. Another example is a ''dynamic coordinate system'' (DCS) node. Once a rotation and/or translation is applied to a DCS node, all its children—''leaf'' (i.e., bottom of the scene graph) geometry, LOD nodes, or whatever—will have this transformation applied to them. This allows the programmer to quickly manipulate subgraphs (subsets of the visual database). This inherited functionality approach of the Performer scene graph allows quick manipulations of large and heterogeneous portions of the scene. This organization also simplifies the code required for database queries and updates.

A portion of a cutplane scene graph, a subgraph that describes a single plane seen on the screen, is shown in Fig.

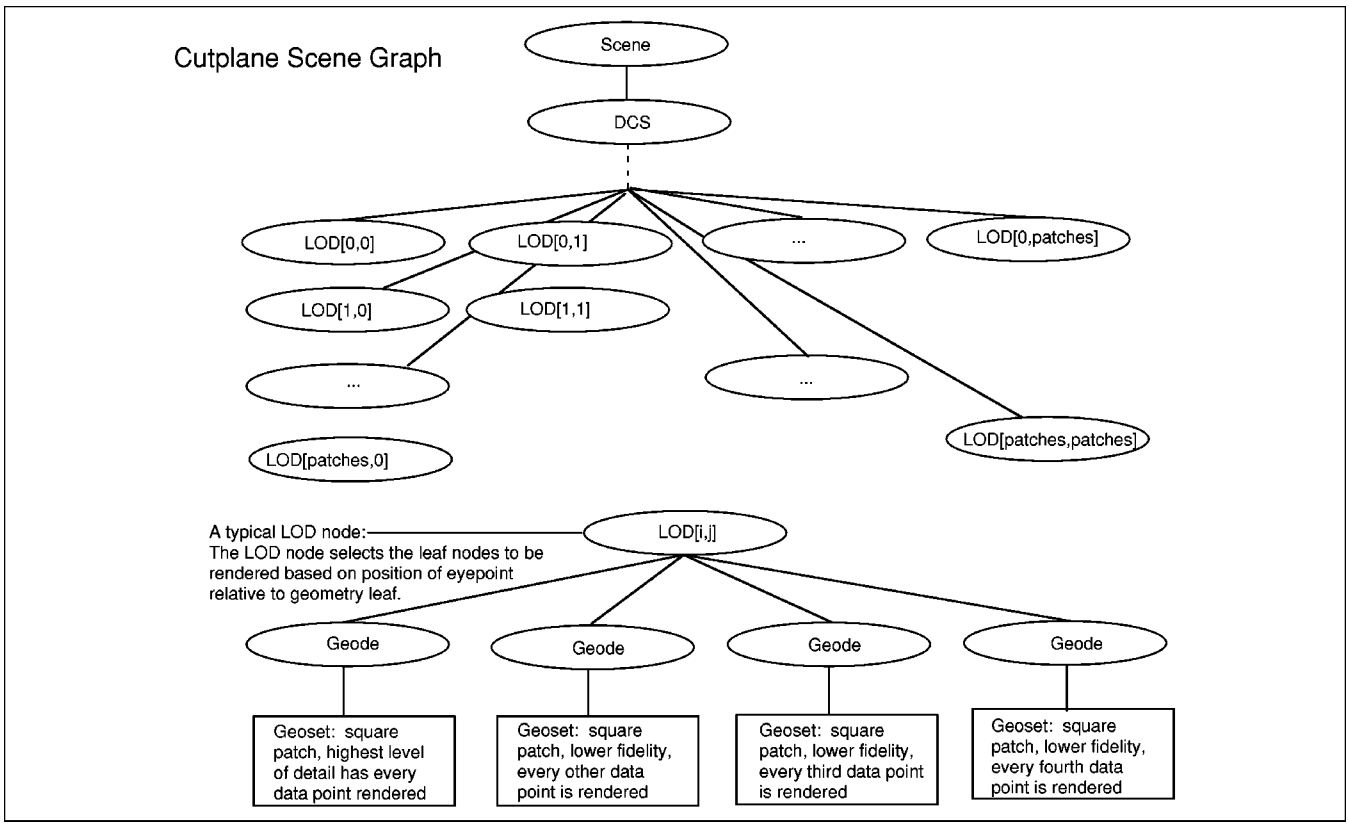

Figure 1. The overall structure of the tree representing cutplane data is shown. The root of the tree is the ''scene'' node at the top. This node's child is a ''dynamic coordinate system'' (DCS) node, which has every ''level of detail'' (LOD) node in the tree as its child. These nodes are ''functional'' nodes, meaning that they perform some function that is applied to all their children. The DCS node can rotate, scale, and translate all the nodes below it. The LOD node ''selects'' one of its children, shown in the bottom half of the diagram, to be rendered based on its (the LOD's) distance from the eyepoint.

1 to exemplify how these visual database structures are built. The actual scene would contain an array of planes, since our machine (the Large Plasma Device, the LAPD, not to be confused with the data server, LAPDDS) is capable of taking volume data sets. This means that Fig. 1 is actually a cross section of what would be the entire scene graph. Because every subgraph representing a single plane is identical, only this one subgraph (a single plane) will be discussed to avoid redundancy.

Since the data change at each time step, one might suspect that the entire scene graph would need to be constantly rebuilt. However, one of the strengths of the tree decomposition approach is that only a subset of its nodes will need to be updated, and much of the graphics information referenced by these nodes will remain static. The intermediate nodes will need only very minor changes, if any, from one step to the next, while the leaf-geometry nodes will need to be continually updated. These leaf-geometry nodes are defined in the Performer library and are called geosets.

The geosets are at the bottom of the scene graph in Fig. 1. The geosets contain geometric data such as shading information, vertices for triangle primitives, normal vectors, etc. By ''contain,'' it is meant [in object-oriented programming (OOP)] that the geoset object pointed to by this tree node stores this information as private class members. The OOP paradigm dictates that the coordinate, normal, vertex, or any information considered ''internal'' to the object's description should be made private members of this class, while operations that can be performed on this object, such as updating or rotating a plane, are public member functions. As mentioned above, the entire tree defines an array of planes to be rendered. Again, this means that the geometric information contained in the geosets shown in Fig. 1 corresponds to the geometry of a single plane.

Each triangle contained in the geoset has vertex information corresponding to where in the LAPD the data were taken and normal vector information that determines shading properties. Since the position at which data are taken is temporally invariant, every scene *may* share positional vertex information (depending on how we display the data). If the vertex positions are static, the corresponding class object member (which is a long array of vectors encoded as pointers to floats) should be declared statically.[11] Static class members are declared at file scope and are shared by all objects of the class. Use of the static storage class will save both memory mapping overhead and possible page faults when the LAPDDS's demands start to reach the computer's system limits.

The geosets also have color vector information at each vertex. This information determines the color combination—corresponding to red, green, and blue intensities—at that vertex. Color values are interpolated between vertices by the graphics engine (a hardware device).

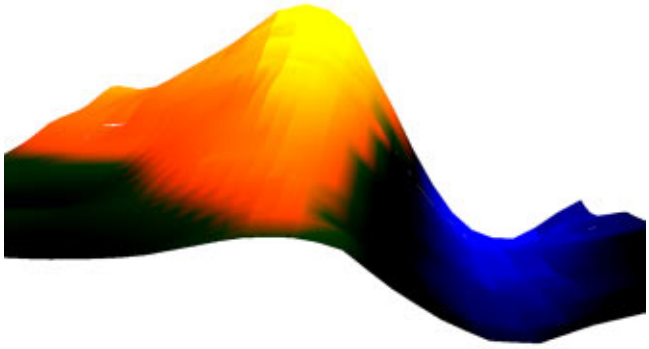

Figure 2. The cutplane shown represents data taken from a ''magnetic beach'' experiment. Magnetic-field values are encoded as red–green–blue (rgb) color and also provide a distortion of the surface from the spatially flat plane at which the data were taken.

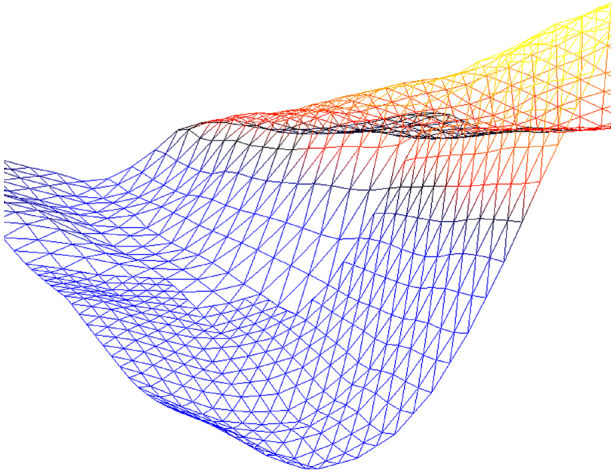

Figure 3. An example of graphical LODs is shown above. This is a cropped wire-frame outline of the cutplane shown in Fig. 2. Notice that the polygon count near the bottom of the page, ''closer'' to the eyepoint, is larger than the more distant portion of the cutplane. Notice also that the high or low polygon counts occur within a square ''patch.'' These are the same patches shown as LOD nodes in Fig. 1.

For the LAPDDS, we have publicly derived a cutplane class from Performer's geoset class. Our derived class references all the internal data and has access to all the member functions of the parent geoset class, but also contains special private class members and functions unique to the LAPD. An example of a special class member is the data taken in an experiment. We store the data at this level of the tree (the leaf-geometry level) so that we can easily relate data taken in our machine to the standard graphics quantities (color, vertex position, etc.) stored at this level by the Performer library. This method ties the data in naturally with native graphics parameters. It is explained in more detail below.

The LAPDDS uses the Performer geosets as the vehicle for data expression. One possible way to express the data is by adding the data at each point to the spatial location at which they were taken. This results in a distortion of the surface out of the plane. Additionally, we can encode field values as colors at each vertex. This is particularly useful for vector-valued fields. We could make a field component—the x component, for example—proportional to the intensity of red at a particular vertex. A bright red region in a cutplane would then imply a large F_x (where **F** is the field we have measured) in that area. If we similarly made the y field component proportional to green, then a bright yellow region would indicate large F_x and F_y values, due to the additive properties of the colors. An example of this is shown in Fig. 2.

But, back to Fig. 1; the next level up the tree is the geodes. Their function (for our prototype) is merely to ''hold'' geosets. In order for the functional intermediate tree nodes to affect them, Performer enforces the presence of geodes as parents. This means that if we wanted to rotate one of our planes by attaching a parent node with that functionality, we would need to have a geode as an ''intermediary'' between the geoset and the node that rotates children. Again, this is a constraint enforced by Performer. The geode is a constructual detail that has been included only for accuracy and can be ignored for the purposes of understanding the functionality.

Moving up the tree, we come to the LOD nodes. This is one of the ''functional intermediate'' nodes mentioned previously. Its function is to select one of its geode children depending on how far the eyepoint is from the LOD node. If the eyepoint is ''close''—a criterion defined by the programmer during tree construction, not by the user at run time—a child comprising a ''large'' number of triangles is selected. The portion of the scene graph defining this LOD behavior, how many levels of detail, at what distance a given level is selected, etc., is constructed prior to any rendering and is not updated with new time steps. To this extent it does not deter the frame rate.

In order for the LOD node to be able to select the correct child, it must have multiple copies to choose from on hand. These copies are the geosets at the bottom of the tree, all describing the same ''patch'' of data with varying degrees of accuracy. They are created by first subdividing each plane into several square patches, where each patch is a separate geoset that contains copies of the data at various resolutions and can be rendered at a level of detail based on the distance from the eyepoint to the patch. After each patch is assigned all the necessary graphics information, it is added to the parent LOD node. It is then up to the LOD node to administer the selection process. A single plane with patches selected by the LOD node at two different resolutions is shown in Fig. 3.

Like everything in software engineering, LODs are a trade-off. For example, how many levels of detail do we want? If they were free, we would like an infinite number, so that the model would ''morph'' continuously from one level to the next in a way undetectable to the naked eye. However, they are not free. Since the LOD needs a copy of each child close at hand, more levels of detail mean more memory used by the application. For example, in our application, the LAPDDS creates four levels of detail. This means that there are now four geodes below each LOD node, even though only one is rendered for a given frame; this is memory lost, but not necessarily wasted.

In addition to memory considerations, LOD nodes create an extra computational burden. Determining the dis-

tance from the LOD to the eyepoint, i.e., determining when to switch levels of detail, costs flops. The memory problem may not affect real-time performance, but the extra computational burden certainly will.

Still, fewer points means fewer triangles and a faster frame rate. Is the extra computation time in the patch selection process compensated for by the lower polygon count? Since this decision hinges on the relative power of the CPU and the graphics engine, the answer varies from one computer architecture to the next. Even on one machine, the answer can be totally dependent on the model rendered. For our application, the answer is yes; we get a faster frame rate.

Looking again at Fig. 1, we see that, the parent of all the LOD nodes in the scene graph is the DCS node. Recall that this graph only shows one plane. When the matrix this node points to is rotated, all the underlying geometry is rotated with it, an example of the scene graph's inherited functionality mentioned previously. This gives the user the ability to rotate the entire model while flying through it, giving the prototype more flexibility in viewing data.

The root of the tree is the scene node. For the cutplane scene graph, it defines the position of the sun and some graphics state information that all child nodes share in common. An example of this would be setting the graphics state at this node to toggle backface culling for all underlying geometry.

Each data type that we would visualize requires its own scene graph. We have created scene graphs for streamlines, isosurfaces, cutplanes, and vector fields. To view two types simultaneously (e.g., an isosurface within a vector field), the LAPDDS connects the individual trees to a common parent (usually the root scene node) as sister subgraphs and treats the whole system as one large tree. As one would suspect, the more trees we put together, the lower the frame rate.

III. STRATEGIC DATABASE DECOMPOSITION

Ideally, the techniques discussed above would give us a system that is capable of displaying all of data space at each time step while maintaining a real-time frame rate. Unfortunately, since the data sets are so large, the number of polygons needed to display all the data at each time step causes the frame rate to plummet, which precludes navigation through the data. Thus, some subset of the visual database at a given time step must be viewed. There are two basic approaches to this database decomposition.

The simpler and less effective approach is to call all the data for a given time step from disk into main memory and use clipping algorithms to determine what subset of those data is rendered. This means that an intersection test between data objects to be rendered and viewing frustum (the truncated pyramid which defines field of vision) boundaries is executed in real time. Even with the best clipping algorithms, this method will necessarily cause some drop in frame rate, since the intersection tests are performed in the graphics pipeline loop.

The selection of the subset of data is most efficient if the database is decomposed into easily indexed sections in secondary storage prior to program execution. This allows a separate process, chip on a multiprocessor machine or machine on a distributed system, to perform a query and retrieval while the graphics pipeline process keeps the frame rate high. The problem then becomes one of how to minimize the query time while retrieving a data subspace which most closely fills the viewing frustum.

The first step in the solution to this problem is to pick a data structure that naturally lends itself to quick retrievals of the scene. Such a structure would be a function of the kind of data being visualized and the method by which they are acquired. The stepper motors involved in our automated data-acquisition system are capable of taking very regularly spaced samples, so the resulting data-point configuration forms a lattice structure in a rectangular solid. This arrangement lends itself naturally to an octree data structure.

Octrees are structures commonly used in computer graphics[12] as hierarchical spatial-occupancy enumeration tools. The structure is a standard tree on which each nonleaf node has eight children. The root of the tree can be thought of as containing the entire rectangular lattice. A note of caution: this is not the same tree as the scene graph that was previously discussed. Our visualization system consists of a forest of two trees per time step. This lattice can be subdivided into octants just like a three-dimensional (3D) Cartesian coordinate system. Each one of these octants would be a child of the root node. In turn, each of these octants comprises eight other octants, corresponding to the next level of the tree. This recursive subdivision can continue until some maximum resolution is reached.

A typical intersection query would start at the root node. Is the entire data space contained within the frustum? If so, retrieve all the data at this time step from disk (or across the network). If not, test each octant for intersection. If any of these octants is contained entirely within the viewable area, retrieve it. If a given octant does not intersect the frustum, test the next octant on this level of the tree. If it intersects the frustum, test its children recursively down the tree until a node is either contained entirely or has no children. This is illustrated [in two dimensions (2D) for simplicity] in Fig. 4.

The next level of optimization would involve tailoring the file system to this data structure. Since the leaf nodes of the octree would be files containing volumes of binary data, traversing the tree could be equivalent to a disk search. One could create a file system based on this octree structure instead of the linked list of blocks used by Unix. However, in the interest of portability we have abandoned this option in favor of an indexing scheme within the Unix file system. A project in the UCLA Department of Computer Science is now addressing this issue.[13]

IV. FLOW OF CONTROL I: STANDALONE VERSION

A skeletal control flow of the standalone prototype is shown in Fig. 5. The session manager's responsibilities include spawning, killing, and suspending child processes, and managing the user interface.

Prior to spawning any new processes, the session manager presents a GUI interface that allows the user a large selection of preferences. If the user wants to glean isosurface information from the raw data, for example, she/he will pick a value, a time step, and possibly some color preferences. In order to be able to dynamically change the

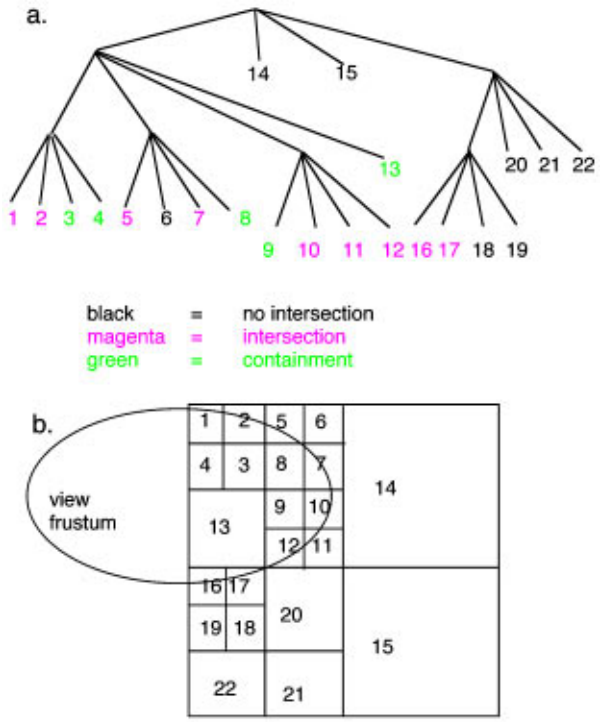

Figure 4. (a) Two-dimensional representation of the intersection of a view frustum with data space. The space is repeatedly subdivided until the squares can most closely approximate the perimeter of the oval. The octree approach (quadtree in this example) makes the determination of which squares to subdivide equivalent to a tree search. Since regions 14 and 15 do not intersect the view frustum, the subtrees below these nodes are not searched. This is illustrated in (b). Since the NW quadrant intersects the viewable area, its children are tested for intersection. When the tree search reaches node 13, all its children are retrieved without further intersection testing, since it is contained entirely within the frustum. In the tree, nodes to be retrieved are shown in green. Nodes that are clipped are in black. Leaf nodes that intersect, and may or may not be retrieved depending on system stress, are shown in magenta.

isosurface value, the current value must be visible to the computational process. Again, in order to change the graphics state information in the rendering process, the updated state information in the session manager must be visible to the rendering process. These processes must also be able to communicate this information quickly.

The fastest (and simplest) interprocess communication (IPC) scheme for our situation is the shared memory arena. In our prototype, there are actually two such arenas. All the shared information independent of the scene graph (e.g., isosurface value and current time step) is allocated a temporary directory called an arena. Each process gets a pointer to this arena and offsets from this address as necessary to access shared information. All information in this arena is available to every process. The other arena is maintained by the rendering engine, and concerns only visual database (scene graph) updates. It is visible only to P2 and P3 in Fig. 5.

After the session manager has received a complete request, it spawns a new process (P2), which sets up the rendering pipeline. An independent process is necessary here, since Motif is event driven and the rendering pipeline is not. If the user generates a window event during the scene graph traversal loop, there must be a process there to handle it. P2 draws the rendering window, initializes the graphical pipeline's shared memory arena, and forks P3. It then continues traversing the scene graph in an infinite loop until suspended or killed by the session manager.

P2's main function is to maintain high frame rate. It descends the entire scene graph in three stages for each frame. The faster it traverses the tree, the higher the frame rate. The first stage is the ''application'' traversal, which effects scene graph updates: changing colors, positions, etc. The cull traversal then clips objects determined to be outside the viewing frustum and does LOD switching. The draw traversal draws.

The pipeline process is synchronized so that the application traversal of frame n does not start until the draw traversal of frame $n-1$ is finished. The database process, P3, runs asynchronously with the pipeline process. It is responsible for building and updating the scene graph. At any stage of the scene graph traversal, P3 could finish its database updates and then request to ''merge'' the current and new scene graphs. P2 must then determine when to honor the request. The IPC problem between P2 and P3 is handled by some function calls provided by the Performer library and is somewhat transparent to the programmer.

The database process also spawns a numerical process (P4). P4 executes the marching cubes algorithm[14] to compute the isosurface and sends vertex information to P3 through a system pipe created prior to the fork. Strictly speaking, this fork is unnecessary, since the tasks executed by P3 and P4 are sequential for a standalone system. The fork was chosen with an eye on the future of the LAPDDS. The pipe could be connected to a socket through which the server and client communicate. The distributed control flow is discussed in Sec. V.

V. FLOW OF CONTROL II: DISTRIBUTED VERSION

There are three major motivations for distributing the LAPDDS. One reason is update speed. The ATM bandwidth (currently 155 Mbits/s and destined to be 655 Mbits/s) provides faster access than our current external SCSI bus arrangement.

Another motivation is increased rendering performance for the client. Ideally, the client would have two processors: one that listens to the socket for the next time step of data and keeps necessary system daemons running, and one devoted solely to the graphics pipeline. Our current client, an SGI Indigo, has only one processor, and so it must be shared by these two tasks. However, we will soon begin porting the client to a two-processor Octane for this purpose.

As far as possible, every task necessary for a graphics system that does not fall under the jurisdiction of the graphics pipeline will be assigned to the server. The server will be responsible for computation, disk lookups, spatial decomposition, vertex generation, etc. This means that the server will have the daunting task of selecting the correct temporal and spatial portion of data based on the client's request, performer necessary computations, creating triangle information, and sending it over the network.

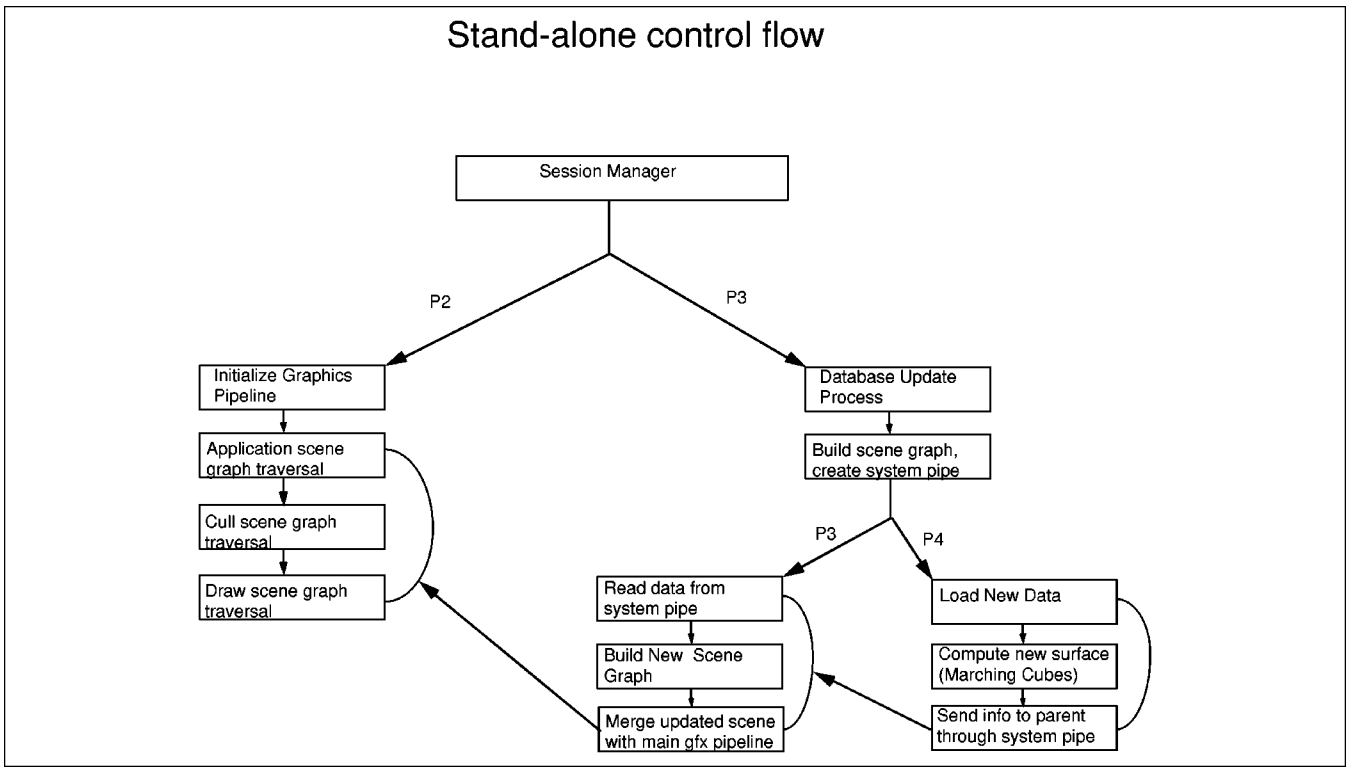

Figure 5. The basic flow of control for the standalone LAPDDS. Separate processes are labeled P1, P2, etc.

Yet another reason for providing a distributed system is information sharing. Someday, a server with large storage capacity and strong computational ability could be located at UCLA, while relatively cheap graphics clients around the world could access the data over ATM. The current prototype is not capable of this, but it is a step in that direction.

The distributed control flow is shown in Fig. 6. Most of the functionality shown has been implemented, but this is a work in progress. The current network connection is fast Ethernet, but after working out the bugs we will put this over ATM.

The basic client–server interaction is as follows. The client sends a request to the server specifying which time step of data it wants to render (there are thousands), how it would like to render it (vectors, streamlines, etc.), and a quality of service specification (data fidelity). The server reads the request, creates a corresponding mock scene graph, encodes it as an array of floats, and sends it across the network. The encoding and decoding modules shown in Fig. 6 are functioning, but to date the ''YES'' option in the flow diagram has not been written. What this means is that vectors and cutplanes can be requested from the client, but not streamlines or isosurfaces. They only work on the standalone system.

The client's flow of control is relatively simple. It spawns one process that listens to the network and translates incoming data, and the parent process's only responsibility is maintaining the graphics pipeline in an infinite loop.

The server's flow of control is more complicated and still under development. The session manager (top of server flow-diagram block) is currently only capable of managing a single session. We are currently developing a session manager that can accept multiple clients. This would bring us closer to the data-sharing objective mentioned above.

VI. EXPERIMENTAL VISUALIZATION

The LAPDDS is being developed to enable us to rapidly find physically interesting regions of space time and isolate new plasma physics effects therein. As a first step, we have developed the prototype that has been discussed here. In order to test it, we re-examined a data set taken in an Alfven-wave-propagation experiment that was initially analyzed using AVS. In this experiment,[15] two out-of-phase shear Alfven waves propagated side by side in a magnetoplasma. Since the waves were excited using structures that had dimension, the collisionless skin depth across the magnetic field, the waves propagated as shear-Alfven-wave cones.[16] Wave vector-magnetic-field data were acquired on 10 planes orthogonal to the background magnetic field, at 400 spatial positions on each plane, and during 2048 time steps at each location. Of interest is the current carried by the wave and its closure in three dimensions. The current was calculated from the fields using $\mathbf{j}=(c/4\pi)\nabla\times\mathbf{H}$ (displacement current is negligible for our experimental conditions). We used the LAPDDS to display isosurfaces of constant j_z, vector fields $[\mathbf{B}(\mathbf{r},t),\mathbf{j}(\mathbf{r},t)]$, current streamlines, and cutplanes.

An isosurface of constant field-aligned current is shown embedded in a vector magnetic field in Fig. 7. The value of the surface shown can be changed dynamically. The frame rate and temporal update rate, as well as the

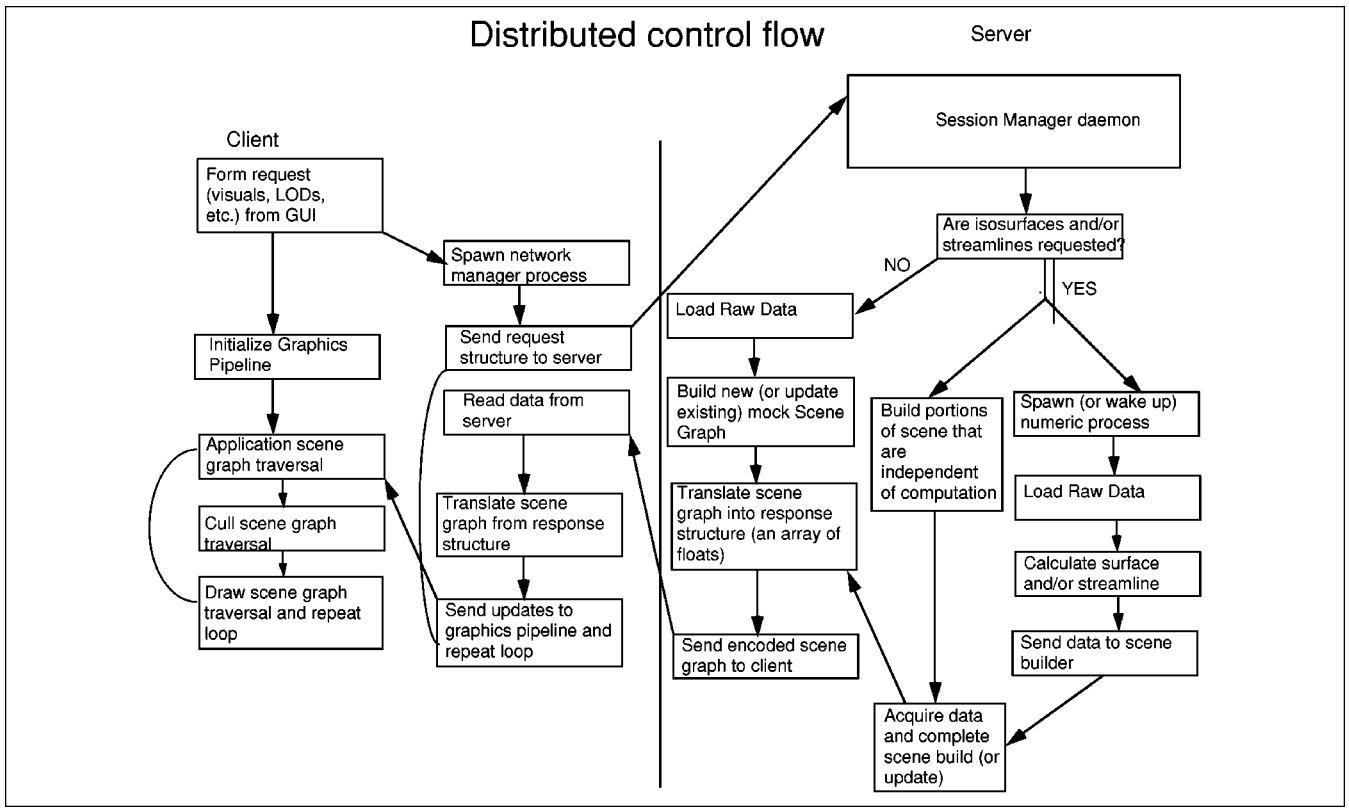

Figure 6. The basic flow of control for the distributed LAPDDS. The vertical line down the center separates the duties performed by the client from those performed by the server.

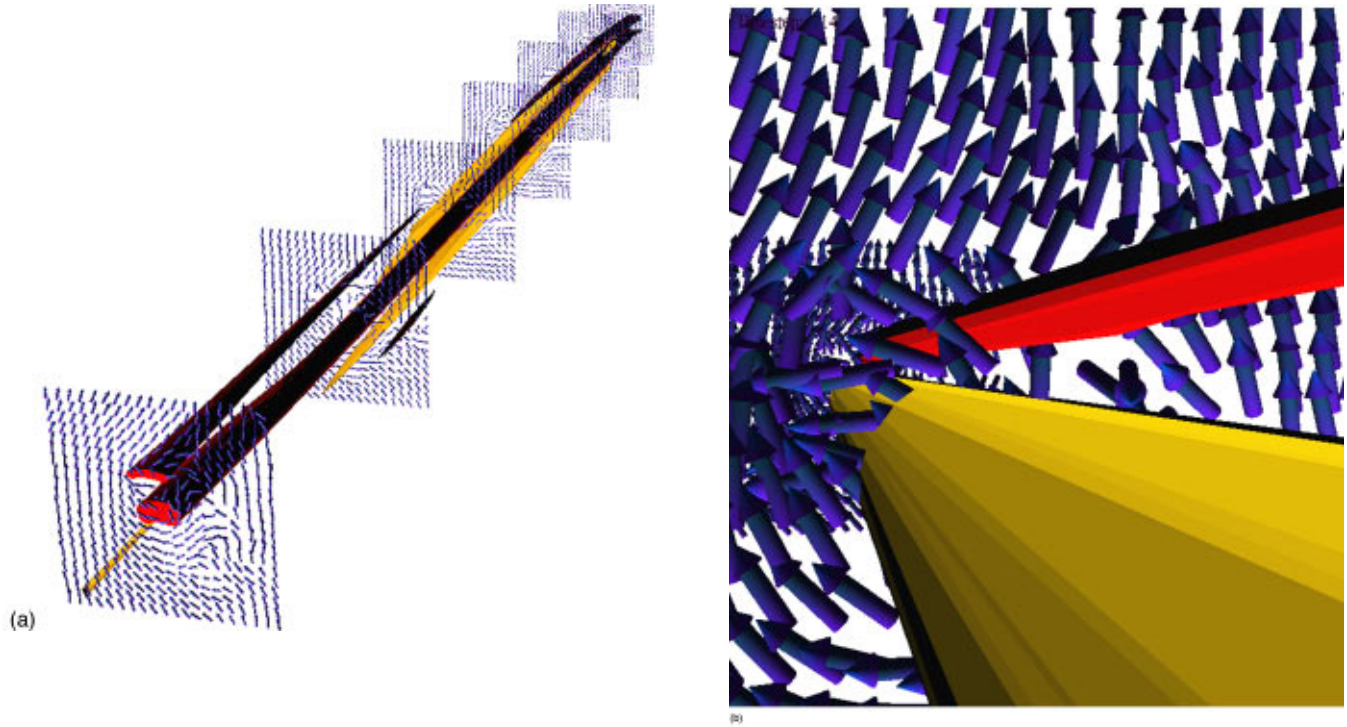

Figure 7. Isosurfaces of constant field-aligned current, J_z, are shown embedded in a magnetic field displayed on seven cutplanes in (a). The blue isosurfaces represent current flowing toward the right, and the yellow surfaces are flowing toward the left. The scene comprises well over 30,000 triangles—too large for a typical interactive flythrough. A smaller subsection, which would typically be viewed during such an exploratory flythrough, is shown in (b).

ability to change perspective, far exceeded the performance of AVS.

The system is by no means complete. At present the streamline computational algorithm can be oppressive, but we expect this problem to be solved when we distribute the application across a client and server. The single-processor SGI client cannot render a sufficient number of triangles for our full vector field with a real-time frame rate. This could be ameliorated with a two-processor client, or with a very efficient decomposition algorithm. (For those who forgot what that means, it refers to the octree/view frustum intersection test and corresponding storage method discussed previously.)

Nevertheless, we feel that with the use of midrange computing equipment, it is possible to build a visualization system with supercomputer performance. The weak link in this scheme is the fact Performer currently runs only on SGI machines, which tend to be expensive. This software may be ported to other platforms someday. There are other software packages under development (VRML 2.0, OpenGL++), which run across platforms, and may eventually allow ports of this system. The design goal of any such software is eventually to allow a great many scientists to run it on inexpensive platforms, and collaborate with each other on high-bandwidth networks.

ACKNOWLEDGMENTS

This work supported by the National Science Foundation under Grant No. NSF IR 1952 7178 003 and by the Office of Naval Research. The authors acknowledge the many useful discussions they have had with Richard Muntz, Bill Jepson, and Scott Friedman. They further acknowledge the many useful suggestions and observations made by the referee. By incorporating many of these we feel the article has been greatly clarified.

REFERENCES

1. W. Gekelman, H. Pfister, Z. Lucky, J. Bamber, D. Leneman, and J. Maggs, Rev. Sci. Instrum. **62**, 12 (1991).
2. L. Mandrake and W. Gekelman, Comput. Phys. **11**, 5 (1997).
3. R. Haimes, Proceedings of the AIAA 32nd Aerospace Science Meeting and Exhibit, Reno, NV, January 1994.
4. D. A. Keim, H.-P. Kriegel, and M. Ankerst, Visualization '95, Atlanta, GA, 1995.
5. S. Smith, G. Grinstein, and R. D. Bergeron, Visualization '91, San Diego, CA, 1991, pp. 248–254.
6. J. Rohlf and J. Helman, Proceedings of the Annual Conference Series, 1994, ACM SIGGRAPH, pp. 381–394.
7. W. Jepson and S. Friedman, I/ITSEC '97 19th Interservice/Industry Training Systems and Education Conference, Orlando, FL, December 1997.
8. W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit* (Prentice–Hall, Englewood Cliffs, NJ, 1996).
9. J. Rohlf, in Ref. 6, Sec. 2.1.
10. T. Cormen, C. Leiserson, R. Rivest, *Algorithms* (MIT Press, Cambridge, 1990), Sec. 5.4.
11. S. Lippman, *Inside the C++ Object Model* (Addison–Wesley, Reading, MA, 1996), pp. 78–80.
12. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics*, 2nd ed. (Addison–Wesley, Reading, MA, 1996), pp. 550–560.
13. J. R. Santos and R. Muntz, UCLA CSD Technical Report, May 1997.
14. W. E. Lorensen and H. E. Cline, Comput. Graph. **21**, 163 (1987).
15. W. Gekelman, S. Vincena, D. Leneman, and J. Maggs, J. Geophys. Res. **102**, A4 (1997).
16. G. Morales, R. Loritsch, and J. Maggs, Phys. Plasmas **1**, 3765 (1994).