LA-UR-15-28026

Approved for public release;
distribution is unlimited.

Title: In-Situ Visualization Experiments with ParaView Cinema in RAGE

Author(s): Robert J. Kares
Applied Computational Physics (XCP) Division
Los Alamos National Laboratory

Intended for: Report
October 15, 2015

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Form 836 (7/06)

This page is intentionally left blank.

In-Situ Visualization Experiments with ParaView Cinema in RAGE

Robert J. Kares

Applied Computational Physics (XCP) Division, Los Alamos National Laboratory
Los Alamos, NM 87545

A previous paper [1] described some numerical experiments performed using the ParaView/Catalyst in-situ visualization infrastructure deployed in the Los Alamos RAGE radiation-hydrodynamics code to produce images from a running large scale 3D ICF simulation. One challenge of the in-situ approach apparent in these experiments was the difficulty of choosing parameters likes isosurface values for the visualizations to be produced from the running simulation without the benefit of prior knowledge of the simulation results and the resultant cost of recomputing in-situ generated images when parameters are chosen sub-optimally. A proposed method of addressing this difficulty is to simply render multiple images at runtime with a range of possible parameter values to produce a large database of images and to provide the user with a tool for managing the resulting database of imagery. Recently, ParaView/Catalyst has been extended to include such a capability via the so-called Cinema framework. Here I describe some initial experiments with the first delivery of Cinema and make some recommendations for future extensions of Cinema's capabilities.

I. Introduction

In-situ visualization is a complementary approach to traditional post-processing visualization methods in which images are produced directly from within a running physics simulation as a problem is being computed in order to avoid writing out large volumes of simulation data to disk. In a previous paper [1] I described a comprehensive set of numerical experiments performed to test the in-situ visualization concept using the ParaView/Catalyst in-situ visualization software [2] deployed in the Los Alamos RAGE radiation-hydrodynamics code [3] to produce images from a running large scale state-of-the-art 3D ICF simulation on the Cielo supercomputer at Los Alamos.

As part of a 2013 Advanced Strategic Computing (ASC) program Level II milestone [4], the Los Alamos ASC code RAGE was modified to allow it to render images directly from a running problem using the ParaView/Catalyst library. These modifications consisted of installing ParaView adaptor library calls in RAGE to copy data from the adaptive mesh refinement (AMR) data structures used by RAGE into a VTK unstructured grid format suitable for use by the ParaView/Catalyst package and linking the RAGE code with the ParaView/Catalyst library. The adaptor library calls in RAGE convert the 3D AMR mesh used by RAGE into an unstructured

3D mesh that is used by the ParaView/Catalyst library to render images directly from the running RAGE problem under the control of a custom ParaView visualization pipeline defined by the user. The visualization pipeline which defines the image to be rendered must be created interactively by the user with the ParaView software in a separate off-line step and is exported to ParaView/Catalyst as a Python .py file containing the pipeline definition. This .py file then becomes a part of the input to RAGE that defines the images to be produced.

Utilizing this infrastructure I was able to successfully produce a variety of interesting and non-trivial in-situ visualizations from 2D and 3D RAGE simulations of the developing turbulence in an imploding ICF capsule as described in Reference [1]. While the in-situ visualization experiments described in Reference [1] demonstrate the power of the in-situ approach, they also reveal some of its difficulties. One such difficulty is illustrated in Fig. 1 which shows two snapshots at the same simulation time taken from a 3.19 billion cell 3D RAGE simulation of the developing turbulence in the ICF capsule. In each of the snapshots the vertical face of the gas is colored by the azimuthal component of vorticity and the horizontal face is colored by the gradient of pressure. The off-white surface is the plastic/gas interface of the imploding capsule. The gray tubes visible in the snapshots are isosurfaces of constant total vorticity and represent counter-rotating vortex rings in the gas. These vortex tubes are colored by the axial component of vorticity whose color palette has been chosen so that the zero level is represented in gray. At time t = 1.4 ns the 3D simulation began from a completely axisymmetric state with the vorticity purely in the azimuthal direction. As the 3D simulation progresses, the appearance of alternating yellow and blue regions on the gray tubes indicate the development of a non-zero axial component for the total vorticity as a result of the growth of turbulence in the gas. Figs. 1(a) and (b) are in-situ generated images taken from the same 3D RAGE simulation and differ only in the degree of magnification of the capsule view and in the choice of constant value for the isosurface of total vorticity that represents the interacting vortex tubes. In Fig. 1(a) a constant value of $2.5\times10^{11}\ \mathrm{sec}^{-1}$ has been chosen for the vorticity isosurface while in Fig. 1(b) a constant value of $1\times10^{12}\ \mathrm{sec}^{-1}$ has been chosen instead.

An examination of the gas bubbles nearest the polar axis of the capsule in Fig. 1(a) shows that the vortex rings in that region are undergoing some type of turbulent evolution as the appearance of the blue and yellow regions on the rings indicate. However, because of the choice of isosurface value, most of the very intricate structures in the evolving vortex tubes in this region are completely invisible in this image. To exhibit these structures with in-situ visualization I had to backup and restart the 3D simulation from t = 1.55 ns with the rendered view zoomed in on the capsule and with a value for the vorticity isosurface of $1\times10^{12}\ \mathrm{sec}^{-1}$ more appropriate for observing the dynamical evolution of the vortex cores in this region of the gas. Fig. 1(b) shows the resulting time snapshot of the evolving vortex tubes. A quick comparison of Figs. 1(a) and 1(b) clearly illustrates how the intricate structure of the vorticity in the developing turbulence can be missed in this problem without a judicious choice for the vorticity isosurface value.

I happened to already know that 1×10^{12} sec^{-1} is a good choice for the isosurface in this case because I had previously done a great deal of interactive data exploration on this problem. Without this a priori knowledge of the correct value to pick, I might have had to do a lot of expensive and time consuming recomputing of this problem in order to obtain a useful understanding of just how the vortex tubes evolve to a fully turbulent state, which is the entire point of doing the simulation in the first place. The choice of isosurface value is further complicated by the fact that magnitude of the total vorticity in the simulation is resolution dependent making it difficult to guess a nominal value for the vorticity isosurface by performing lower resolution simulations that are computationally less expensive.

So in this example a less than optimal choice for the value of the vorticity isosurface resulted in the production of a visualization of the vortex tubes in which the dynamics of the tube interactions, the principal phenomenon we wished to exhibit with the simulation, was essentially invisible. Correcting that error required significant recomputing to back up and render time snapshots with a value of the vorticity isosurface more appropriate for capturing the central phenomenon of interest. In fact, the problem of correctly choosing parameters for the in-situ visualization is perhaps the most challenging aspect of the in-situ approach.

Proposals have been made to circumvent this problem by rendering in-situ multiple views with a range of parameter values. It seems clear that in the absence of a priori knowledge that comes from interactive data exploration in a post-processing methodology, the ability to render multiple views with a range of parameters and to organize the resulting database of imagery is a minimal requirement for the successful application of in-situ visualization for scientific discovery with large scale computing. A specific framework for managing the production and viewing of large collections of images rendered from a running physics code using Catalyst, the so-called Cinema framework, is currently being developed [5].

In this paper I report results from some initial experiments using the first (specA) delivery of Cinema to address some of the problems encountered with the in-situ visualization of the 3D ICF simulation described in Reference [1]. I also present some recommendations for the development of future Cinema capabilities.

II. Capabilities of the specA Delivery of Cinema

The first or so-called specA delivery of Cinema has been released for user beta testing in ParaView/Catalyst Version 4.3.1 and provides a limited set of extensions to Catalyst that allow the user to easily generate a JSON (JavaScript Object Notation) database of images by moving the camera through a defined range of views and/or by varying the parameters of clips and isosurfaces through a defined range of values. When interactively creating the Catalyst pipeline, the user defines a range of camera views or parameter values to be rendered. Then when this pipeline is run in the Catalyst/Cinema-enabled application code, a directory called “cinema” is

created containing the subtree of rendered imagery for the various timesteps together with a file called “info.json” that organizes these images into a database with accompanying metadata. An image viewer called the Qtviewer is provided that takes the .json file as input and allows the user to browse the various images in the database over a range of times, camera views and parameter values. SpecA is intended to be a prototype for a more comprehensive set of future Cinema capabilities.

In specA of Cinema two different camera types are available: “Static” and “Spherical”. The static camera option gives the single camera view defined by the user when the Catalyst pipeline was created. In contrast, the spherical camera allows the user to specify a range of discrete values for the angular position of the camera in theta and phi from which to render multiple views of the same scene.

SpecA of Cinema also introduces the notion of a track. A track is a set of discrete parameter values that are used to control the definition of a filter in a visualization pipeline. An example of a track is a range of isovalues for some isosurface. Cinema walks through the parameter values in a track rendering an image of the visualization pipeline for each value using the currently selected camera type. In specA of Cinema two types of track are supported: a track that controls the position of a clip along an axis, and a track that controls the isovalue of some isosurface. An arbitrary number of tracks may be defined simultaneously, and an image is rendered for each independent combination of track parameters using the currently selected camera type.

III. Cinema Experiments with the 3D RAGE ICF Simulation Problem

The specA delivery of Cinema was made available to me for testing though a specialty version of ParaView/Catalyst 4.3.1 that was built and maintained on the open Moonlight and Lightshow clusters at LANL by John Patchett of the ASC Research Viz team. John also built the latest version of the adaptor library required to use ParaView/Catalyst with RAGE on Moonlight and provided me with a complete build tree for RAGE containing the Cinema-capable in-situ visualization components from ParaView/Catalyst that could be used to modify and rebuild the RAGE code. As discussed in Reference [1] such a build tree is needed because the ICF capsule problem requires the installation of a custom energy source in the esrcs.f90 module of RAGE that needs to be compiled into the code to produce the asymmetric pressure drive used in the ICF problem. The use of the Moonlight platform facilitated the ease of installing required updates to the Cinema code but it also had the effect of limiting the size of the 3D ICF simulation that could be run because of the number of processors practically available on the open Moonlight machine. As a result I had to restrict my experiments to the initial phase of the 3D simulation just after link time t = 1.4 ns where the total cell count is limited to only about 226 million hexahedral AMR cells, a size that comfortably fits on 512 Moonlight processors. I also had to recreate by hand on the open Moonlight machine essentially all of the infrastructure of the 3D RAGE ICF problem

originally run on Cielo including the input decks, the problem geometry and EOS files and the ParaView state files, a process that required a considerable investment in time to complete.

Once the complete infrastructure for the ICF implosion problem was in place on the Moonlight machine, I used this infrastructure to run the initial 2D phase of the implosion from $t = 0$ out to the link time of $t = 1.4$ ns with in-situ visualization turned on and was easily able to successfully reproduce the images that I previously obtained on the Cielo machine (cf. Fig. 3 of Reference [1]) as a non-trivial check of both the code and the problem setup. With the confidence thus obtained I then attempted to run the 3D phase of the implosion, again with in-situ visualization turned on, starting from the link at $t = 1.4$ ns in order to reproduce the 3D images that I previously generated on Cielo (cf. Fig. 5 of Reference [1]), and here I immediately encountered a problem. When running the 3D simulation any call to the in-situ visualization package resulted in an immediate core dump of RAGE. This was, of course, rather surprising since I had just run the 2D simulation with the same calls to ParaView/Catalyst, without any apparent problem. And Cinema was not an issue in this case because no Cinema calls were included in these initial tests.

This puzzling situation was eventually resolved when it was realized that, between the time of my Cielo tests and the tests performed on Moonlight, modifications had been made to the ParaView/Catalyst adaptor library for RAGE in order to improve its treatment of 3D ghost cell generation for the VTK data structure. Apparently, the new ghost cell code contained a problem that only appears with a relatively large number of CPUs and my use of 512 CPUs for the 3D ICF simulation was sufficient to trigger this problem resulting in a crash. The short-term fix was to comment out the new ghost cell code in the adaptor library effectively returning to the Cielo version of the adaptor. This experience provides an interesting case in point about the difficulties that can be encountered in tracking down the source of failures when using in-situ calls embedded in a large physics application.

Once this issue was resolved and the basic in-situ visualization functionality was working correctly in RAGE on Moonlight, I turned my attention to testing of the new Cinema features. In the course of this testing two minor bugs in the Cinema code were discovered and fixed and their description is relegated to Appendix A. Here I focus instead on some selected results of the testing process.

IV. Experiments with the Cinema Spherical Camera

Fig. 2 shows the basic Python pipeline file with the Cinema extension for the spherical camera used to generate in-situ images from the 3D RAGE simulation. This pipeline is called by adding the following command lines for ParaView/Catalyst to the RAGE input desk:

```
!==============================================================================
! ----- PLOTS PARAVIEW IN SITU
!==============================================================================
```

```
do_pv_insitu = .true.                      ! if this is false next two must be also
do_pv_insitu_gate = .false.                ! allow gate filter on output
do_pv_insitu_camera = .false.              ! allow camera to move by data

pv_use_python = .true.                     ! python pipeline vs hardcoded pipeline
pv_python_script = 'CinemaVortTubes.py'    ! if python, execute this script

pv_insitu_dt = 0.0005e-9                   ! time delta for coprocessing

npv_insitu_mesh = 9                        ! number of insitu variables and names
pv_insitu_mesh(1) = 'rho', 'grd', 'mat', 'prs', 'v02', 'vel', 'xdt', 'ydt', 'zdt'

pv_insitu_camera(1) = 0,0,0,0,2,0,0,0,0    ! 0=NOAUTO, 1=ZOOMOUT, 2=ZOOMOUTIN

pv_insitu_xmn(1) = 9*-400e-4               ! initial camera bounds
pv_insitu_xmx(1) = 9*400e-4
pv_insitu_ymn(1) = 9*-400e-4
pv_insitu_ymx(1) = 9*400e-4

pv_insitu_camera_weight(1) = 9*1           ! number of weights used in auto-camera
pv_insitu_camera_previous_bounds(1) = 9*1 ! number of previous bounds to use
pv_insitu_camera_max_frames(1)= 9*1        ! maximum of frames before zooming-in
pv_insitu_camera_x_bins(1)=9*200           ! number of spatial bins in x dimension
pv_insitu_camera_y_bins(1)=9*400           ! number of spatial bins in y dimension
pv_insitu_camera_z_bins(1)=9*1             ! number of spatial bins in z dimension
pv_insitu_camera_s_bins(1)=9*100
```

in order to produce in-situ images.

The Cinema camera extension in this pipeline consists of a modification to the coprocessor.RegisterView call which adds a new argument of the form cinema={list}:

```
coprocessor.RegisterView(renderView1,filename='image_%t.png', freq=1, fittoscreen=0,
magnification=1, width=1920, height=1080, cinema={"camera":"Spherical", "phi":[-180,
-150,-120,-90,-60,-30,0,30,60,90,120,150], "theta":[-180,-150,-120,-90,-60,-30,
0,30,60,90,120,150] })
```

that defines a spherical camera and provides the discrete set of 12 X 12 = 144 angular positions theta and phi from which Cinema will automatically render the camera view at each output time. Setting cinema={} in this call turns Cinema off and results in the production of the single in-situ image defined by the rendering pipeline with its associated view.

When the pipeline of Fig. 2 is executed in RAGE, Cinema creates a subdirectory called “cinema/image” in the RAGE problem base directory into which it writes individual named subdirectories for each time dump, labelled by the corresponding simulation time of the dump in e-format. Each of these dump directories in turn contains further subdirectories with the 12 X 12 = 144 camera images, organized first by phi value and then by theta value. Also included in “cinema/image” is a small metadata file called “info.json” that organizes the camera images by time and by angular position for the Qtviewer image browser.

The resulting image database can then be browsed interactively using the Qtviewer by typing:

qtviewer info.json

In the Qtviewer holding down the left mouse button and dragging vertically and/or horizontally along the screen directions can be used to smoothly move through theta and phi positions in the image database in a reasonable imitation of interactive rendering of the 3D scene. The right mouse button can also be used to zoom the image. Time step can be changed using a slider control on the right hand side of the Qtviewer's GUI. Fig. 3 shows a sequence of 8 images from the 3D ICF simulation at time t = 1.41 ns generated with the spherical camera that have been chosen to give some sense of the experience of viewing scene transformations with the Qtviewer.

One capability that the Qtviewer does not have at the moment and that is much needed is the ability to easily save a sequence of images from the browser as a consecutively numbered sequence of image files with a prefix name. I found, for example, that preparing the simple image sequence of Fig. 3 was rather difficult by directly copying image files out of the database. This would have been much easier if I were able to save a sequence of files directly out of the image browser instead.

V. Experiments with Cinema Tracks

I performed some simple experiments with the new track concept in Cinema by creating a Cinema-aware pipeline to automatically generate visualizations of the 3D ICF problem with a range of values for the isosurface of total vorticity using a contour track. For these experiments I chose a static camera view so that the RegisterView call in the pipeline file took the form:

```
coprocessor.RegisterView(renderView1,filename='image_%t.png', freq=1, fittoscreen=0,
magnification=1, width=1920, height=1080, cinema={"camera":"Static"})
```

The track is implemented in the pipeline script by the addition of a call to the function coprocessor.RegisterCinemaTrack corresponding to the contour filter to be varied, in this case contour2 which is the isosurface of total vorticity:

```
# create a new 'Contour'
contour2 = Contour(Input=calculator3)
contour2.ContourBy = ['POINTS', 'MagVort']
contour2.Isosurfaces = [5.e10]
contour2.PointMergeMethod = 'Uniform Binning'

# register the filter with the coprocessor's cinema generator
coprocessor.RegisterCinemaTrack('contour', contour2, 'Isosurfaces', [5.e10, 6.e10,
7.e10, 8.e10, 9.e10, 10.e10])
```

Fig. 4 shows the resulting set of 6 images generated by the contour2 track for the 3D ICF problem at time t = 1.41 ns.

I also performed some simple experiments with the new slice track in Cinema by creating a Cinema-aware pipeline to automatically generate visualizations of the 3D ICF problem with a range of positions for the slice plane that is used to display the pressure gradient inside the gas. For these experiments I again chose a static camera view so that the RegisterView call in the pipeline file took the form:

```
coprocessor.RegisterView(renderView1,filename='image_%t.png', freq=1, fittoscreen=0,
magnification=1, width=1920, height=1080, cinema={"camera":"Static"})
```

The track is implemented in the pipeline script by the addition of a call to the function coprocessor.RegisterCinemaTrack corresponding to the slice filter to be varied, in this case slice2:

```
# create a new 'Slice'
      slice2 = Slice(Input=computeDerivatives2)
      slice2.SliceType = 'Plane'
      slice2.SliceOffsetValues = [0.0]

      # init the 'Plane' selected for 'SliceType'
      slice2.SliceType.Origin = [1e-10, 0.050000000745058101, 0.050000000745058101]

      # register the filter with the coprocessor's cinema generator
      coprocessor.RegisterCinemaTrack('slice', slice2, 'SliceOffsetValues', [0, 0.001,
0.002, 0.003, 0.004, 0.005 ])
```

It should be noted that slice2 is not displayed directly in the visualization. Rather, I have routed the output of the slice2 filter to a sequence of further filters that extracts the region of the plane containing the gas by taking an isovolume of the gas volume fraction between 0.5 and 1, reflects this region of the slice plane about Z, and colors the resulting reflection by the gradient of the pressure in order to display the position of the shocks inside the gas. When the slice track is executed to update the slice filter, then everything that depends on the output of this filter is updated as well. Fig. 5 shows the resulting set of 6 images generated by the slice2 track for the 3D ICF problem at time t = 1.41 ns.

VI. Resource Usage by Cinema in RAGE

It is of interest to examine the cost in run time and memory usage of generating images in-situ with Cinema in RAGE, particularly for the case of the spherical camera where images from a large number of views need to be generated in order to provide the user with an interactive viewing experience. To get some idea of the magnitude of these costs I performed a series of simple tests using my 512 CPU, 226 million cell RAGE 3D ICF simulation. I selected a short 66 cycle long time segment from this simulation beginning at simulation time t = 1.40488 ns and ran this segment of the simulation forward in time with several different in-situ visualization options in RAGE starting from the same 68 GB restart dump. This time segment of the full simulation was chosen because during this period I forced uniform 0.2 μm zoning within a fixed

spatial radius of 140 μm which encompasses the entire capsule so that while the AMR mesh adaptation is active during this period, the total AMR cell count remains essentially constant over time because the active region of the problem is already zoned at the maximum permitted AMR resolution. The 66 cycle length for the test segment was chosen to include a total of 11 calls to the in-situ visualization package. Both run time and memory use were measured using diagnostics obtained from the RAGE log file.

For measuring run time RAGE prints upon completion of a segment of the simulation the wallclock time for the execution of that segment. For measuring memory usage RAGE periodically prints to its log file a quantity called RSS_MAX which is the memory high water mark for the Resident Set Size, the total size of the process residing in memory, obtained from a call to the operating system. (Note that RSS_MAX is reset to zero at every restart.) This value is computed on each node of the job, and then the Min, Mean and Max for RSS_MAX over all the nodes is reported to the RAGE log file as a percentage of the total available memory on a node. For the test runs over the chosen time segment I report the observed Mean value of RSS_MAX over the nodes.

Six separate test runs over the chosen time segment were made. In the first baseline run the generation of in-situ images was completed turned off. In the second, I used the basic pipeline of Fig. 2 in Catalyst with Cinema turned off (cinema={} in the coprocessor.RegisterView call) to generate a single in-situ image per time dump. The third test used a Cinema contour track similar to the one discussed above to generate a total of 7 images per time dump, one for each value of the vorticity isosurface from $4\times10^{10}\ \mathrm{sec}^{-1}$ to $10\times10^{10}\ \mathrm{sec}^{-1}$ in steps of $1\times10^{10}\ \mathrm{sec}^{-1}$. The fourth test used a Cinema slice track similar to the one discussed above to generate 7 images per time dump, one for each value of the slice position along the X axis from X = 0. To X = 0.06 in steps of 0.01. The fifth test used the Cinema spherical camera to generate 12 X 12 = 144 independent views of the basic pipeline per time dump. The sixth and final test combined the use of the slice track with the spherical camera to generate 7 X 144 = 1008 images per time dump. Table 1 summarizes the timing and memory use results of these simple tests.

	RSS_MAX Mean %	Run Time (sec)	# of In-Situ Images
No in-situ viz calls	56.58	3988.1	0
Catalyst, no Cinema	71.01	4106.9	11
Cinema static camera with contour track	71.05	4153.3	11 x 7 = 77
Cinema static camera with slice track	71.11	4499.1	11 x 7 = 77
Cinema spherical camera	71.44	5213.4	11 x 12 x 12 = 1584
Cinema spherical camera with slice track	71.76	13108.1	11 x 7 x 12 x 12 = 11088

Table 1. Timing and memory usage results from six RAGE test runs to examine resource utilization of Cinema features for in-situ visualization.

A brief examination of the memory usage date in Table 1 suggests that Cinema adds little to the basic memory overhead incurred by Catalyst which for these test cases amounts to a roughly 26% increase in application memory usage relative to the baseline case that makes no calls to the in-situ visualization package. In terms of run time, however, the spherical camera is a bit more expensive. The use of the spherical camera with a default range of views results in an increase in run time of 5213.1/3988.1 = 1.307 or 30.7% relative to the baseline case that makes no calls to the in-situ package. Combining Cinema features further increases this run time overhead. In the final test in Table 1 above, combining the use of the spherical camera with a single slice track with 7 slices produces a total of 11,088 images with a corresponding increase in run time by a factor of 13108.1/3988.1 = 3.287 relative to the baseline case that makes no calls to the in-situ visualization package.

VII. Conclusions and Recommendations

The in-situ visualization capability provided by Catalyst in RAGE provides a host of new opportunities to produce a variety of useful images, plots, histograms and other humanly meaningful data products in real time directly out of the running code at a much higher frequency than is practical with conventional post-processing. With some thoughtful effort this capability could also be applied to automate the production of a standardized set of simulation outputs useful in comparing results across a suite of simulation models. Cinema further extends

this capability by providing simple mechanisms for producing variations on the output of the basic visualization pipeline.

The specA prototype delivery of Cinema, with some very minor corrections, works well as intended and provides two basic and useful extensions to Catalyst. First, the spherical camera provides a mechanism to automatically generate a large database of images of the visualization rendered from a range of camera views. And second, the ability to define tracks to control pipeline filters provides an easy means of generating a range of variations in the basic visualization. Combining these two capabilities results in the generation of a database of imagery that can be browsed using the Qtviewer to produce a credible, albeit limited imitation of interactive data visualization. SpecA Cinema, as a prototype, provides track controls for only two filters, clips and isosurfaces. Future Cinema deliveries will provide more.

It is of interest to return to the basic problem which began this investigation, the problem illustrated in Fig. 1 of creating a useful representation of the vorticity field in the gas by an appropriate choice of value for the isosurface of total vorticity and to ask: does the current delivery of Cinema help us solve this problem ? The answer is: yes, but only partially. With specA of Cinema we now have the ability to automatically render a range of values for the vorticity isosurface, but a comparison of Figs. 1(a) and 1(b) shows that in selecting an appropriate representation of the vorticity field, I changed not only the isosurface value chosen but also the palette range of the vorticity vector component that was used to color the isosurface. This example suggests another capability that would be useful for Cinema, namely, the ability to control not only pipeline filters with a track but also to control variable palette ranges.

One issue that arises in our 3D ICF simulation and many other LANL problems as well is the shrinking in the physical size of the simulation region of interest over time. For the ICF problem, for example, the diameter of the imploding capsule decreases by a factor of 8 over the time range of the full simulation so this is quite a substantial issue. A fixed camera view of the capsule would simply see it shrink out of sight. In Reference [1] I discussed how to use the automatic camera to deal with this problem in a limited way for the 2D portion of the ICF simulation. A more general and perhaps in many respects more satisfactory way of addressing this problem is to add to Cinema a third major capability, the capability to move the basic camera position over time in a well-defined way. With this capability it would be possible to follow the capsule as it shrinks down in size in order to maintain a useful view of the visualization. I often do this in keyframe animations produced in post-processing, and a similar capability in Cinema would be of great use.

Another potential new Catalyst capability of some interest to me would be improved support for the production of stereo images. At the moment it is possible to use the EyeAngle stereo feature of ParaView with Catalyst to render left-right stereo pairs by making changes by hand to the Catalyst pipeline file that define and render left and right images separately by explicitly setting

the stereo parameters in the current RenderView and then making calls to the Render function (see Appendix B). However, the whole process is quite awkward, and a more user-friendly way of producing stereo pairs would be a welcome addition to Catalyst. Correspondingly, the definition of the Cinema image database would need to be expanded to include stereo pairs.

As I have already noted some simple improvements to the Qtviewer image browser would be useful. I would particularly like to see something like a Record button added to the browser interface that provides the user with the ability to easily save out a series of images from the browser as a consecutively numbered sequence of image files with a specified prefix name. And if the Cinema image database were expanded to include stereo images, then the addition of stereo support in QtViewer would be essential.

Finally, as I have pointed out in Section VI above, the memory overhead associated with the use of Cinema for rendering multiple images is not significantly greater than the basic memory overhead incurred in the use of Catalyst for rendering single images of the visualization pipeline. In the above tests the application memory usage by the in-situ visualization package, whether using Cinema features or simply rendering single images of the visualization pipeline, was about 26% greater than for the case in which no calls to the in-situ visualization package were made. As we have seen above, however, the run time overhead of Cinema features like the spherical camera can be significant. In the test described above the use of the spherical camera increased the application run time by 31% relative to the case in which no calls to the in-situ visualization package were made. And, of course, the combined use of more than one Cinema feature quickly multiplies the number of images that need to be produced. For example, combining the use of the spherical camera with a slice track with 7 slice plane positions in the final test of Table 1 above resulted in the production of a total of 7 x 1584 = 11,088 images with a corresponding increase in the run time of the simulation by a factor of 3.29 relative to the baseline case with no calls to the in-situ package. Clearly, a successful application of Cinema to the problem of scientific discovery with numerical simulation will require a thoughtful choice of track parameters that balances overhead cost against physics understanding.

VIII. Acknowledgements

My sincere thanks to John Patchett of the LANL Research Visualization team for his help in building and maintaining the Cinema-capable ParaView/Catalyst 4.3.1 software installation on Moonlight and Lightshow. Thanks also to David DeMarle of Kitware, Inc. for his help in resolving several problems with specA of the Cinema software.

IX. References

1. R. J. Kares, "Experiments at Scale with In-Situ Visualization Using Paraview/Catalyst in RAGE", Los Alamos National Laboratory report LA-UR-14-28528, (2014). http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-14-28528

2. A. C. Bauer, B. Geveci, and W. Schroeder, "The ParaView Catalyst User's Guide v1.0", Kitware, Inc. (2013). http://www.paraview.org/Wiki/images/4/48/CatalystUsersGuide.pdf

3. M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, D. Ranta, and R. Stafan, "The RAGE radiation-hydrodynamic code", *Comput. Sci. Disc.* **1**, 015005 (2008).

4. J. Patchett, J. Ahrens, B. Nouanesengsy, P. Fasel, P. O'Leary, C. Sewell, J. Woodring, C. Mitchell, L. Lo, K. Myers, J. Wendelberger, C. Canada, M. Daniels, H. Abhold and G. Rockefeller, "Case Study of In-Situ Data Analysis in ASC Integrated Codes", Los Alamos National Laboratory report LA-UR-13-26599 (2013). http://datascience.lanl.gov/data/papers/2013-2.pdf

5. J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. Rogers and M. Peterson, "An Image-based Approach to Extreme Scale In Situ Visualization and Analysis", paper submitted to Super Computing 2014 (2014). http://datascience.lanl.gov/data/papers/SC14.pdf

Appendix A. Problems Encountered with specA of the Cinema Software

A few problems were encountered with the specA delivery of the Cinema software during the course of the investigation that are worth documenting here. They are listed below:

1. In order to avoid a problem of having all the processors try to write Cinema output , the definition of class ImageExplorer in the file /usr/projects/paraview/2015-06-16/ml/lib/site-packages/paraview/cinemaIO/pv_explorers.py had to be corrected. The original version of class ImageExplorer:

```
class ImageExplorer(explorers.Explorer):
   """
   An explorer that connects a paraview script's views to a store
   and makes it save new images into the store.
   """
   def __init__(self,
                cinema_store, parameters, tracks,
                view=None):
       super(ImageExplorer, self).__init__(cinema_store, parameters, tracks)
       self.view = view

   def insert(self, document):
       # FIXME: for now we'll write a temporary image and read that in.
       # we need to provide nicer API for this.
       extension = self.cinema_store.get_image_type()
       simple.WriteImage("temporary"+extension, view=self.view)
       with open("temporary"+extension, "rb") as file:
           document.data = file.read()

       #alternatively if you are just writing out files and don't need them
       #in memory
       ##fn = self.cinema_store.get_filename(document)
       ##simple.WriteImage(fn)

       super(ImageExplorer, self).insert(document)
```

was replaced with the corrected version:

```
class ImageExplorer(explorers.Explorer):
    """
    An explorer that connects a paraview script's views to a store
    and makes it save new images into the store.
    """
    def __init__(self,
                 cinema_store, parameters, tracks,
                 view=None,
                 iSave=True):
        super(ImageExplorer, self).__init__(cinema_store, parameters, tracks)
        self.view = view
        self.iSave = iSave

    def insert(self, document):
        # FIXME: for now we'll write a temporary image and read that in.
        # we need to provide nicer API for this.
        extension = self.cinema_store.get_image_type()
        simple.WriteImage("temporary"+extension, view=self.view)
        with open("temporary"+extension, "rb") as file:
```

```
        document.data = file.read()

    #alternatively if you are just writing out files and don't need them
    #in memory
    ##fn = self.cinema_store.get_filename(document)
    ##simple.WriteImage(fn)
    if self.iSave:
       super(ImageExplorer, self).insert(document)
```

that introduces the iSave variable which controls whether or not a processor will save an image.

2. I also encountered a problem with Cinema trying to create time labels for the names of its time dump directories in fixed point format. This results in a problem for the 3D ICF simulation where the code unit of simulation time is seconds but all the times involved are of the order of 10^{-9} sec . To correct this problem a correction was added to the return value of function float_limiter in the file /usr/projects/paraview/2015-06-16/ml/lib/site packages/paraview/coprocessing.py to change the output time label returned from %6f to %6e format.

3. I noticed a minor problem with using the Qtviewer image browser to view a sequence of images with increasing values of the vorticity isosurface generated by the contour track of Section V. Using the contour slider to page through the image sequence, I discovered that when the slider reaches the last isosurface value, the corresponding image is not displayed. Instead the graphics window of the Qtviewer goes blank. For some reason this only happens with the contour slider. All other sliders behave as expected.

Appendix B. Producing Stereo Images with Catalyst

Here I document the modifications to the Catalyst pipeline script required to produce left and right stereo color image files from Catalyst using the EyeAngle stereo capability of ParaView. The first modification is to set the value of the EyeAngle parameter in the definition of the renderView that appears at the top of the exported Catalyst script:

```
# ------------------------ CoProcessor definition ------------------------

def CreateCoProcessor():
  def _CreatePipeline(coprocessor, datadescription):
    class Pipeline:
      # state file generated using paraview version 4.3.1-729-g70155c1

      # ----------------------------------------------------------------
      # setup views used in the visualization
      # ----------------------------------------------------------------

      #### disable automatic camera reset on 'Show'
      paraview.simple._DisableFirstRenderCameraReset()

      global renderView1
      # Create a new 'Render View'
      renderView1 = CreateView('RenderView')
      renderView1.ViewSize = [1920, 1080]
      renderView1.OrientationAxesVisibility = 0
      renderView1.CenterOfRotation = [0.0051310141570866099, 0.0052890516817569698,
1.00000001335143e-10]
      renderView1.StereoType = 0
      renderView1.EyeAngle = 1.0
      renderView1.CameraPosition = [0.015510679163475642, 0.02582817641414506, -
0.016881874614716776]
      renderView1.CameraFocalPoint = [0.0040111196479768237, 0.0047644642890671066, -
0.0015618841184303258]
      renderView1.CameraViewUp = [0.91455513671569855, -0.31283540234699175,
0.256364804420041]
      renderView1.CameraParallelScale = 0.0073689465985661397
      renderView1.Background = [0.0, 0.0, 0.0]

      # register the view with coprocessor
      # and provide it with information such as the filename to use,
      # how frequently to write the images, etc.
      coprocessor.RegisterView(renderView1,
          filename='image_%t_l.png', freq=1, fittoscreen=0, magnification=1,
width=1920, height=1080, cinema={})
```

All of the modifications to the standard exported Catalyst script are indicated in blue. Note the appearance of the line:

renderView1.EyeAngle = 1.0

in the above definition of the renderView used to create views of the 3D ICF problem with Catalyst. You will have to set an appropriate value of the EyeAngle parameter by trial-and-error. The value chosen here produces a stereo separation for the left-right pairs that is comfortable for the current problem. Larger values of EyeAngle increase the stereo separation. Note also that the

renderView variable has been promoted to a global variable so that it can be manipulated in another later routine. And finally, the file name has been modified to append an "_l" to the name to indicate it is the left image of a left-right stereo pair.

The second modification required to the pipeline script appears in the definition of the DoCoProcessing routine at the end of the script:

```
# ------------------------ Processing method ------------------------

def DoCoProcessing(datadescription):
    "Callback to do co-processing for current timestep"
    global coprocessor

    # Update the coprocessor by providing it the newly generated simulation data.
    # If the pipeline hasn't been setup yet, this will setup the pipeline.
    coprocessor.UpdateProducers(datadescription)


    # Write output data, if appropriate.
    # coprocessor.WriteData(datadescription);

    # Render the left eye view
    renderView1.StereoRender = 1
    renderView1.StereoType = 'Left'
    Render(renderView1)

    # Write image capture (Last arg: rescale lookup table), if appropriate.
    coprocessor.WriteImages(datadescription, rescale_lookuptable=False)

    # Render the right eye view
    renderView1.StereoRender = 1
    renderView1.StereoType = 'Right'
    fnamel = renderView1.cpFileName
    fnamer = fnamel.replace("_l","_r")
    renderView1.cpFileName = fnamer
    Render(renderView1)

    # Write image capture (Last arg: rescale lookup table), if appropriate.
    coprocessor.WriteImages(datadescription, rescale_lookuptable=False)

    # Restore the original file name
    renderView1.cpFileName = fnamel

    # Live Visualization, if enabled.
    # coprocessor.DoLiveVisualization(datadescription, "localhost", 22222)
```

Here again the modifications appear in blue. I have added code to the DoCoProcessing routine that first renders the left eye view and saves the image file. Then it switches the output image file name to append an "_r" to the file name, renders the view and saves the right eye image out to a file. Finally, it switches the name of the image file back to the left-hand name in preparation for the next output time step. The resulting numbered sequence of left-right image file pairs can then be opened and viewed with the CEI movie player EnVideo or some comparable tool. Fig. 6 shows a left-right stereo pair produced in this way from the 3D ICF simulation using Catalyst.

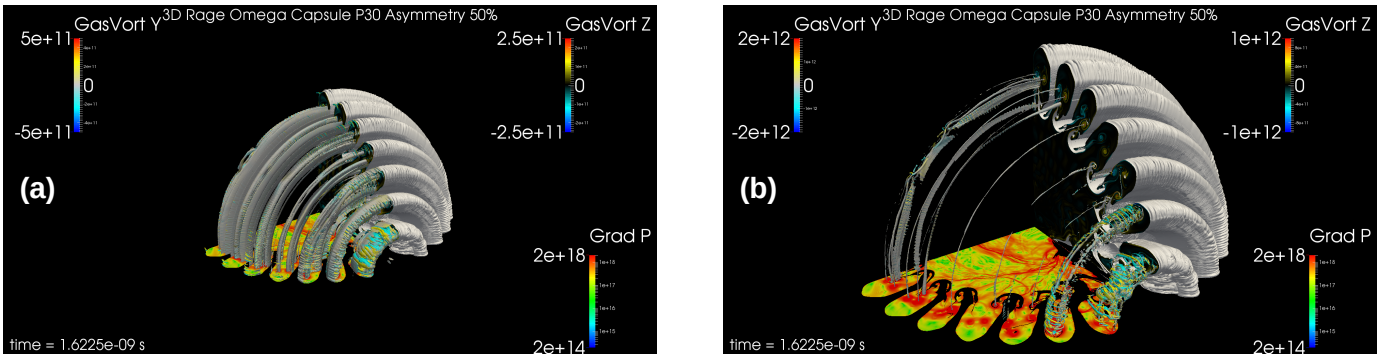

Fig. 1. Two snapshots at the same simulation time *t* = 1.6225 ns taken from the 0.05 μm 3D RAGE simulation of the P30 OMEGA capsule generated using the in-situ visualization capabilities of RAGE. The gray vortex tubes with yellow and blue regions are isosurfaces of total vorticity at a constant value of $2.5\times10^{11}\,\mathrm{sec}^{-1}$ in (a) and $1\times10^{12}\,\mathrm{sec}^{-1}$ in (b).

```
from paraview.simple import *
from paraview import coprocessing


#--------------------------------------------------------------
# Code generated from cpstate.py to create the CoProcessor.
# ParaView 4.3.1-729-g70155c1 64 bits


# ----------------------- CoProcessor definition -----------------------

def CreateCoProcessor():
  def _CreatePipeline(coprocessor, datadescription):
    class Pipeline:
      # state file generated using paraview version 4.3.1-729-g70155c1

      # ----------------------------------------------------------------
      # setup views used in the visualization
      # ----------------------------------------------------------------

      #### disable automatic camera reset on 'Show'
      paraview.simple._DisableFirstRenderCameraReset()

      # Create a new 'Render View'
      renderView1 = CreateView('RenderView')
      renderView1.ViewSize = [1920, 1080]
      renderView1.OrientationAxesVisibility = 0
      renderView1.CenterOfRotation = [0.0051310141570866099, 0.0052890516817569698,
1.00000001335143e-10]
      renderView1.StereoType = 0
      renderView1.CameraPosition = [0.015510679163475642, 0.02582817641414506, -
0.016881874614716776]
      renderView1.CameraFocalPoint = [0.0040111196479768237, 0.0047644642890671066, -
0.0015618841184303258]
      renderView1.CameraViewUp = [0.91455513671569855, -0.31283540234699175, 0.256364804420041]
      renderView1.CameraParallelScale = 0.0073689465985661397
      renderView1.Background = [0.0, 0.0, 0.0]

      # register the view with coprocessor
      # and provide it with information such as the filename to use,
      # how frequently to write the images, etc.
      coprocessor.RegisterView(renderView1,
          filename='image_%t.png', freq=1, fittoscreen=0, magnification=1, width=1920,
height=1080, cinema={"camera":"Spherical", "phi":[-180,-150,-120,-90,-60,-
30,0,30,60,90,120,150], "theta":[-180,-150,-120,-90,-60,-30,0,30,60,90,120,150] })

      # ----------------------------------------------------------------
      # setup the data processing pipelines
      # ----------------------------------------------------------------

      # create a new 'XML MultiBlock Data Reader'
      # create a producer from a simulation input
      grid_0vtm = coprocessor.CreateProducer(datadescription, 'input')
```

Fig. 2(a). Listing of the Python pipeline file "CinemaVortTubes.py" used with Catalyst in RAGE to generate the time snapshots from the spherical camera shown in Fig. 3.

```
# create a new 'Cell Data to Point Data'
cellDatatoPointData1 = CellDatatoPointData(Input=grid_0vtm)

# create a new 'Compute Derivatives'
computeDerivatives2 = ComputeDerivatives(Input=cellDatatoPointData1)
computeDerivatives2.Scalars = ['POINTS', 'prs']
computeDerivatives2.Vectors = [None, '']
computeDerivatives2.OutputTensorType = 'Nothing'

# create a new 'Slice'
slice2 = Slice(Input=computeDerivatives2)
slice2.SliceType = 'Plane'
slice2.SliceOffsetValues = [0.0]

# init the 'Plane' selected for 'SliceType'
slice2.SliceType.Origin = [1e-10, 0.050000000745058101, 0.050000000745058101]

# create a new 'Iso Volume'
isoVolume2 = IsoVolume(Input=slice2)
isoVolume2.InputScalars = ['POINTS', 'v02']
isoVolume2.ThresholdRange = [0.5, 1.01]

# create a new 'Reflect'
reflect1 = Reflect(Input=isoVolume2)
reflect1.Plane = 'Z Min'
reflect1.CopyInput = 0

# create a new 'Contour'
contour1 = Contour(Input=cellDatatoPointData1)
contour1.ContourBy = ['POINTS', 'v02']
contour1.Isosurfaces = [0.5]
contour1.PointMergeMethod = 'Uniform Binning'

# create a new 'Calculator'
calculator1 = Calculator(Input=cellDatatoPointData1)
calculator1.ResultArrayName = 'cell_velocity_pn'
calculator1.Function = 'iHat*xdt+jHat*ydt+kHat*zdt'

# create a new 'Compute Derivatives'
computeDerivatives1 = ComputeDerivatives(Input=calculator1)
computeDerivatives1.Scalars = ['POINTS', 'prs']
computeDerivatives1.Vectors = ['POINTS', 'cell_velocity_pn']
computeDerivatives1.OutputVectorType = 'Vorticity'
computeDerivatives1.OutputTensorType = 'Nothing'

# create a new 'Cell Data to Point Data'
cellDatatoPointData2 = CellDatatoPointData(Input=computeDerivatives1)

# create a new 'Calculator'
calculator2 = Calculator(Input=cellDatatoPointData2)
calculator2.ResultArrayName = 'GasVort'
calculator2.Function = 'Vorticity'

# create a new 'Calculator'
calculator3 = Calculator(Input=calculator2)
calculator3.ResultArrayName = 'MagVort'
calculator3.Function = 'mag(GasVort)'
```

Fig. 2(b). Listing of the Python pipeline file “CinemaVortTubes.py” used with Catalyst in RAGE to generate the time snapshots from the spherical camera shown in Fig. 3.

```
      # create a new 'Contour'
      contour2 = Contour(Input=calculator3)
      contour2.ContourBy = ['POINTS', 'MagVort']
      contour2.Isosurfaces = [50000000000.0]
      contour2.PointMergeMethod = 'Uniform Binning'

      # create a new 'Reflect'
      reflect2 = Reflect(Input=contour2)
      reflect2.Plane = 'Z Min'

      # create a new 'Slice'
      slice1 = Slice(Input=computeDerivatives1)
      slice1.SliceType = 'Plane'
      slice1.SliceOffsetValues = [0.0]

      # init the 'Plane' selected for 'SliceType'
      slice1.SliceType.Origin = [0.050000000745058101, 0.050000000745058101, 1e-10]
      slice1.SliceType.Normal = [0.0, 0.0, 1.0]

      # create a new 'Iso Volume'
      isoVolume1 = IsoVolume(Input=slice1)
      isoVolume1.InputScalars = ['POINTS', 'v02']
      isoVolume1.ThresholdRange = [0.5, 1.01]

      # ----------------------------------------------------------------
      # setup color maps and opacity mapes used in the visualization
      # note: the Get..() functions create a new object, if needed
      # ----------------------------------------------------------------

      # get color transfer function/color map for 'GasVort'
      gasVortLUT = GetColorTransferFunction('GasVort')
      gasVortLUT.RGBPoints = [-100000000000.0, 0.0, 0.0, 1.0, -50000000000.0, 0.0, 1.0, 1.0,
0.0, 0.80000000000000004, 0.80000000000000004, 0.80000000000000004, 50000000000.0, 1.0, 1.0,
0.0, 100000000000.0, 1.0, 0.0, 0.0]
      gasVortLUT.LockScalarRange = 1
      gasVortLUT.ColorSpace = 'RGB'
      gasVortLUT.NanColor = [0.49803900000000001, 0.0, 0.0]
      gasVortLUT.ScalarRangeInitialized = 1.0
      gasVortLUT.VectorComponent = 1
      gasVortLUT.VectorMode = 'Component'

      # get opacity transfer function/opacity map for 'GasVort'
      gasVortPWF = GetOpacityTransferFunction('GasVort')
      gasVortPWF.Points = [-100000000000.0, 0.0, 0.5, 0.0, 100000000000.0, 1.0, 0.5, 0.0]
      gasVortPWF.ScalarRangeInitialized = 1
      # get color transfer function/color map for 'Vorticity'
      vorticityLUT = GetColorTransferFunction('Vorticity')
      vorticityLUT.RGBPoints = [-100000000000.0, 0.0, 0.0, 1.0, -50000000000.0, 0.0, 1.0, 1.0,
0.0, 0.0, 0.0, 0.0, 50000000000.0, 1.0, 1.0, 0.0, 100000000000.0, 1.0, 0.0, 0.0]
      vorticityLUT.LockScalarRange = 1
      vorticityLUT.ColorSpace = 'RGB'
      vorticityLUT.NanColor = [0.49803900000000001, 0.0, 0.0]
      vorticityLUT.ScalarRangeInitialized = 1.0
      vorticityLUT.VectorComponent = 2
      vorticityLUT.VectorMode = 'Component'
```

Fig. 2(c). Listing of the Python pipeline file "CinemaVortTubes.py" used with Catalyst in RAGE to generate the time snapshots from the spherical camera shown in Fig. 3.

```
      # get opacity transfer function/opacity map for 'Vorticity'
      vorticityPWF = GetOpacityTransferFunction('Vorticity')
      vorticityPWF.Points = [-100000000000.0, 0.0, 0.5, 0.0, 100000000000.0, 1.0, 0.5, 0.0]
      vorticityPWF.ScalarRangeInitialized = 1

      # get color transfer function/color map for 'ScalarGradient'
      scalarGradientLUT = GetColorTransferFunction('ScalarGradient')
      scalarGradientLUT.RGBPoints = [200000000000000.0, 0.0, 0.0, 1.0, 2000000000000000.0, 0.0,
1.0, 1.0, 20000000000000000.0, 0.0, 1.0, 0.0, 2e+17, 1.0, 1.0, 0.0, 2e+18, 1.0, 0.0, 0.0]
      scalarGradientLUT.UseLogScale = 1
      scalarGradientLUT.LockScalarRange = 1
      scalarGradientLUT.ColorSpace = 'RGB'
      scalarGradientLUT.NanColor = [0.49803900000000001, 0.0, 0.0]
      scalarGradientLUT.ScalarRangeInitialized = 1.0

      # get opacity transfer function/opacity map for 'ScalarGradient'
      scalarGradientPWF = GetOpacityTransferFunction('ScalarGradient')
      scalarGradientPWF.Points = [200000000000000.0, 0.0, 0.5, 0.0, 2e+18, 1.0, 0.5, 0.0]
      scalarGradientPWF.ScalarRangeInitialized = 1

      # ----------------------------------------------------------------
      # setup the visualization in view 'renderView1'
      # ----------------------------------------------------------------

      # show data from isoVolume1
      isoVolume1Display = Show(isoVolume1, renderView1)
      # trace defaults for the display properties.
      isoVolume1Display.ColorArrayName = ['CELLS', 'Vorticity']
      isoVolume1Display.LookupTable = vorticityLUT
      isoVolume1Display.ScalarOpacityUnitDistance = 0.00030382981714722801

      # show color legend
      isoVolume1Display.SetScalarBarVisibility(renderView1, True)

      # show data from contour1
      contour1Display = Show(contour1, renderView1)
      # trace defaults for the display properties.
      contour1Display.ColorArrayName = [None, '']

      # show data from reflect1
      reflect1Display = Show(reflect1, renderView1)
      # trace defaults for the display properties.
      reflect1Display.ColorArrayName = ['CELLS', 'ScalarGradient']
      reflect1Display.LookupTable = scalarGradientLUT
      reflect1Display.ScalarOpacityUnitDistance = 0.00030251818748908398

      # show color legend
      reflect1Display.SetScalarBarVisibility(renderView1, True)

      # show data from reflect2
      reflect2Display = Show(reflect2, renderView1)
      # trace defaults for the display properties.
      reflect2Display.ColorArrayName = ['POINTS', 'GasVort']
      reflect2Display.LookupTable = gasVortLUT
      reflect2Display.ScalarOpacityUnitDistance = 0.00042791784907172302
```

Fig. 2(d). Listing of the Python pipeline file "CinemaVortTubes.py" used with Catalyst in RAGE to generate the time snapshots from the spherical camera shown in Fig. 3.

```
    # show color legend
    reflect2Display.SetScalarBarVisibility(renderView1, True)

    # setup the color legend parameters for each legend in this view

    # get color legend/bar for vorticityLUT in view renderView1
    vorticityLUTColorBar = GetScalarBar(vorticityLUT, renderView1)
    vorticityLUTColorBar.Position = [0.89135338345864701, 0.55617283950617302]
    vorticityLUTColorBar.Title = 'GasVort'
    vorticityLUTColorBar.ComponentTitle = 'Z'

    # get color legend/bar for scalarGradientLUT in view renderView1
    scalarGradientLUTColorBar = GetScalarBar(scalarGradientLUT, renderView1)
    scalarGradientLUTColorBar.Position = [0.89511278195488697, 0.036049382716049502]
    scalarGradientLUTColorBar.Title = 'Grad P'
    scalarGradientLUTColorBar.ComponentTitle = 'Magnitude'

    # get color legend/bar for gasVortLUT in view renderView1
    gasVortLUTColorBar = GetScalarBar(gasVortLUT, renderView1)
    gasVortLUTColorBar.Position = [0.072109022556390998, 0.55456790123456801]
    gasVortLUTColorBar.Title = 'GasVort'
    gasVortLUTColorBar.ComponentTitle = 'Y'

    annotationColor = [1.0, 1.0, 1.0]
    VersionText = Text(Text='3D Rage Omega Capsule P30 Asymmetry 50%')
    VersionRep = Show()
    VersionRep.WindowLocation = 'UpperCenter'
    VersionRep.Color = annotationColor
    VersionRep.FontSize = 10
    VersionRep.Orientation = 0
    VersionRep.TextScaleMode = 'Viewport'

    AnnotateTimeFilter1 = AnnotateTimeFilter()
    TimeRep = Show()
    AnnotateTimeFilter1.Format = 'time = %6.4e s'
    TimeRep.WindowLocation = 'LowerLeftCorner'
    TimeRep.Color = annotationColor
    TimeRep.FontSize = 10
    TimeRep.TextScaleMode = 'Viewport'

  return Pipeline()

class CoProcessor(coprocessing.CoProcessor):
  def CreatePipeline(self, datadescription):
    self.Pipeline = _CreatePipeline(self, datadescription)

coprocessor = CoProcessor()
# these are the frequencies at which the coprocessor updates.
freqs = {'input': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
coprocessor.SetUpdateFrequencies(freqs)
return coprocessor
```

Fig. 2(e). Listing of the Python pipeline file "CinemaVortTubes.py" used with Catalyst in RAGE to generate the time snapshots from the spherical camera shown in Fig. 3.

```
#--------------------------------------------------------------
# Global variables that will hold the pipeline for each timestep
# Creating the CoProcessor object, doesn't actually create the ParaView pipeline.
# It will be automatically setup when coprocessor.UpdateProducers() is called the
# first time.
coprocessor = CreateCoProcessor()

#--------------------------------------------------------------
# Enable Live-Visualizaton with ParaView
coprocessor.EnableLiveVisualization(False, 1)


# ---------------------- Data Selection method ----------------------

def RequestDataDescription(datadescription):
    "Callback to populate the request for current timestep"
    global coprocessor
    if datadescription.GetForceOutput() == True:
        # We are just going to request all fields and meshes from the simulation
        # code/adaptor.
        for i in range(datadescription.GetNumberOfInputDescriptions()):
            datadescription.GetInputDescription(i).AllFieldsOn()
            datadescription.GetInputDescription(i).GenerateMeshOn()
        return

    # setup requests for all inputs based on the requirements of the
    # pipeline.
    coprocessor.LoadRequestedData(datadescription)

# ------------------------ Processing method ------------------------

def DoCoProcessing(datadescription):
    "Callback to do co-processing for current timestep"
    global coprocessor

    # Update the coprocessor by providing it the newly generated simulation data.
    # If the pipeline hasn't been setup yet, this will setup the pipeline.
    coprocessor.UpdateProducers(datadescription)

    # Write output data, if appropriate.
    # coprocessor.WriteData(datadescription);

    # Write image capture (Last arg: rescale lookup table), if appropriate.
    coprocessor.WriteImages(datadescription, rescale_lookuptable=False)

    # Live Visualization, if enabled.
    # coprocessor.DoLiveVisualization(datadescription, "localhost", 22222)
```

Fig. 2(f). Listing of the Python pipeline file "CinemaVortTubes.py" used with Catalyst in RAGE to generate the time snapshots from the spherical camera shown in Fig. 3.

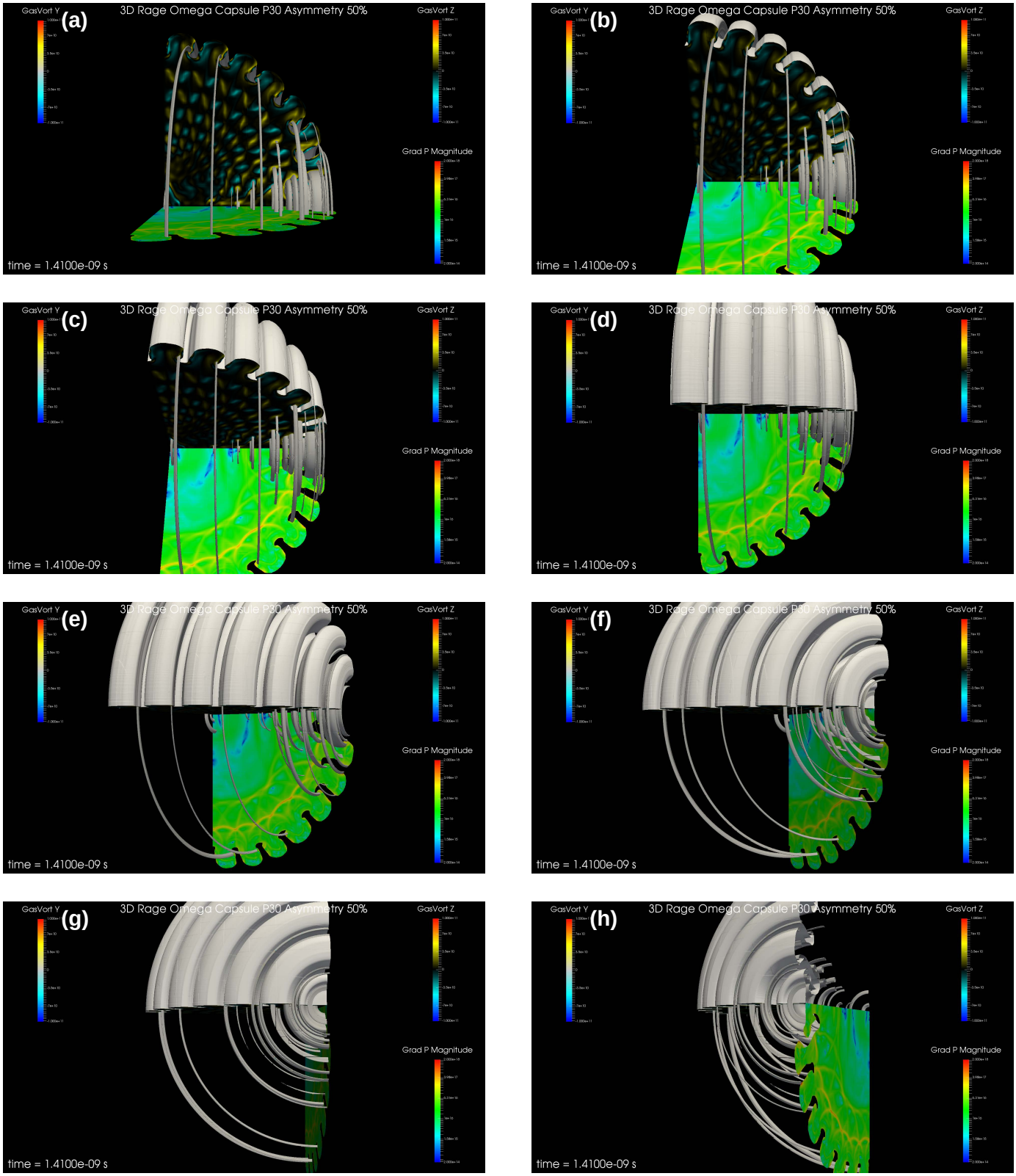

Fig. 3. Eight of the 144 separate views at time *t* = 1.41 ns of the 3D RAGE simulation of the P30 Omega capsule generated automatically using the Cinema spherical camera with the ParaView/Catalyst based in-situ visualization capabilities of RAGE.

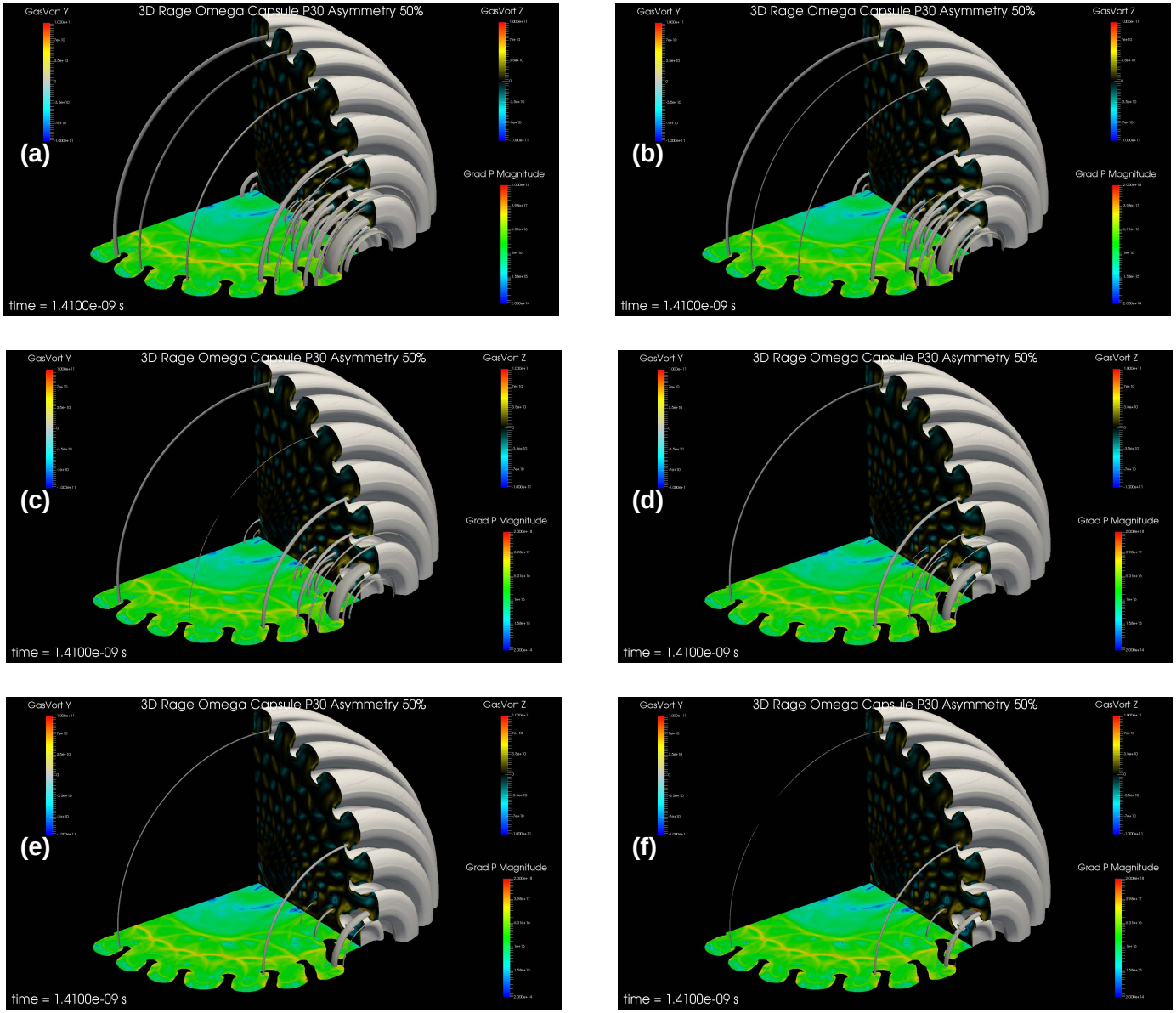

Fig. 4. Six snapshots at time *t* = 1.41 ns from the 3D RAGE simulation of the P30 OMEGA capsule generated using a Cinema contour track with the ParaView/Catalyst based in-situ visualization capabilities of RAGE. In snapshots (a) through (f) the gray vortex tubes are isosurfaces of total vorticity at constant values of $5\times10^{10}\ \mathrm{sec}^{-1}$ through $1\times10^{11}\ \mathrm{sec}^{-1}$ in increments of $1\times10^{10}\ \mathrm{sec}^{-1}$.

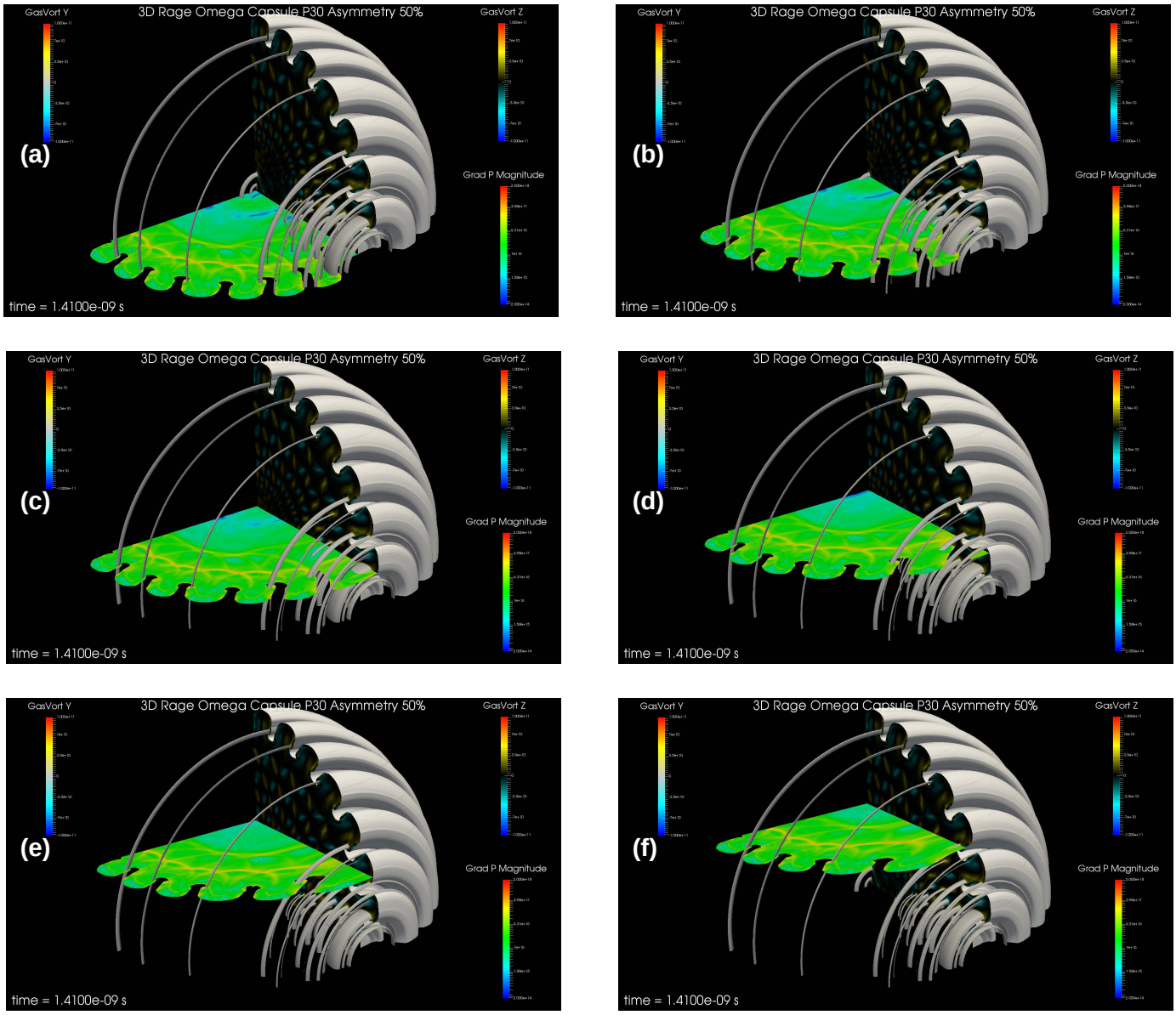

Fig. 5. Six snapshots at time *t* = 1.41 ns from the 3D RAGE simulation of the P30 OMEGA capsule generated using a Cinema slice track with the ParaView/Catalyst based in-situ visualization capabilities of RAGE. In snapshots (a) through (f) the slice plane position has been moved along the X direction from X = 0. to X = 0.005 in steps of 0.001.

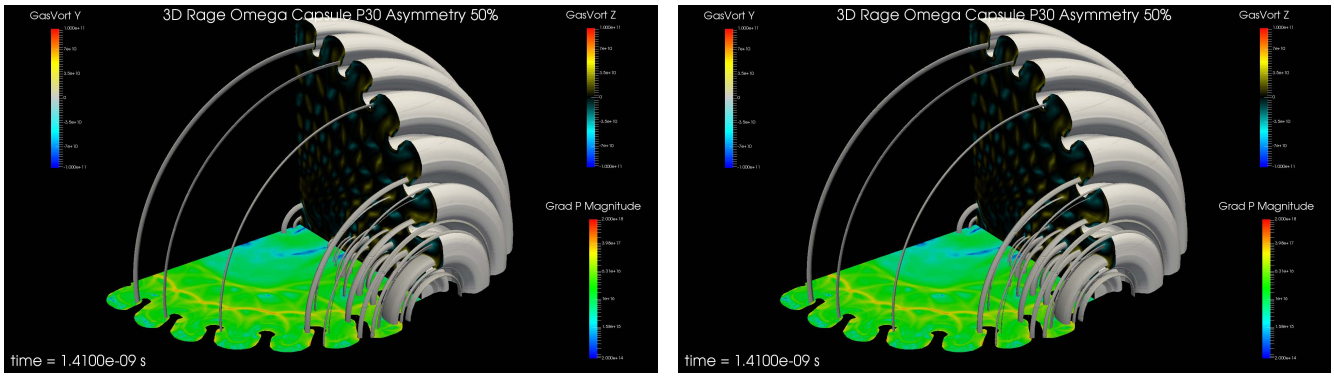

Fig. 6. Left-right stereo pair produced using the EyeAngle stereo capabilities of ParaView/Catalyst by the procedure described in Append B. The image pair is arranged on the page to be suitable for freeviewing.