CS 780/880 Semester Project Report

Anthony Westbrook

Introduction

The following paper provides a comprehensive overview and detailed description of my CS880 semester project. An end-user copy of the usage and installation instructions may also be found in the README file included in the submitted archive.

Primarily, this project focused on providing interoperability between the ParaView/VTK framework and the UNH Granite Scientific Database System (hereby simply referred to as *Granite*). During planning and development, emphasis was placed on reading and writing data from Granite into ParaView, thereby simultaneously taking advantage of the data retrieval efficiency of Granite and the manipulation/visualization benefits of the VTK framework.

As ParaView is designed to be modular in nature, this implementation took shape in the form of an encapsulated plugin, providing reader and writer style functionality to serve as the interface between Granite and ParaView. Additionally, since ParaView provides robust volume rendering capabilities in both a single resolution and via adaptive mesh refinement, focus was placed specifically on rendering single resolution uniform and non-uniform rectilinear datasets, as well as data provided by Granite's multi-resolution classes. Lastly, the plugin implements matching write capabilities for each style of reading, both to aid in development, but also for end-user file conversion use.

Background

Some visualization concepts and technologies were key in implementing the Granite plugin:

- ParaView / VTK Framework [1] - Developed by Kitware Inc, VTK (Visualization Toolkit) is an open source, widely used framework providing image processing, visualization, and data modeling capabilities. Built upon this Framework, ParaView, also developed by Kitware, provides a detailed user interface to easily make use of virtually all of VTK's functions.
- Adaptive Mesh Refinement (AMR) - A method of visualizing multiple resolutions of data, AMR overlays a collection of grids with differing resolution, depending on the accuracy required for the respective spatial area being visualized. In this way, certain areas of interest can be visualized via higher density grids, while areas of lower interest can use lower density grids, saving on storage and processing.

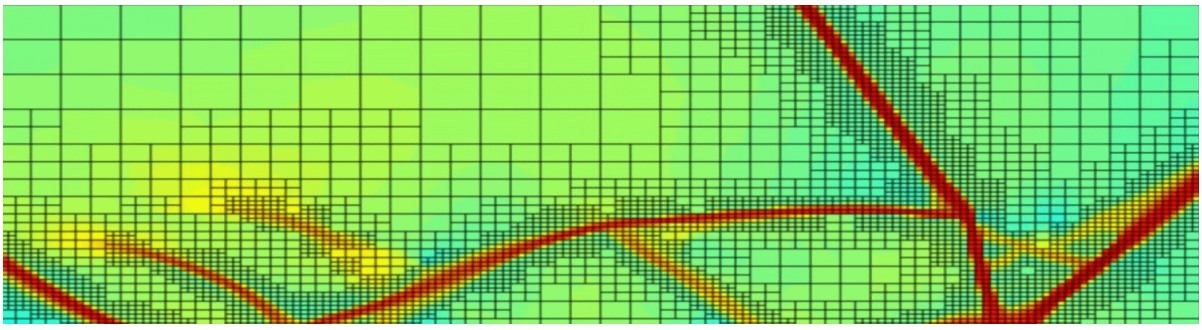

Figure 1: Adaptive Mesh Refinement (AMR) example [2]

Project Description

Functionality

1. Reader
    a. Opens standard Granite XFDL files to visualize uniform rectilinear data
    b. Opens ParaView created Granite XFDL files to visualize non-uniform rectilinear data
    c. Opens standard multi-resolution Granite XFDL files to visualize multi-resolution uniform rectilinear data in a streaming overlapping AMR / adaptive-resolution fashion
    d. For single resolution data, allows data extents to be user specified pre-read, to visualize a specific VOI (volume of interest)

2. Writer
    a. Writes uniform and non-uniform rectilinear datasets (VTK, DICOM, binary, etc) to Granite XFDL/BIN files
    b. Supports resampling output data so data extents match the spatial bounds
    c. Supports writing uniform rectilinear datasets to Granite multi-resolution XFDL/BIN files and directory structures at a specified number of resolution levels and steps per level
    d. Supports the following custom meta-data tags for increased functionality with ParaView:
        A. CustomParaViewOrigin - Spatial origin on axes (3 doubles)
        B. CustomParaViewSpacing - Spacing/magnification of data against spatial coordinates per axis (3 doubles)
        C. CustomParaViewGrid - Spatial coordinate arrays per axis representing the spacing of each point lattice in a non-uniform rectilinear grid (3 double arrays)
        D. CustomParaViewType - VTK data type to be used for this dataset. vtkImageData for uniform rectilinear (default if not specified), or vtkRectilinearGrid for non-uniform rectilinear data
        E. "Array.Component" formatting for attribute names. Since VTK supports the concept of multiple arrays of data, each having its own components, the plugin will emulate importing this information from Granite by parsing dots found in component names into "Array.Component" - e.g. "VectorVel.x", "VectorVel.y", "VectorVel.z", "temperature.amount" would create two arrays, one named "VectorVel" with 3 components "x", "y", and "z", and one array "temperature" with a single component "amount"

3. General
    a. Directly interfaces with Granite library via JNI, and allows standard command-line arguments to be specified within the GUI for the Java VM (memory allocation, debugging, garbage collection, etc)
    b. Allows the number of AMR subdivisions during the rendering process to be specified
    c. Adheres to (mostly) all VTK standards and implementation requirements for maximum compatibility with all filters, mappers, and other ParaView functionality
    d. Supports datasets as large as ParaView and physical memory permits
    e. Successfully tested on all major platforms (Windows, Linux, OSX)

Approach

A design overview of the Granite plugin can be separated into 3 distinct sections. See detailed specifications for further detail.

1. Granite Interface - As VTK is written in C++ and therefore executes under the native architecture of the machine, a method of interoperability with the Java VM must be implemented, as Granite is a Java library. To achieve this, the JNI (Java Native Interface) framework was employed in a wrapper design pattern, with each pertinent Granite method called from a JNI connected C++ method. Additionally, Granite oriented helper functions were also added to aid in ease-of-use and efficiency.

2. VTK Classes - For purposes of modularity and standardization, VTK is designed heavily with an OO design. When implementing a new plugin, newly created classes should inherit from existing standard VTK classes, and implement a set of required methods that will be called as part of the VTK pipeline. VTK classes related to the functionality of reading and writing of uniform, non-uniform, and AMR datasets (see detailed specifications) were used as the base classes when implementing the Granite plugin VTK classes.

3. Support Library - While the Granite Interface and the VTK classes serve as both ends of the plugin's interface, a large amount of functionality was common to all classes involved, including basic dataset oriented calculations, IO, parsing, etc. The support library contains methods heavily called amongst all classes in the implementation.

Detailed Specifications

Please refer to figures 2 and 3 as reference to relevant class relationships and data flow

As VTK operates in a pipeline fashion (as shown in figure 2) [3], plugins implement segments of this pipeline, taking in data from the previous segment, performing an action upon the data, and then providing an output to the next portion of the pipeline. Given this architecture, two families of VTK classes are key to implementation: those describing the structure of the segment, and those describing the data the segment processes.

Structure

All segments of the pipeline associated with sourcing, filtering, and sinking data in VTK ultimately inherit from the *vtkAlgorithm* class. This class handles two important functions. First, it associates with a matching *vtkExecutive*, the class responsible for message processing and execution of the VTK pipeline. Second, it contains base functionality common to dataset-oriented processing. VTK then provides a number of child classes with extended functionality matching the type of data being processed.

In the case of the single-resolution portion of the Granite plugin, the reader derives directly from the base *vtkAlgorithm* class, as opposed to a more specialized child class. The reason for this pertains to the plugin supporting both uniform and non-uniform rectilinear data. These are normally handled by two different child classes - however, it was a goal to dynamically handle either data type without extra overhead, and hence the plugin provides its own specialized functionality to handle both input types, dynamically reconfiguring the VTK pipeline in the process.

In the case of multi-resolution data, the VTK functionality involved is quite complex, and it was necessary to implement a separate class which inherited from the *vtkAMRBaseReader*, a "great grandchild" class of *vtkAlgorithm*. *vtkAMRBaseReader* provides the framework necessary to provide meta-information about AMR block structures and locations, as well as asynchronous streaming of data through the pipeline.

Both of these classes in the Granite plugin implement a *CanReadFile* method, indicating whether they support a particular XFDL file or not. Upon opening a new XFDL file, both *CanReadFile* methods determine from Granite if the dataset is single or multi-resolution, and will reply true/false respectively, allowing ParaView to choose the appropriate class for the data.

Additionally, these two classes also override all necessary methods (either directly or through child methods) for proper execution of the VTK pipeline. The most noteworthy are as follows:

- ProcessRequest - This receives incoming requests from connected segments of the VTK pipeline, and delivers them to subsystems within the plugin accordingly.
- ProcessInformation - This provides meta-information regarding the data set (data extents, spatial bounds, spacing, etc) to connected segments of the VTK pipeline.
- RequestData - This is the heart of any *vtkAlgorithm*, and provides the data itself to connected segments of the VTK pipeline
- FillInputPortInformation, FillOutputPortInformation - These control the data types this segment of the pipeline will accept, and will provide.

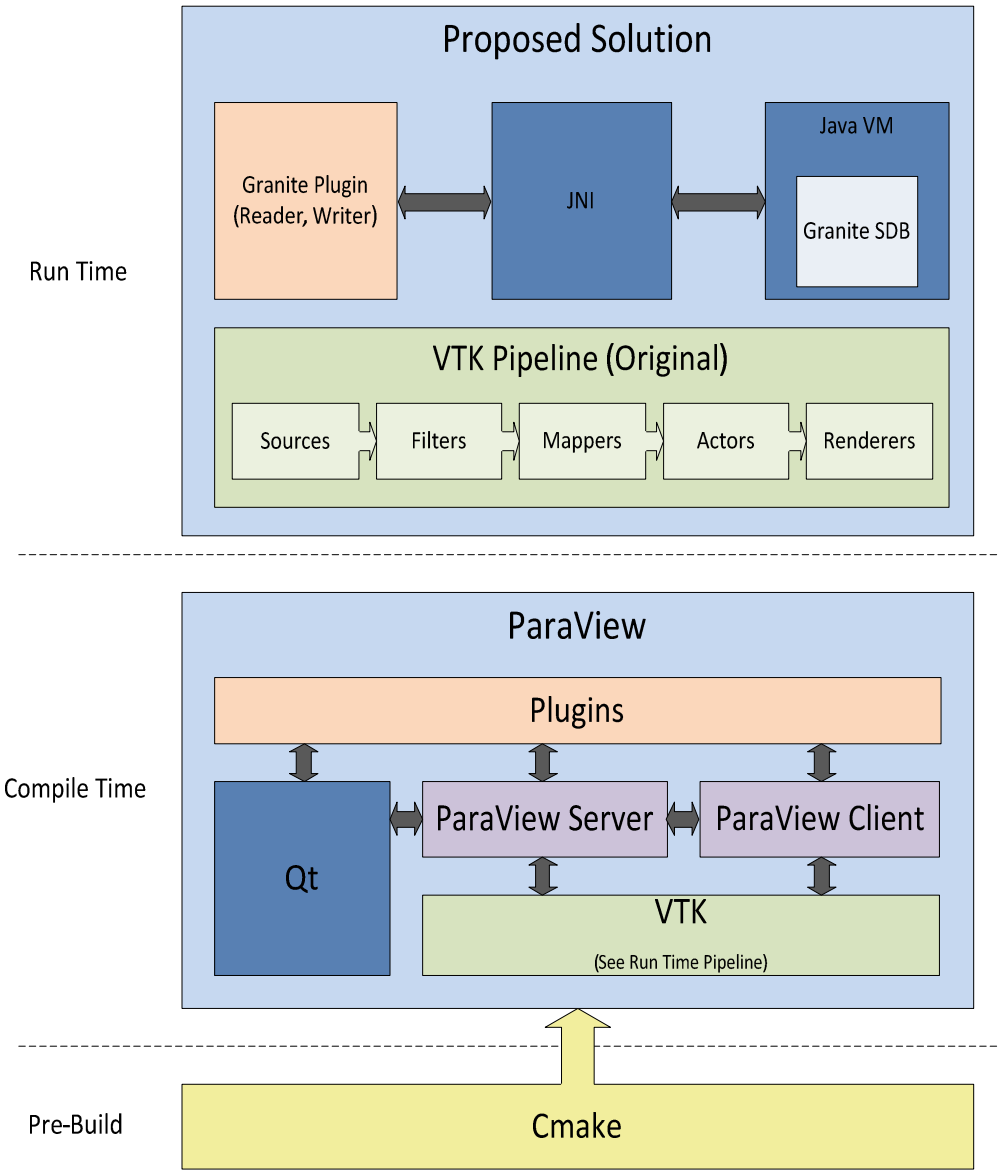

Figure 2: ParaView/VTK Structure and Data Flow

Data

The data types specified by the methods above ultimately inherit from the *vtkDataSet* class (as shown in figure 3). Since arrays of data are common to all datasets, regardless of their associated geometry, storage and functionality related to point and cell data is handled by the *vtkDataSet* class. Functionality related to spatial structure and topology is handled by each respective child class.

The Granite plugin makes use of 3 different VTK supplied data types:

- vtkImageData - designed to handle uniform rectilinear data, in which spacing along each major axis is regular and user-definable in magnitude

- vtkRectilinearGrid - designed to handle non-uniform rectilinear data, in which spacing must be defined between every intersection in the lattice grid
- vtkOverlappingAMR - designed to handle a composite set of overlapping grids of uniform rectilinear data. These blocks are assigned a spatial location in respect to the data origin, as well as an ordering level-of-resolution. During rendering, this positioning and resolution level are used to hide (in volumetric space) existing lower-resolution blocks.

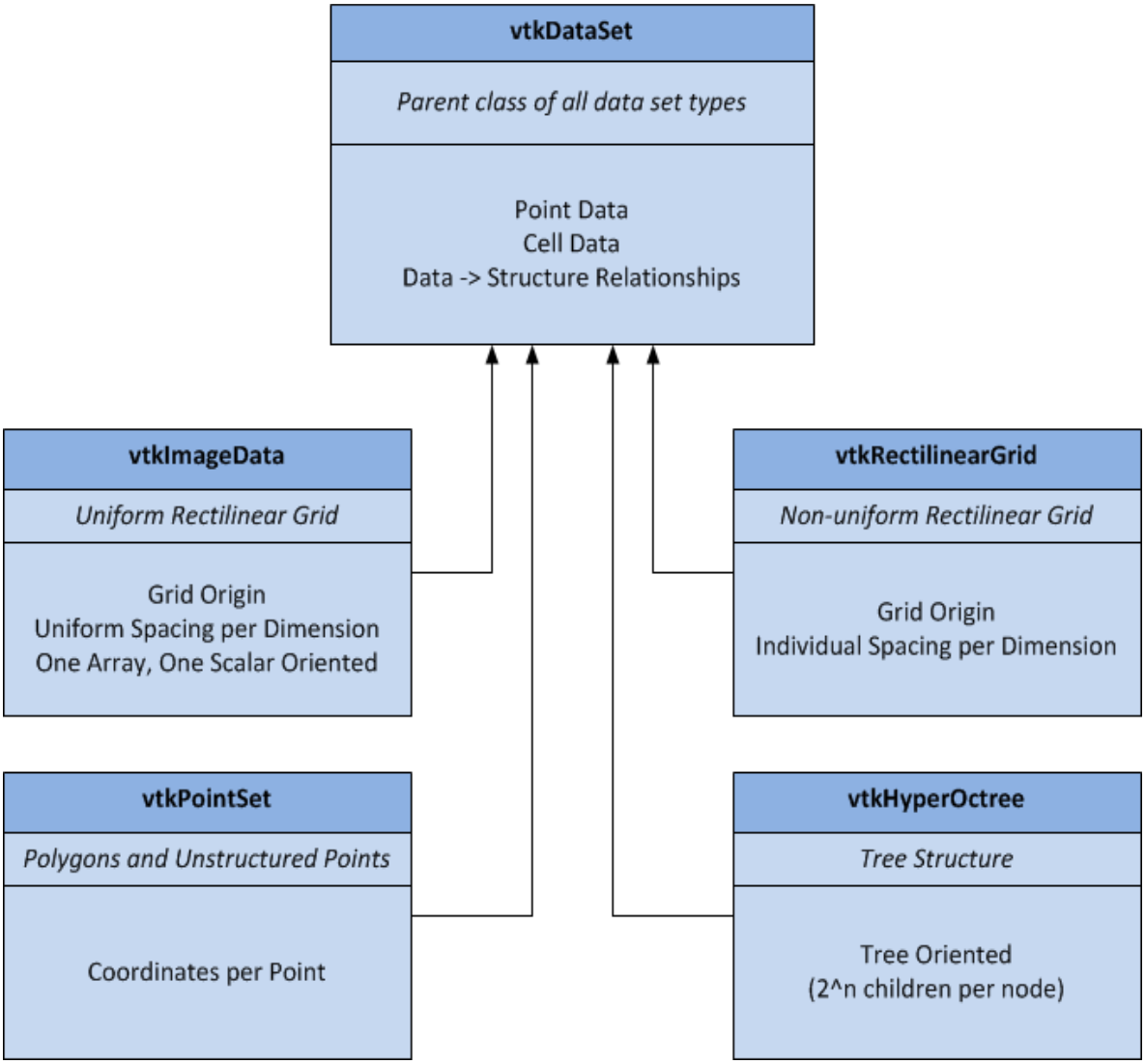

Figure 3: VTK Dataset Class Relationships

Results

Overall, the Granite plugin successfully achieved the stated goals of rendering Granite-originating single and multi-resolution data with no major functional issues. Reader performance, while reasonable, was inherently limited by the overhead incurred while interfacing between native and JNI/JVM execution (see conclusions

section for further consideration). Specific results and minor functional and visual correctness issues are as follows:

- Uniform Rectilinear Data, Read/Write, Single Resolution - This was the most successful and robust portion of the implementation due to availability of VTK documentation and lower complexity. As VTK supports point data for this type of dataset, visual correctness was as expected (see figures 4, 5 and 6). Known issues are:
    - Data obtained from Granite, and written from ParaView, is always treated as floating point numbers. While not a significant issue, it can potentially increase file sizes and processing time if the original data was stored as characters or short integers. Fixing this issue would be relatively simple, but due to time constraints, priority was placed on implementing other major features of the plugin.
    - Due to weaker documentation in the area of UI interaction in ParaView, the VOI fields will not automatically be set to the data extents of the dataset upon opening a new file - they will remain "0, 0, 0, 0, 0, 0". Because of this, the Granite plugin will initially ignore the VOI bounds, and will only make use of them if one of the values is updated by the user. While this does not impair functionality, it can be confusing

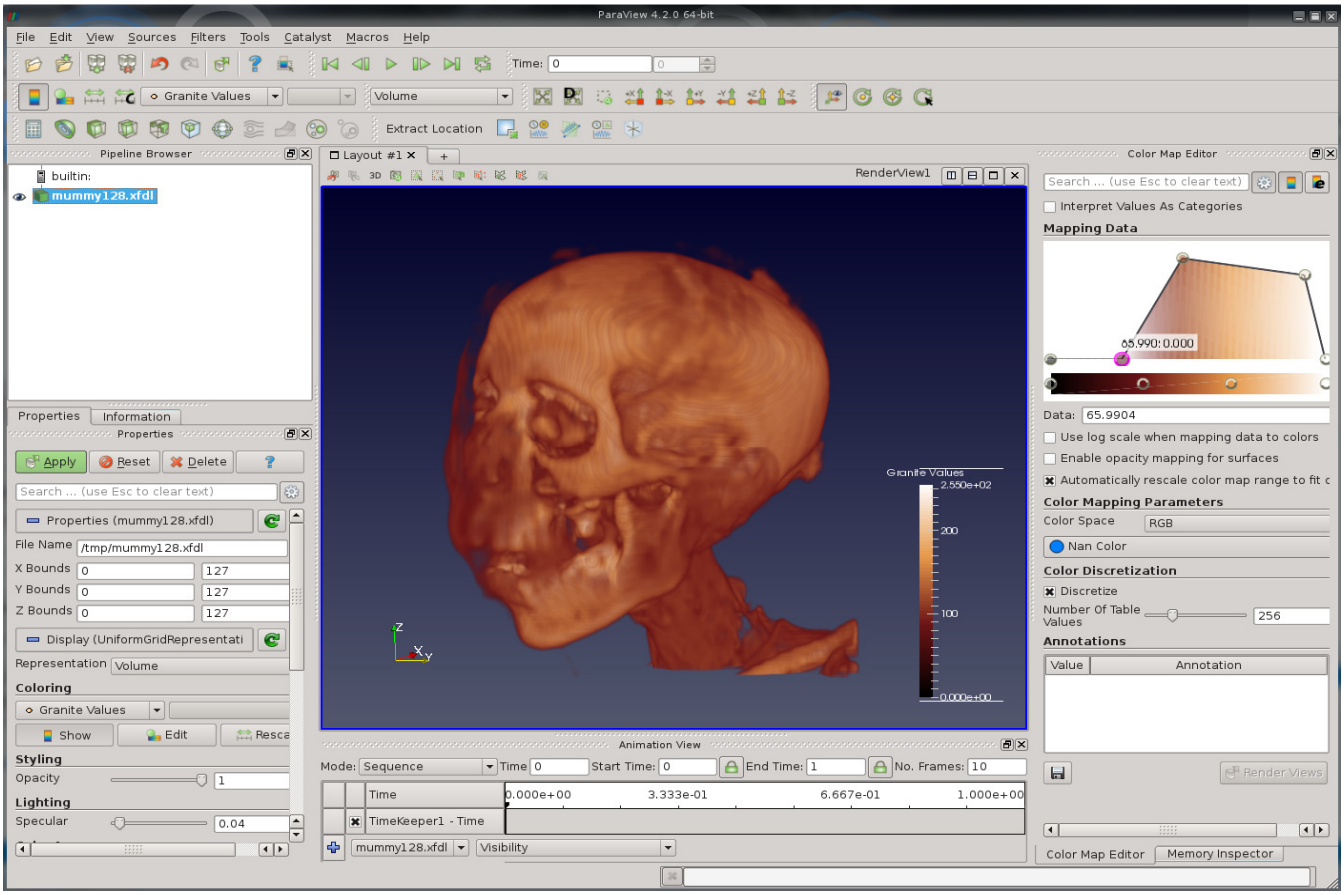

Figure 4: Written from VTK format, Read from XFDL format (ParaView interface shown as reference) [4]

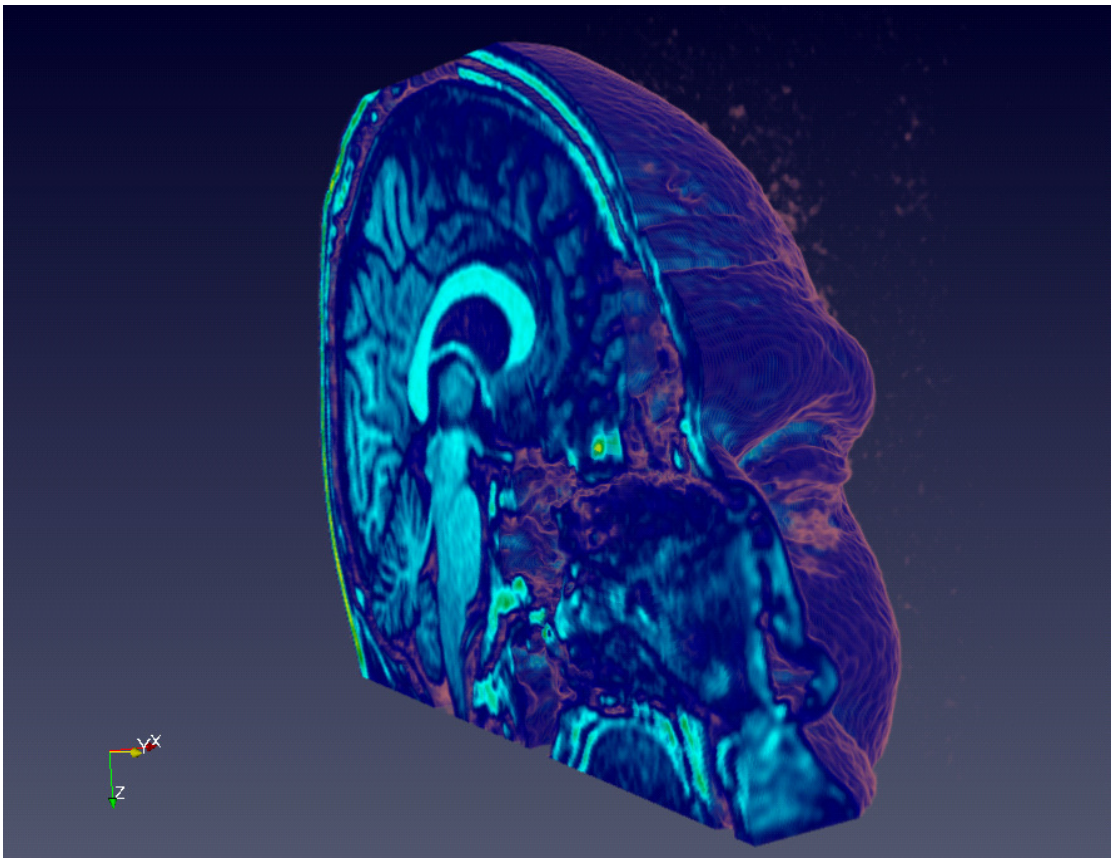

Figure 5: Written from DICOM format, Read from XFDL format, VOI set across sagittal plane [5]

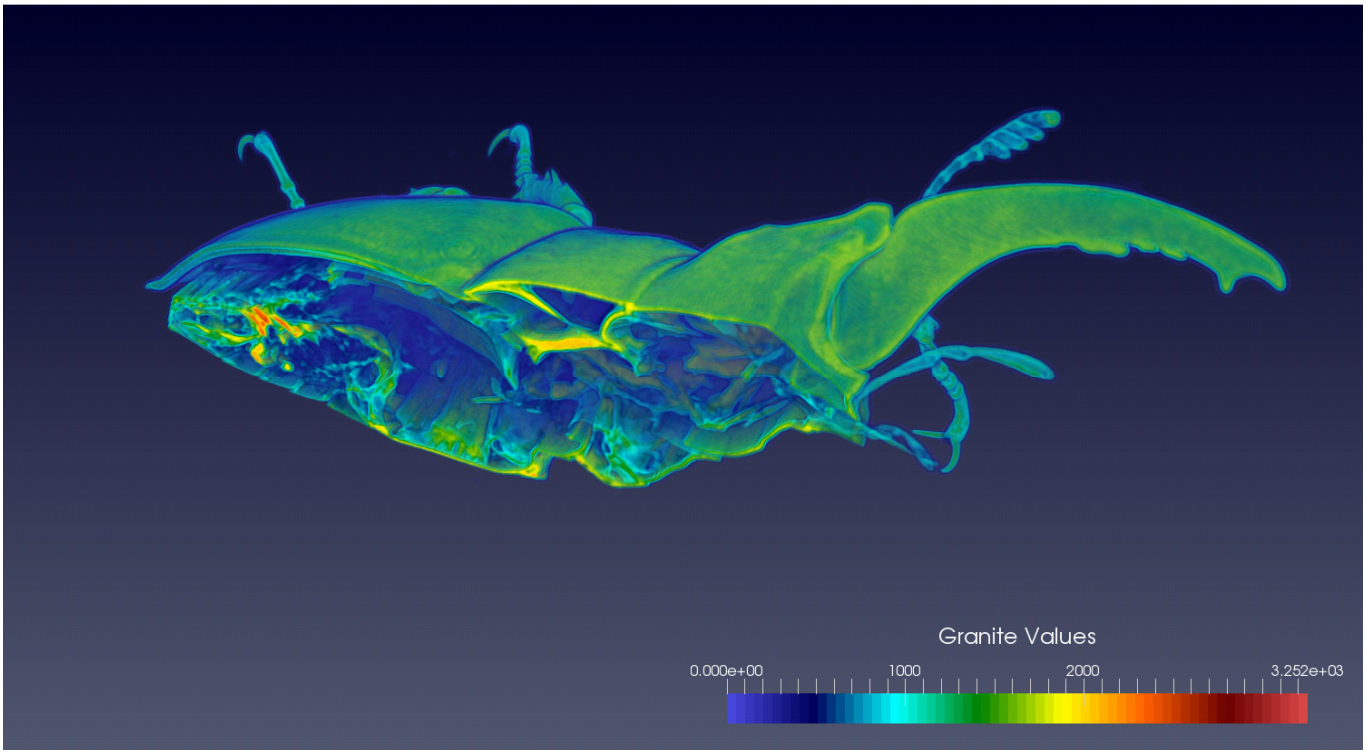

Figure 6: Read from XFDL format, larger dataset (1.3GB), VOI set across sagittal plane

- Non-Uniform Rectilinear Data, Read/Write, Single Resolution - Success and known issues for non-uniform data matches that of uniform data due to the shared implementation.

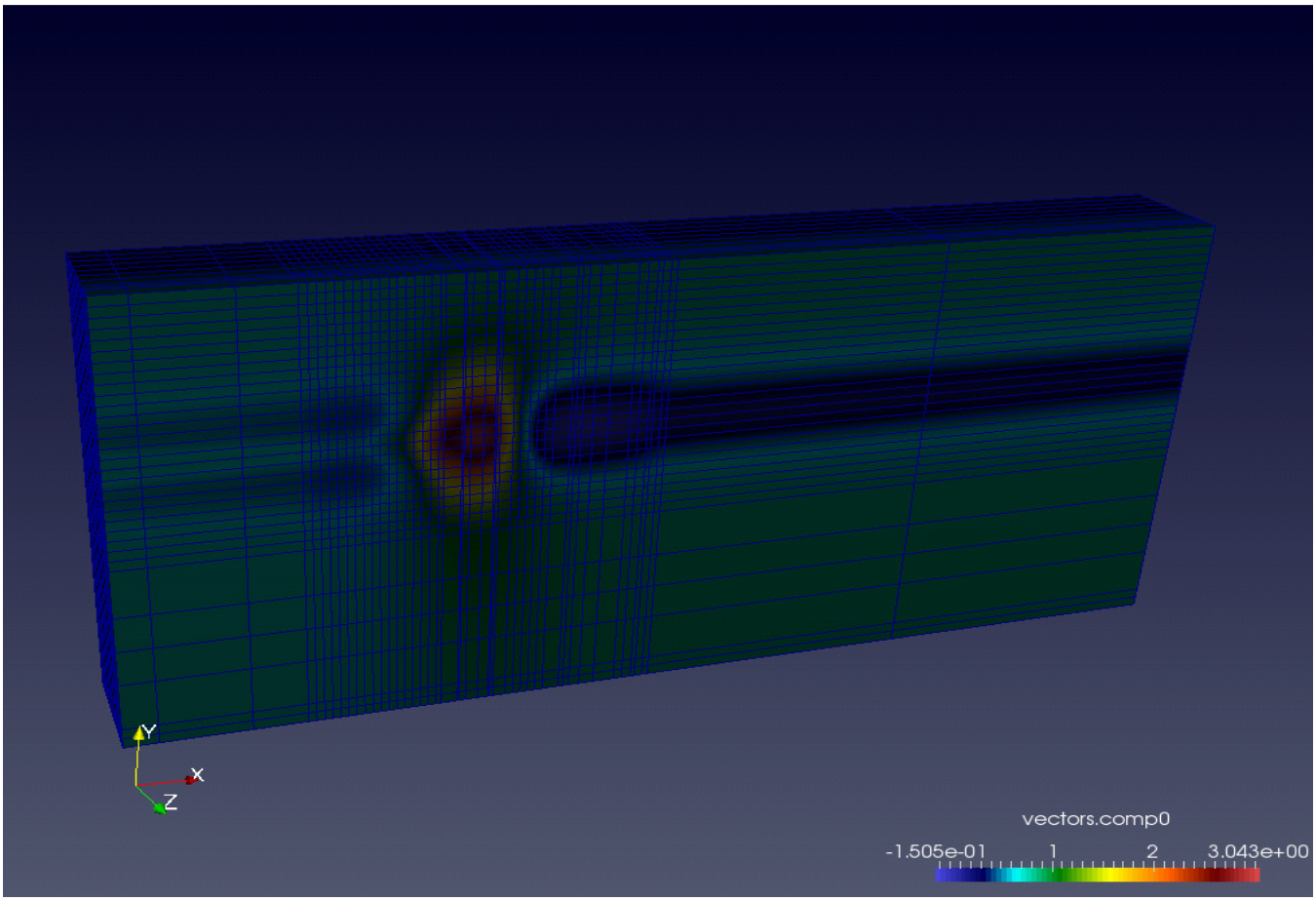

Figure 7: Written from VTK format, Read from XFDL format

- Uniform Rectilinear Data, Read/Write, Multi-resolution (AMR) - While this portion of implementation yielded the most interesting results, it also has a number of minor known issues. The overlapping AMR functionality within ParaView/VTK is a relatively newer addition to the package as of version 5.6, and documentation is almost exclusive to a single Kitware white-page document. Reversing the VTK code was necessary for the majority of implementation, presenting its own challenges due to the sheer size and complexity of the VTK codebase. That stated, implementation was still an overall success, with the following exceptions:
    - At this time, the AMR reader class within VTK does not support point data arrays (stub functions are in place, but are commented to be completed in the future), and only supports cell data arrays. The plugin handles this by adjusting the data extents, and loading the point data into the cell arrays. However, this results in a "blockier" volume render than the original, point based render. Partial compensation can be achieved by then applying a cell-to-point data filter on the plugin output - however, due to multiple interpolations, the overall fidelity is still reduced from the original. Once Kitware implements point array handling within the AMR code, adjusting the plugin to make use of it will be trivial.

- Due to an apparent OS/JVM dependent issue within the Granite library itself of processing relative paths in relation to the root multi-resolution XFDL file, in Linux (and assumed in OSX), a multi-resolution dataset must be opened from the current working directory, or Granite will fail. This problem does not occur in Windows. The problem was confirmed via direct Java tests (without JNI/ParaView).
- As non-uniform functionality relies on directly reading custom XML fields from the XFDL file, only uniform data is supported under multi-resolution (at this time). Future versions would require the plugin to iterate through each directory corresponding to the resolution level and read in these custom XML fields to support non-uniform data.
- Upon rendering the final block of the highest resolution level, ParaView currently displays a "Queue Empty" error. After examining the source code, it's not certain if this is actually an error, or a message indicating that rendering is fully complete.
- If too many levels of resolution or steps between levels are specified when writing multi-resolution datasets from ParaView, the resulting XFDL file/directory fails upon reading back into ParaView. The root cause is unclear at this time, but it's likely related to the heavy interpolation resulting in erroneous data extents at extremely low resolutions. Fortunately, these levels of resolutions are virtually unnecessary as they contain little to no remarkable visualization information.

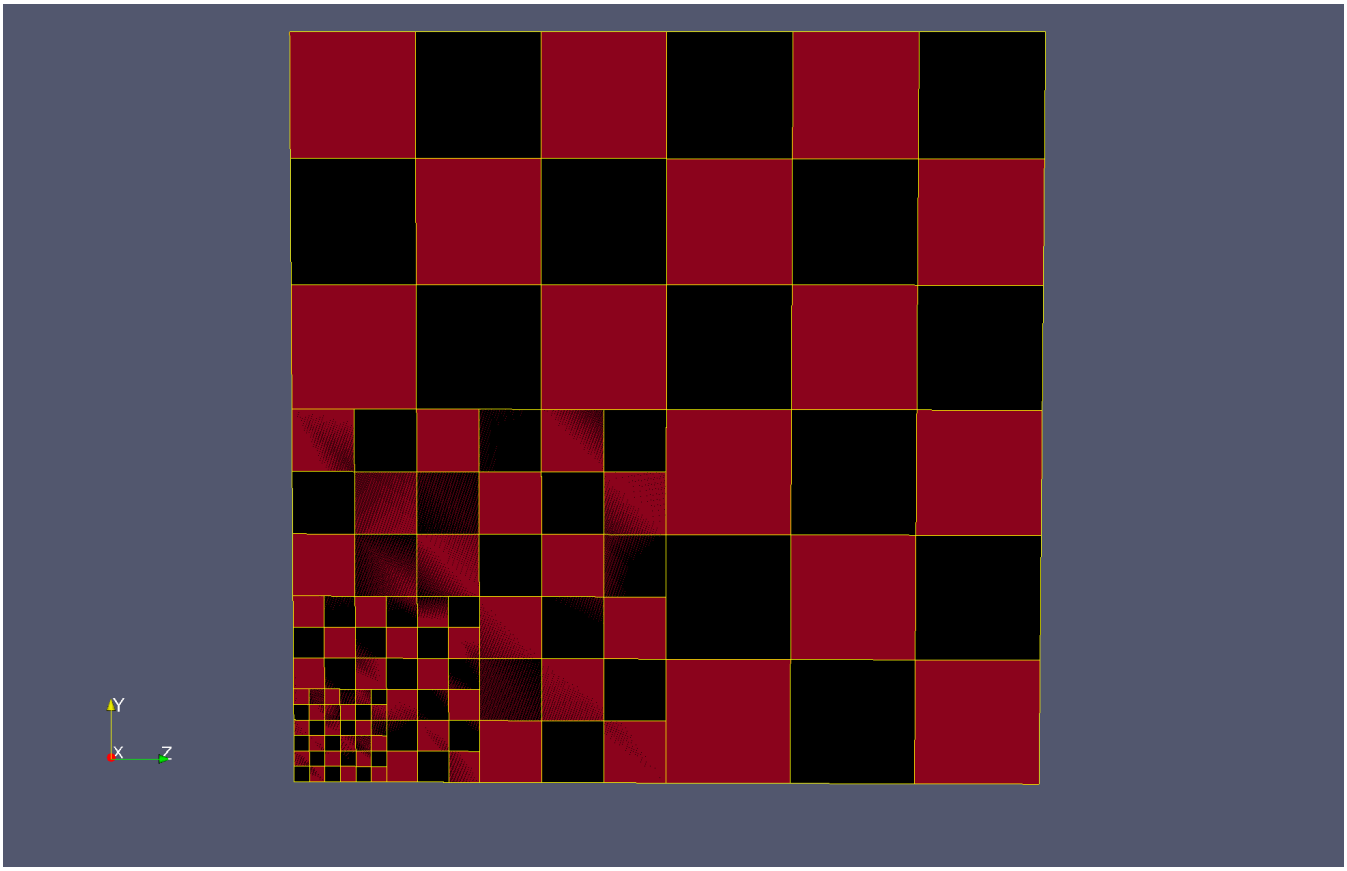

Figure 8: Procedurally generated data demonstrating overlapping AMR (2D surface)

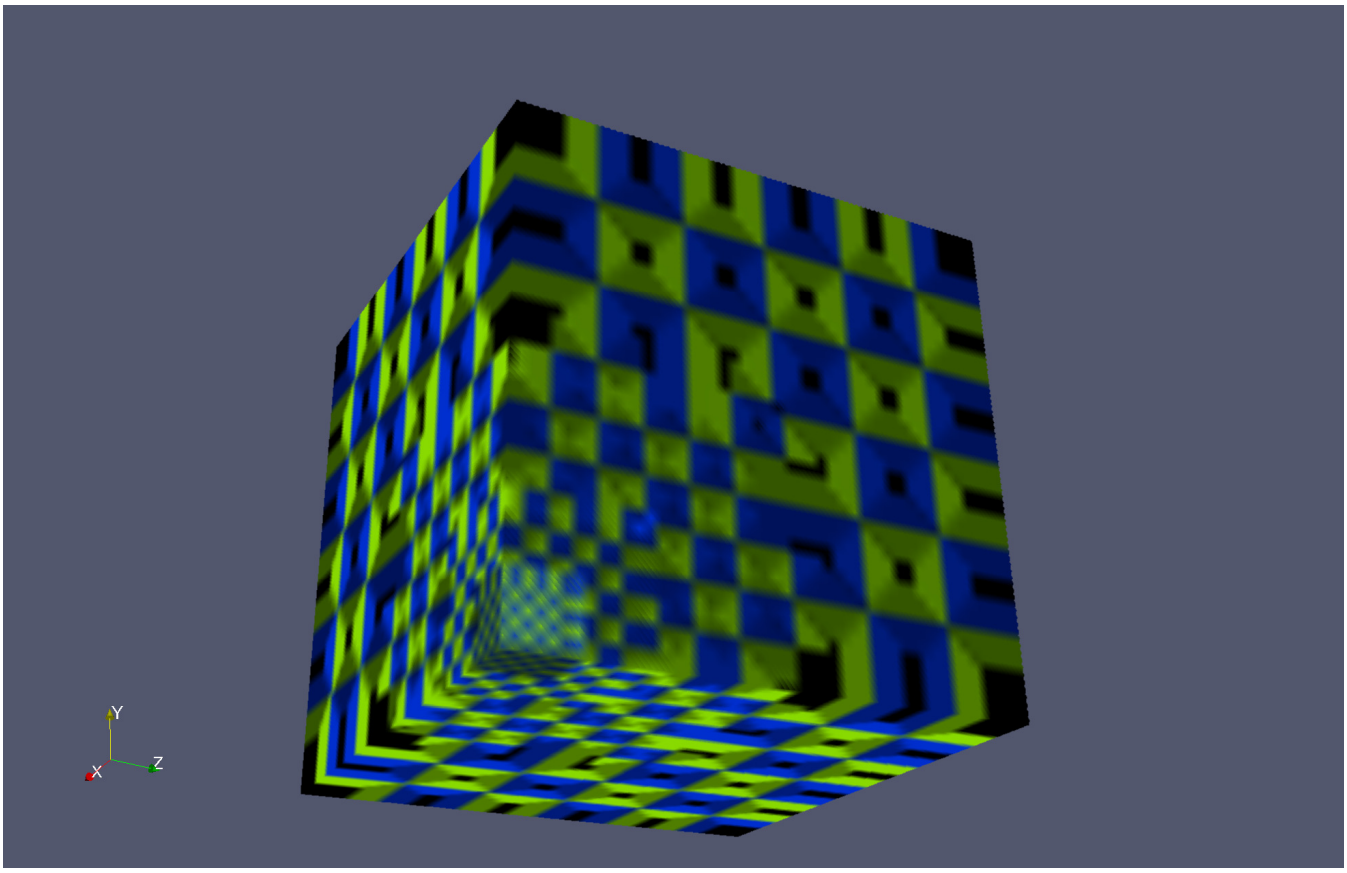

Figure 9: Procedurally generated data demonstrating overlapping AMR (3D volume)

The following 3 images (Figures 9-11) show a real-time volume render progression of a multi-resolution dataset containing 3 levels of resolution, with the AMR blocks divided into a 3x3 grid. Figure 9 shows the render at the starting position, with most blocks at the lowest resolution. Figure 10 shows partially through the streaming process, with the blocks closest to the camera viewport rendered at a higher resolution, and blocks further from the viewpoint still at the lower resolution. Figure 11 shows the end of the streaming process, with all blocks resolved at the highest resolution.

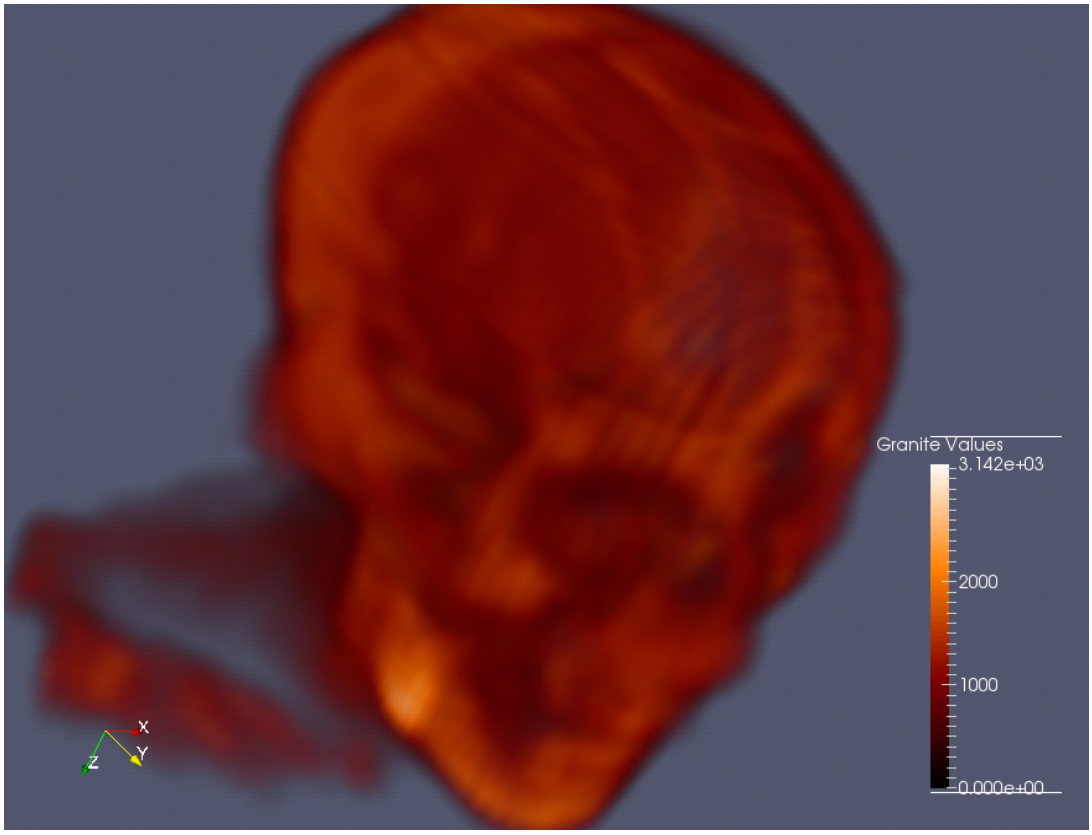

Figure 10: Streamed overlapping AMR, mostly low resolution [4]

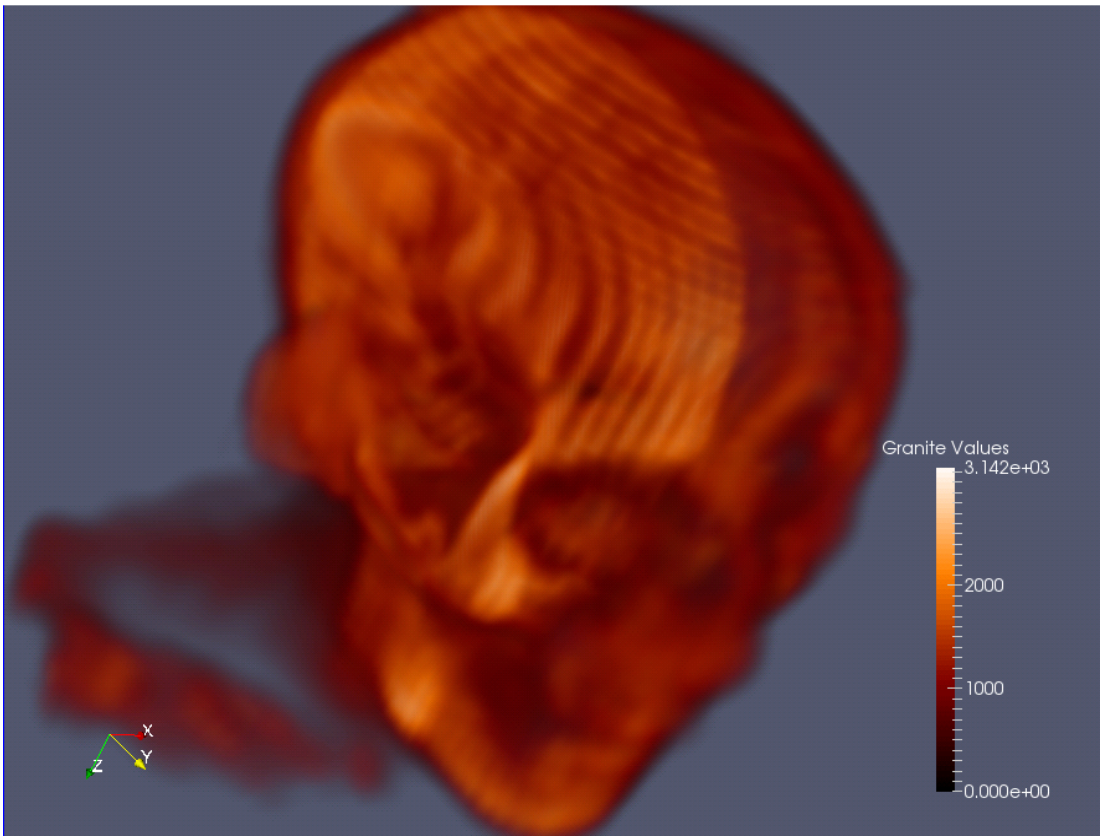

Figure 11: Streamed overlapping AMR, closest blocks at higher resolution

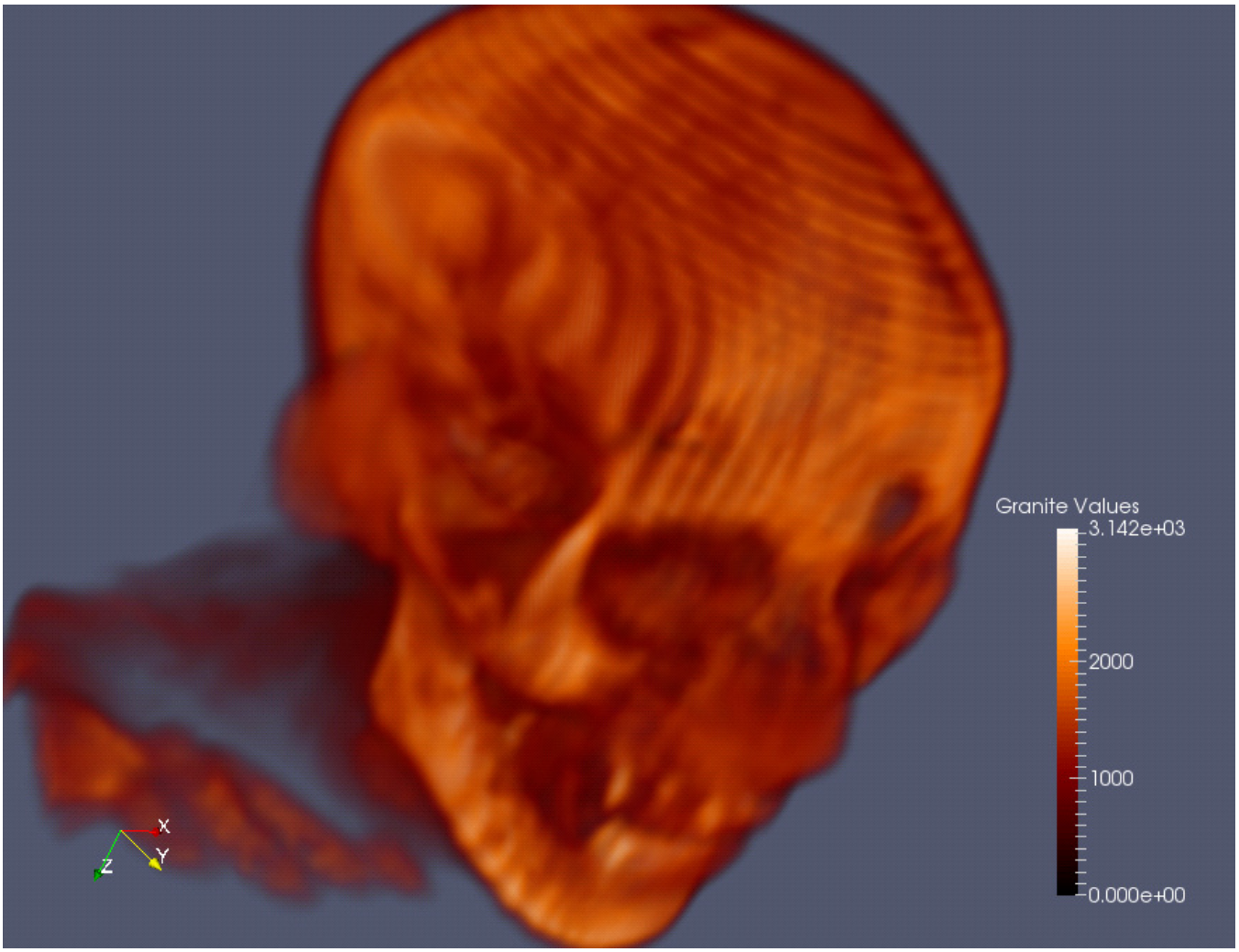

Figure 12: Streamed overlapping AMR, fully resolved

Conclusions

In summary, the goal of implementing a ParaView/VTK plugin to provide interoperability with the Granite library was an overall success, with some minor limitations involving the AMR functionality, as previously noted. Many of these require updates by Kitware (or the community) to the VTK codebase.

The primary performance based limitation, mainly the overhead and virtualization involved in interfacing between native and JVM execution via JNI, is difficult to overcome. While small changes could be made to the Granite library to provide concurrent push technologies into native memory, or other similar strategies, the penalty will always remain to some degree due to virtualization overhead. However, depending on context, user requirements, and machine resources, this may not be an issue.

Ultimately, the Granite plugin proves to be a very useful and versatile tool for Granite, both in simplifying the potentially laborious task of converting foreign (and poorly documented) file formats, but also in providing all the power of ParaView's filtering, rendering, and user interface to Granite stored datasets.

On a personal note, I have learned a large amount concerning both visualization in general, but also the VTK framework, both of which before this class were completely unknown to me. In regard to VTK, the deeper I have explored both the source code and the documentation, the more I am impressed by the sheer amount of functionality it provides - it is a very valuable tool which I am thankful to now have at my disposal for future work. Overall, this project was a very illuminating and valuable experience.

References and Datasets

[1] Kitware, Inc. (2014, Dec.) ParaView. [Online]. http://www.paraview.org/

[2] Ria Fischer. (2014, Dec.) Ria Fischer Ph.D. homepage. [Online]. http://jupiter.ethz.ch/~rfischer/publications.php

[3] Kitware, Inc. (2014, Dec.) VTK / Tutorials / New Pipeline. [Online]. http://www.vtk.org/Wiki/VTK/Tutorials/New_Pipeline

[4] Bill Lorensen / Albany Medical Center. (2014, Dec.) MIDAS at the National Library of Medicine. [Online]. http://placid.nlm.nih.gov/community/21

[5] University of Oldenburg. (2014, Dec.) FTP. [Online]. ftp://dicom.offis.uni-oldenburg.de/pub/dicom/images/car97cd.zip

Appendices

Software Requirements

The following software is required to built ParaView and the Granite Plugin:

1. ParaView source (verified with 4.2.0)
2. CMake (verified with 3.0.2)
3. Qt libraries (verified with 4.8.6 - version 5 has issues with CMake)
4. Java JDK of matching architecture as C++ build configuration (32/64) (verified with 1.7.0.710)
5. Granite.jar
6. CMake supported build environment (Make, Visual Studio, Xcode, etc)
7. Compiler supporting C++11 standard (Preconfigured to set gcc and VC for correct standard. Other compilers may need additional switches in Cmakelists.txt)
8. POSIX compliant OS (Pre-compiler adjusts for Windows libraries)

Build and Install Instructions

1. Ensure a system restart has occurred since Qt installation
2. Copy "Granite" directory from this package in its entirety to the "Plugins" directory in the ParaView source
3. Execute CMake GUI. Select ParaView source directory, and a choose a new target build directory (different than source)

4. Run Configure in CMake. Will likely require location of QMake executable. Choose from Qt installation, and configure any further desired options. Run Configure again until no errors are returned. Then run Generate
5. Open generated project in build directory using specified IDE/compiler. Build "ParaView" and "Granite" projects.
6. Before executing ParaView, ensure the following have been added into your system path:
    a. Qt bin directory (contains Qt dynamic link libraries)
    b. "jre/bin/client" directory of the JDK installation (contains jvm.dll). NOTE: This is the biggest area where the plugin can fail - you must be sure to choose the JDK installation that matches the architecture of the build, and you must include the directory containing jvm.dll in your path - due to Java restrictions, you cannot simply copy jvm.dll to the directory containing paraview.exe - the plugin will fail
7. Run ParaView, then navigate to Tools->Manage Plugins. Expand "Granite" plugin (if it is not listed, there was an error above, do not manually specify a DLL/XML file). Select "Auto Load" and "Load Selected". Close Plugin Manager
8. Navigate to Edit->Settings->Granite Settings, and specify the path and filename of granite.jar. Quit ParaView
9. Installation complete. When running ParaView for normal use, specify the "--enable-streaming" switch for AMR support

Note: After activating the plugin, if ParaView will not load, displays a "T()" error, or crashes, double check step 6b above.

Team Responsibilities

This project originally started as a two-person team, with Bence Cserna. Specifically, I would handle Granite/ParaView integration, and Bence would be handling visualization of a OpenFOAM dataset obtained from another department at UNH, with the two pieces integrating either on the ParaView or Granite side. As we progressed further, it was obvious due to the nature of the OpenFOAM geometry and underlying structure that integration, while possible, was not the best use of time given the number of higher priority tasks we both had. Because of this, the project split, though for collaboration purposes, we still operate as a team.

Additionally, I also shared the plugin (earlier revisions) with Erol Aygar and Andrew Boutin (though subsequently, Andrew's project took a different direction). Erol and I had a large amount of very productive collaboration with each other. Overall, it was a very positive team experience.