Three-Dimensional Computation Visualization

Technical Report GIT-GVU-92-20

John T. Stasko

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
E–mail: stasko@cc.gatech.edu

Abstract

Systems supporting the visualization and animation of algorithms, programs, and computations have focused primarily on two-dimensional graphics to date. In this paper we identify the need for three-dimensional graphics in these types of displays, and we describe how 3D imagery best can be used for visualizing computations. We also introduce an animation toolkit that supports simplified development of 3D computation visualizations. A few examples of computation visualizations created with the toolkit are described and included. Our methodology, although specifically applied to computation visualization, is general-purpose and can be used to build a variety of 3D information visualizations and animations.

Keywords: information visualization, 3D computer graphics, program visualization, algorithm information, software understanding

1 Introduction

For many years the primary way to understand a computer program's or process' execution was to examine its source code and utilize a debugger. Recently, a number of systems for visualizing and animating computational process have been developed as aids for program understanding. These systems have mostly been restricted to exhibiting two-dimensional graphical imagery, however.

In this paper we motivate the need and importance of providing three-dimensional application-specific imagery for visualizing computations and programs. We discuss why 3D graphics are necessary for a particular class of computations and how 3D graphics can provide valuable information for views in which they are not strictly required. We also introduce a 3D animation methodology and package that is particularly well-suited for visualizing software and that does not require a background in 3D graphics to build animations with it.

Although our techniques and our support package could be used to build many different types of *information visualizations*[CRM91], the focus of this paper is on *computation visualization* and *computation animation*. Computation visualization[1] is the use of computer graphics to explain, illustrate, and show how computer hardware and software function. Computation visualizations provide graphical depictions of typically inanimate processes and entities. If a visualization is designed well, viewers will gain an understanding of the inherent process it presents in a way not possible using traditional program understanding methods such as tracing and debugging.

The terms algorithm animation[Bro88], program visualization[Mye90], and software visualization[PSB92] all have been used to describe systems seeking to aid program understanding. Our view of the scope of these terms is shown in Figure 1. Algorithm animation (visualization) is the most specific category, focusing on abstractions of an algorithm and its operations. Program visualization illustrates data structures, program state and program code as well. Software visualization illustrates computer processes and data in addition to regular programs. Finally, computation visualization includes both software and hardware views. We will use the term "computation visualization" in this paper as a general encompassing notion.

Why are 3D graphics desirable for visualizing computations? Briefly, 3D graphics 1) match the 3D data some programs manipulate and 2) provide an extra dimension to encode more information about programs manipulating non-3D data.

Frederick Brooks, in his *No Silver Bullet* paper, gave the following view[Bro87]:

> ...As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These graphs are usually not even planar, much less hierarchical.

[1] For brevity, from this point on we will just use the term "computation visualization" to mean both computation visualization and computation animation. Identifying when a visualization becomes an animation is often a subjective matter. Nevertheless, our system does provide smooth, incremental animation effects.

Figure 1: The scopes of different terms used to characterize visualizations of programs, processes, and computations.

This article states Brooks' opinions about the inadequacy of graphical views in general for portraying software. The specific types of displays referenced, however, were 2D views such as flowcharts.

Andy van Dam, in his keynote speech at the 1991 UIST conference, implored researchers to "escape flatland" and explore how 3D graphics can be used in user interfaces and information displays[vD91].

Three-dimensional graphics have begun to appear in certain information displays, primarily those of data. For example, the Information Visualizer from Xerox Parc[CRM91, RMC91] is a system that uses three-dimensional imagery to present structured information such as computer directories and project plans. The system's designers noted that the three-dimensional displays help shift the viewing process from being a cognitive task to being a perception task. This transfer helps to enable humans' pattern matching skills.

Scientific visualization presents complex scientific data from areas such as meteorology, biology, and chemistry in a 3D format. Computation visualization presents yet a further challenge, because it involves not just static data views, but dynamic views of the operations and actions computer processes and programs make over time.

2 Requirements

One approach to building 3D computation visualizations would be to use an existing sophisticated 3D animation package or library such as PHIGS+, GL, or RenderMan. For example, in our work with 2D algorithm animation, we developed an animation of the post office problem, a simulation often used in operating systems courses to teach queueing. This simple animation showed a top view of a post office with the patrons depicted as stick people. It would be conceivable to build a 3D animation of this problem with highly realistic ray-traced imagery, sophisticated illumination models, and kinematics of the human figures — basically making a computer-generated movie.

Although appealing in a general sense, the effort involved in such a project would not warrant the understanding or comprehension gain achieved, if any, by the animation for two key reasons: First, graphics systems for building such realistic imagery have steep learning curves and they require familiarity with 3D graphics concepts, terminology, and techniques. Second, computation visualization does not require all the power these systems provide. A

package with lesser capabilities could be able to well provide the kind of information displays that computation visualizations exhibit. Our goal herein is to find a middle ground in the inevitable trade-off of image sophistication versus ease-of-use.

Computation visualizations must illustrate how computers and software are used to perform computations and solve problems. The domain of computer hardware and software is a somewhat restricted world: In hardware, we use processors, memory registers, CPU's, and so on. Software involves programs with statements, data structures, code, and operations.

A structured image model involving block diagrams, circles, lines, and text is a natural choice to illustrate how computations are performed. Existing 2D software visualization systems almost always use this type of imagery.

Below we list three basic requirements for 3D computation visualizations. The requirements are the three-dimensional analogs of the structured image model described above.

1. The visualizations must provide a suite of graphical objects including lines, rectangles, circles, polygons, arrows, blocks, spheres, and text. Color displays are critical.
2. The graphical objects must be able to undergo changes in position, size, color, and visibility, thereby providing a sense of animation.
3. The visualizations must be able to adapt to run-time execution description data that controls how the display appears.

The final requirement also precludes the use of many preexisting 3D animation toolkits, which are typically used for predetermined, precisely-scripted scenarios. Computation views are parameterized animations; they cannot be hard-wired, predefined animation sequences. The views must be adaptable, presenting imagery that corresponds to programs' run-time conditions, data and inputs.

Another useful feature, although not listed as a requirement, is the ability to easily create groups of objects, such as rows, columns, and grids that are appropriately spaced and scaled. We have found this capability to be invaluable in visualizing computations. Lists, arrays, processor architectures, etc., are typically targets of this type of imagery.

3 Dimension Usage

In a 2D computation visualization, a designer has three dimensions with which to encode information about the computation: two spatial dimensions (vertical and horizontal) and one temporal dimension (dynamic changes over time). For example, a bar chart view of a sorting algorithm uses the horizontal dimension to encode array position, the vertical dimension to encode array value, and the time dimension to encode current program state. By adding a third spatial dimension, we supply visualization designers with one more possibility for describing some aspect of a program or system.

In fact, we have found it useful in developing 3D computation visualizations to characterize how each of the four dimensions (x, y, z, and time) are being utilized. Below we list the uses of dimensions common in the animations we have developed.

- **Value** - A dimension or axis is being used to encode values of computational elements such as variables. Typically, a scaling factor or mapping is established.
- **Positional** - A dimension is being used to encode position or index within a structure, e.g., processor number, array index, etc.
- **History** - A dimension is being used to encode a history of a computation. For example, a visualization might show which messages were sent at each step of an execution by using a timeline graphical artifact.
- **State** - A dimension is being used to encode the instantaneous state of a computation. Typically, the time dimension fits this role.
- **Aesthetics** - A dimension is being used for no real purpose but to improve or refine the appearance of an animation.

These dimensions help make it possible to see how 3D imagery can be utilized by the designers of computation visualizations. For instance, consider a program that manipulates a two-dimensional matrix of values, an extremely common occurrence. Two of the spatial dimensions of the visualization can be used to encode positional information, the third spatial dimension can encode value, and time can reflect the program state. Even a program manipulating a list (1D) of elements may have two relevant attributes to be shown, such as value and most recent modification time. Here, the three spatial dimensions are beneficial again.

4 Types of 3D Computation Visualizations

The 3D computation visualizations that we have developed fall into three categories which are useful for characterizing the visualizations. They are

1. Augmented 2D views – This category includes visualizations of computations whose information display requires only two spatial dimensions (e.g., position and value) but a third spatial dimension is added for aesthetic or presentation purposes. For example, a traditional 2D bar chart sorting view can be transformed into a 3D rectangular solid view by adding constant depth to the bars.[2] Figure 4 illustrates this type of computation visualization.
2. Adapted 2D views – This category again involves computations minimally requiring two spatial dimensions. Here, however, the third spatial dimension encodes some other value or attribute of the computation and its data. For example, rather than add constant depth to the bar chart sort above, the third dimension can be used to show a history of the pairs of elements exchanged at each step of the program. Figure 5 illustrates this type of computation visualization.
3. Inherent 3D application domain views – This category includes computations involving inherent three-dimensional entities such as visualizations of non-planar graph algorithms, volume packing rather than bin packing, and a visualization of a cube parallel architecture. Figures 6 and 7 illustrate computation visualizations of this type.

[2] It has been suggested that this category be labeled *USA Today* chart views.

Of the categories just described, category 3 is a natural in the sense that it is clear how 3D imagery will be utilized. Category 2 promises to involve innovative new views of some tried and true algorithms and programs. But it is debatable whether there is utility in category 1. Tufte believes that when two dimensions are sufficient to portray information, adding a third dimension can be detrimental to the view[Tuf83]. Others disagree and believe that 3D imagery can provide important cognitive cues for the human visual system. Spence found that people can process information from 3D displays such as charts more quickly and just as accurately as from two-dimensional displays[Spe90]. He speculated that the attractiveness of the 3D displays may be an important influence in this result.

Nonetheless, the capability of 3D computation display is one whose time has come. Hardware capabilities are advancing at rates to make this type of display practical in the near future. It is not difficult to imagine a time when 3D imagery is the norm rather than the exception on desktop computers.

5 Visualization System

To support the development of 3D computation visualizations we have designed and created an animation methodology and toolkit/class library. Our system[3] is called Polka-3D, and its primary focus is on supporting 3D animation development by programmers who do not have an in-depth knowledge of 3D graphics techniques. Rather, programmers must only position graphical objects in a three-dimensional world coordinate space and use system primitives to evoke action and motion in these objects.

The system is general purpose, that is, it does not utilize predefined views. Rather, it supports the development of *application-specific* computation views. One other style of system presents a particular visualization style for all program automatically. Lieberman has presented an innovative 3D program display system, primarily showing Lisp code, that uses color and depth to show recursion and history[Lie89].

The Cognitive Coprocessor architecture[RCM89] supports 3D information and user interface displays which could be used to visualize computations. This system gives users access to predefined, highly tailored views. Our work differs in that our primary purpose is to allow external programmers to develop animations on their own using the system.

Creating a computation visualization with Polka-3D is a two-part process (they can be done concurrently). The first part involves identifying the computation abstractions to be visualized. Currently, we utilize an event-based method—programmers identify and insert named, parameterized events into their application program. The events are implemented as procedure calls to an animation control routine. The second part of the process is animation development.

Polka-3D is an object-oriented animation methodology. It contains classes that model 1) an entire animation 2) individual views of or windows onto the animation 3) the entities which help define a view such as graphical objects and action or motions. Figure 2 depicts the different classes used in a computation visualization. Let us examine each of these

[3]We use the word "system" in a broad sense here. We have not created an integrated, monolithic visualization system.

Figure 2: The different types of classes involved in a POLKA-3D animation. The levels in the hierarchy illustrate the "has-a" relationships.

components in more detail.

An entire computation visualization is encapsulated in an *Animator* class. The Animator base class minimally defines a routine that receives and handles events from the program or system being visualized. Animation developers subclass Animator, adding individual animation views as data members and a mapping routine that specifies which view(s) are called in response to program events.

The second level of animation development involves the different visual appearances or *Views* of a computation. For example, in animating a graph algorithm, one might construct a vertex-edge view and an adjacency matrix view. Views must contain individual *Scenes* (logical activities or routines into which an animation is partitioned) and the View data members, such as graphical objects and motions, manipulated in the scenes. The graph algorithm vertex-edge view example might include, for instance, scenes for displaying a graph, visiting nodes, altering the appearance of a node, etc., and it might include data members such as the spheres and lines making up the graph. Each view always contains (inherited from View base class) an integer *time* variable that contains the current time or frame count of the view, and an important method called *Animate* we will describe later.

To design animation imagery and action within a View, we wanted a method which would not require a background in 3D graphics. Basically, we wanted to hide or provide useful defaults for as much of the 3D specification as possible, and allow developers to simply position and move objects within a 3D coordinate system. In computation visualizations, most objects' actions or motions occur with respect to other objects.

The animation methodology we have developed is a combination of a simple path-based method we created for 2D algorithm animation[Sta90] and the more sophisticated frame-based animation clock methods of high-end 3D graphics systems. Programmers design views using objects from *AnimObject*, *Location*, and *Action* classes.

The AnimObject class provides a simple object modeling capability. It has subclasses for particular types of 2D and 3D objects such as lines, circles, rectangles, blocks, spheres, text, etc. These subclasses correspond to the object types in the Requirements section earlier in the paper.

A Location object is simply a user-placed marker or "positional buoy" sitting somewhere within the 3D coordinate system. Locations are used to reference AnimObject positions, and they frequently mark object creation points or end-points of motions.

Actions encapsulate changes to AnimObjects, such as changes in position, size, color, visibility, etc. Actions simply are three-dimensional paths (sequences of control points) with a type attribute. When an Action's type is "MOVE" or "RESIZE," the control points correspond to animation coordinate system values. If an Action's type is "COLOR," the control points correspond to an RGB color space, and so on.

AnimObjects have an important member function, *Program*, with which animation developers associate or bind Actions to AnimObjects. *Program* has two parameters: the Action to be used, and an integer time parameter that specifies the View time at which the Action should commence.

The View method *Animate* actually generates animation frames and advances the *time* member variable. Therefore, animations in Polka-3D typically consist of scenes that specify or designate Actions to occur and a simple scene that advances the animation.

Polka-3D provides useful defaults for some 3D viewing attributes such as surface properties and a lighting model so that programmers need not specify their values. The hooks for changing these attributes are included for advanced programmers, however. Other fundamental attributes such as eye position can be changed via dials and sliders during runtime of an animation.

Polka-3D also has a number of other interesting features. AnimObjects can be created not only individually, but as rows, grids, or meshes. As discussed in the earlier Requirements section, this capability is extremely valuable for computation visualizations. When a developer specifies a composite structure of AnimObjects, s/he must designate the axes to be used as positional, value, aesthetic, etc., dimensions. The system then creates the individual objects, positioning and scaling elements as necessary.

Polka-3D has a special Eye or Viewer object that controls the position and direction of the viewpoint. This feature allows designers to create animations that move beyond passive detached views of program executions evident in all existing systems. Now, viewers can be brought into or actively involved with the computation they are observing. Rather than simply watch a graph search, for instance, animations can be created in which the viewer "experiences" the search first-hand, traversing the graph in a very simplistic virtual reality fashion.

Implementation

Polka-3D is implemented in C++ on SGI workstations using the GL graphics library. The code fragment in Figure 3 provides an example of how end user-created Polka-3D animation design code appears. In this sample, we create a sphere AnimObject and move it to another

```
void
MyView::Scene1()
{
   Location *from, *to;
   int len;
   Color c("red");

           // 1. Construct AnimObject and Location
   Sphere *s = new Sphere(this, 1, 0.2,0.2,0.2, 0.3, c );
                // Parameters are View, visibility, xpos, ypos, zpos, radius, color
   to = new Loc(0.8, 0.8, 0.8);
                // Parameters are x, y, z

           // 2. Retrieve a loc corresponding to the center of the sphere
   from = s->Where(CENTER);
           // 3. Construct a straight interpolated movement between _from_ and _to_
   Action a("MOVE", from, to, STRAIGHT);
           // 4. Program the action into the sphere starting at frame _time_
           //    Return value is length in frames of the action
   len = s->Program(time, &a);
           // 5. Animate for that number of frames, return new _time_
   time = Animate(time, len);
                // _time_ is a View member variable
}
```

Figure 3: Small Polka-3D code segment that creates a sphere and moves it to another location.

position (Location object).

Example Animations

In this section we describe a few computation visualizations created with Polka-3D and representative of the three classes of animations described earlier. Still-frame figures from the animations are included at the end of this paper, and a videotape showing these animations and others accompanies too.

The first example illustrates an inherently 2D algorithm to be animated, bubblesort, in which we have added constant depth to the prototypical bar chart or block view common in so many algorithm animation systems. Figure 4 shows a frame from this animation.

As preparation for the second animation to be shown, consider another sorting algorithm view, a "scatter-plot" style view, in which two display dimensions are sufficient (position and value). The view shows the elements to be sorted as dots; horizontal position represents index into an array and vertical position indicates relative values of the elements. A completely sorted array is therefore shown as a diagonal line from the lower left to the upper right.

The next example animation is a member of the second category identified earlier, an

adapted 2D view. We have taken the simple 2D scatter plot view described above and used the depth or z-dimension to illustrate history. Each time that two elements are exchanged in the sort, we draw a plane (each plane is drawn slightly behind the previous one, the first is directly behind the scatter dots) using the two dots to specify opposite corners along a diagonal in the plane. Figure 5 shows this view, rotated around the y-axis. The dots are the small blue boxes toward the right edge of the figure. The multi-colored "exchange" planes depict a visual history of the indices of elements exchanged in the sort.

Many other examples of adapted 2D views are possible. Consider planar graph algorithms in which values are associated with each vertex. Drawing both a representation of a graph vertex and its value at the vertex may lead to a cluttered view. But if we encode the vertex values as graphical objects that project out in the z-dimension, a much more comprehensible view would result.

Finally, Figures 6 and 7 show two computations involving inherent 3D data. Figure 6 shows a frame from a depth-first search animation on a graph embedded in 3D space. Figure 7 shows an animation of a simple particle chamber simulation. In this animation, particles move smoothly about the chamber, colliding with walls, and changing shape, velocity and color. Less than 100 lines of Polka-3D code was required to implement this view.

6 Conclusion

We have presented a system for creating 3D visualizations and animations of programs, algorithms, and computations. Although computation visualization is the domain on which our research focuses, Polka-3D is a straightforward, general purpose 3D animation methodology that could be used to build many other types of information visualizations and animations. The emphasis on simple object descriptions and action primitives, along with useful defaults for 3D viewing parameters, helps make the methodology accessible to many programmers who are not graphics experts. As such, Polka-3D is one of the first systems to bring 3D graphics capabilities to many programmers who desire 3D views, but who do not wish to spend a great deal of time learning 3D techniques and tools.

We also have identified the value of 3D graphics for presenting views of computations, and we have characterized how the third spatial dimension can be utilized. These different usages result in different classes of visualizations. Finally, we presented a few example computation visualizations created with Polka-3D.

In future work, we plan to examine how well 3D visualizations help people understand computations. For example, it would be interesting to discover if augmented 2D views such as the 3D bar chart sorting view are better at conveying a program's methodologies than 2D views.

We also plan to create more canonical program views so that they can be generated automatically by a variety of software engineering tools. These types of views can be helpful for program tracing, debugging, and profiling.

Acknowledgments

Joe Wehrli helped port and implement Polka-3D. His comments and discussions were instrumental in realizing this work.

References

[Bro87] Frederick P. Brooks. No silver bullet, Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[Bro88] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[CRM91] Stuart K. Card, George G. Robertson, and Jock Mackinlay. The Information Visualizer, an information workspace. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 181–188, New Orleans, LA, May 1991.

[Lie89] Henry Lieberman. A three-dimensional representation for program execution. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 111–116, Rome, Italy, October 1989.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume II, pages 597–606, Kauai, HI, January 1992.

[RCM89] George G. Robertson, Stuart K. Card, and Jock Mackinlay. The Cognitive Coprocessor architecture for intercative user interfaces. In *Proceedings of the ACM SIGGRAPH '89 Symposium on User Interface Software and Technology*, pages 10–18, Williamsburg, VA, November 1989.

[RMC91] George G. Robertson, Jock Mackinlay, and Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 189–194, New Orleans, LA, May 1991.

[Spe90] Ian Spence. Visual psychophysics of simple graphical elements. *Journal of Experimental Psychology: Human Perception and Performance*, 16(4):683–692, 1990.

[Sta90] John T. Stasko. The Path-Transition Paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.

[Tuf83] E. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT, 1983.

[vD91] Andries van Dam. Escaping Flatland. Plenary Address, UIST '91, November 1991. Hilton Head, SC.

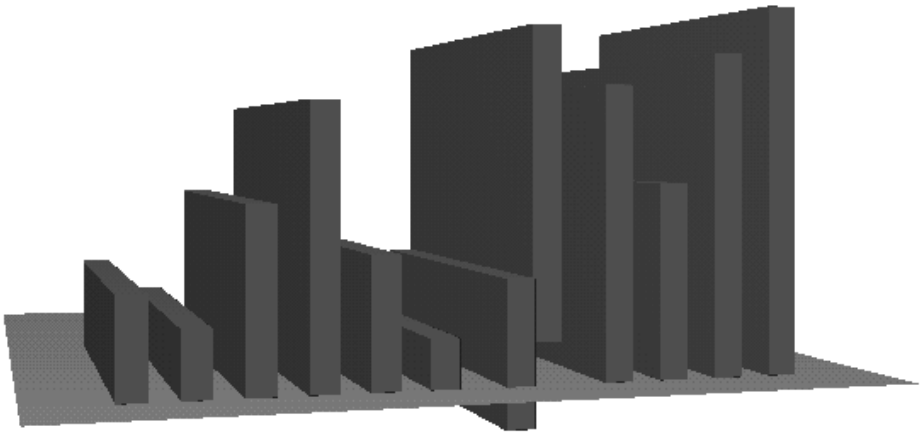

Figure 4: An augmented 2D bar chart view of a bubblesort. Constant depth is added to each array element.

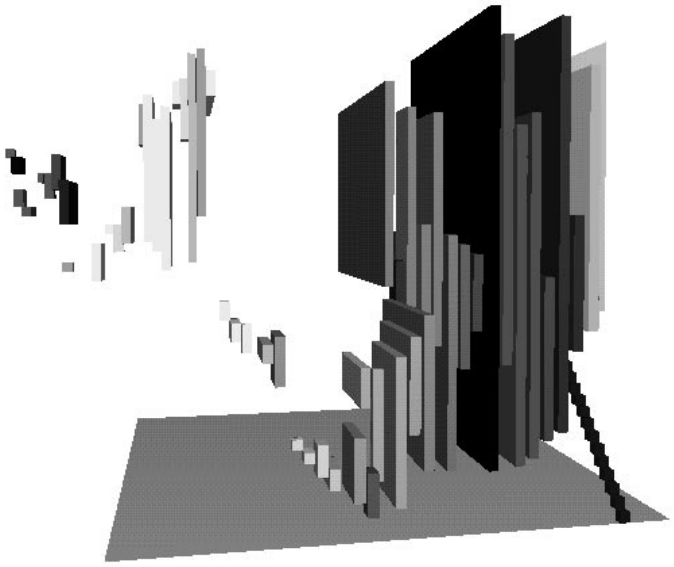

Figure 5: A scatter plot view of quicksort. The dots to the right indicate array values and the planes to the left detail a history of the exchanges performed.

Figure 6: A frame from a depth-first search of a 3D embedded graph. (Figure missing)

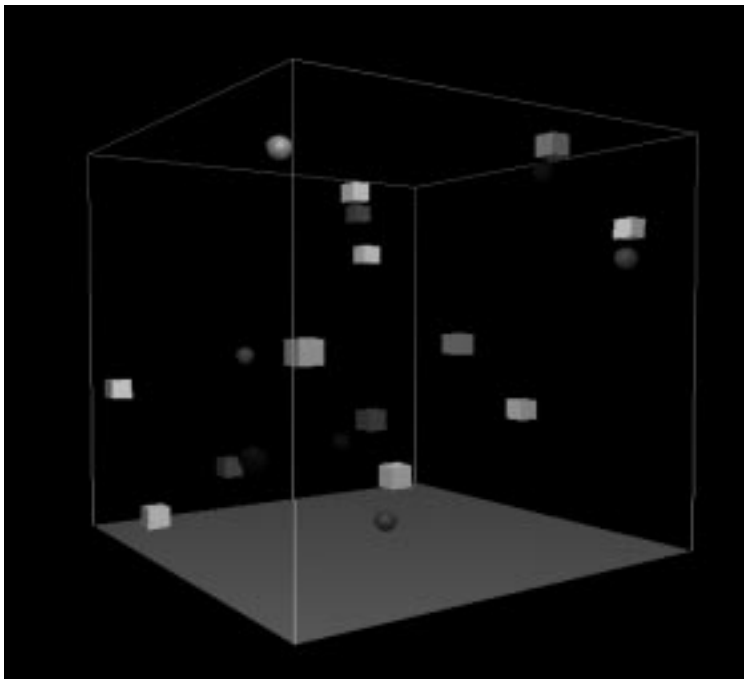

Figure 7: A frame from a three-dimensional particle simulation.