A Framework for the Visualization of N-Body Simulations: Independent Study Final Report

Jonathan Coles

August 29, 2004
Introduction

Testing theories in astrophysics involving the interaction of stars is often very difficult. The time and distances are so vast that it is impossible to study a system in real-time. Astrophysicists want to be able to simulate their theories on computers, using realistic data sets consisting of the mass, position, velocity, and acceleration of stars, or more generally, “particles.” The interactions of these particles are computed to construct an accurate theoretical model. The problem is that these interactions are very difficult to compute on the scale that scientists need. The number of particles is often in the millions and each star enacts a gravitational influence on all other particles. Thus, the number of computations needed to calculate the gravitational vector forces for each time step in these so-called N-body problems simulation grows as $O(2^n)$.

To solve this exponential growth problem, researchers in Tokyo, Japan developed specialized hardware called GRAPEs (GRAvity PipElines). The original design was a single, large circuit board to which the particle data was downloaded from a host machine. The board then performed each set of computations and the results were uploaded back to the host machine. The design had two major drawbacks. First, the amount of the data that could be analyzed at one time was limited to the amount of memory that the board held. Second, the GRAPE board was much faster than the host computer, so while the host was preparing for the next round of computation, the board was idling and not being used to its full potential.

The newer designs of the board replace the single, large board with many smaller PCI cards that plug into computers connected in a cluster. Given $n$ machines, the data set is divided evenly across the $n$ boards, with each machine holding $1/n$ the data. This portion of the data remains fixed in each board's memory. The amount of memory on each board is double that needed to hold the $1/n$ data so that there will be room to temporarily store another $1/n$ data for comparison. To calculate the effects of all of the particles, each board needs to see the particles which are not held in its memory. The cluster of machines rotates the complete data set through the network so that each board has a chance to see the portion of the data set not held fixed in its memory. After each phase of rotation, the computed values are uploaded back to the host machine. After a complete rotation, the computers pool the results to form the new data set [1]. See Figure 1.

The cluster solves the two problems that the original board design faced. To handle more particles, another computer with a GRAPE board can be added to the network. The cluster can expand as much as is needed. The idle time for each board is also reduced because the host machine only needs to analyze the results of $1/n$ the data before using the GRAPE again.

Professor David Merritt of the Astrophysics Department at RIT has recently received funding from the NSF, NASA, and RIT to build such a GRAPE cluster. The initial cluster will have 8 boards, but there is hope to expand, with further funding, to 32 boards. While these boards can generate the data needed to build a simulation, there is no software to visualize the results. To this
Figure 1: Diagram of how multiple GRAPEboards can be used to analyze large data sets. The complete data set is broken into pieces which are then cycled across the network and into half of the GRAPEboard memory at each time set.

end, Professor Merritt has asked Professor Hans-Peter Bischof of the Computer Science Department to lead the design of such a software system [2].
Software Requirements, Goals, and Design

The software requirements called for a system that allows the user to display the particle data from several different views. The particles are represented as points on the computer screen and each view from the point of view of a virtual camera. The user must be able to change the position of the cameras and create “flight paths” so that the data can be viewed as a movie of a camera moving through the virtual space. Since the particle data contains information about how the particles change over time, the user can animate the particles and control which time frame is displayed. In addition, the way the points are drawn should reflect the data in some way. For instance, the color and size of each point could be based on the velocity and mass, respectively, of the particle it represents.

In designing the system, one primary goal was flexibility. The entire system is controlled using a very simple language called Sprache. This language controls the creation, movement, and positioning of the cameras, and includes commands for loading particle information and changing the virtual time frame that is being viewed. Using this language provides the ability to easily create complex sequences of commands without having to modify the system. Scripts can be written and loaded on demand to be executed and played repeatedly. The language at the time of this writing is given in BNF in Appendix A.

The general system design consists of three pieces: the Switchboard, the Feeder, and the ViewController. The Switchboard is the module responsible for maintaining the current state. All commands that affect the position of the cameras, or anything else in the system are first sent to the Switchboard. When commands are sent to the Switchboard, the Switchboard distributes them to all of the different displays. There is no limit to the number of displays that can be connected. Before commands can be sent to the Switchboard, however, they must first be sent to the Feeder.

The Feeder acts as a central point to which other pieces of the system send commands. The commands are then ordered and sent to the Switchboard one at a time. The Feeder also allows an additional layer of interpretation. If a different, more complex, language were to be developed, the Feeder could provide a translation mechanism to convert the higher level language into Sprache commands. The rest of the system would be unaffected, but complex sequences could more easily be programmed, either directly by the user, or through an interface of some sort.

The ViewController is the visual part of the system. The ViewController is connected to the Switchboard and receives instructions from the Switchboard on how to update the display. Within the ViewController is the MapView and the CameraViews. The MapView offers a global view of the positions of the cameras relative to each other and the coordinate system of the virtual space. The CameraViews mechanism manages the different CameraViews. Each

1Priority of commands is currently an issue that has not been dealt with, although this will be handled in the future.
CameraView shows the particles as they appear from the point of view of one camera.

The two pieces of the ViewController also act as an interactive interface. The user can manipulate the cameras by directly clicking and dragging the cameras in the map view, or by clicking and dragging in one of the CameraView displays. The commands to execute the movements are first sent to the Feeder and then to the Switchboard. The Switchboard then decides to send the actual movement command to the ViewController, which then updates its display with the current state of the system.

It is important to understand this sequence of events. The ViewController will not act on a user command unless instructed to do so by the Switchboard. This is to prevent the state in the Switchboard from became unstable.

The three components communicate using one of two mechanisms. Either they can talk over an TCP/IP socket connection, or, if they are running on the same machine, they can be connected directly in memory. The advantage of communicating in memory is one of speed. However, allowing for network communication yields two powerful features. First, any language that supports a network library can be used to design any portion of the system. If one ViewController would be better written in C++ or Python then it would be easy to plug in to the system. The code would simply have to understand how to talk to the Feeder and Switchboard using Sprache commands over a TCP/IP connection. If there is a new input device that is better suited to manipulate three dimensional images, such as a glove, then the driver would only have to support talking to the Feeder. This allows for an incredible degree of freedom for anyone wishing to add to the system. Second, it means that the ViewControllers, for example, can be spread across the Internet, allowing researchers to view, discuss, and manipulate the data simultaneously. In order for researchers to do this, however, they must have access to the same data.

There is currently only a tentative design which allows the ViewController to retrieve particle data by talking to a module called the ParticleView. The ParticleView would have direct access to the data and would allow the ViewController to query for particle information at a given point in time. The data is stored in a format which gives initial values and then incremental changes over time. The time intervals for each change may differ between particles, but using some simple algorithms it is possible to extrapolate the position, velocity, etc. of every particle at a given time. The ParticleView performs the extrapolation at the request of a ViewController.

A picture of the design is in Figure 2 on page 5.

**Current State**

The code written thus far has been in Java, using Java3D for the graphics. Initially, it was thought that C++ and OpenGL would be best, but for portability and certain aspects of coding, Java was deemed to be better suited to the task. Java3D can take advantage of both OpenGL and DirectX libraries and does not
Figure 2: System design of the visualization software
require compiling C++ code that may not be portable.

The Switchboard has been mostly implemented. The design allows it to talk to multiple ViewControllers, locally or across a network. The Feeder is also operational and can receive commands from multiple ViewControllers. The ViewController is only partially complete. The user interface portions of the ViewController need a lot of work, and the appearance will certainly change. The MapView has a tentative user interface, but the practicality of it has yet to be determined. The CameraView can display data, but at this point, the data is randomly generated. The ParticleView has not been fully designed or written, but it should plug into the current code base of the ViewController and CameraView.

Future Plans

As stated above, the user interfaces need significant improvement. This is simply a matter of time and acquiring a more thorough understanding of how best to interact with the system. Eventually, it will be necessary to conduct a usability study.

There will need to be a complete way to access data through a ParticleView. There are some thoughts that the data may reside on one machine, and that if another machine wanted to view the data, the ParticleView would transmit it over the network. This is one possibility. Another would simply be to transfer the data, either via network or storage medium in advance.

There is no simple way to play scripts other than feed them to the system as user input. There will need to be more extensive command line options and a user interface to supports this.

There are also plans to have a feature that can automatically generate a movie from a Sprache script. There is research being conducted in the area, and there are promising results that this will be done.

Conclusion

A lot has been developed since the project’s beginnings, but there is still a reasonable amount of work left to do. A fully operational program is foreseeable well within the next two months. From that point on the program will require fine tuning and experimentation to provide the best combination of control, flexibility, and usability.
Appendix A

Sprache Command Language

```
SPRACHE: COMMANDS;*
   | # COMMENT

COMMANDS: CAMERA_CMDS
   | WORLD_CMDS
   | updateview
   | +frame
   | -frame
   | load FILENAME
   | DEFINE_CMD
   | PAUSE_CMD
   | SELECT_CMD
   | DESELECT_CMD

CAMERA_CMDS: camera CAMERA_CMD*

CAMERA_CMD: location POS
   | translate POS
   | lookat POS
   | rotate ANGLES
   | up VECTOR
   | select #
   | join # [#]

WORLD_CMDS: world WORLD_CMD*
WORLD_CMD: size DIMENSIONS

DEFINE_CMDS: define DEFINE_CMD*
```
DEFINE_CMD: timestep #
   | CAMERA_SURFACE_CMD

CAMERA_SURFACE_CMD: camera_surface CAMERA_SURFACE_TYPE

CAMERA_SURFACE_TYPE: sphere center=POS radius=#
   | sphere_lookat_center center=POS radius=#
   | none

PAUSE_CMD: pause #

SELECT_CMD: select SELECTION

DESELECT_CMD: deselect SELECTION

SELECTION: all
   | [# - #]
   | #, #*

POS: # # #
ANGLES: # # #
DIMENSIONS: # # #
VECTOR: # # #