Visualization Viewpoints

Editor: Theresa-Marie Rhyne

Visual Debugging

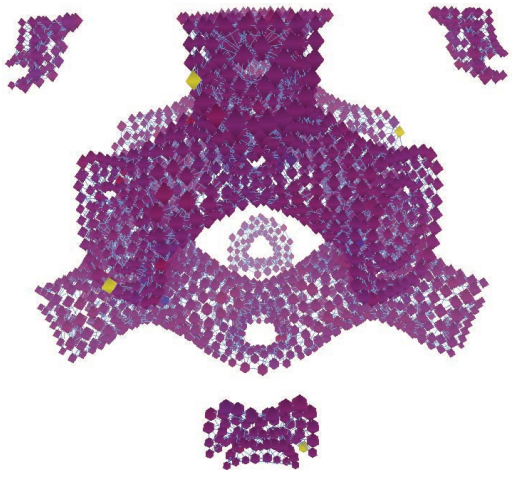
Patricia Crossno and David H. Rogers

Sandia National Laboratories

Our understanding of the world is based on mental models of objects and processes and their relationships. We think about the world using abstractions that we must then organize into a coherent whole. When working with complex, dynamically changing systems, visual representations, such as drawings, graphs, images, or animations, help us gain insight by seeing patterns that wouldn't be discernable in a numeric representation. This is the essence of scientific visualization. We use similar ideas as part of the debugging process to understand the complexities and dynamics of software and hardware systems.

Although graphics and visualization programmers already use a simple form of visual debugging to evaluate their code when they view their output, debugging is generally done using a conventional alphanumeric debugger. This works well in many situations, but there are some types of problems that can be more effectively solved using visual debugging tools.

For example, imagine that you're debugging the energy minimization of a particle system in which each particle's energy is based on its location in space and the number of nearby particles. This particle system grows to thousands of particles and you want to know each particle's energy at any time interval over hundreds of cycles. Two factors make a conventional debugger useless: the large number of particles and the need to correlate the information spatially. So, what do you do?



1 **A particle system with 3,798 particles. Because of color-coding, we can quickly locate four high-energy particles (shown in yellow) correlated with their positions, relationships to each other (links are drawn in cyan), and repulsive forces (indicated by their sizes).**

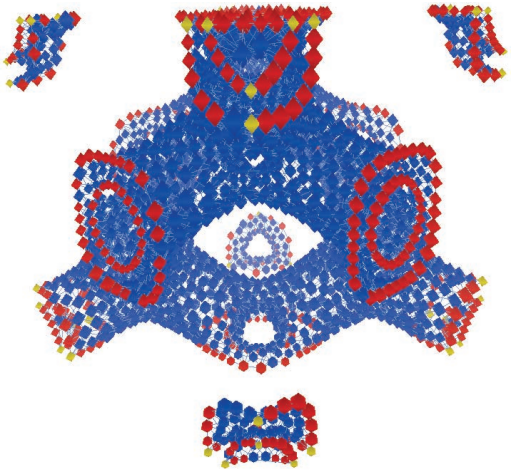
Figure 1 shows our solution to this problem. We developed an alternate approach that uses our innate visual pattern recognition skills as part of the debugging process.[1] Inspired by Huang's[2] use of color to visualize energy distributions while untangling knots, we represented the particles graphically and color-coded them by energy value. Thus far, we've applied this approach to three domains: particle systems, cluster hardware configurations, and physics codes using finite element models. This debugging paradigm differs from software or program visualization in that we don't visualize software elements such as procedures, message passing between processors, or graph-based representations of data structures (that is to say, boxes and arrows for linked-list structures or binary trees).

In most application domains developers that use algorithm visualization tools must make decisions about what kind of visualization would best represent their code, and they must, in effect, code this visualization in addition to their application. For many developers, the time investment is too great compared to their perceived benefit, so they return to a traditional debugging approach.[3] Like Tal and Dobkin,[4] we believe that restricting the application domain increases the ease of use of visual debuggers. However, we go one step further by creating a visual tool tailored to a particular application domain that can use either captured data or simulation outputs and requires no coding effort on the part of the user.

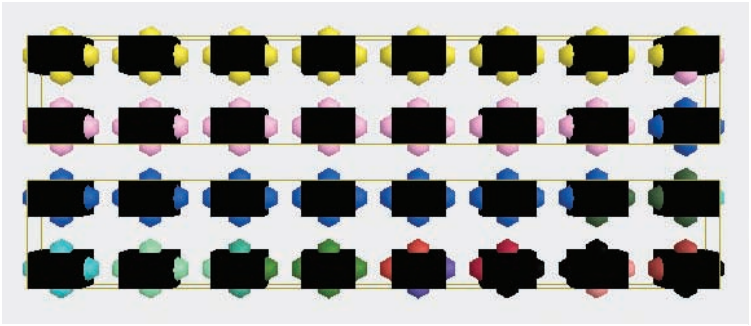
Model components

Despite many programming strategies to reduce complexity, we still find ourselves faced with software that doesn't work as expected. For problems that involve complex interactions between many model pieces or components (such as the particles in Figure 1), correlating multiple attributes (for example, energy, location, number of neighbors) and understanding their evolution over time is often key to tracking down subtle logical errors in the system. Looking at any one attribute or piece of the model in isolation is usually insufficient.

The representation of these model components



2 Particles are colored by type. Red edge particles move on the curve. Yellow corner particles are fixed. Blue surface particles float over the surface. Blue particles on the top and bottom edges of the lowest cluster reveal errors.



3 Model of cluster switches and processor ports for a cluster of 128 processors. Port color represents the job running on that processor. Black ports are idle.

depends on the application. Because each application is based on a domain-specific abstraction, a visual representation that closely matches that abstraction reduces developers' difficulty in translating between the visualization and their internalized representation.

In each of the domains we've worked with, we've focused our visualizations on simple representations of the systems' key conceptual components. For the particle system, we drew the particles as spheres and the nearest-neighbor links as lines between particles. In the cluster hardware application, we drew network switches as cubes, switch ports as spheres on the cubes, and network links as lines connecting the ports. In our most recent work with finite element codes, we depict finite elements as hexahedral cells and their nodes as spheres on the corners of the cells. Additional detail added on top of the central basis components provides a richer representation of the system in each application.

Attributes

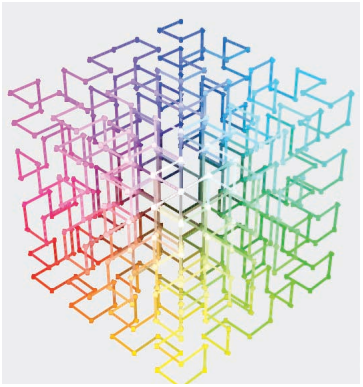
Although the components in each application's model differ, visualizing component attributes has a common theme: components are drawn using various color-coding schemes to show attribute values. This approach is powerful because the user can simultaneously see relationships between multiple aspects of the system. Visualization of proximity relationships in combination with values, especially when animated through time, can increase the likelihood of discovering patterns or anomalies. This is especially true when important features are highlighted in bright colors, while less important components are displayed using dark colors or are culled from the image.

Interactivity plays an important role in viewing attributes, not only in switching between attributes, but also by specifying the range of values that should be highlighted. In both the particle system and the cluster hardware applications, the user inputs high and low values that represent the expected range of normal values for a particular attribute. The components whose values fall within this range are color-coded between blue and red with colors gradually transitioning from blue at the low end, to purple in the middle, to red at the high end. Components with values above the high end are typically drawn in yellow, while those with values below the low end are highlighted in green or drawn in black. Figure 1 shows how this color-coding scheme displays the energy values for each particle. The user can interactively query component values by adjusting the range. Furthermore, the colors used for highlighting the outlier values should be user definable to aid color-blind users in selecting colors that they can distinguish.

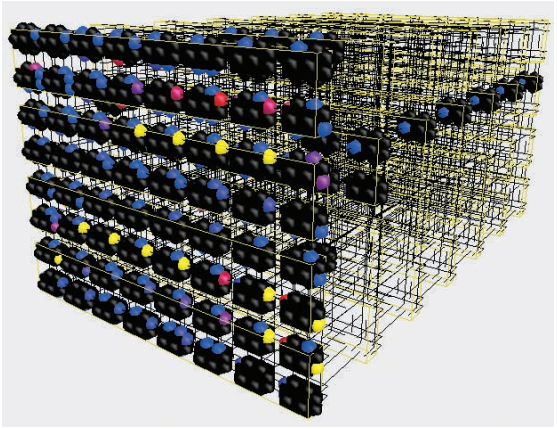
We can effectively color-code attributes in several ways. If an attribute's value varies continuously, the coloring scheme previously mentioned works well because 'hotspots' or other oddities stand out against a common background. For other types of attributes, this smooth shading approach isn't useful. For example, the components that we study often have an attribute that identifies the processor on which each was computed (this is a useful attribute when studying parallel simulations). When viewing data like these, it's useful to see distinct groups, so we need contrasting colors. Figure 2 shows an example of this type of color encoding in which three types of particles are shown by highly contrasting colors. If there are many groups to be represented, it may be difficult to find unique and visually distinct colors for this type of visualization.

For the cluster hardware application shown in Figure 3, coloring processor ports by job identifier was challenging. Potentially, there can be a large number of jobs, and the colors must be visually distinct. We found that randomly selecting RGB component values frequently led to similar colors being assigned to different jobs. Our solution was to create a 3D Hilbert curve through the RGB color cube and to randomly assign the evenly distributed vertices (shown as thickened points along the curve in Figure 4, next page) to jobs.

When we started to apply these techniques to finite element simulations, we had to revise our color-coding scheme to fit with what physicists expected. Physicists typically use a "Roy G. Biv" rainbow spectrum to visualize value ranges, so the red to blue depiction of the range didn't make sense to them. To remember the spectral color order, the "Roy G. Biv" mnemonic provides the first



4 Three-dimensional Hilbert curve through RGB color cube.



5 The model is a 1,792-processor cluster. Boxes are switches; spheres are network ports colored by error counts. Switches with no errors have been culled.

letters of the colors in order (red, orange, yellow, green, blue, indigo, and violet). This type of coloring is now an option in our software.

We represent attributes that are vector quantities with scalable arrows. These can then be combined with scalar attributes in the same image. Although this increases the information content of the visualization, these arrows can be interactively switched on and off so that the user can control the visual clutter on the screen. Visual clutter is a significant issue in presenting information. On the one hand, you want to show multiple attributes simultaneously so the user can see connections and interactions. On the other hand, once a certain threshold of visual overload is surpassed, users can't make sense of what they're seeing.

Applications

In applying our visual debugging approach, we've created applications targeted to three diverse problem areas. The common theme uniting these problems is that they all involve evaluating very large numbers of variable values that are correlated in a spatial domain and changing over time. The three applications we've targeted are particle systems, switching hardware for clusters of PCs, and physics simulation codes that use finite elements.

Particle systems

We first used visual debugging while developing a particle system to find isosurfaces in volumetric data.[5] The central component in this application was the particle. There were three types of particles, each of which could move in different ways. During visual debugging, these particles were drawn as color-coded spheres as shown in Figure 2. Attributes colored by their value within a range included energy (shown in Figure 1), surface curvature, size (repulsive range), and age. The repulsive range was also indicated by the particle's size. The normal to the isosurface at the particle's location was displayed using a vector, and particles that were 'nearest neighbors' were connected with a line. Each of these attributes was used to understand and debug particle movements, repulsive and attractive force calculations, and particle population. By understanding the details of how the particles behaved, the programmer could make useful changes to the code. In addition, these attributes were used to evaluate the system's convergence to determine when the isosurface was good enough for the algorithm to terminate.

PC cluster hardware

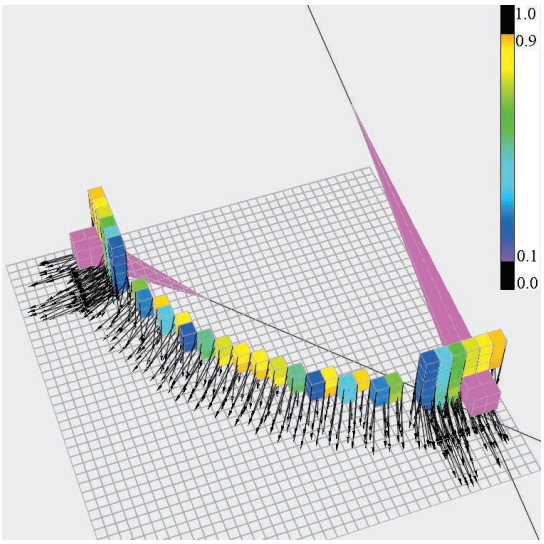
We next applied these techniques to understand an unusually high message failure rate in a cluster of PCs.[6] Researchers were looking for a way to understand how the cluster was behaving, so we modeled its switch hardware and let them look for patterns in the distribution of errors (bad packets, bad routes, dead routes, and timeouts). Switches, ports, and network cables formed the model's components. By coloring the ports according to their error counts and then correlating ports experiencing high error counts over time with both routing and job information, we traced part of the problem back to errors generated and propagated during rebooting subsections of the cluster.

Later, we expanded our model definitions to include multiple planes and crossbar switches. Figure 5 is an example of how large the models have now become. Keeping up with the increasing complexity of the model definitions has been a major obstacle. For the system to remain useful, it must accurately reflect true hardware configurations. Consequently, we removed any assumptions about regularity in port definitions or layout, and virtually all aspects of the model are configurable. We're currently using the cluster tool to evaluate scheduling and processor allocation strategies.

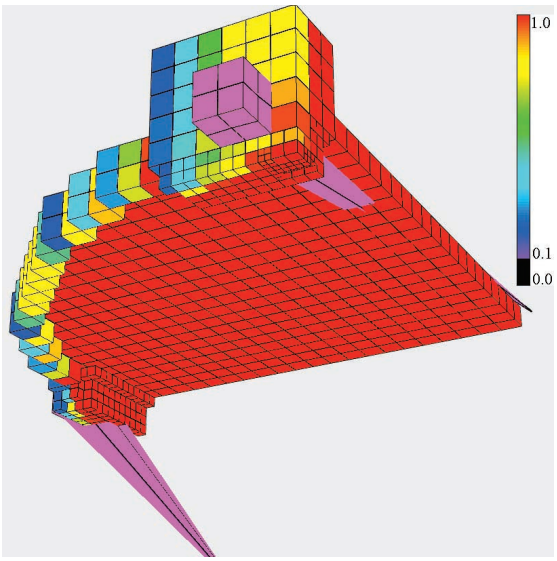
Finite element codes

Our current work is in the area of debugging physics codes that use finite element models.[7] In this context, the central components are elements (or cells) and nodes. Although finite element models can have a variety of cell types, we've concentrated on hexahedral elements. We're now starting to work with adaptive mesh refinement (AMR) grids.

Finite element models can contain millions of elements, so interior elements are obscured by exterior elements. This means that we can't view all the cells simultaneously. At this scale, the visual debugging approach breaks down, and it becomes useful to automatically search for features in each model. In the case of the physics codes, elements whose shapes are highly distorted signal the source of a problem, so these cells



6 Cells extracted from a simulation of a rod impacting a plate include two clusters of inverted elements (in pink). The cells are on the boundary where the rod (a quarter-circle) meets the plate. Vectors display velocity at each node.



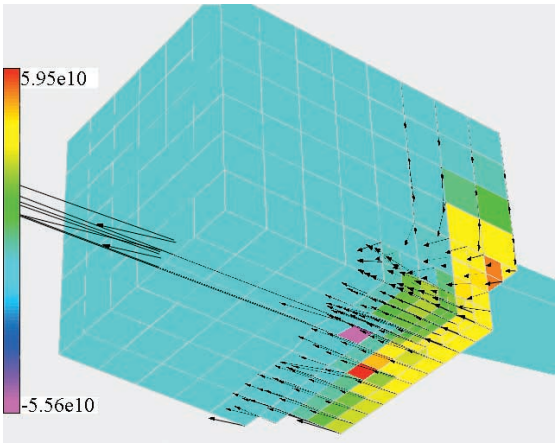
7 Elements are colored by the percentage of rod material, with red as high and blue as low. Pink indicates no rod material. The grid subdivision is directly below the inverted elements.

are extracted. To provide context when viewing these bad cells, we also extract a clump of cells around them and planes of cells that pass through them. These subsets are then viewed using color-coding to display attribute values such as pressure or the percentage of a given material in the cell. In AMR grids there's an attribute that measures the simulation's need for the cell to subdivide. Vector arrows show quantities such as the velocity or acceleration at a node.

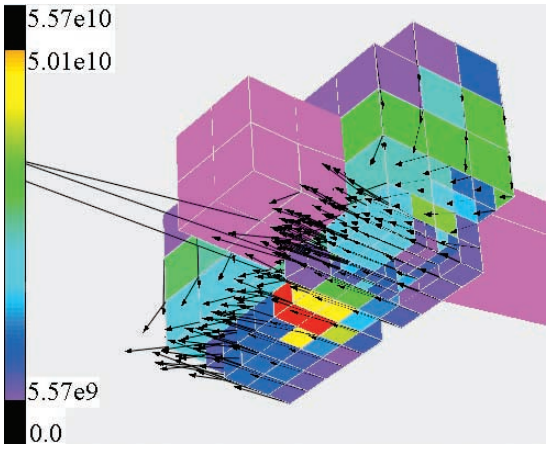
Figures 6 through 9 are views of an AMR grid subset from a simulation of a rod smashing into a plate. The rod is represented as a cylindrical quarter section. The region where the rod impacts the plate has two levels of refinement. Normally, we expect all cells to be box-shaped, but the bad cells are distorted because at least one vertex has moved unexpectedly. We searched the full model for bad cells and found two clusters of them, which are shown in pink in Figure 6. When that central node is incorrect, all eight cells sharing that node are distorted. That's what has happened in both the clusters found by the automated search. Some of the distorted elements have negative volumes since the cell faces are interpenetrating. These are inverted elements.

In Figure 6, each node's velocity vector is shown with a black arrow. Note that the velocity vectors originating from the displaced nodes are anomalous compared to the regular pattern of the other vectors.

Figure 7 shows the underside of the same data. The velocity vectors have been removed, and the culling operation has been changed to display both the inverted elements and those elements having some amount of the rod material. The percentage of the element containing the rod material is shown using the spectrum coloration described earlier, with red showing high concentrations and blue showing low concentrations. The



8 Elements colored by pressure. The pink and red elements, representing zero and high-pressure values, are anomalous with respect to the pressure values of their neighbors.



9 Elements are colored by their need to subdivide. Red marks a hotspot where cells need more refinement.

pink of the inverted clusters indicates that there's virtually no rod material in any of those elements.

Three interesting features are immediately apparent in Figure 7. First, the rod material is unevenly distributed in the area to the left of and below the inverted cells (there are orange and red elements where we would expect yellow elements). Second, the inverted elements have no rod material within them. Third, cell subdivision begins in the layer of elements immediately beneath the inverted elements.

Figure 8 shows a closer view of the inverted cells in Figure 7. This time the cells are colored using the pressure attribute. The contextual plane of cells has been removed and velocity vectors are displayed. The proximity of the pink (zero-pressure) and red (highest-pressure) elements in the bottom layer is anomalous and is indicative of the problem in this region. In Figure 9 the same cells are colored according to their need to subdivide. Elements with low values have been culled except for the inverted elements. Note that a hotspot appears in the same area as in Figures 7 and 8.

These features lead us to believe that the problem was created by the incorrect propagation of a velocity vector from a node whose elements contained the material (and hence had a velocity) to a node shared by elements outside the rod. This propagation appears to have happened during the grid subdivision. The combination of the velocity with elements of essentially zero mass led to the inversion of the elements. Although the inverted cell was anomalous, it was the context around that cell that provided the indications of where the problem existed in the code.

Conclusions

Visual debugging isn't a novel idea. Scientists use visualization indirectly to find bugs when simulation results are rendered. However, scientific visualization techniques hinder the debugging process because they hide the underlying data structures used to generate the visualization. For instance, isosurfaces and direct renderings of volumetric data often create the illusion that the data are continuous rather than sampled. These techniques suppress the cell or processor boundaries. But for debugging purposes, the structures should be explicitly shown, and the developer should be able to interactively query structures for detailed information about their contents.

We've shown that visual debugging provides insights into certain types of problems that aren't available through conventional debugging approaches. Although the components used by each of the described applications differ, they all represent models using simple abstractions, such as boxes, spheres, and lines. It's essential to provide these simple abstractions while retaining the key details within them. Abstracting the model lets users focus on seeing patterns and anomalies in the attributes of the model without being distracted by unimportant or unintentional features.

We're continuing to explore techniques for abstracting attributes. In particular, we need to find an intuitive way to represent tensor quantities. As we represent more complex quantities, it becomes more difficult to design useful representations. ■

Acknowledgments

We want to thank Rena Haynes, Eric Russell, Vitus Leung, and Rachel Rubin for their work on the switch hardware application. Thanks to Daniel Carroll for the finite element data and help on the Visual Tools project. The Department of Energy's Mathematics, Information, and Computer Science Office funded this research. The work was performed at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

References

1. P. Crossno and E. Angel, "Visual Debugging of Visualization Software: A Case Study for Particle Systems," *Proc. Visualization 1999*, ACM Press, New York, 1999, pp. 417-420.
2. M. Huang et al., "Untangling Knots by Stochastic Energy Optimization," *Proc. Visualization 1996*, ACM Press, New York, 1996, pp. 279-286.
3. S. Mukherjea and J. Stasko, "Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger," *ACM Transactions on Computer-Human Interaction*, vol. 1 no. 3, Sept. 1994, pp. 215-244.
4. A. Tal and D. Dobkin, "Visualization of Geometric Algorithms," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, June 1995, pp. 194-204.
5. P. Crossno and E. Angel, "Isosurface Extraction Using Particle Systems," *Proc. Visualization 1997*, ACM Press, New York, 1997, pp.495-498.
6. P. Crossno and R. Haynes, "Case Study: Visual Debugging of Cluster Hardware," *Proc. Visualization 2001,* ACM Press, New York, 2001, pp. 429-432.
7. P. Crossno, D.H. Rogers, and C.J. Garasi, "Case Study: Visual Debugging of Finite Element Codes," to appear in *Proc. Visualization 2002*, ACM Press, New York, 2002.

Readers may contact the authors by emailing Patricia Crossno at pjcross@sandia.gov or David H. Rogers at dhroger@sandia.gov.

Readers may contact department editor Theresa-Marie Rhyne by email at tmrhyne@ncsu.edu.