LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-411205

Mercury + VisIt: Integration of a Real-Time Graphical Analysis Capability into a Monte Carlo Transport Code

M. J. O'Brien, R. J. Procassini, K. I. Joy

March 11, 2009

International Conference on Mathematics, Computational Methods & Reactor Physics
Saratoga Springs, NY, United States
May 3, 2009 through May 7, 2009

Disclaimer

International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)
Saratoga Springs, New York, May 3-7, 2009, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2009)

MERCURY + VISIT: INTEGRATION OF A REAL-TIME GRAPHICAL ANALYSIS CAPABILITY INTO A MONTE CARLO TRANSPORT CODE

Matthew O'Brien and Richard Procassini
Lawrence Livermore National Laboratory
Mail Stop L-95, P.O. Box 808
Livermore, CA 94551
United States of America
mobrien@llnl.gov and spike@llnl.gov

Kenneth Joy
Computer Science Department
University of California at Davis
Davis, CA 95616
United States of America
kijoy@ucdavis.edu

ABSTRACT

Validation of the problem definition and analysis of the results (tallies) produced during a Monte Carlo particle transport calculation can be a complicated, time-intensive processes. The time required for a person to create an accurate, validated combinatorial geometry (CG) or mesh-based representation of a complex problem, free of common errors such as gaps and overlapping cells, can range from days to weeks. The ability to interrogate the internal structure of a complex, three-dimensional (3-D) geometry, prior to running the transport calculation, can improve the user's confidence in the validity of the problem definition. With regard to the analysis of results, the process of extracting tally data from printed tables within a file is laborious and not an intuitive approach to understanding the results. The ability to display tally information overlaid on top of the problem geometry can decrease the time required for analysis and increase the user's understanding of the results. To this end, our team has integrated ***VisIt***, a parallel, production-quality visualization and data analysis tool into ***Mercury***, a massively-parallel Monte Carlo particle transport code. ***VisIt*** provides an API for real time visualization of a simulation as it is running. The user may select which plots to display from the ***VisIt*** GUI, or by sending ***VisIt*** a ***Python*** script from ***Mercury***. The frequency at which plots are updated can be set and the user can visualize the simulation results as it is running.

Key Words: Visualization, Geometry Validation, Combinatorial Geometry, Monte Carlo Transport.

1 INTRODUCTION

Rather than "reinventing the wheel" by developing custom software to visualize the results from our Monte Carlo particle transport simulations, our team has chosen to use ***VisIt*** [1], an existing scientific visualization and data analysis tool. For years, the ***Mercury*** Monte Carlo particle transport code [2],[3] had the ability to write restart and graphics files that could be opened by ***VisIt*** for post-processing visualization. Recently, our team has connected ***Mercury*** and ***VisIt*** via in-memory data transfers. This has been achieved through the use of ***VisIt*** application program-

ming interface (API) function calls [4] which provide ***VisIt*** with the data that it needs for plotting, based on user requests.

VisIt is capable of visualizing domain decomposed mesh based data on structured and unstructured meshes. This feature is used to visualize mesh-based geometry from ***Mercury***. ***VisIt*** can also visualize "point" based data, a feature which is used to visualize particle-based data from ***Mercury***. A relatively new feature in ***VisIt*** is the ability to discretize and visualize constructive solid geometry (CSG) data, which is also known as combinatorial geometry (CG) data. The user provides ***VisIt*** the coefficients of the surfaces that define the cells, as well as how the surfaces are combined together to form cells, and ***VisIt*** will automatically discretize and visualize the CG data. Before ***VisIt*** had this capability, the user would direct ***Mercury*** to convert its internal CG data into mesh based data by overlaying the CG onto a graphics mesh and sampling the CG at the mesh points. If a mesh cell intersected multiple CG cells, a "mixed" mesh cell is created that contains the volume fractions of the partial CG cells. This mesh sampling algorithm will be described later in more detail.

Mercury has recently been extended to support an interactive ***Python*** interface. This allows the user to issue the "*visit()*" command to launch ***VisIt*** and connect it to the running ***Mercury*** simulation. Once this connection has been made, the user may select various plots from the ***VisIt*** graphical user interface (GUI). Alternatively, the user may request plots directly from the ***Python*** interface to ***Mercury*** by feeding ***VisIt*** a ***Python*** script [5], such as "*visit('myScript.py')*". This will send the contents of the ***Python*** script *'myScript.py'* to ***VisIt*** for visualization. As part of the ***Mercury*** input, the user may also set the frequency at which the ***VisIt*** plots are updated, even as the simulation is running.

2 TYPES OF PLOTS SUPPORTED BY *VISIT*

VisIt supports many different types of plots, including mesh, material, pseudocolor, domain, curve, histogram, contour, particle, vector, label and volume rendering. It also has many operators that can be applied to these plots, such as clipping and material subselection. Material subselection is one of the most useful operators in ***VisIt***. It allows the user to enable / disable visualization of any material in the problem. This allows the user to focus in on exactly what they are looking for, while hiding what they are not interested in.

In this section, the various plots that are available in ***VisIt*** will be demonstrated, as they are applied to the **"**Criticality of the World**"** test problem. This problem is a 7 x 7 x 7 lattice of ^{235}U spheres embedded in a block of low density material. The problem was set up using combinatorial geometry with $7^3 + 1 = 344$ cells. Each sphere has a radius $r = 5.0\,cm$, with the exception of the center sphere, which has a radius $r = 8.7407\,cm$. The density of all of the spheres is $\rho = 19.1\,g/cm^3$. The centers of each of the spheres are separated by $\Delta = 24.0\,cm$. Low density ^{235}U ($\rho = 10^{-10}\,g/cm^3$) surrounds the lattice of spheres. Each of the spheres is subcritical, with the exception of the center sphere, which is supercritical. The initial source of particles is in a sphere at a corner of the lattice. A static *k* eigenvalue calculation is performed on this system using 100,000 simulated neutrons. The plots presented in the following subsections display the iterative evolution of the simulation, as the particles "find" the center sphere.

2.1 Mesh Plots

For the mesh plots shown in Figure 1, the user has requested a 100 x 100 x 100 cell "graphics mesh" be overlaid on top of the CG for visualization. The problem was run in parallel on 64 processors using 64 (4 x 4 x 4) spatial domains. The figure shows plots of (a) the graphics mesh, (b) the graphics mesh which is color coded by the domain that the cells are assigned to, (c) the graphics mesh overlaid by a material plot showing the ^{235}U spheres, and (d) both the mesh and low density ^{235}U filler have been "hidden" via ***VisIt***'s material subselection feature.

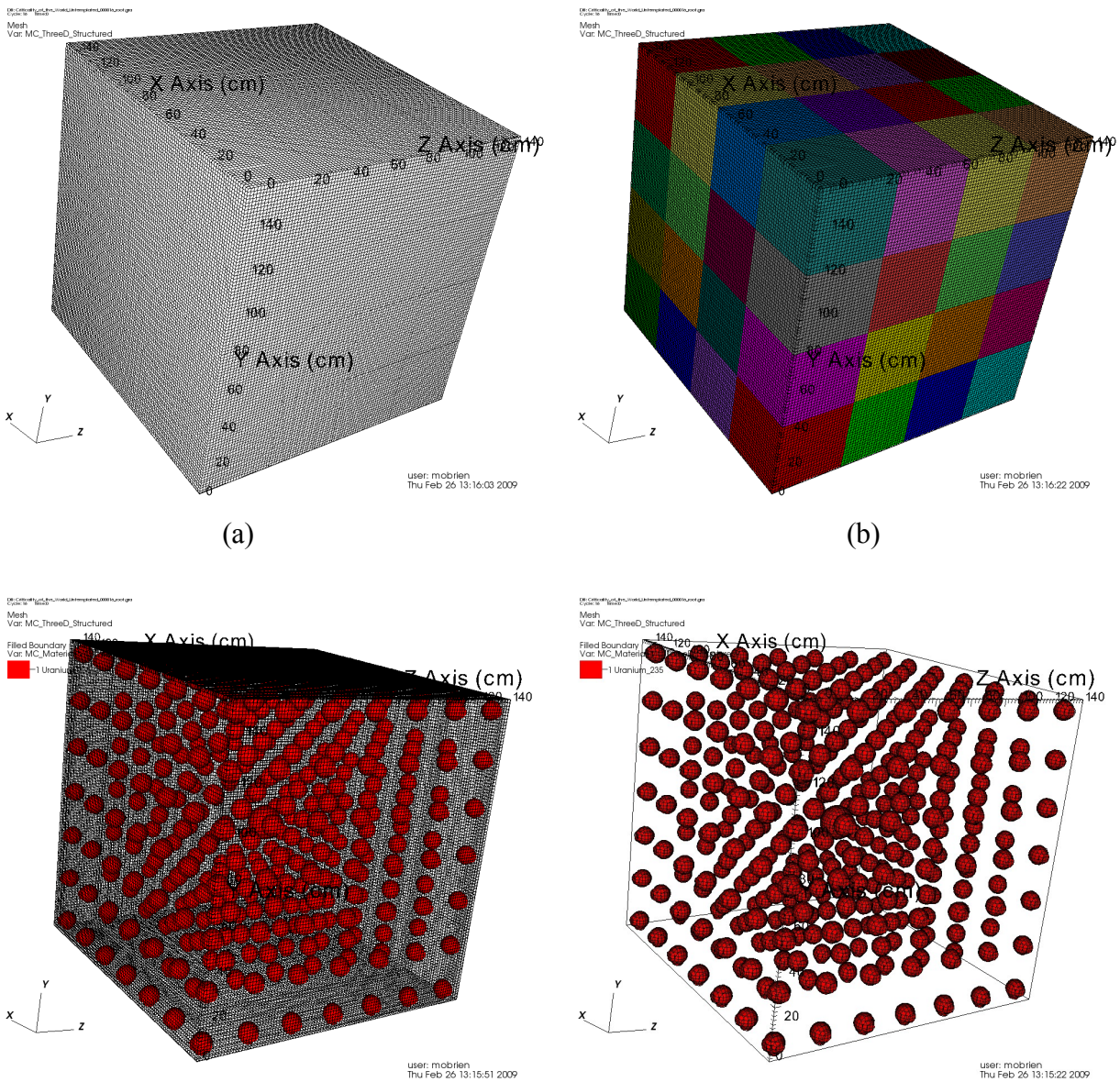


Figure 1. Examples of mesh plots from *VisIt*: (a) mesh only, (b) mesh plus domain, (c) mesh plus material, (d) mesh is hidden and the low density ^{235}U is removed via the material subselection feature.

2.2 Material Plot

VisIt has the ability to color code various problem attributes by material. For instance, the ^{235}U spheres in Figures 1c and 1d are colored red. Similarly, Figure 2 is a plot of both the underlying geometry and the particles, which is color coded by material. Every mesh cell has at least one, (or multiple) material(s) associated with it for "clean" ("mixed") cells. Similarly, every particle a;so has a background material associated with it. In Figure 2, the opacity of the low density filler ^{235}U (shown in green) has been reduced to make it look transparent, while the high density ^{235}U has been completely hidden. This permits us to plot the particles which are color coded by their background material. The figure clearly indicates that the particles are residing almost ex-

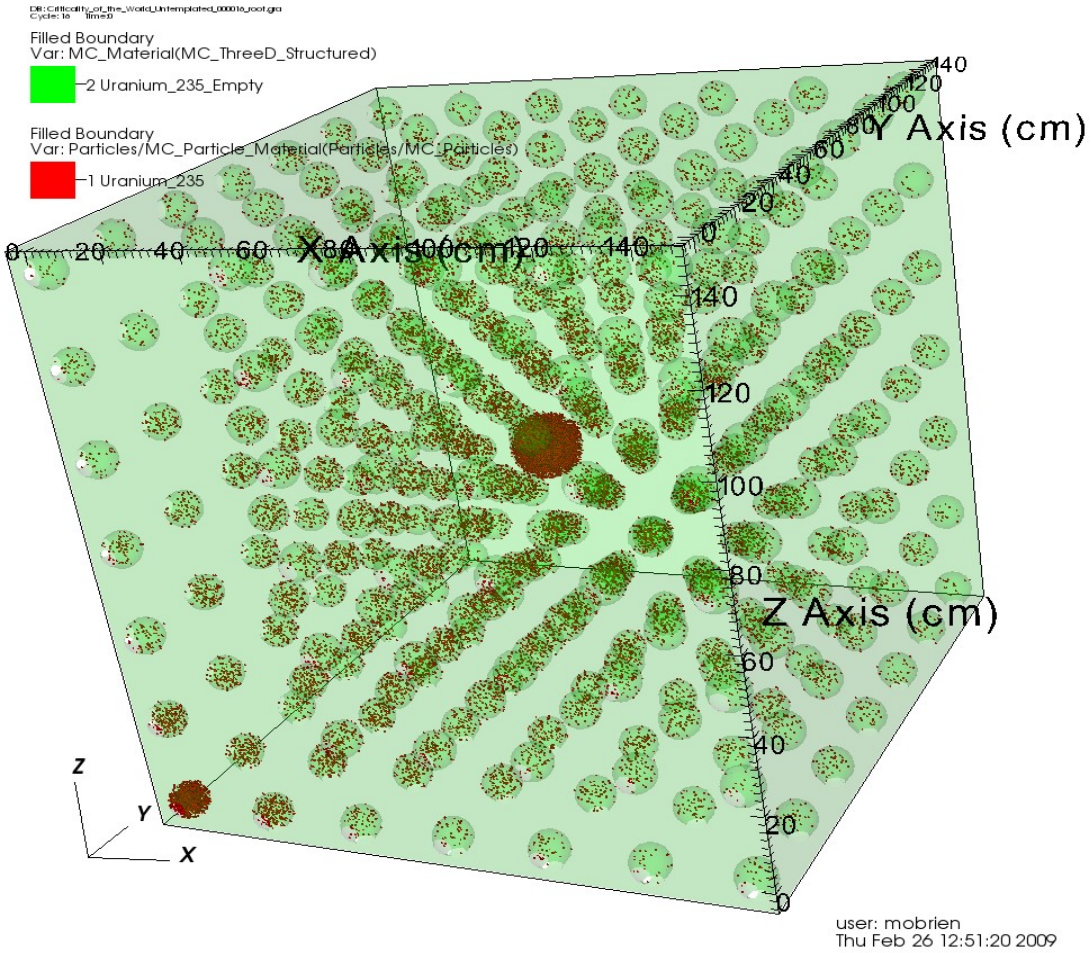


Figure 2. The particles in the problem are color coded according to their background material in this *VisIt* material plot. Note that the particles reside almost exclusively in the high-density 235U spheres.

clusively within the high-density ^{235}U spheres. Since the particle distribution has not yet converged, remnants of the initial particle source are visible in the lower-left sphere.

When plotting particles in ***VisIt***, one may plot them as simple "points", as shown in Figure 3a, or as "spheres", as in Figure 3b. Figure 3c shows particles color coded by material and the low-

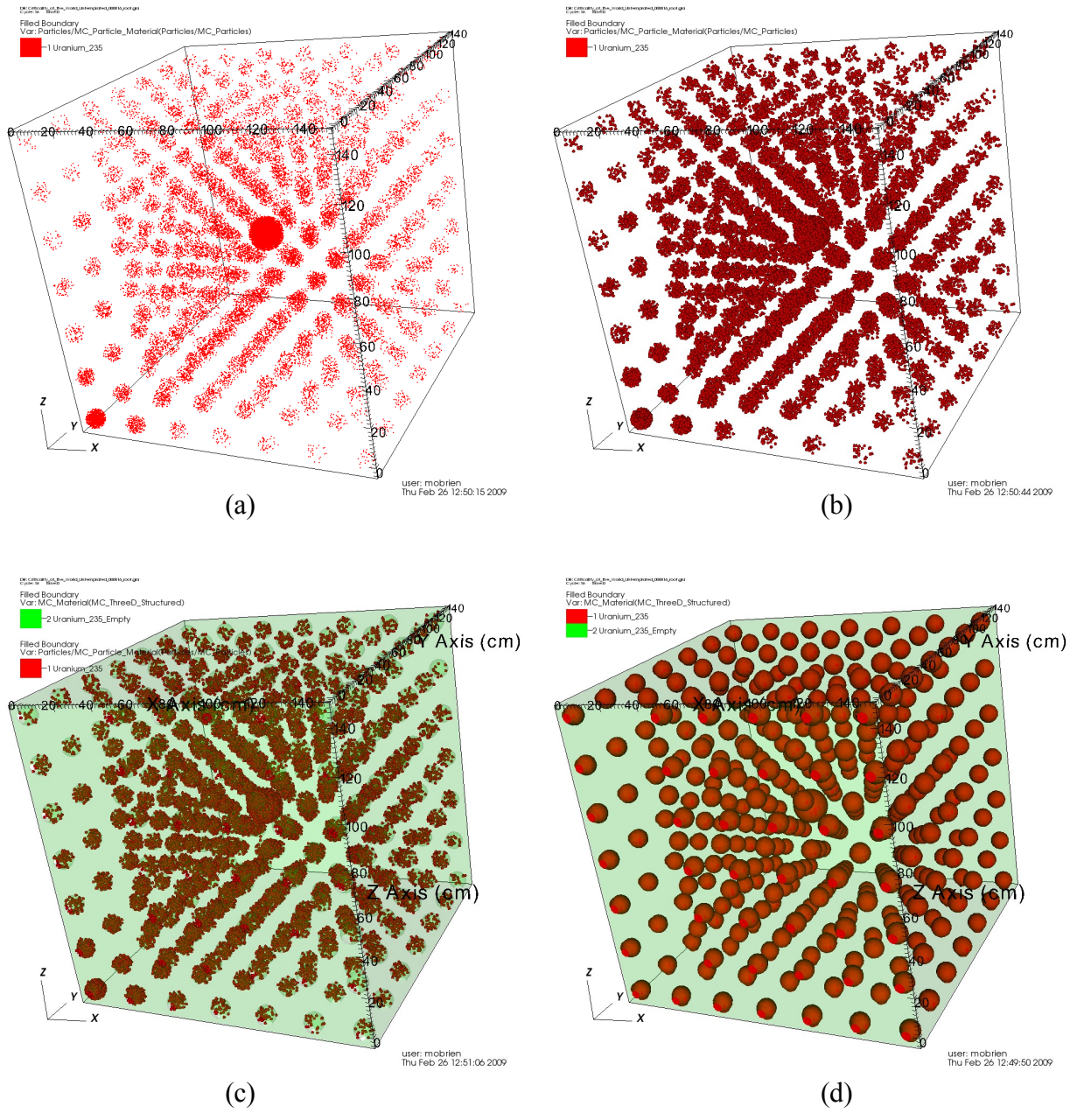


Figure 3. Examples of material plots from *VisIt*: (a) "point" particles color coded by material, (b) "sphere" particles color coded by material, (c) particles plus low opacity filler material, and (d) high density spheres plus low opacity filler material .

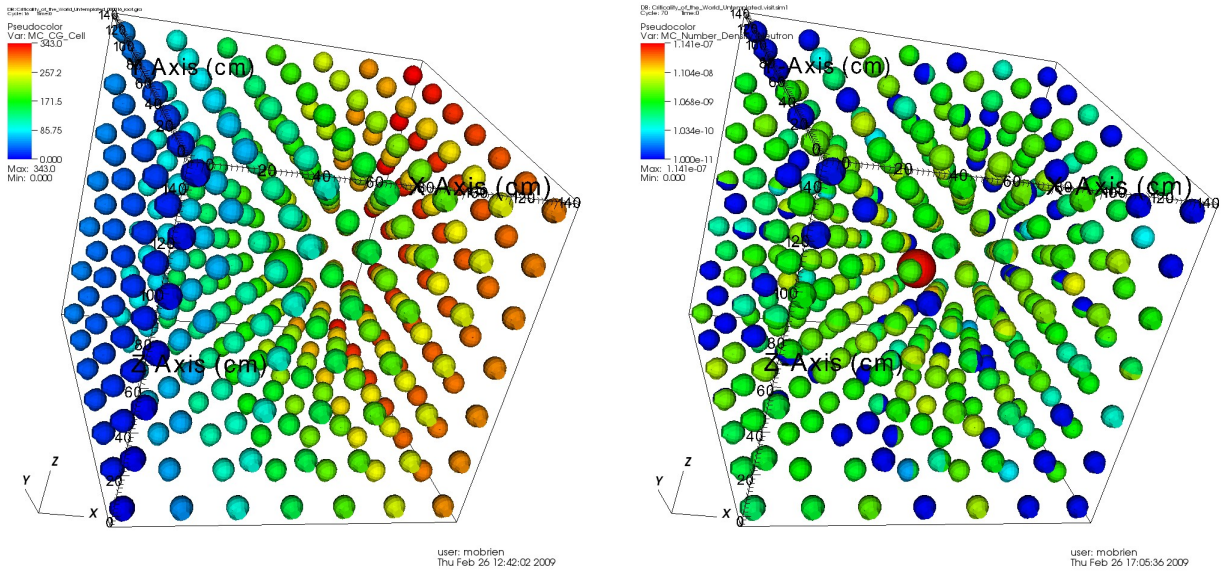
density ^{235}U material with a low opacity, which is analogous to Figure 2. Finally, in Figure 3d, the high-density ^{235}U material is plotted with full opacity, while the low-density ^{235}U material is plotted with low opacity.

2.3 Pseudocolor Plots

A pseudocolor plot maps values of the chosen parameter to colors. ***VisIt*** supports pseudocolor plots of a wide variety of problem parameters. Some examples of pseudocolor plots are shown in Figure 4. Note that the material subselection feature has been used to hide the low-density filler material on this figure. Figure 4a is a pseudocolor plot of the CG cell index. Note that the "color palette" indicates that the range of values is 0 to 343, for a total of 344 cells. This type of plot has been extremely useful for debugging the definition of combinatorial geometries. This plot, along with ***VisIt***'s "pick" tool, allows one to click on a cell and query the CG cell index. Figure 4b is a pseudocolor plot of the base-10 logarithm of neutron number density in each CG cell, which is the eigenvector associated with the *k* eigenvalue. ***VisIt*** allows linear or logarithmic color scales, which can be used to "spread" out the color values. In this case the *log* scale provides greater dynamic range than the *linear* scale, thereby showing a greater ranges of colors.

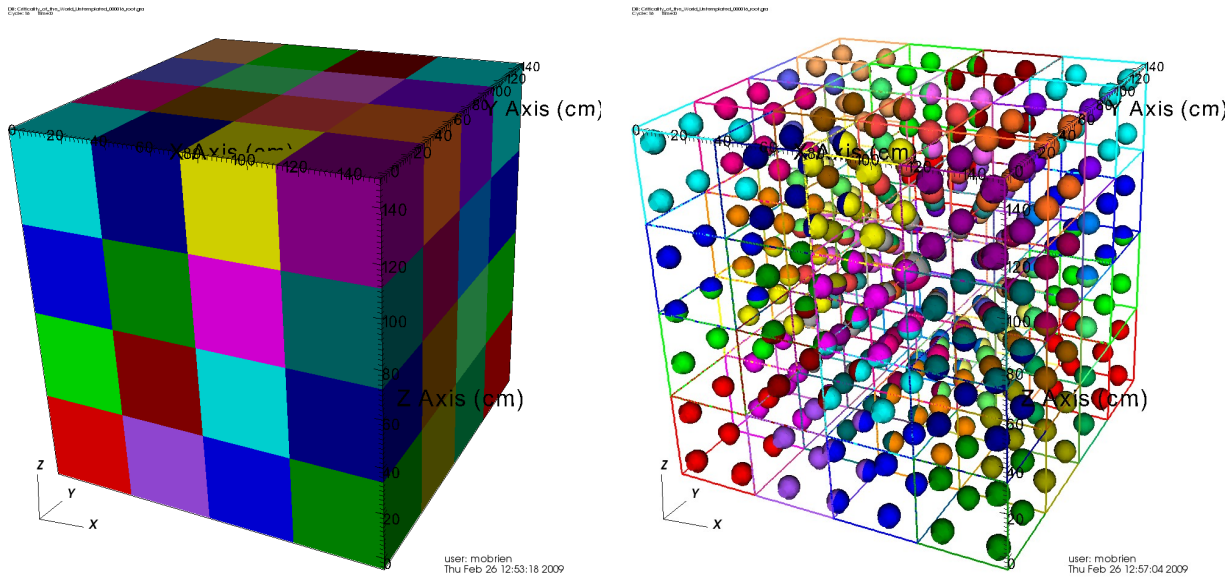
2.4 Domain Plots

In a typical problem, the user specifies the physical extents of the Cartesian graphics mesh and the number of cells in each (x,y,z) direction. The ***Mercury*** Monte Carlo code automatically domain decomposes the graphics mesh into *N* spatial domains, where *N* is the number of processors



(a) (b)

Figure 4. Examples of pseudocolor plots from *VisIt*: (a) a pseudocolor plot of CG cell index, and (b) a pseudocolor plot of base-10 logarithm of the neutron number density.



(a) (b)

Figure 5. Examples of domain plots from *VisIt*: (a) the graphics mesh is color coded by spatial domain, and (b) the CG cells are color coded by spatial domain.

used for the calculation. This makes the generation of the plots faster, since each processor has to render only a portion of the entire problem. The problem shown in Figure 5 was run on 64 processors, hence the graphics mesh is shown as 64 color-coded domains in Figure 5a. In Figure 5b, the CG cells in each domain are assigned the color which indicates the domain decomposition of the graphics mesh. Note that this is only the domain decomposition of the graphics mesh, which, in general, is different than the domain decomposition of the combinatorial geometry. For additional information on the CG domain decomposition methods used in ***Mercury***, please see the paper by Green man, *et al.* entitled "Enhancements to the Combinatorial Geometry Particle Tracker in the ***Mercury*** Monte Carlo Transport Code".

2.5 Volume Rendering Plots

VisIt also supports volume rendering of both mesh data (as shown in Figures 6a, 6b, 6c and 6e) and particle data (as shown in Figure 6d). Figures 6a through 6c show that mass density in the problem for various numbers of volume rendering ray samples. When a low number of ray samples are used (5×10^4 ray samples in 6a), the image looks fuzzy. However, as the number of samples is increased, the image becomes sharper (5×10^5 ray samples in 6b and 5×10^6 ray samples in 6c). The kinetic energy of each neutron particle is rendered in Figure 6d using 5×10^7 ray samples, and the neutron number density in each cell of the graphics mesh is rendered in Figure 6e using 5×10^5 ray samples.

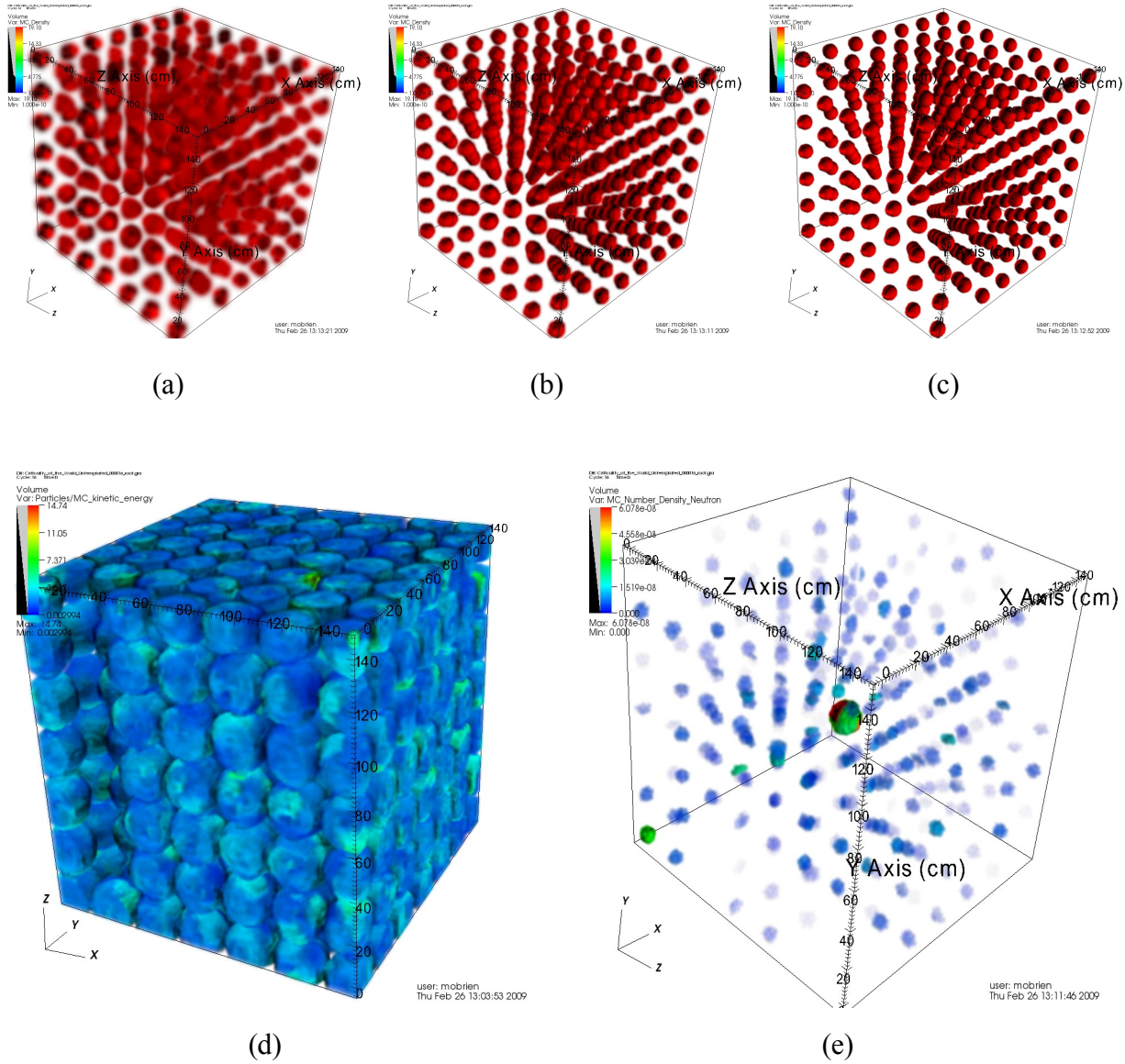
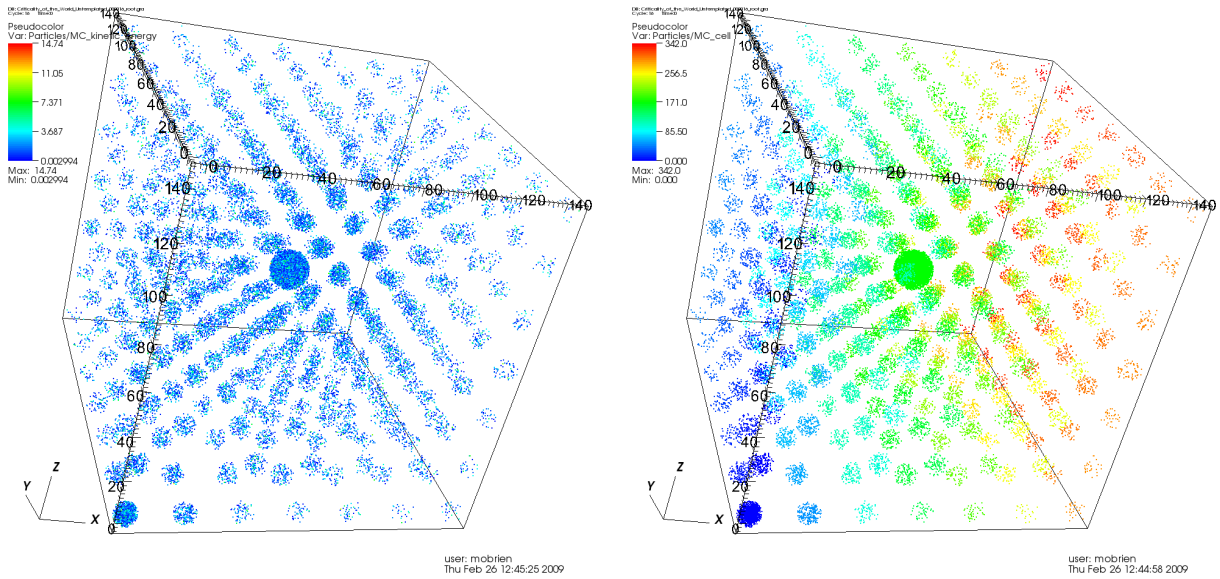


Figure 6. Examples of volume rendering plots from *VisIt*: (a) - (c) are renderings of the mass density in the problem is volume rendered for (a) 5×10^4 samples, (b) 5×10^5 samples, and (c) 5×10^6 samples, (d) is a rendering of the neutron kinetic energy and (e) is a rendering of the neutron number density.

2.6 Particle Plots

VisIt supports particle plots in which the pseudocolor can be associated with any particle attribute. For instance, Figure 7a shows the particle plot where the neutrons are color coded by their kinetic energy. This provides an indication of the particle distribution in both space and energy. The figure clearly shows a concentration of particles in the center sphere. In addition since the eigenvector has not yet converged, one can see some remnants of the initial particle source in the lower left sphere.



(a) (b)

Figure 7. Examples of particle plots from *VisIt*: (a) the particles are color coded their kinetic energy, and (b) the particles are color coded their CG cell index.

Figure 7b shows the particle plot where the neutrons are color coded by the index of the CG cell that contains the particle coordinates. This type of plots is very useful for finding particle tracking bugs. It is very easy for the human eye to spot an incorrectly colored particle. By employing a "discontinuous" color scale, in which adjacent cells have drastically different colors instead of continuously varying colors, it is even easier to determine when a particle has an incorrect cell attribute.

2.7 Vector Plots

Figure 8 illustrates the velocity vector plotting feature in ***VisIt***, where the velocity vectors for each particle in Figure 8a. The vectors are colored by the speed (magnitude of the velocity) of each particle. Since plotting all of this data can result in a very dense image, ***VisIt*** has an option to specify the stride with which particles are drawn. For example, in Figures 8b, 8c and 8d plot every 4^{th}, 8^{th} and 16^{th} particle , respectively.

2.8 Histogram Plots

Figure 9 is a histogram of the number of particles in various kinetic energy bins (the particle spectral distribution), as plotted on a log(E) - log(f[E]) scale. This type of plot permits easy determination of the energy distribution of the particles. The histogram plot is also useful for looking at the particle weight distribution. Histogram plots can be produced for any particle- or mesh-based data.

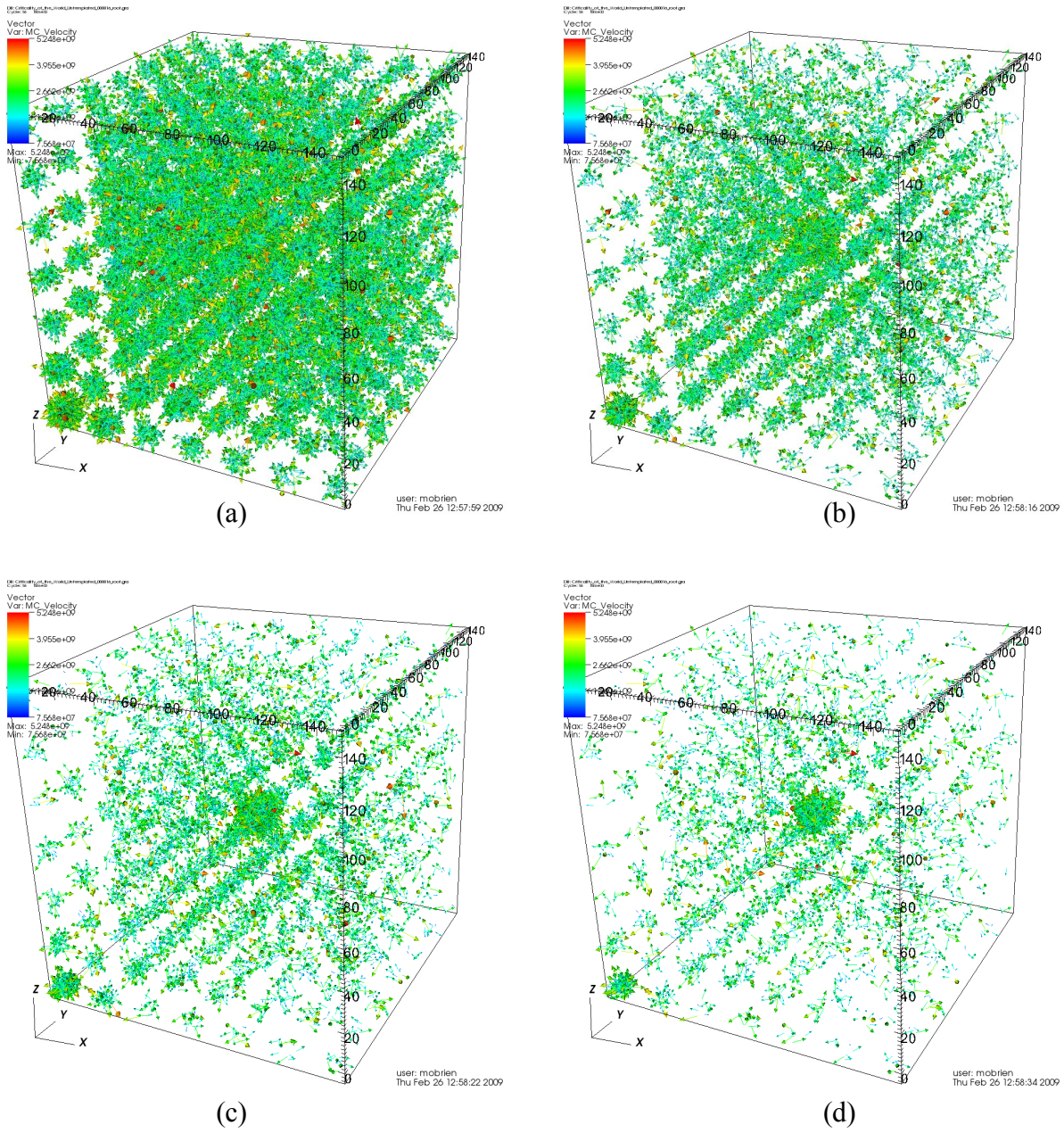


Figure 8. Examples of velocity plots from *VisIt*, where the particles are color coded by their speed: (a) every particle is plotted, (b) every 4th particle is plotted, (c) every 8th particle is plotted, and (d) every 16th particle is plotted.

2.9 Curve Plots

In addition to three-dimensional visualization and rendering, ***VisIt*** can also plot data as a function of a single independent variable, *aka* curve plots. An example of this ***VisIt*** feature is Figure 10, which shows the iteration history of (1) the k eigenvalue for each iteration (generation) in red, (2) the iteration (generation) averaged eigenvalue $\langle k \rangle$ in green, (3) the averaged eigenvalue plus

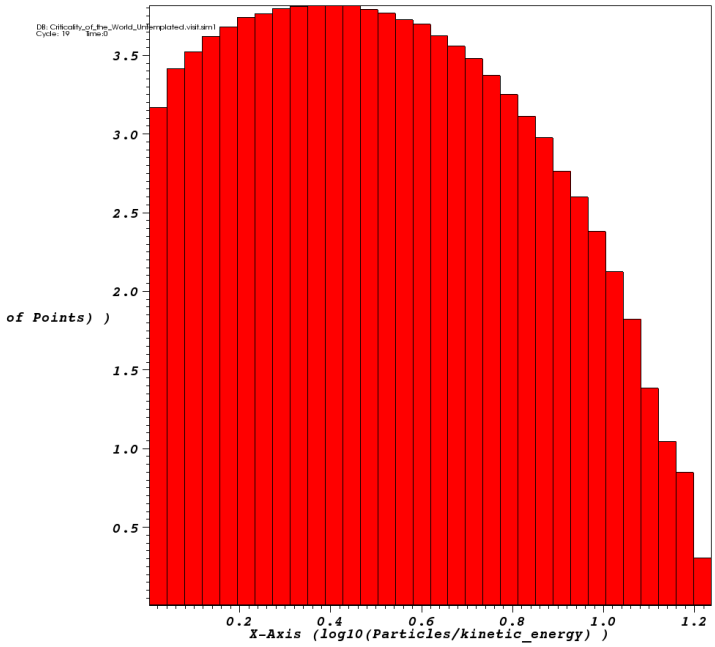


Figure 9. An example of histogram plots from *VisIt*. This is a log(E) – log(f[E]) plot of the particle spectral distribution.

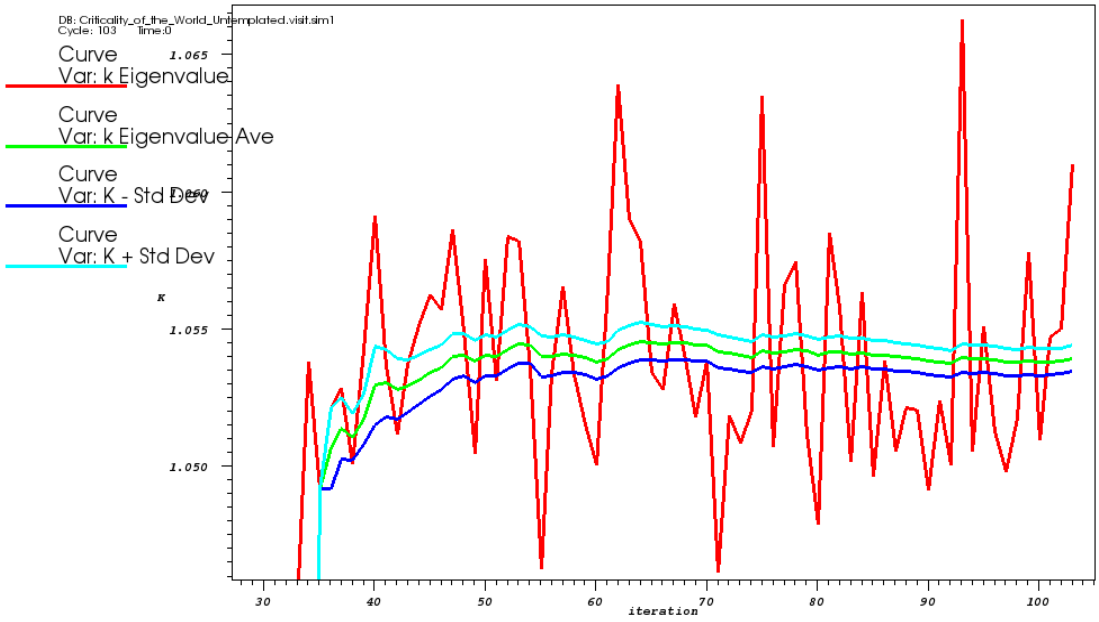


Figure 10. An example of curve plots from *VisIt*. This shows the iteration history of the k eigenvalue (red), averaged eigenvalue $\langle k \rangle$ (green) and averaged eigenvalue +/- one standard deviation $\langle k \rangle \pm 1\sigma$ (cyan, blue).

one standard deviation $\langle k \rangle + 1\sigma$ in cyan, and (4) the averaged eigenvalue minus one standard deviation $\langle k \rangle - 1\sigma$ in blue. The figure clearly shows that, as the iteration count increases, the standard deviation decreases and the averaged eigenvalue $\langle k \rangle$ is converging. Note the discontinuity in the average $\langle k \rangle$ eigenvalue at 34 iterations. This is due to a change from initial (inactive) to final (active) "settle" iterations after iteration 34. This method assumes that the initial 34 iterations is sufficient to "settle" the neutron distribution through the transient phase and towards a stationary distribution. After 34 iterations, the averaging begins anew with what is assumed to be a equilibrium distribution.

3 USING THE INLINE INTERFACE TO VISIT

Once the *'visit()'* function has been executed via ***Mercury***'s ***Python*** interface, the ***VisIt*** GUI and at least one data window will be launched. The user is then able to select the data sets of interest for display within multiple data windows.

3.1 Time Evolution of The "Criticality Of The World" Problem

Figures 11 – 14 show the ***VisIt*** GUI and 4 data windows during the evolution of a *k* eigenvalue simulation of the **"**Criticality of the World**"** problem. These figures are displaying data for iterations (generations) 1, 6, 26 and 63, respectively. The GUI control panel is shown on the left side of these images, and the four data windows are displaying (Upper Left) a pseudocolor plot of the base-10 logarithm of the neutron number density, (Upper Right) a curve plot of the iteration history of k, $\langle k \rangle$, $\langle k \rangle + 1\sigma$ and $\langle k \rangle - 1\sigma$, (Lower Left) a particle plot of base-10 logarithm of the neutron kinetic energy, and (Lower Right) a curve plot of the flux entropy, which is used as an eigenvector convergence diagnostic. The screen snapshots shown in these figures were obtained while the tally data was being updated in real-time during a ***Mercury*** simulation of a test problem.

Let us examine the evolution of the plotted data. In Figure 11 (Iteration 1), there are not yet two points for the curve plots, so they are blank. One can see that after only a single iteration, the particles are still concentrated in the vicinity of the lower left sphere where they were sourced. After another 5 iterations, Figure 12 (Iteration 6) shows that while the particles have begun to reach the far corners of the system, they are still concentrated around the source sphere**.** By Iteration 26 (Figure 13), a large number of particles are being produced within the central sphere and the particle distribution is approaching its equilibrium state. Finally, by Iteration 63 (Figure 14), a large fraction of the particles are within the central sphere, and the particle distribution has become stationary. Note the a discontinuity at Iteration 34, which is when ***Mercury*** switched from inactive to active "settle" cycles.

3.2 Command Recording, Playback and Scripting Features in *VisIt*

Figure 15 shows the ***VisIt*** window that permits recording and playback of your mouse clicks to the main ***VisIt*** GUI. When recording is enabled, ***VisIt*** automatically translates mouse clicks into a ***Python*** script. Later, that ***Python*** script can be "played back" to achieve the same behavior as your mouse clicks. It is also possible to save the ***Python*** script to a text file, which can be handed back to ***VisIt*** via the interactive ***Mercury Python*** prompt. For example, suppose the user

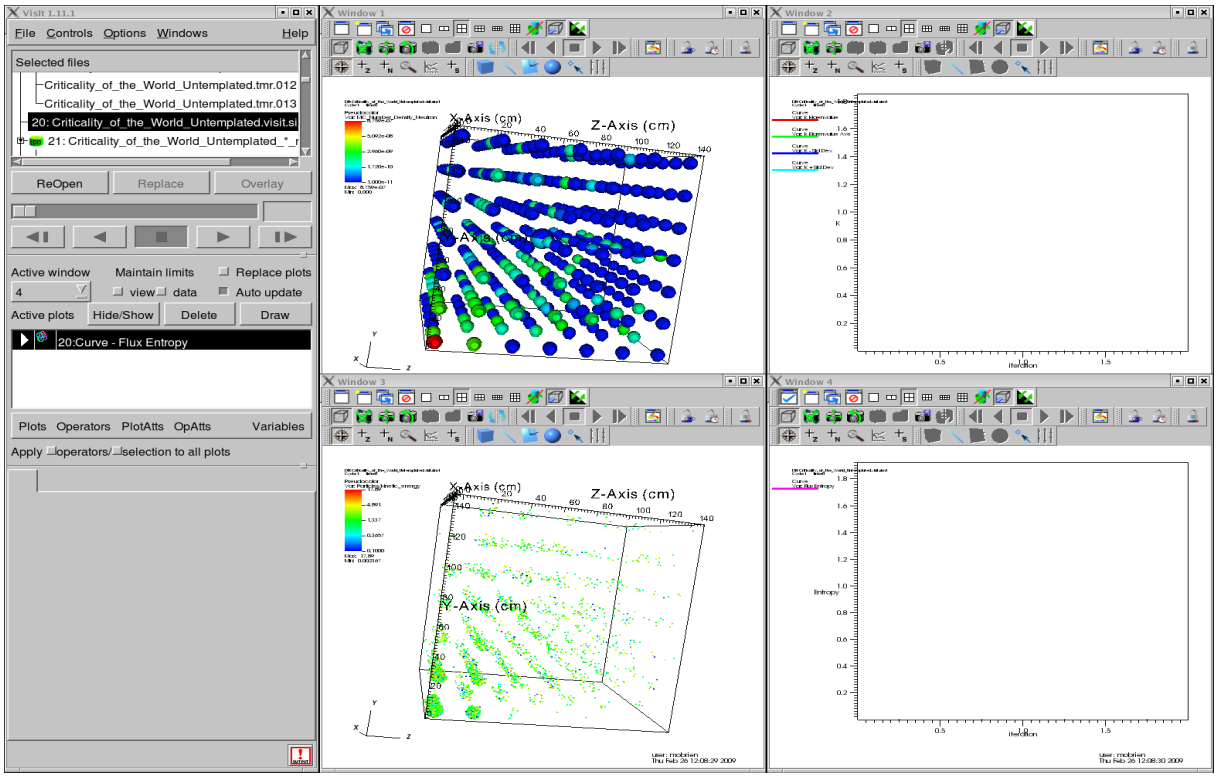


Figure 11. Time evolution of the "Criticality of the World" test problem: Visualization of various data items at Iteration 1.

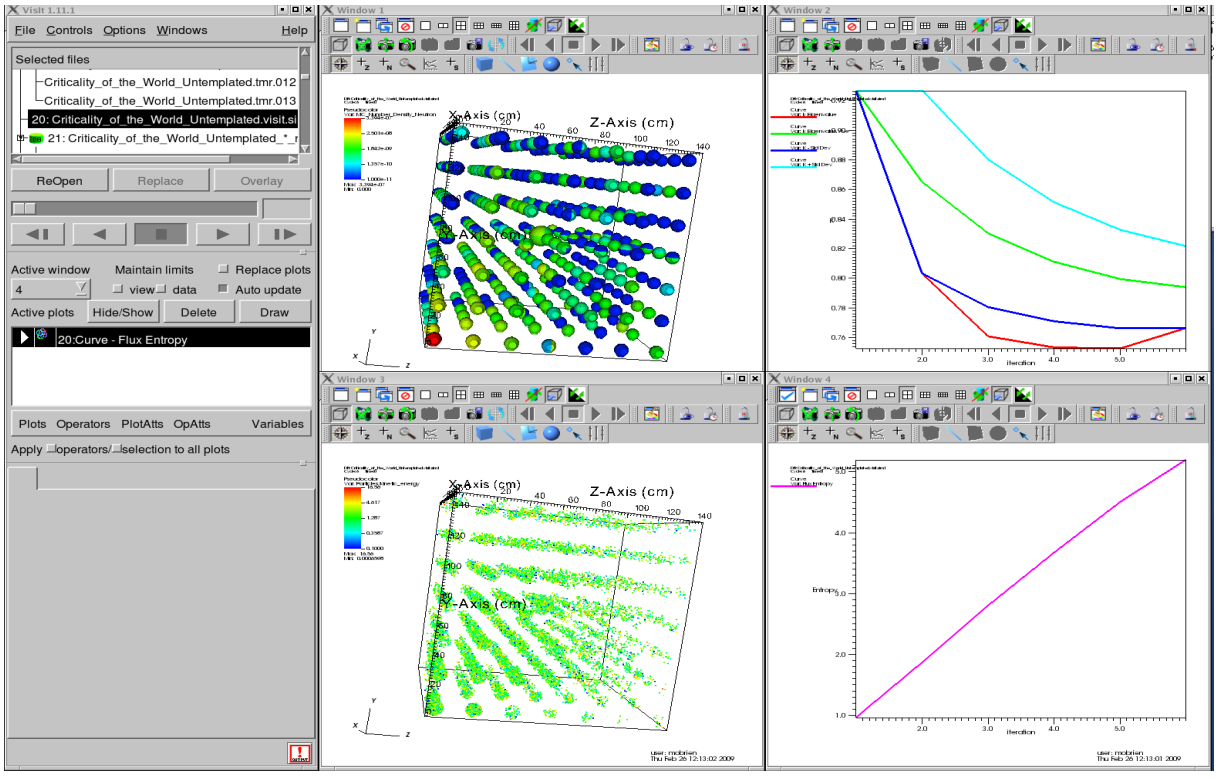


Figure 12. Time evolution of the "Criticality of the World" test problem: Visualization of various data items at Iteration 6.

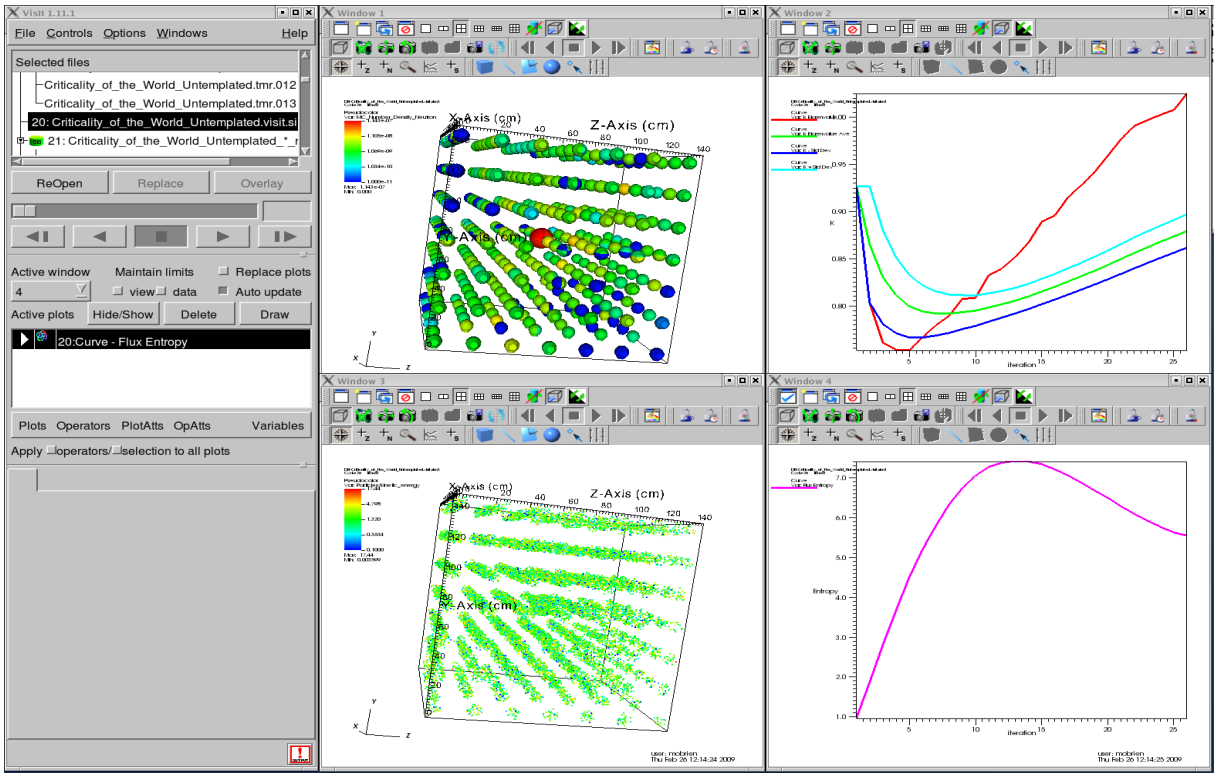


Figure 13. Time evolution of the "Criticality of the World" test problem: Visualization of various data items at Iteration 26.

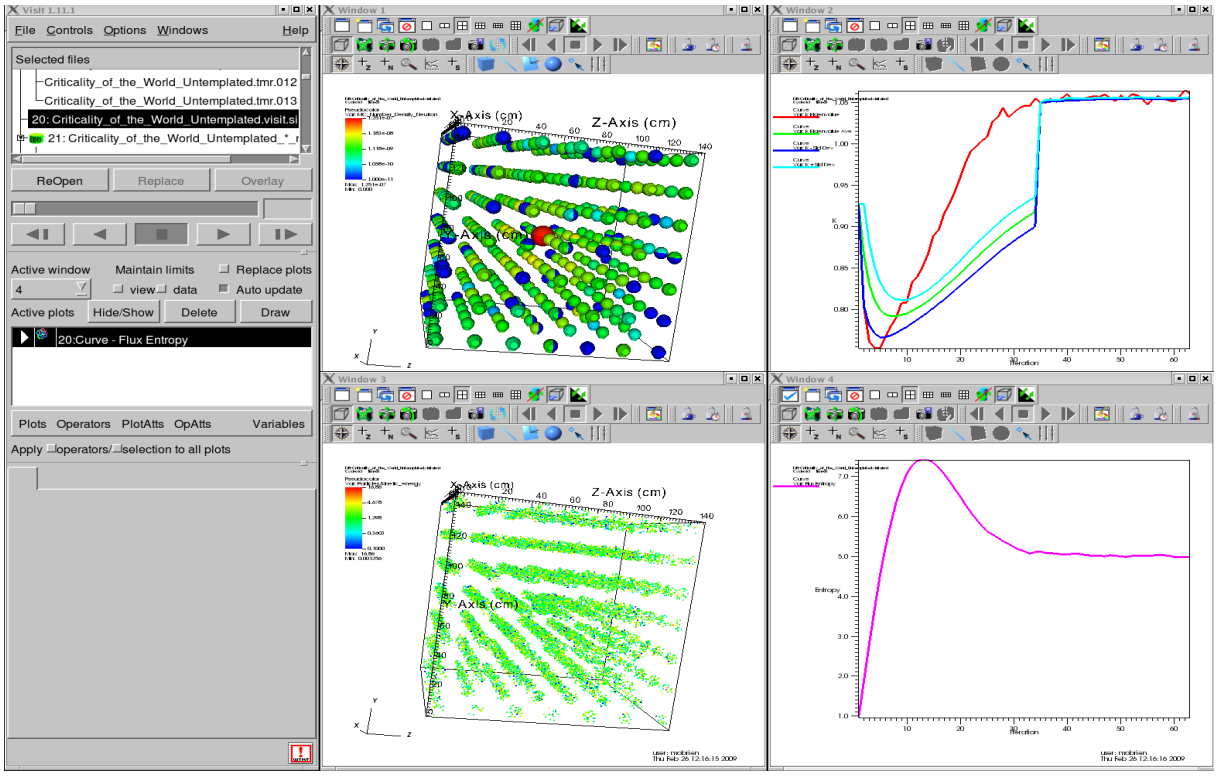


Figure 14. Time evolution of the "Criticality of the World" test problem: Visualization of various data items at Iteration 63.

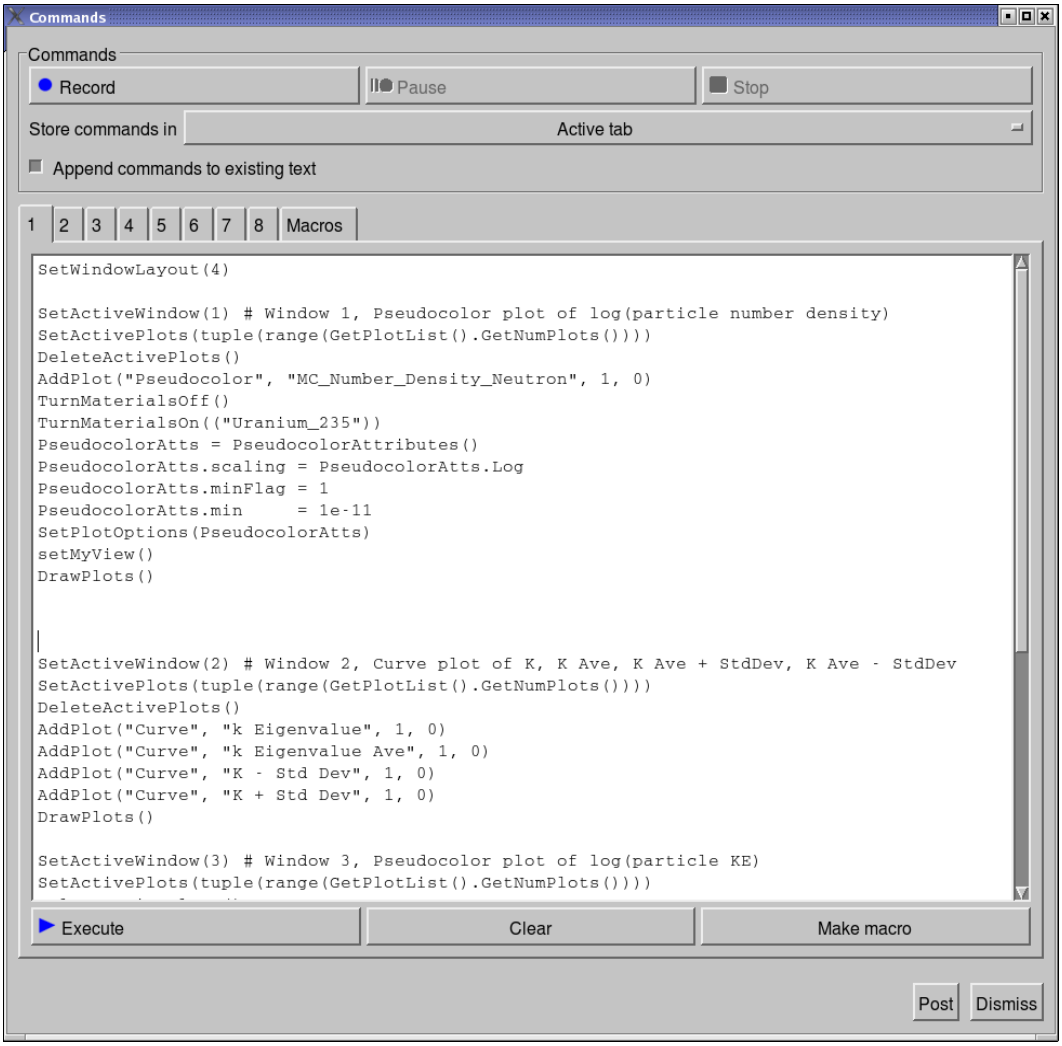


Figure 15. The *record* and *playback* window in *VisIt*, showing the *Python* script that is automatically generated from the user's mouse click in the main GUI window.

saves the script to a file named '*cow.py*', as shown in Figure 16. Later, at the ***Mercury Python*** prompt, the user would type *'visit('cow.py')'* to feed the ***Python*** script back to ***VisIt***. This permits the user to quickly recover all of the plots, without having to reselect them via ***VisIt***'s main GUI window.

4 COMBINATORIAL GEOMETRY SAMPLING AND DISCRETIZATION METHODS

The problem at hand is: Given the analytic surfaces that define the CG, how does one visualize them in a mesh-based visualization tool like ***VisIt***? The approach taken by the ***Mercury*** and ***VisIt*** teams is to translate the analytic, surface-based representation of the CG to a mesh-based representation, by sampling the CG onto a graphics mesh. Before ***VisIt*** was modified to internally discretize combinatorial geometry (map it onto a graphics mesh), ***Mercury*** would perform that discretization, writing out a mesh file, which ***VisIt*** could then visualize. While this method works well, it has the drawback of having to store a (potentially) large graphics mesh in memory

```
################################################################################
# File Name: cow.py
# Purpose: This is an example python script that sets up 4 VisIt Windows.
# 1. Pseudocolor plot of: log(particle number density)
# 2. 4 curve plots: K, K Ave, K Ave + StdDev, K Ave - StdDev.
# 3. Pseudocolor plot of log(particle KE)
# 4. 1 curve plot: Flux Entropy
#
# Usage:  Mercury>  visit('cow.py')
################################################################################

def setMyView():
  # Set view
  View3DAtts = View3DAttributes()
  View3DAtts.viewNormal    = (-0.973501, 0.0961496, -0.207485)
  View3DAtts.focus         = (72.5, 72.5, 72.5)
  View3DAtts.viewUp        = (0.11985, 0.987242, -0.104832)
  View3DAtts.viewAngle     = 30
  View3DAtts.parallelScale = 134.234
  View3DAtts.nearPlane     = -268.468
  View3DAtts.farPlane      = 268.468
  SetView3D(View3DAtts)

SetWindowLayout(4)

SetActiveWindow(1) # Window 1, Pseudocolor plot of log(particle number density)
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Pseudocolor", "MC_Number_Density_Neutron", 1, 0)
TurnMaterialsOff()
TurnMaterialsOn(("Uranium_235"))
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.scaling = PseudocolorAtts.Log
PseudocolorAtts.minFlag = 1
PseudocolorAtts.min     = 1e-11
SetPlotOptions(PseudocolorAtts)
setMyView()
DrawPlots()

SetActiveWindow(2) # Window 2, Curve plot of K, K Ave, K Ave + StdDev, K Ave -
StdDev
SetActivePlots(tuple(range(GetPlotList().GetNumPlots())))
DeleteActivePlots()
AddPlot("Curve", "k Eigenvalue", 1, 0)
AddPlot("Curve", "k Eigenvalue Ave", 1, 0)
AddPlot("Curve", "K - Std Dev", 1, 0)
AddPlot("Curve", "K + Std Dev", 1, 0)
DrawPlots()
```

Figure 16. The *Python* script that is generated by the *record* feature in *VisIt*.

and on disk. Several techniques for sampling the CG onto a graphics mesh have been implemented in either ***Mercury*** or ***VisIt***. These results of these techniques are contrasted in Figure 17 for a single, spherical CG cell.

4.1 "Lego" Method

The very first implementation of this sampling technique that was implemented in ***Mercury*** is reminiscent of "Legos" building blocks. The user was required to define a graphics mesh, and each graphics cell was assigned exactly one CG cell. That cell was chosen by answering the question "Which CG cell is the center of each graphic cell in?" The accuracy with which this method reproduces the underlying CG could be severely limited, as evidenced in Figure 17a. Obviously, a better method is needed.

4.2 Adaptive Mesh Refinement (AMR) Method

In this method, ***Mercury*** recursively samples, up to some user specified limit, each graphics cell to get an accurate volume fraction of each CG cell that it overlaps. As a result, the graphics cells on the boundary of a CG cell are known as mixed-material cells, or simply, mixed cells ***Mercury*** only writes out volume fraction information and then relies on ***VisIt***'s material interface reconstruction (MIR) algorithm to subdivide the graphics cells into parts, corresponding to the volume fraction of the material within the cell. This method can produce images that are superior to the "Lego" method, as shown in Figures 17b and 17c.

As seen if Figure 17c (which shows the AMR mesh on a cut place through the center of the CG spherical cell), if the 8 corners of a 3D Cartesian graphics cell all contain the same CG cell, the discretization algorithm terminates. If any of the corners contains a different CG cells, the algorithm recursively subdivides the cell into 8 subcells, and continues sampling. There are user settable parameters for controlling the minimum and maximum recursion limits. The goal of the algorithm is to compute the volume fraction of each CG cell within each graphics cell. ***VisIt*** uses the volume fraction information to perform a material interface reconstruction (MIR) to attempt to reconstruct the position of the CG cell interface within the graphics mesh.

4.3 Conformal Cell Method

Starting with a Cartesian graphics mesh, "launch" a particle from the center of that mesh to every node of the mesh. If the particle intersects a CG surface before it reaches the neighboring node in the graphics mesh, that mesh node is moved to lie on the CG surface. The idea is to conformally move the graphics mesh nodes directly onto the CG surfaces, such that the graphics mesh contains cells of exactly one material. Since this mesh is only used for visualization, and not a finite difference or finite element calculation, it does not matter how distorted the graphics cells are. At current, the ***Mercury*** team is still researching the best implementation of this algorithm.

Results from the AMR and conformal cell discretization methods are compared for a set of nested CG spheres in Figure 18. The AMR and conformal meshes are shown on a cut plane through the center of the spheres in Figures 18a and 18b, respectively. In the AMR method, one can see that the base mesh is Cartesian. ***Mercury*** has computed volume fractions of each CG cell within each graphics cell, and then ***VisIt*** use its MIR method to draw the CG cell boundaries. In the conformal method, the figure shows that while the starting point was a Cartesian mesh, the nodes of the mesh were moved onto CG cell boundaries.

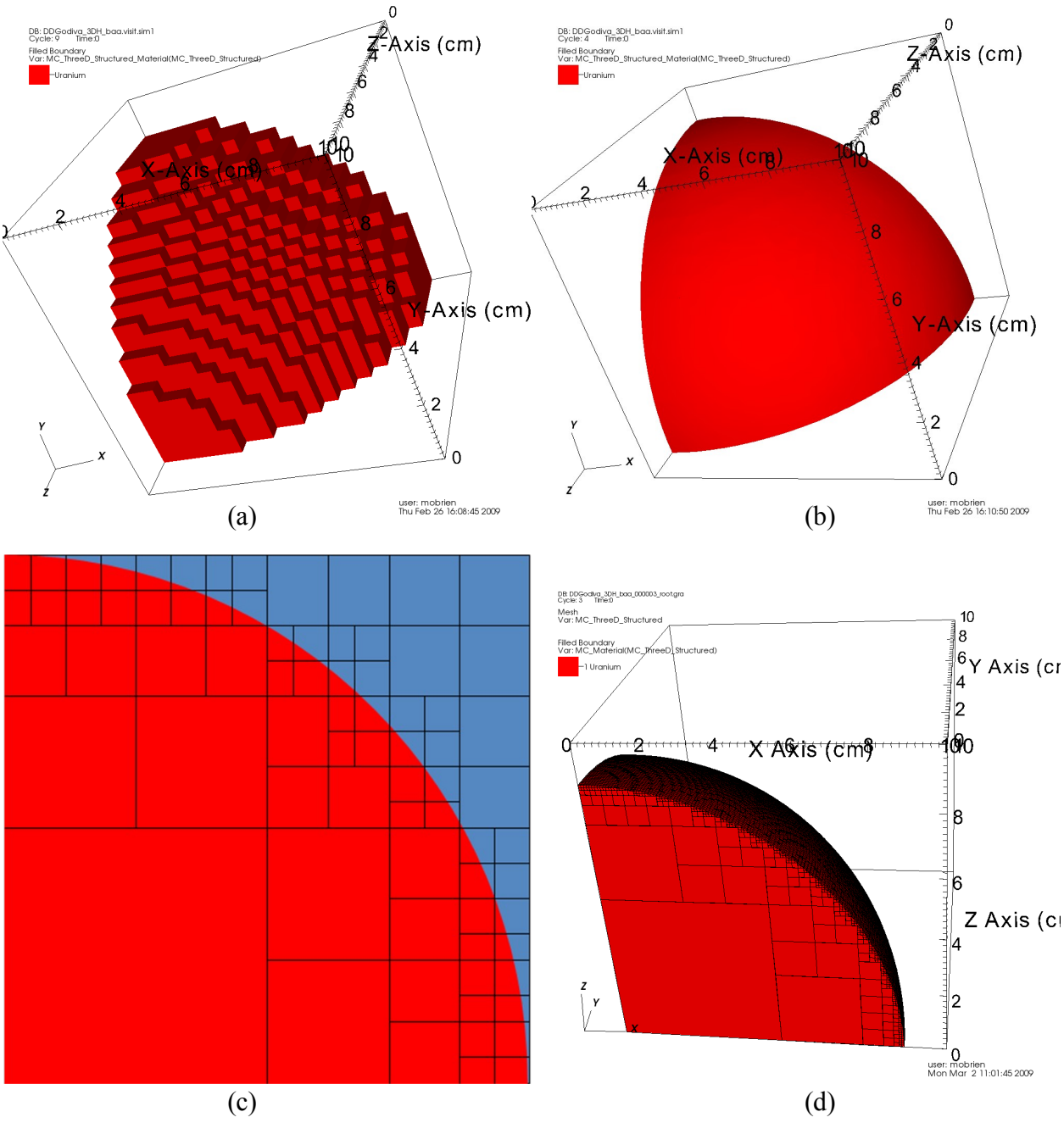
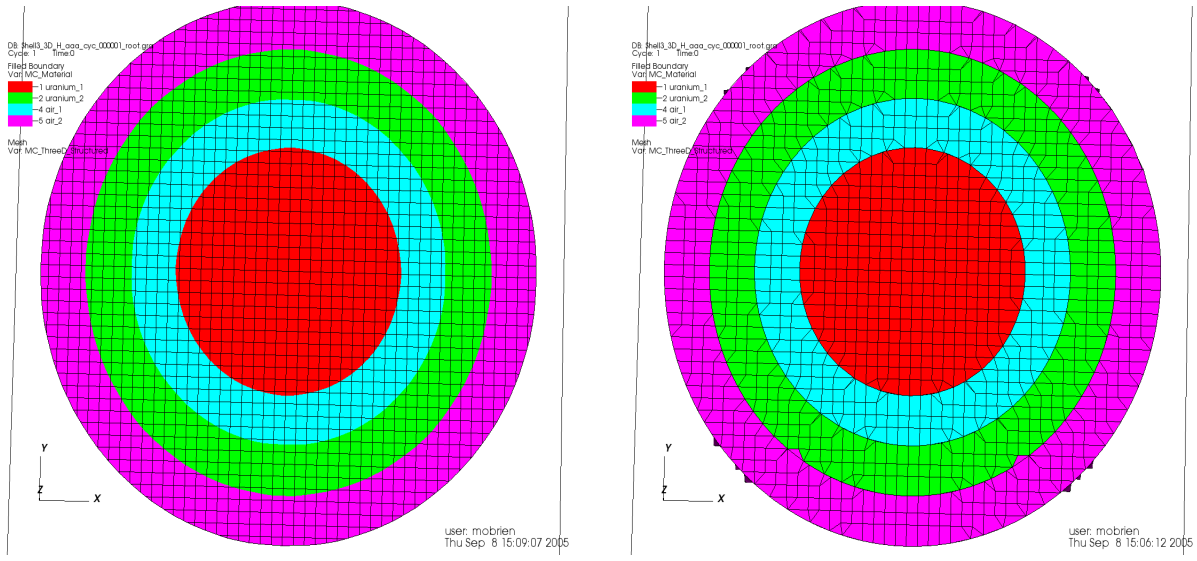


Figure 17. Illustration of the various methods for discretizing and sampling a combinatorial geometry onto a graphics mesh: (a) the "Lego" method, (b) - (c) the adaptive mesh refinement (AMR) (mixed cell) methods, and (d) native *VisIt* method.

4.4 Native *VisIt* Method

In this method, ***Mercury*** does not convert its internal CG representation to a mesh representation. Rather, it provide the CG representation directly to ***VisIt***. The coefficients of the analytic sur-



(a) (b)

Figure 18. Comparison of the (a) adaptive mesh refinement (AMR) (mixed cell) method and (b) the conformal mesh method for the discretization of a set of nested CG spheres.

faces, and how those surfaces are combined to form CG cells, are transferred between the codes. ***VisIt*** then performs an AMR discretization of the CG, and creating an AMR mesh on which to visualize the CG. The results of this method, including the adaptive structure of the graphics mesh, are shown in Figures 17d.

4.5 AMR Method: Algorithm Error Analysis

In the AMR, or mixed cell, method, ***Mercury*** recursively samples each graphics mesh cell, attempting to calculate an accurate volume fraction for each CG cell the overlays the graphics mesh cell. For this error analysis, consider a sphere of radius $r = 8.7407\,cm$, centered at the origin (0,0,0). Now assume that it is contained within a single graphics mesh cell with limits [0, 10] x [0, 10] x [0, 10], such that an octant of the sphere will be sampled. Let the minimum and maximum refinement levels vary from 1 to 11. Now enforce that the minimum refinement equal the maximum refinement, which shall be termed constrained AMR. For a 3D AMR mesh, at refinement level n, there are 8^n sample points: each cell is subdivided in half in each of the 3 coordinate directions $2^3 = 8$. Since this mesh is an octant of a sphere, the exact volume is:

$$V = \left(\frac{4}{3}\pi 8.7407^3\right)/8 = 349.6530057623\,cm^3 \tag{1}$$

As the refinement level is increased by 1 level, one would expect the error to go down roughly by a factor of 8, since each graphics cell is subdivided into 8 subcells. Defining the error at refinement level n as:

$$E[n] = [\![V_{exact} - V_{calculated}]\!] \tag{2}$$

Table I. Error analysis of constrained adaptive mesh refinement (AMR) algorithm for calculating volume fractions.

Refinement Level	*Volume Error E[n]*	*Error Ratio E[n-1] / E[n]*	*Calculated Volume*
1	4.36057623183E-02		349.6094000000
2	1.64010057623E+01	2.65872489469E-03	333.2520000000
3	4.78942376817E-02	3.42442150793E+02	349.7009000000
4	9.59594237682E-01	4.99109267240E-02	350.6126000000
5	5.94057623183E-02	1.61532181430E+01	349.5936000000
6	2.20942376817E-02	2.68874460274E+00	349.6751000000
7	1.63942376817E-02	1.34768313786E+00	349.6694000000
8	3.69423768171E-03	4.43778638360E+00	349.6567000000
9	9.87387681732E-04	3.74142573384E+00	349.6539931500
10	1.36472318275E-04	7.23507663837E+00	349.6528692900
11	2.10223182648E-05	6.49178252157E+00	349.6529847400
Exact Volume:		349.6530057623	

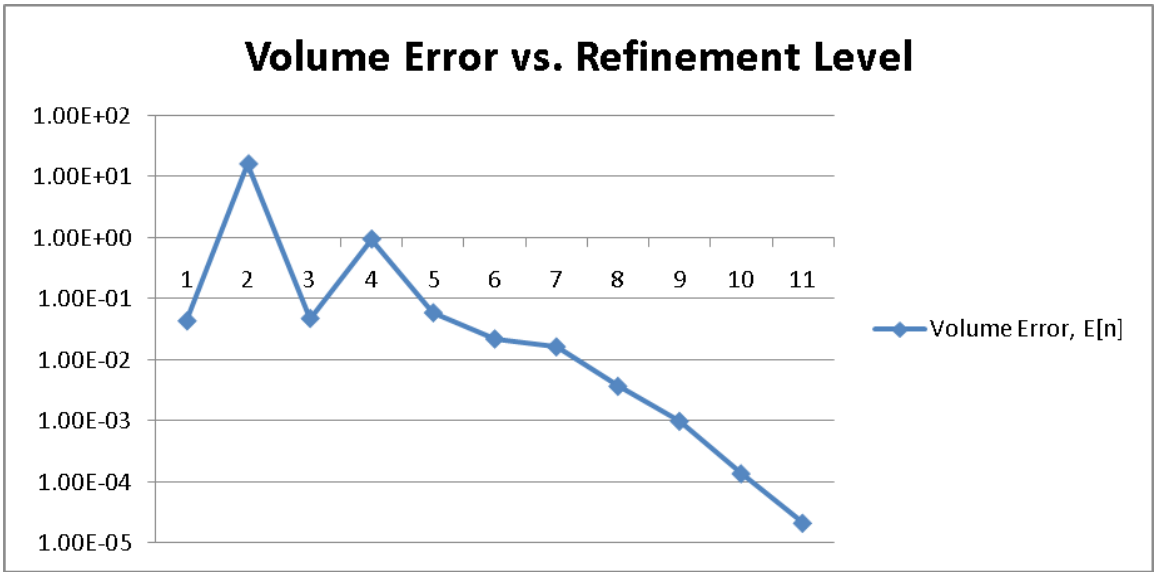
The error for constrained AMR discretization of the CG sphere is shown in Table I and Figure 19 for various refinement levels. Beyond the noise at small refinement levels, Figure 19a shows a roughly linear relationship on a log scale, hence the error is decaying exponentially. Figure 19b shows the error for refinement levels n = 7, 8, 9, 10, 11 as fit to an exponential. The calculated an expoential fit is:

$$E[n] = 1.028 \times 10^{-1} (5.27)^{n} \tag{3}$$

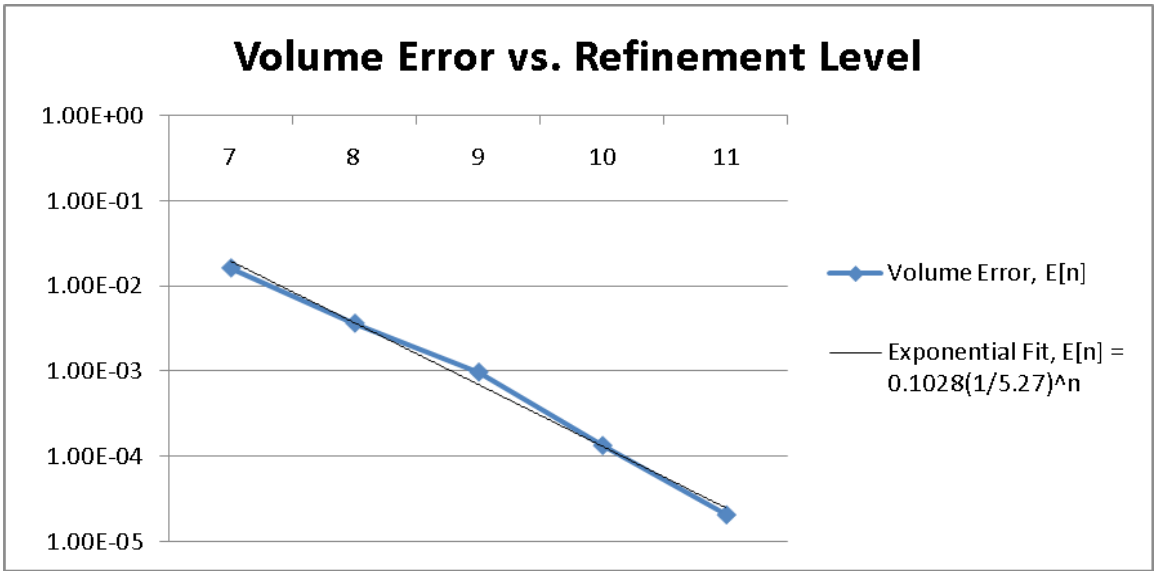
This says that the error decreses by a factor of about 5.27 when the refinement is increased by one level. This is roughly what is expected, since each graphics cell is divided into 8 subcells as for each increase in refinement level. Therefore, one would expect the error to go down by at most a factor of 8 for each increase in refinement level.

4.6 Geometry Error Detection

It can be very difficult to ensure that your CG is defined correctly. ***Mercury*** has "gap" and "overlap" detection built into its graphics mesh options. Anytime a graphics mesh is created, the code will automatically find "gaps", which are defined to be points in space where no CG cell claims ownership of that location. This implementation employs the ***Mercury*** routine CG racking routine "Which cell am I in?" at each node in the graphics mesh. If no CG cell claims ownership of the point, then a "gap" has been found. Overlap detection must be requested by the user, since it is a more expensive calculation. For each graphics mesh point, the code loops over each CG cell and determines if the CG cell thinks it owns that point in space. If more than one CG cell claim ownership of the point, then "overlapping" cells have been found. The void and overlap detection are based on the graphics mesh resolution that the user requests. Since ***Mercury*** samples the CG at the nodes of the graphics mesh, if gaps or overlaps are smaller than the mesh resolution, this sampling method will not detect those errors.



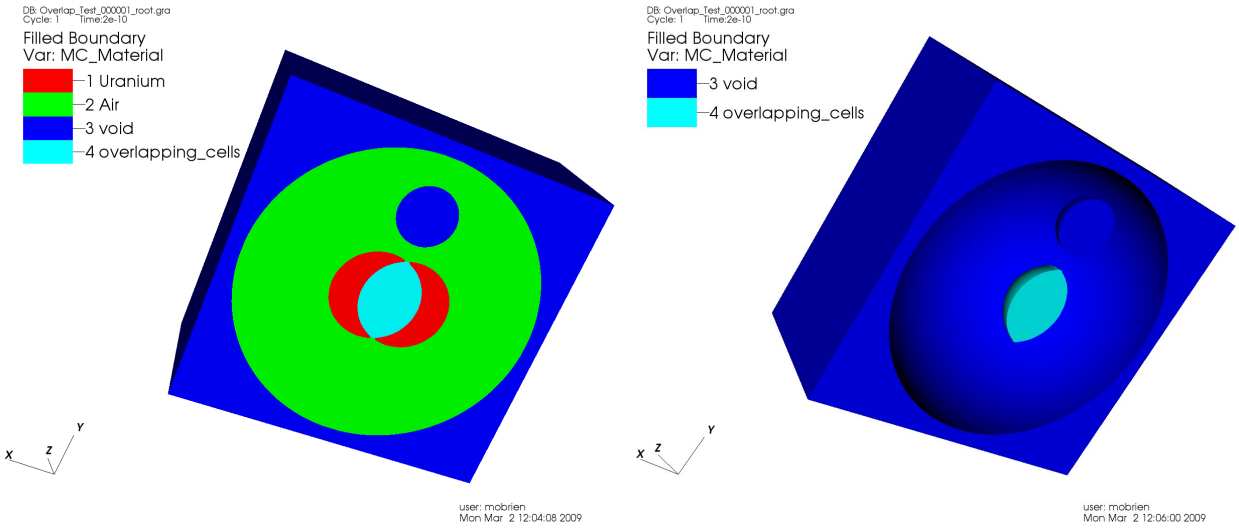
(a)



(b)

Figure 19. Plot of (a) the calculated error and (b) the error fit to an exponential in the volume of a sphere versus refinement level in the constrained AMR method.

A simple example of gaps and overlaps is shown in Figure 20. Figure 20a shows 2 red uranium spheres that are overlapping (shown as cyan *overlapping_cells*), a blue "gap" sphere (shown as *void*) that has been excluded from the green air sphere, and the blue"gap" volume outside the air sphere. The volumes assigned to the *void* and *overlapping_cells* regions is shown in Figure 20b. It is the user's responsibility to decide if the *void* and *overlapping_cells* regions are valid. For example, in this problem, a vacuum boundary condition has been applied to the outer surface of the green air sphere. Therefore, the *void* region outside of the green sphere is valid. However,



(a) (b)

```
Material Uranium            Mass 2.0408300111e+03 Volume 5.4451174255e+01
Material Air                Mass 2.3926665283e+00 Volume 1.9938887736e+03
Material void               Mass 0.0000000000e+00 Volume 3.2463340822e+03
Material overlapping_cells  Mass 0.0000000000e+00 Volume 2.9325969955e+01
-------------------------------------------------------------------------
Total                       Mass 2.0432226776e+03 Volume 5.3240000000e+03
```

(c)

Figure 20. A simple example of gaps and overlaps in a combinatorial geometry: (a) a material plot showing all regions, (b) a material plot showing only the *void* (gap) and *overlapping_cells* (overlap) regions, and (c) a printed tale of the "masses" and volumes of each region.

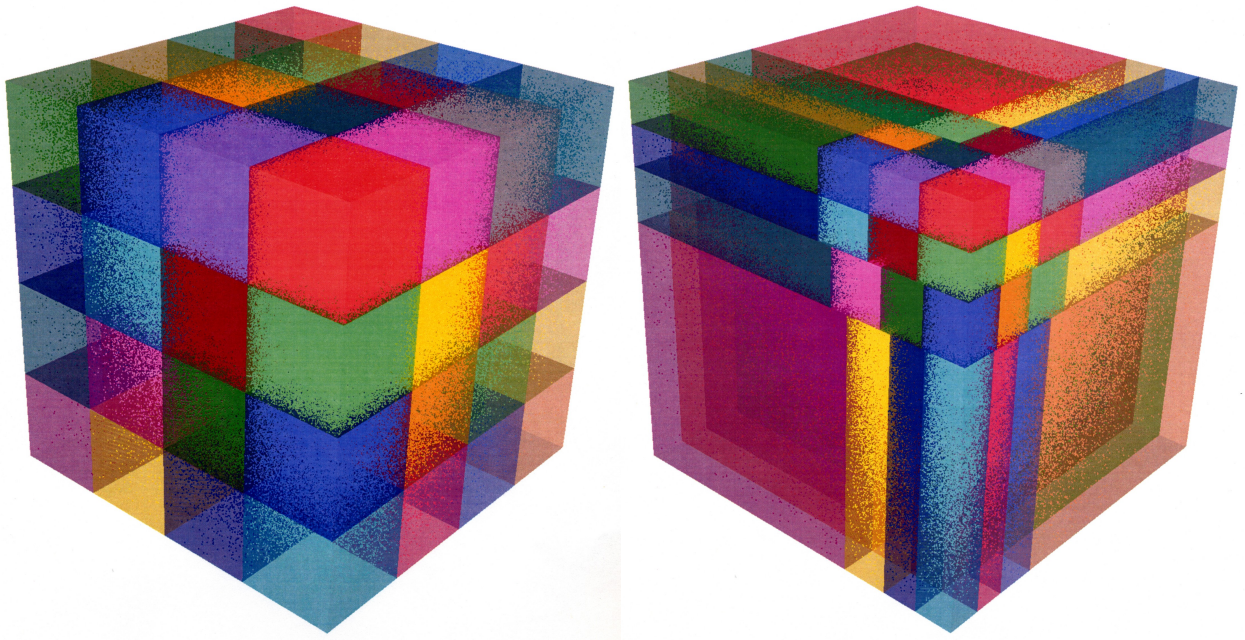
no boundary condition has been applied to the small, internal blue *void* sphere, hence this is a geometry setup error. When discretizing the CG onto the graphics mesh, ***Mercury*** prints of the mass and volume of all regions / materials in the problem, including any *void* or *overlapping_cells* "material" found., as shown in Figure 20c.

5 IMAGE GALLERY

This section presents images from ***VisIt*** that demonstrate how visualization has been used to aid in understanding and verification of ***Mercury*** code development issues.

5.1 Dynamic Load Balancing

Mercury supports a method of dynamically balancing the particle work load by redcomposition of the spatial domains. ***VisIt*** can be used to visualize the domain decomposition to verify that it intuitively makes sense. Figure 21 shows the spatial domain decomposition of a 3D Cartesian

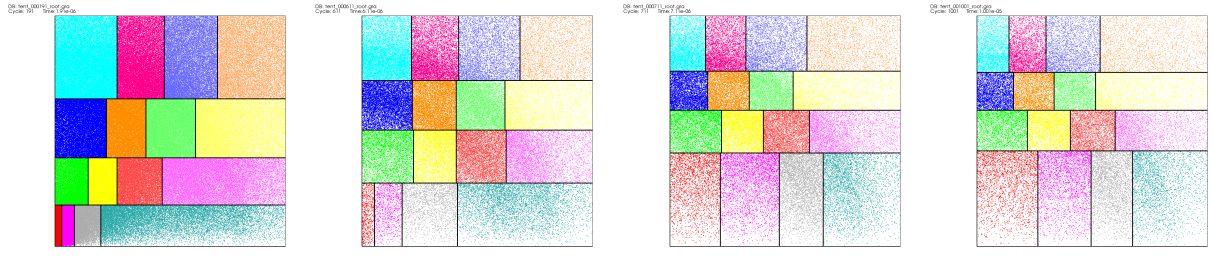


(a) (b)

Figure 21. Two possible decompositions of supercritical-uranium-sphere criticality problem into 64 spatial domains: (a) a uniform decomposition, and (b) a load balanced decomposition.

mesh that is used to model an octant of a supercritical uranium sphere, where the center of the sphere is located at the upper, central corner of the problem. The particles in Figure 21 are color coded by the domain that they reside in. The density of the particles indicate that most of the work load is at the center of the sphere than father out radially. Figure 21a shows a uniform domain decomposition of the problem in 64 domains, while Figure 21b shows the domain decomposition that ***Mercury*** has chosen to balance the work load in each domain.

While particles in ***Mercury*** are tracked in 3D Cartesian space, it is plots the particles on a 2D "cylindrical projection". Figure 22 shows such a projection, where the particle's Cartesian coor-



(a) (b) (c) (d)

Figure 22. Evolution of the domain decomposition in a supercritical-uranium-sphere criticality problem in response to the evolution of the particle workload.

dinates (x, y, z) have been mapped to cylindrical coordinates $(z, r) = \left(z, \sqrt{(x^2+y^2)}\right)$, and the particles are color coded by domain. Moving from left to right (a to d) in Figure 22, one can see that the domain decomposition is varying dynamically in response to the evolution of the particle workload, as the particles migrate from their source at the center of the sphere to the outer reaches of the sphere.

5.2 Material Interface Reconstruction

Figure 23 shows a particle plot for a 2D cylindrical mesh problem, where a cylindrical projection has been applied to the particle coordinates. This image has been used to verify the material interface reconstruction (MIR) algorithm within ***Mercury***. First note the bold black line that identifies the material interface. It is computed to be normal to the gradient of the material volume fraction. The position of the line has found to match the input volume fractions of the materials in the underlying mesh. Next note that each particle is colored by the material that it is in. It took many iterations of code development, followed by visualization, to obtain this image. These iterations relied upon the ease of spotting a particle that is colored incorrectly based upon its location. This type of visualization is extremely valuable for validating code development.

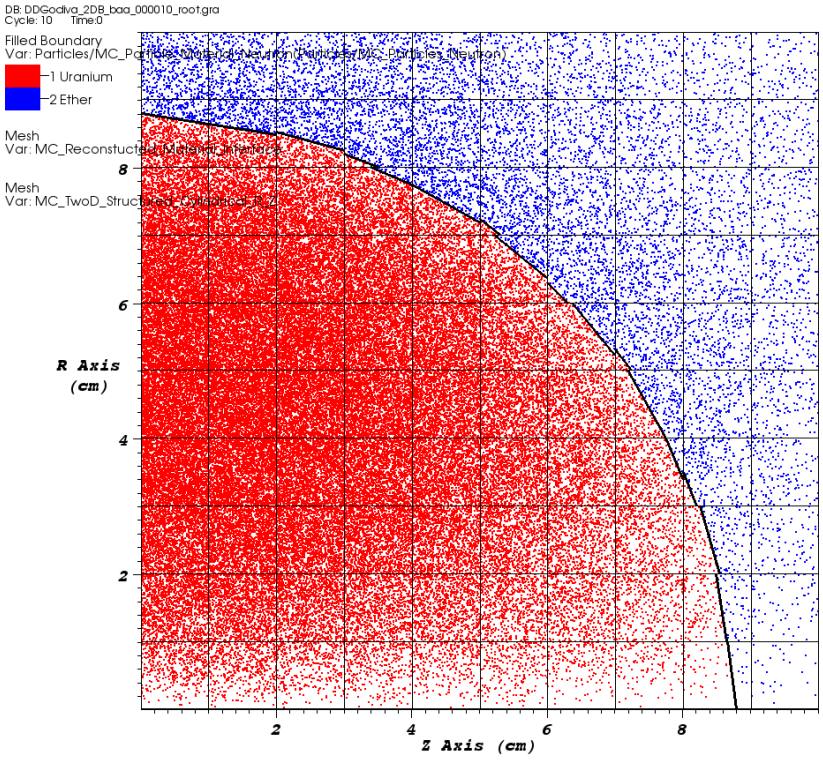
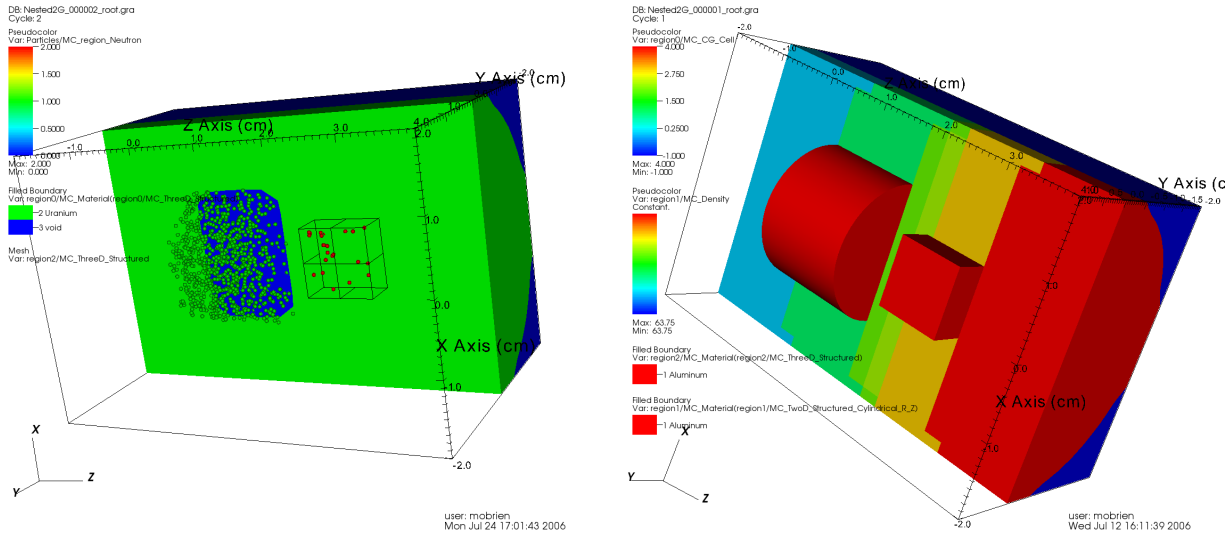


Figure 23. A particle plot which demonstrates the validity of the material interface reconstruction (MIR) algorithm in *Mercury*: particles which are color coded by material are on the correct side of the black material interface.



(a) (b)

Figure 24. Plots of am embedded mesh problem with a combinatorial geometry (CG) region surrounding a 2D cylindrical and 3D Cartesian mesh: (a) the particles are color coded by the region they reside in, and (b) the CG cells are color coded by their cell index.

5.3 Embedded Mesh in Combinatorial Geometry

Mercury has the ability to embed a mesh region with a combinatorial geometry region (see the paper by Green man, *et al.* in these proceedings for further information). Figure 24 shows a combinatorial geometry region with 2 embedded mesh regions: one 2D cylindrical mesh and one 3D Cartesian mesh. In Figure 24a, the particles are color coded by region. In this case, particles are blue if they reside within the CG region (none are present), green if they reside within the 2D cylindrical mesh region, and red if they reside within the 3D Cartesian mesh region. Figure 24b shows a cutaway of the CG cells that are color coded by cell index, as well as the outer boundaries of the 2D and 3D embedded meshes.

5.4 Setup Verification for Complex Geometries

Several examples of complex combinatorial geometries that have been developed with ***Mercury*** and visualized with ***VisIt*** are shown in Figure 25. These images are (Figures 25a and 25b) the National Ignition Facility (NIF) target chamber and support structures, (Figure 25c) a fusion shield test facility with particles that are color coded by kinetic energy, (Figure 25d) a concentric sphere criticality test problem also with particles that are color coded by kinetic energy, and (Figures 25e and 25 f) a uranium and beryllium fast-spectrum critical assembly.

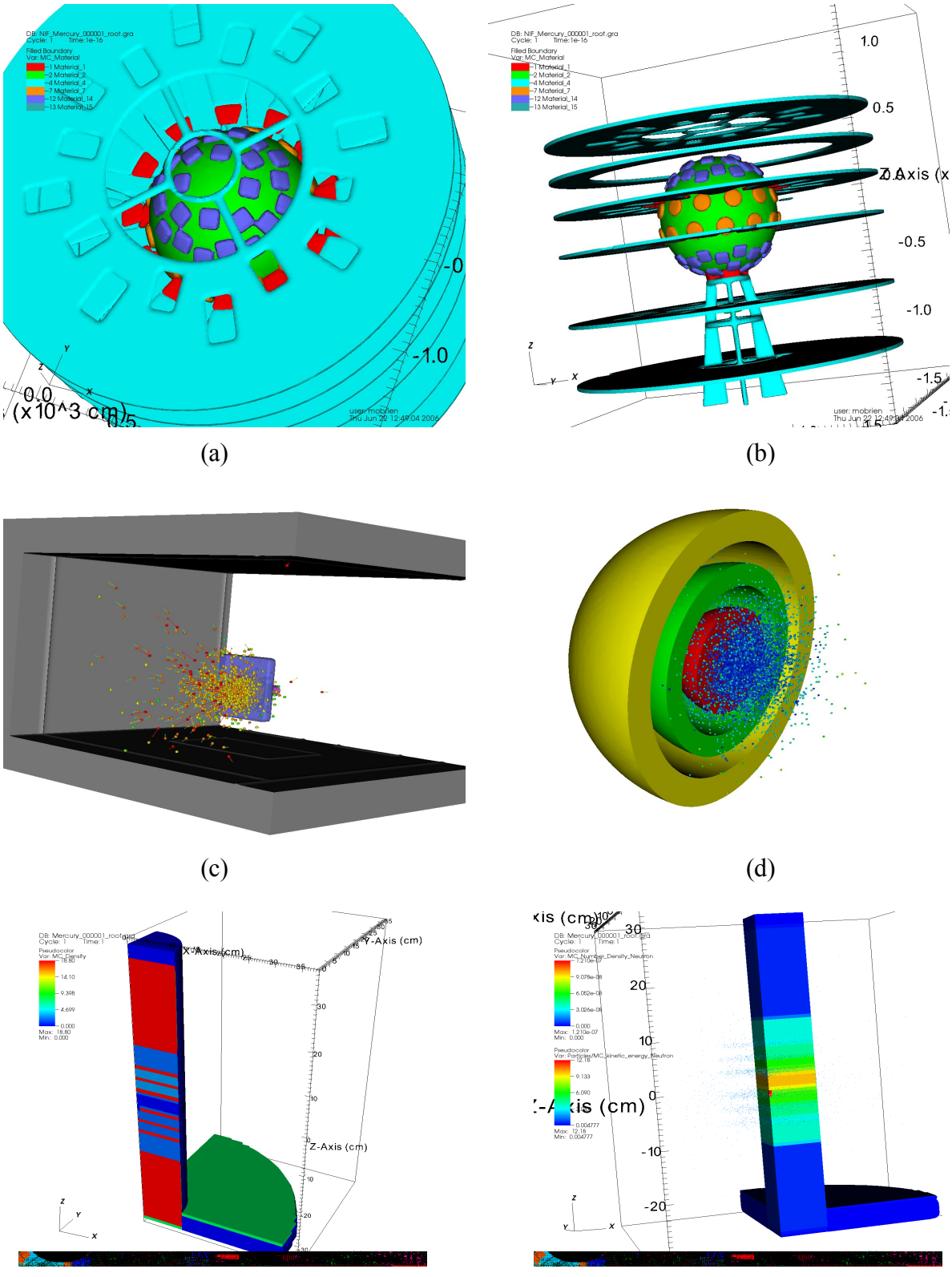


Figure 25. A variety of complex geometries that were developed in *Mercury* and visualized in *VisIt*.

6 SUMMARY

This paper has demonstrated how the ***VisIt*** visualization tool has been integrated with the ***Mercury*** Monte Carlo particle transport code to provide real-time visualization and analysis capabilities of running problems. This integrated analysis tool has been shown to be invaluable for verification of algorithms during code development, user verification of problem geometry setup, and visualization of simulation results. Several algorithms for converting combinatorial geometry data to mesh data for visualization with mesh-based tools (such as ***VisIt***) have been presented. Extensions to ***VisIt*** that permit native discretization and visualization of combinatorial geometry data have also been described. Finally, the paper has discussed the embedding of a Python interpreter within ***Mercury***. This feature can be used to launch and attach ***VisIt*** to ***Mercury***, and to send ***VisIt Python*** scripts which contain plot commands to automatically create plots in ***VisIt*** data windows.

ACKNOWLEDGMENTS

This work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

The authors would thank ***VisIt*** code developers Brad Whitlock for implementing the inline ***VisIt*** interface and for adding support for combinatorial geometry to that interface, and Mark Miller for implementing the ***VisIt*** discretization of combinatorial geometry data for visualization in ***VisIt*** and for various performance and usability enhancements.

REFERENCES

1. "VisIt Web Site", Lawrence Livermore National Laboratory, http://www.llnl.gov/visit (2008).
2. R. J. Procassini, et al., "*Mercury User Guide (Version c.2)*", Lawrence Livermore National Laboratory, Report UCRL-TM-204296, Revision 1 (2008).
3. "Mercury Web Site", Lawrence Livermore National Laboratory, http://www.llnl.gov/mercury (2009).
4. The VisIt Code Team, "*Getting Data Into VisIt (Version 1.5.4)*", Lawrence Livermore National Laboratory, Report UCRL-SM-224277 (2006).
5. The VisIt Code Team, "*VisIt Python Interface Manual (Version 1.4.1)*", Lawrence Livermore National Laboratory, Report UCRL-SM-209589 (2005).