University of Oslo
Department of Informatics

Visualization of Scientific Datasets Obtained From Parallel Simulation

Gunnar Sletta
gunnarsl@ifi.uio.no

30th April 2002

Acknowledgements

This document contains my thesis for the Cand Scient degree in informatics at the University of Oslo (UiO), Department of Informatics (Ifi). The work has been done in asscociation with my advisors, Xing Cai, Ph. D, and Hans Petter Langtangen, Prof. II. The Cand Scient degree was began in January 2000. Implementation of the software and writing on the thesis was begun in the Spring of 2001 and concluded in May 2002.

The work I have done in this time is the implementation of a visualization system including many interresting and sometimes frustrating challenges, over which I have lost several nights of sleep. The work has been documented, tested and discussed in this thesis document. Now that it is over, I feel it has been an experience that has taught me to keep focus and perspective, and perhaps above all else, it has taught me to be patient and keep going. I believe it has been an experience I will benefit from in years to come.

I would like to take this opportunity to express my gratitude to my primary advisor Xing Cai for his time and effort in giving me encouraging feedback through the writing and implementation process, and for his ability to respond to every mail within the hour. I would like to thank my mother and father for their moral and financial support, and I would also like to thank my fellow students and friends for all the fertile discussions on the technical finesses.

Finally, I would like to express the utmost reverence for my fiancee, Elin E. Johansen, for staying with me and supporting me in hard times, and for her love which has been my perpetual source of inspiration, guiding me through this to the end.

Gunnar Sletta
University of Oslo
30th April 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scientific visualization and simulation

Visualization is a part of our everyday life. It is used in weather forecasts, bar charts of stock market prices and tour maps of the local mountainside.

It has been formally stated in [17] that "visualization is concerned with exploring data and information in such a way as to gain understanding and insight into the data". In this thesis, we restrict such information to be the result of a computer simulation of some physical phenomenon. Such scientific simulations are run to enable us to predict or study the original phenomenon, by describing it mathematically and producing results for various parameters. This is done because we want to predict the outcome of the physical phenomenon, such as with tomorrow's weather forecasts. This weather forecast simulation can be based on today's air pressure, strength of the wind and other factors, and is simplified so that we can describe them by a mathematical model. The mathematical model can be implemented in a simulation program that we then can use to run various simulations of the weather for tomorrow.

The result of the simulations can very often be a vast amount of numerical values. Since the human mind is not capable of processing many numbers at the same time, the result in its original form is not very useful.

Imagine that a simulation has been done to study how fast heat spreads across a metal plate. We then want to view the data and see what is happening as time passes. Unfortunately, the large list of rapidly changing numbers is not very descriptive to us. We therefore need to introduce an added form of perception to enable us to see more than the simple mass of numbers. Our eyes are responsible for the major understanding of the world around us, so using this powerful tool of sensory input, we can have the computer transform the numerical values into images for us to view. By studying the images, we can more easily extract information that is relevant.

For the above heat conduction simulation, we introduce different colors for different temperatures, red for hot and blue for cold. These are colors we relate to

warm and cold in the real world. Then we place the color values in a coordinate system to form an image. If we let the image change over time according to the simulation, we have an animation that represents our phenomenon in a way that we can understand and study.

The importance of visualization is best stated as in [1]. “Since visualization directly engages the vision system and human brain, it remains an unequaled technology for understanding and communicating data”.

1.2 Parallel simulation

All scientific simulations are based on mathematical models, and as these models grow in size and complexity, it is necessary to use computers to run simulations. Computers, however, have limitations, such as memory size and processor speed. This limits the accuracy and the size of the simulation we can run. In recent years one has started to use parallel computers to run scientific simulations. A parallel computer is a set of processors that are able to work together cooperatively to solve a computational problem. This includes clusters of PCs, networks of workstations, supercomputers and any other architecture that combines more than one processor in a single task.

Parallel computers are an important tool because they offer us the possibility to concentrate computational resources, such as processing power and memory, on a specific problem. For scientific simulation this enables us to solve our physical problems more efficiently by dividing the computation among the processors. This makes it possible to compute more accurate or complex solutions.

Another gain is that we are now able to solve problems that were previously too large. A simulation that is too memory-consuming to compute on one processor alone, can be decomposed into smaller domains and processed in parallel by a group of processors

A common approach to parallel simulation is divide-and-conquer, described in e.g. [19]. It is concerned with decomposing the original solution domain into subdomains where each subdomain is processed by an individual simulator, called a subdomain simulator. For the simulation to produce the same result as if were done on one processor, the subdomain simulators must exchange information during processing. Each subdomain simulator creates its own set of data.

1.3 Parallel datasets

In this thesis, we refer to a dataset as a collection of discrete data on the form of fields and grids. The grid defines the topology of the data, such as the sample points in a surface. The field defines a function based on a grid such that for each sample point in the grid, there exists a field value for it. The dataset is the collection of all the grids and fields that has been produced during a simulation.

Recalling our example about how heat spreads across a metal plate, the grid will represent the topology of the plate, and the field will contain the temperatures for each sample point in the grid.

Parallel simulations are done by decomposing a domain into subdomains. The individual datasets are the output from the individual subdomain simulators. A parallel dataset is a collection of individual datasets that are the result of a parallel simulation. We will use the term parallel dataset or global dataset to address the union of the individual datasets. We will use the term subdomain dataset to address these individual datasets. We use the term single dataset to address datasets produced by non-parallel simulators.

1.4 Main result of the thesis

The primary result of this thesis is the implementation of a visualization system, named PVis, that is able to load and display parallel datasets. The visualization system is composed of a data loader, a user interface and filter modules. The main feature that separates PVis from other visualization tools is its ability to load a parallel dataset and treat the individual subdomain datasets collectively as a single dataset.

The visualization system has been implemented using the Java programming language[3] and use a Java 3D[8] based package named VisAD[10] for rendering. Details on these topics are covered in later chapters. The datasets of interest are from Diffpack[20, 21], a library for solving partial differential equations.

1.5 Organization of the thesis

This thesis is composed of eight chapters. The next chapter covers the software and packages used as part of the implementation in this thesis, because some knowledge about these existing software and packages is helpful for understanding the content of the forthcoming chapters.

Chapter 3 covers visualization of parallel datasets and describes how they differ from single datasets. We propose in this chapter three methods for representing parallel dataset in a visualization system and discuss how these methods will affect standard visualization techniques.

Chapter 4 covers the implementation of PVis, which is a visualization system based on a processing pipeline that loads datasets, filters them and displays them. The execution order of the pipeline and and all its building blocks are covered here.

Chapter 5 covers the graphical user interface of PVis. It describes the implementation details, such as the classes that have been written and the relationship between them.

Chapter 6 describes case studies done with the system. The object of the case studies is to test the performance and usability of the PVis, as well as finding its limits.

Chapter 7 discusses the PVis system by analyzing the implementation of the system and the software it is based on. It will also give explanations to some of the limitations that were uncovered during the case studies.

Chapter 8 concludes the thesis by summarizing the work that has been done and points out some issues for future extension.

The products of this thesis are located on the web site

`http://www.ifi.uio.no/~gunnarsl/pvis`

The web site contains installation guide, source code, compiled code, a screen shot gallery and a user tutorial.

For all example code and pseudo code we will use a monospaced font and a syntax that is close to the Java Coding Conventions [12], although trivial keywords such as private and public often have been omitted. Package names use all lowercase letters. Class and interface names are simple or complex nouns with the first letter of each word capitalized. Method names are written with lowercase for the first word and uppercase for the first letter in any subsequent word, and ending with parenthesis. Variable names follow the same standard as methods save for the parenthesis at the end. Functions and subroutines are called methods. Dot (.) is used to dereference object methods and variables. The following is an example.

```
package a.b.c;
class Example {
  int value;
  int getValue();
}
```

Chapter 2

Background, tools and software

This chapter covers the background material, tools and software used in this thesis.

2.1 The Java programming language

The Java Programming Language started out as a simple programming language for running small applications in web browsers in 1996. It is based on the concept of compiling source code to byte code[1] that could be run on a Java Virtual Marching (JVM), described in [5]. A JVM has been implemented for all the major existing operating systems and platforms, making Java applications platform independent (compile once, run anywhere). The Java language has continuously evolved and expanded since the then. It exists today commercially in version 1.3, and is a widely used programming language.

2.1.1 The Java standard library

The Java language is shipped with an extensive standard library that contains general APIs[2] assisting the programmer in the most common tasks, such as user interfaces, networking, input/output and database connections. Most applications written in Java take advantage of these APIs, which mean that a large community of programmers use the same standardized code.

Most common programming tasks, such as file access, networking and user interfaces, that would normally require low level programming and hardware access have also been encapsulated in the standard library in a platform independent way. This has been done so that programmers need not worry about the target platform of their applications. Should it be needed to access the hardware directly however, that can also be done, through Java Native Interface covered in 2.1.5.

[1] Bytecode is targeted at a virtual hardware. Machine code which is the compiled result from a language such as C++, is targeted at a specific CPU type, such as the Pentium processor.

[2] Application Programming Interface, a collection of routines targeted at performing one or more tasks.

The disadvantages of having an extensive library is that it takes time to learn it and get familiar with it.

2.1.2 Dynamic class loading

Java uses dynamic class loading, which is the ability to load classes on demand at runtime. The JVM uses this feature to load the classes needed for a program to execute. Dynamic class loading is, among other things, used in distributed settings to pass references of objects between JVMs. If an object being passed as reference from one JVM to another does not have an implementing class file on the receiving JVM, its implementation is loaded from the sender and the object will function equally on both JVMs. In this thesis, class loading has been used to dynamically load implementations of modules in the PVis system.

2.1.3 Java language reflection

Closely related to dynamic class loading is reflection. Reflection is the ability to analyse the program at runtime. All Java objects hold a reference to its implementing class. From the class one can extract all declared methods and fields. Methods can be invoked and fields can be read and written. This technique is for instance used to call the `main` method that is the starting point of any standard Java application. Methods and fields are also objects so once extracted from the class implementation, they can be passed as arguments to other methods, thus enabling passing of methods and references to fields as arguments, a feature that is not available in the core Java language. Java language reflection is provided in the `java.lang` and `java.lang.reflect` packages in the standard library.

2.1.4 Object serialization

Reflection can also be used to make exact clones of objects, iterating over every field in an object and copying them. This particular technique is called object serialization. It is the process of dumping an entire object structure to an output stream. The process of loading a serialized object structure into memory is called deserialization. This can be used to pass objects structures between JVMs or as a generic storage format.

2.1.5 Java native interface

Java Native Interface (JNI) is Java's means of communicating with other programming languages. This is done by creating a shared library[3] primarily written in C, C++ or Assembly, but any language that can be compiled to shared libraries can be used. To access a shared library from Java, one defines a class that loads the shared library and declares one method for each of the functions that should be available.

[3] Dynamic Link Libraries (.dll) on Windows and Shared Object (.so) on Unix

JNI can also be used to access Java from a native language such as C. Technical details and examples can be found in [6].

2.1.6 Threads

The Java programming language has the ability to spawn a program's execution into multiple threads. A thread is a task that executes separately from the rest of an application, while still having access to the applications memory.

Threads differ from processes, as they would run on a parallel simulator, in that threads share the same memory while processes may have separate memory. This means that a multithreaded program can have multiple threads accessing the same variables, whilst an application running with multiple processes, have separate variables for each process and have to communicate data through more complex means, such as piping or message passing.

Threads are implemented in the class `java.lang.Thread`. The fact that threads are a part of the standard library is a strong feature, opposed to many other languages where threads are supported through extension packages.

2.1.7 References and pointers

Java uses references in stead of pointers. References are different from C style pointers in that one does not allow access and manipulation the pointers, only the objects they are referring to. In addition, references can only refer to objects, not variables. This means that methods can only have one return value, and one cannot make references to methods. Workarounds are possible, such as returning an array of values or using reflection, but these methods are not equally efficient.

2.1.8 Memory model

The weakest side of Java is its memory model. Java uses a Garbage Collector (GC), a background process that locates and removes all objects that are no longer referenced by the application. This is convenient for the programmer as it leaves deallocating memory for objects up to the system. The problem however, is that one cannot through the programming language directly control the GC. The GC runs separately and has two drawbacks. First, one cannot tune the memory usage of an application, since memory may pass far beyond an estimated maximum before the GC runs. Secondly, the GC can decide to run during a critical part for the application, drastically reducing performance when it matters.

2.1.9 Runtime compilation

The original JVM (version 1.0) compiled byte code into machine code at run-time, one instruction at a time, which resulted in that java programs executed very slowly. To achieve better performance a technique called Just In Time (JIT) compilation

was introduced. JIT compilation is the process of evaluating the byte code at run-time and compile critical parts of the code to machine code. The JVM will then run a combination of byte code and native code. Given the fact that most of a program's execution time is spent on minimal parts of the code, this has greatly increased the performance of the JVM. JIT compilation also covers dynamic inlining of methods, loop unrolling and elimination of bounds checking and null pointer checks. The details on the JVM and JIT are covered in [5].

2.1.10 Array management

Arrays in java are objects, not only memory blocks, as they are be in C/C++. These objects have been supplied with a length field, telling the program in runtime its length. Multi-dimensional arrays are in fact a 1D array containing recursively new independent arrays that may have different lengths for each row. The impact this has on memory is discussed further in Section 7.1.1.

2.1.11 Datatypes

The Java programming language has a standard set of primitive datatypes, that all have a fixed resolution independent of the underlying hardware or platform.

```
boolean      1 bit
byte         1 byte
char         2 bytes
short        2 bytes
int          4 bytes
long         8 bytes
float        4 bytes
double       8 bytes
```

All Java classes are derived from the class `Object` in the `java.lang` package. This means that one can always reference any object with an `Object` reference.

2.1.12 Summary

By itself, the Java Programming Language is considered a low threshold programming language. The basic features are stripped down compared with more complex languages like C++. This results in that Java is a language that is easy to learn but takes long time to master, and once mastered, the language is a highly functional language, which is probably why it has been adopted in many communities worldwide.

Compared with C/C++, Java is slower and more memory consuming as a result of its abstraction from hardware, but these downsides are outweighed by the gain in shorter development time and more stable applications.

2.2 Java 3D

Java 3D is an object-oriented class library for 3D graphics written in Java. Its basic function is to define virtual universe composed primarily of visual objects and lights. Java 3D renders this virtual universe "behind the scenes" by using existing graphic libraries such as OpenGL[23] and DirectX[24].

Java 3D is built to serve as a building block for java applications and libraries that require 3D graphics. For that reason, it only contains general functionality for setting up three-dimensional environment and having the environment displayed. It does not include specialized functionality such as field representation or filters needed for scientific visualization.

The Java 3D Specification[9] states the following: "Java 3D allows the programmer to think about geometric objects rather than about triangles - about the scene and its components rather than about how to write the rendering code for efficiently displaying the scene."

2.2.1 Scene graph

The rendering process in Java 3D is based on a scene graph, which defines a virtual universe. The scene graph is composed of various components, called nodes, that serve as real world abstractions from low level graphics components. Geometric primitives such as triangle and quadratic strips, are abstracted as shape nodes together with attributes that determine how the geometry should be rendered.

The color attribute defines the basic color for the shape. The rasterization mode determines whether the shape is rendered as a wireframe, as points or as a filled polygon. The backface culling attribute can be set to avoid rendering primitives facing away from the viewer. The material properties define the ambient, diffuse, specular and emissive color used to determine the coloring effect of light sources.

Matrices used to control translation, rotation, scale and skew are abstracted in transformation nodes that are connected to the scene graph for the nodes they operate on. Some other node types that exist in the scene graph are lights, camera and fog.

The scene graph can be either mutable (read and write enabled) or immutable (only readable once the graph is rendered) at the programmer's choice. Immutable scene graphs can be compiled and optimized to achieve better performance.

2.2.2 Example of basic flow

We will now describe how the components of Java 3D fit together. The first thing is to have input data. It can be on the form of a file, a mathematical function or any other discrete or sampleable form. From this input data one must create a shape composed of polygonal geometry, such as triangles or triangle strips. For each point in the polygon one can also add colors, normals and texture coordinates. We can then set up appearance for the geometry by using a set of attributes.

Polygonal geometries can be created in several ways. The standard distribution of Java 3D contains a utility library that among other things can process a general set of points into polygons, such as triangles or triangle strips. The programmer can also implemented his own routines for this that can be optimized based on specific topologies.

A virtual universe is created and the shape is added to it, together with one or more light sources to illuminate the object. Then a display is connected to the virtual universe and the display is added to a user interface. That concludes the scene graph. If one wishes to interact with the graph one can add behavior nodes to the graph that respond to mouse and key events to change the appearance of the graph.

2.2.3 Application to scientific visualization

As stated earlier, Java 3D is a general library that can be applied to different applications. We will now exemplify how it can be applied to the field of scientific visualization.

A visualization system based on Java 3D must have some form of internal representation of data that is convenient for visualization based processing. It must also contain a module for transforming that data to a format understood by Java 3D.

2D scalar data

2D scalar data, be it iso surfaces, slice planes or plain 2D fields, are trivial to render in Java 3D. They are mostly represented as mono- or multicolored planes or surfaces in a three dimensional space. They can be rendered by extracting the geometry info from the grid representation and the colors from the field values. Shading effects can be achieved by using the grid and field values to generate normals.

3D scalar data

3D scalar data can be represented in several ways in Java 3D. The first is to use the volume rendering technique described e.g. in [25] that creates a series of slice planes through the data and display them to give the impression of a filled volume.

The other is to take advantage of 3D texturing hardware. Java 3D defines a 3D texture that can be applied to a volume. 3D texturing is not safe to use in a general visualization system however, since it is not emulated in software and is rarely supported by hardware. More information about 3D textures can be found in e.g. [9].

Vectors

Vector data can also be represented, since Java 3D supports lines as a primitive type. Each vectors position can be extracted from the grid, and its direction from the field.

To draw a vector one could the for each sample point in the dataset, create a line in the direction of the vector, with length and color relative to the field value, starting at the sample point. Multiple lines could also be combined to create arrows.

2.2.4 Abstraction from hardware

Java 3D by itself includes no code for rendering, only a structure of graphical data. The rendering is done in a low-level 3D graphics library such as DirectX or OpenGL. Java 3D is connected to a given low-level API using Java native interface, see Section 2.1.5, that translates its functionality to the graphics library. Currently, there exists implementations of Java 3D for OpenGL for Linux, Solaris and Windows, and a DirectX implementation for Windows.

OpenGL is a graphics library, originally from Silicon Graphics. DirectX is a multimedia package for sound and graphics delivered by Microsoft for the Windows platform. A common feature of OpenGL and DirectX is that most of their functions use hardware routines directly, or emulate them using software routines if they are not supported by the underlying hardware. An example is texturing of a polygon. Most of today's computers have a video card that supports this function. When running on these computers OpenGL or DirectX will recognize this, which means that the CPU is free to do other computations. On a more stripped down computer, like a laptop, the videocard may only support for 2D graphics, in which case, the libraries will emulate texturing using software routines.

Since Java 3D relies on these low-level graphic libraries, which use hardware routines or emulate them when not supported, the same feature will apply to Java 3D. This means that Java 3D will render properly independently of the underlying hardware, and will use hardware acceleration if the underlying hardware supports it.

Note that the Java 3D class library is identical for the two implementations, only the shared libraries are different, meaning that a Java 3D application written on DirectX on the Windows platform can be run with OpenGL version of Java 3D on Linux without recompiling.

2.2.5 Performance and quality

The commercial version of Java 3D available today is 1.2. This release offered great improvement to memory management, with the introduction of referenced geometry. In earlier releases, the polygonal geometry could not be updated, only replaced, which would result in massive reallocation of large arrays. This means use of extra memory and recalling that the GC runs at unpredictable intervals, it may greatly reduce the performance as well.

Java 3D has only been available for a few years, and has reduced quality on some areas. One of these areas is rendering transparent objects. Java 3D lacks depth sorting for transparent polygons which results in that transparent surfaces are not rendered properly.

Java 3D is abstracted from hardware and hardware drivers, means that one cannot, through Java 3D, access hardware directly. The programmer is for that reason not able to tune the software to perform at maximum for specific hardware. Nor is it possible to take advantage of special features that may be present on individual graphics hardware.

2.2.6 Platform dependence

As stated above, Java 3D is based on access to existing graphic libraries. These graphic libraries are platform dependent and are accessed through native interfaces, which also makes the implementation of Java 3D platform dependent. This is a violation of one of the basic philosophies of Java. The Java 3D class library however, is the same for all platforms, so applications using Java 3D are portable to any platform that has Java 3D pre-installed.

2.3 VisAD

VisAD [10] is a visualization library written in Java, which uses Java 3D for rendering. It is developed at Space Science and Engineering Center at University of Wisconsin-Madison. Its basic functionality is to define grids and fields, describing them with a general mathematical model and rendering them in a display. A strong feature with VisAD is its compact programming style.

2.3.1 Dataset representation

VisAD supports a variety of data types. Below is given a short introduction to those data types that are relevant for this thesis. Recalling our definition of data from Section 1.3, we have that data is on the form of fields and grids. Grids exist in VisAD as implementations of the class `Set`. The grids described below are all derived from the class `SampledSet` which defines the topology through explicitly or implicitly defined sample points. Fields exist in VisAD as implementations of the class `Field`.

The linear set classes

The simplest form of a grid is the linear sets, which are implemented in classes `Linear1DSet`, `Linear2DSet` and `Linear3DSet`. Linear sets define a uniform ordering of points such that all sample points can be implicitly defined by a start value, a stop value and the number of sample points for each axis. Neighboring sample points will compose rectangles (2D) and boxes (3D) with angles that are

all 90 degrees. The linear set classes have the fastest access times and is the most memory efficient representation of data topology available in VisAD, but is limited to data that is on a regular form.

The gridded set classes

The gridded set defines an ordered topology where coordinates of all sample points are explicitly given. Gridded sets are implemented in the classes `Gridded1DSet`, `Gridded2DSet` and `Gridded3DSet`. A gridded set is composed of sample points and the number of sample points for each direction are called nx, ny and nz. The sample points are stored linearly in a point array which is $nx \cdot ny \cdot nz$ long, where the x coordinate is the most rapidly changing and z is the least rapidly changing coordinate.

`Gridded1DSet` can be used to represent the topology of an arbitrary line in 1D, 2D or 3D composed of nx sample points, where the distance between each sample point may differ. `Gridded2DSet` can be used to represent a plane in 2D or a surface in 3D that is composed of $nx \cdot ny$ sample points. `Gridded3DSet` can be used to represent an 3D structured grid of $nx \cdot ny \cdot nz$ sample points.

The IrregularSet classes

Irregular sets define the topology for irregular data that is composed of sample points and elements, where the elements are triangles for 2D and tetrahedra for 3D. Irregular sets are implemented in the classes `Irregular1DSet`, `Irregular2DSet` and `Irregular3DSet`.

The set of elements is represented as an object of the `Delaunay` class. The `Delaunay` object is composed of an element array describing which points make up the elements, an array describing which elements share the same points, an array describing for each point, which element it is a part of an finally an edge matrix that describes which elements share edges.

The irregular set implementations are ineffecient due to the irregular topology of the data as well as memory consuming due to the arrays for element representation in the `Delaunay` object.

The UnionSet class

The `UnionSet` class defines a union of an array of `Set` objects which enables one field to be mapped across multiple grids. The number of sample points in the `Field` object will equal the sum of the number of sample points in all the `Set` objects used in the `UnionSet` object.

The FlatField classes

The `FlatField` class is, according to [11], “a class for finite samplings of functions whose range type and range coordinate systems are simple enough to allow

efficient representation". An object of the `FlatField` class stores an array of discrete function values, each corresponding to the sample point in the underlying `Set` object. Multiple arrays can also be used within one field to represent multidimensional field values such as vectors.

2.3.2 Metadata description of VisAD datatypes

The VisAD datatypes, such as a grid or a field, contains mathematical metadata called a `MathType`. There are several subclasses of `MathType`, `ScalarType` representing 1D real values, `TupleType` representing tuple of other `MathTypes` and `FunctionType` representing a function mapping one type to another type. For instance, a field of temperatures over a 2D grid can be described as the following hierarchy of types:

```
FunctionType
|
+--domain: TupleType
|          |
|          +--ScalarType, X coordinate
|          |
|          +--ScalarType, Y coordinate
|
+--range: ScalarType, Temperature
```

The temperature field can be written on a more formal form: $(x, y) \mapsto temp$. When creating the grid that forms the topology for the field, it is described by the domain, a tuple of x and y coordinates. It is of course required that the tuple and the grid have same dimensions. The first type in the tuple represents the first dimension, the second type the second dimension etc. The field values are described by range, which is the scalar type temperature. The field is described by the function of domain and range. The field will then contain a description of its data, in addition to the numerical values.

2.3.3 Type based display

When a field is passed to the display module, the fields `MathType` is used to define how it is rendered. The Display has its own set of `MathType` definitions such as `XAxis`, `YAxis`, `ZAxis`, `RGB`, `Alpha` and `IsoContour`. The programmer creates connections, called maps, from the fields `MathType` to the display types. For the temperature field example above we would create the following maps:

`X coordinate` $\rightarrow$ `XAxis`
`Y coordinate` $\rightarrow$ `YAxis`
`Temperature` $\rightarrow$ `RGB`

These maps would result in that the x coordinate of a point is used to displace the point along the x axis and that the y coordinate of a point is used to displace

point along the y axis, thus all points would be located in the XY plane. The plane will be colored according the the temperature values. We could add some additional maps:

`Temperature` $\rightarrow$ `ZAxis`
`Temperature` $\rightarrow$ `IsoContour`

This would cause the temperature values to elevate the sample points along the Z axis and would render contour lines instead a surface. It is also possible to define constant maps, for instance that `Alpha` equal to 0.1 resulting in a semi transparent surface. This is contrary to a scalar map which would result in the field being close to transparent for low temperature values and close to opaque for high temperature values.

One `MathType` can be mapped to several display types, but one display type cannot have more than one `MathType` mapped to it. This means that two fields must share range and domain types if they are going to be viewed with the same filtering.

Some of the display types have widgets[4], that can be used to configure how the display type renders the data that is mapped to it. `RGB` for instance, has a simple color table editor. `IsoContour` has a widget for adjusting the threshold value.

It is not possible to change maps in the VisAD display if there is data in it. this means that if one wishes to change the way something is rendered through the replacement of a map, the data in the display must be removed first and added back after the map has been changed, which means regeneration of geometric primitives.

A result of `MathType` based data and display is that one can with basic mathematical knowledge, easily define mappings that generate useful output, and in this concept lies VisAD's strength. It also means that building a generic and flexible system based on VisAD is difficult since all the functionality is located in the display module, and not as independent modules that can be interconnected as the programmer sees fit.

2.3.4 Animation in VisAD

Animation in VisAD is not done in the traditional way, where one iterates through a loop that loads data and displays it for each step. It is done by reading all time steps of the data and placing them in a 1D field, which then forms a line of time steps, where the field value at each sample point is the field representing the time step.

The `MathType` of the new field will be on the form $t \mapsto ((x, y) \mapsto temp)$, which indicates that it is a field mapping time to fields. In the display module, a map mapping the `MathType` time in the time field is to the display type `Animation`. This map will enable animation and display a standard time string as can be seen

[4] A simple graphical user interface component

in e.g. Figure 3.2.

Since all the data is present in the display modules when animation starts, animation is fast and smooth. This approach to animation however, consumes vast amount of memory since all steps must be present in memory at the same time. For large datasets with many time steps, one will run out of memory and animation will not be possible. For such cases, animation must be split into several parts where each part has fewer time steps.

2.3.5 Rendering techniques

The rendering techniques in VisAD are partially based on display types as discussed above and partially on the implementation of the set that defines the topology of the data. For 1D and 2D sets one can define display type mappings to render a surface or contours and one can map the field values to colors.

For 3D data, the rendering technique depends on what set type is used. For `Gridded3DSet` and `Irregular3DSet` the data is rendered as points. If the data is based on a `Linear3DSet` however, one can set a parameter in the display module that generates slice planes through the data that are used for volume rendering. The technique is described in detail in [25]. Contouring is available for all set types in 3D, the same as in 2D.

It is also possible to configure some Java 3D rendering attributes from VisAD, such as the fill mode attribute that controls if polygons are rendered as points, lines or as filled polygons. This attribute can be used to view data as wireframe models.

The display module in VisAD does not support rendering of vectors or tensors. If an application wishes to visualize vector data, each individual vector has to be represented as a new set and field pair, containing the start and end point for that vector. Creating vector visualization for a $64 \cdot 64 \cdot 64$ regular set would thus result in the creation of 262144 set and field objects, even before the rendering process starts. These facts result in that vector visualization is not practically possible in VisAD.

2.3.6 Summary

Despite these disadvantages and their consequence for the implementation of this thesis, VisAD was chosen since it was the only existing visualization library for Java that used hardware acceleration when this thesis was begun.

2.4 Diffpack

Diffpack is a numerical library written in C++ mainly concerned with solving partial differential equations. Diffpack provides a set of generic C++ class hierarchies for use in applications that solve problems arising in the fields of scientific computing. By making extensive use of well-tested libraries and high-level abstractions, the time spent on writing and debugging code is moderate. Diffpack provides a

file format, called simres, that is used to store the result of simulations done in Diffpack.

2.4.1 Diffpack's simres format

A simres database is composed of five different files. The first file is the information file, named `.xxx.simres`. Each line in the `.simres` file contains the field, grid, time step, number of points, etc. The fields are located in a file named `.xxx.field` and the grids in a file named `.xxx.grid`. To aid in searching into the grid and field files an offset file is supplied for each of them describing where in the file a given field or grid starts. These files are named `.xxx.field_ix` and `.xxx.grid_ix`.

The content of the field and grid files can be in both ASCII and binary format. One file can contain some ASCII components and some binary. Which representation is used for a given component can be determined by indicators in the file.

2.4.2 Parallel simulation in Diffpack

Parallel simulation in Diffpack is achieved using a strategy of divide-and-conquer where the total domain is decomposed into multiple subdomains where each subdomain is computed in its own process and interaction between processes are handled through an abstracted Diffpack communication layer.

The simres files from parallel datasets follow a naming convention where a _p and four digits are appended to the name of each database. The number represents the domain number within the parallel simulation, starting on 0. To give an example, the fourth domain of a parallel simulation named `wave` would have its data information summarized in the file `.wave_p0003.simres`.

2.4.3 Visualization of datasets produced by Diffpack

Diffpack is written to aid the programmer in writing simulators that solve problems that arise in scientific computing. The output of these simulators are given in the simres format described above. Diffpack does not contain classes targeted at direct visualization, but provides a set of filter classes that are used to convert Diffpack's simres format into external file formats that can be read by other visualization tools.

Filter classes, can be used separately to convert Diffpack datasets from a simres database into an external file format, or they can be used in a simulator to produce the results in the external file format directly. Each external file format is represented as an individual class.

As for visualization of parallel datasets, Diffpack provides the means for converting each individual subdomain dataset into a format that can be read by other visualization tools. When these datasets are imported to other visualization tools they are interpreted as separate datasets, not as subdomains datasets in a parallel dataset. This separation will lead to several problems as will be discussed in further in Chapter 3.

Chapter 3

Visualization of parallel datasets

This chapter discusses how working with parallel datasets in a visualization system differs from working with a single dataset. We will describe several possible representations that can be used for parallel datasets in a visualization system and how these representations affect the way the visualization system works.

3.1 Overview of parallel datasets

As stated in Section 1.3, a parallel dataset is a collection of all the subdomain datasets that are produced during a parallel simulation. We will now discuss how parallel datasets differs from single datasets.

3.1.1 Large amount of data

The first thing to recognize is that a parallel dataset can be very large. This comes from the fact that parallel simulation is often used to solve large problems. A solution domain that is to large too be processed on a single computer is decomposed into several smaller subdomains that each can be processed on individual computers. When we want to visualize such a parallel dataset the problem of size arises again, since all the subdomain datasets are present at once in the visualization computer. To work around this problem there are several approaches.

One can resample the parallel dataset into a dataset that has fewer sample points, which would result in a dataset that fit into memory and could be processed. The new dataset however will both have less resolution and may contain faulty approximations compared with the original parallel dataset.

Another way to manage the large amount of data is to view only some of the subdomain datasets at once. The actual number of subdomain datasets that can be viewed is determined by their size and the amount of processing power and memory that is present on the visualization computer.

A third way of managing the large amount of data is to take advantage of the parallelism already present. Visualization could, just as the simulation, be run in

parallel, such that each subdomain dataset was processed by a separate processor

3.1.2 Overlapping boundaries

A common feature for the parallel datasets, whether they are obtained through a divide-and-conquer method or another means, is that each subdomain dataset has overlapping boundaries with its neighboring subdomain dataset. The subdomain dataset boundaries overlap in most cases such that for each point that is on the outer boundary for one subdomain dataset that point is part of the inner boundary of its neighbor subdomain dataset. An illustration of overlapping boundaries is shown in Figure 3.1.

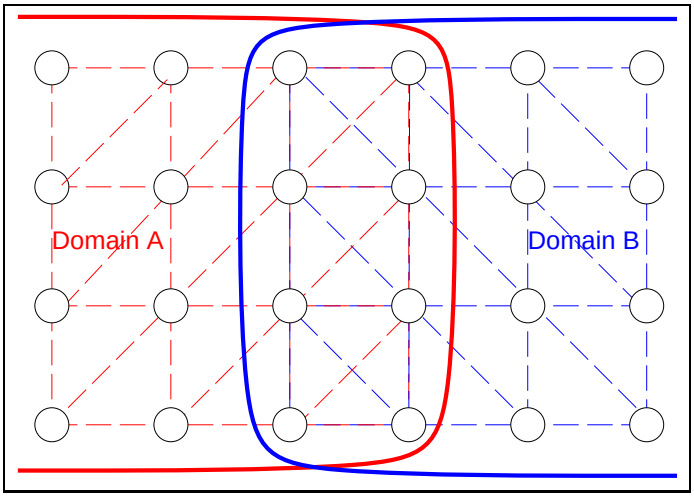


Figure 3.1: The illustration shows how two neighboring subdomain datasets, A and B, can have overlapping boundaries and different element composition.

Although the points in the overlapping boundaries are the same, the cells created from the points may differ, as seen in Figure 3.1. This is the result of that the cell topology may have been created after the global domain has been divided into subdomains.

3.1.3 Multiple data sources

A simulation that has been done in parallel will produce a parallel dataset. The individual subdomain datasets of the parallel dataset can be stored in different files, or even different file systems, depending on the design of the parallel simulator.

3.2 Representation of parallel datasets

This section covers methods that can be used to represent parallel datasets in visualization systems. Only the first, collection of subdomain dataset described in Section 3.2.1, have been used in the implementation. The others have been included for comparison.

3.2.1 Collection of subdomain datasets

One approach to modeling a parallel dataset in a visualization system is to keep the subdomain datasets separated. Subdomain datasets are loaded and filtered individually, then added to the same display to be viewed collectively. Maintaining the parallel datasets would in this case be simple and it is possible to study subdomain datasets individually as well as collectively.

What reduces the quality of this approach is that algorithms will in most cases have to be extended to take into account that there is multiple subdomain datasets as input instead of a single dataset. The changes that must be made for the various visualization techniques are specified below.

Surface rendering

Surface rendering is a visualization technique where one renders the surface of a field. The geometry of the surface can be a 2D grid such as plane or a height map or an iso surface extracted from a 3D volume. Field values are often associated with color values to increase the visual representation. The rendering is done by mapping the grid topology into graphic primitives. Example of surface rendering is given in Figure 3.2

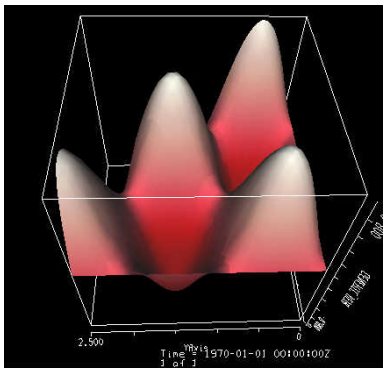


Figure 3.2: Visualization of a surface composed of 6 subdomains

Transforming 2D topologies into graphic primitives is a simple process and is described in detail in [1]. On the boundaries where subdomains overlap, the polygons will also overlap, resulting in more polygons that have to be drawn. Overlap-

ping polygons have grid points and field values originating from the same global dataset, so the points and values that the polygons are composed of will be the same. If the cells are equal and only field values are used to determine the color of the polygons, there will be no visual difference.

This however, is rarely the case, since light is often used to give an increased sense of depth. The amount of luminance at each point is determined by the normal vector for each point[1]. The normal vector for each point is resolved by calculating an average of all the normal vectors of the polygons that the point is a part of. As can be seen by Figure 3.3, this introduces an error on the boundary where the polygons from the intersecting subdomain should also have been used when calculating the average.

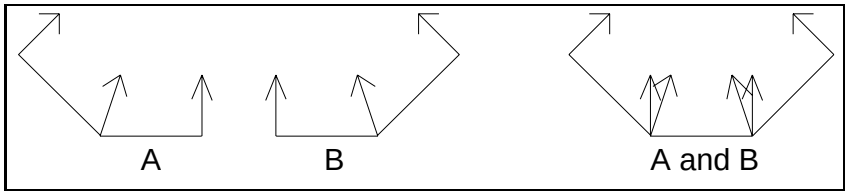


Figure 3.3: Illustrates normal vectors on overlapping boundaries

To remedy this, one would have to join the grids before rendering, as described in Section 3.2.2 or to change the rendering code to take the overlapping domains into account.

Contouring

Contouring is, according to [1], "a scalar visualization technique that creates lines (in 2D) or surfaces (in 3D) representing a constant scalar value across a scalar field. Contour lines are called isovalue lines or isolines. Contour surfaces are called isovalue surfaces or isosurfaces.".

There are two algorithms for generating contours, *marching cubes* and *dividing cubes*. There exist variations depending on the underlying element type and dimension such as *marching squares, marching tetrahedron* and *dividing squares*. The marching algorithms are based on creating a set of lines or triangles through the cells that contain the threshold value as illustrated in Figure 3.4. The other variant *Dividing cubes* is based on subdividing the cells into points where the isosurface crosses through. Dividing cubes is not implemented in VisAD and has for that reason not been discussed further.

The marching algorithms work per cell and since each subdomain dataset is composed of its own set of cells the marching algorithm will produce correct isolines or isosurfaces for each individual subdomain dataset.

[1] Actually, luminance is composed of three components, ambient, diffuse and specular light, where only the diffuse and specular lighting are affected by the normal vector.

On the boundaries of these datasets there will be overlapping isosurfaces as a result of overlapping boundaries. Iso surfaces will suffer the same side effects as other surfaces such as discussed in Section 3.2.1, but isosurfaces are generally smooth and the side effects of incorrect lighting will thus not be visible in most cases.

For marching squares and marching cubes there are certain combination of values that give rise to multiple solutions e.g. the combination of values illustrated in Figure 3.5. When processing, the algorithm will select one of the two solutions (which one is selected depends on the implementation) to be a part of the isosurface. If this happens on the border between two subdomain datasets, one can imagine that different solution may be selected for the two subdomain datasets and the result is that both solutions are present in the isosurface and the isosurface will have intersections that would not exist in an isosurface created from a single dataset.

Slice planes

Slice planes are a scalar visualization technique where one extracts the field values of a volume along a plane and displays the values in the plane.

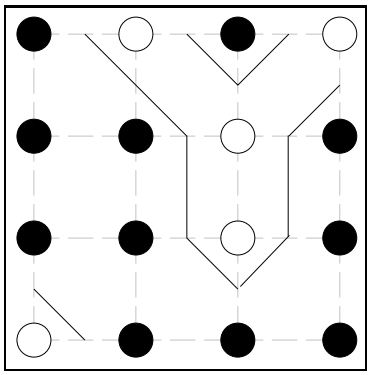


Figure 3.4: Illustration of marching squares. The black points are over the threshold value and the white points are below the threshold value.

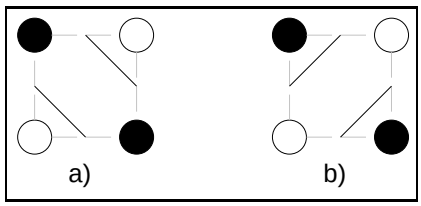


Figure 3.5: Two correct solutions to the combination of points and values for marching squares

When applying this technique to parallel datasets, one will define the function for slice plane and apply the technique to each subdomain dataset. The result will be one slice plane for each of the subdomain datasets. The subdomain dataset slices will have overlapping bounds, and suffer from the same results as surface rendering described in Section 3.2.1. It is worth noting however, that slice planes are often described by the function $ax + by + cz + d = 0$ which describes a flat plane such that normal vectors for all points will be equal resulting in that the side effects of lighting described above will not occur.

Hedgehogs

Hedgehogs is a vector visualization technique where one for each sample point draws a line or an arrow that points in the direction of the vector and has a length relative to the absolute size of the vector value.

As for the scalar techniques discussed above, this method will draw overlapping arrows on the boundaries between subdomain datasets that overlap. The sample points and their corresponding field values however, are equal so the overlapping arrows will be identical. This means that although more arrows are drawn on the overlapping boundaries, the visual quality is not reduced.

Stream lines, streak lines and particle traces

There exists several vector visualization techniques that describe flow in a vector field. From [1], we have the following.

- *Particle traces* are trajectories traced by fluid particles over time.
- *Streaklines* are the set of particles traces at a particular time t_i that have previously passed through a specific point x_i.
- *Streamlines* are integral curves along a curve s satisfying the equation $s = \int_t \vec{V} ds$, with $s = (x, \bar{t})$ for a particular time $\bar{t}$, and $\vec{V}$ describes the vector field.

These vector visualization techniques use the means of numerical integration, which implies the use of at least two sample points and their corresponding field values for each computation. This gives rise to a problem when applied to a parallel dataset. Using Euler's method which the simples form of numerical integration we have,

$$\vec{x}_{i+1} = \vec{x}_i + \vec{V}_i \Delta t$$

To solve this equation by iterating through a dataset untill it reaches the boundary is simple. If x_i and x_{i+1} are located in different subdomain datasets however, the curve should not stop, but continue into the second subdomain dataset. The

result of this is that when the iteration leads the algorithm out of one subdomain dataset, it must search through all other subdomain datasets and located which one it enters if any and continue the iteration in that subdomain dataset.

3.2.2 Preprocess subdomain datasets into one global dataset

Another approach to modeling the subdomain datasets is to preprocess all of them into one global dataset. The process would in general create a new dataset and put the sample points and field values from each subdomain dataset into the new dataset, thus creating a joined dataset.

For the preprocess to work properly we must define how to deal with certain non-trivial issues, such as potentially overlapping points and cells. Overlapping grid points will have the same field value, since they originated from the same global domain. To solve the issue one must avoid adding duplicate grid points, and cells with their corresponding field values to the total domain.

It is trivial to avoid adding duplicate grid points into the global datasets by searching through the global dataset and not add points already present, although such a method may be inefficient. Avoiding duplicate cells is less simple. This is because the points that make up a cell in a subdomain dataset can have been discarded due to duplicity in the global dataset. Also, it is not given that cells in an overlapping boundary are composed of the same combination of points in two subdomain datasets, as was shown in Figure 3.1.

If it is known that the subdomain datasets are on a structured form such as the `LinearSet` and `GriddedSet` described in Section 2.3.1 the cells can easily be regenerated for the overlapping boundaries since the points and the structure of the points are known. For unstructured subdomain datasets one has to use more sophisticated techniques. One could recreate all cell data based on the points in the global dataset, which would be a memory and time consuming process. Another way is to isolate all points that are overlapping and regenerate cells for these points.

The preprocess can be done at runtime or before the visualization starts. Joining at runtime would consume extra resources since data has to be copied from the subdomain datasets into the global dataset, and possible merge conflicts have to be resolved. Joining the subdomains before visualization starts however, will prohibit the possibility of working with them independently.

The joined dataset will be a single dataset, allowing the use of conventional visualization techniques without any modifications.

3.2.3 Virtual global domain

The two approaches discussed above have certain advantages and certain drawbacks. It would be most preferable if we could take advantage of the existing visualization techniques without the loss of detail or the overhead of preprocessing.

This can be done, but the implementation must match a few requirements. The first is that datasets are represented completely through an abstract definition. The

second is that all visualization techniques are implemented based only on this abstract definition and not of the specific dataset implementation such as regular and irregular grids. These requirements are covered below.

- Access to sample points. All sample points in the grid can be accessed independently and as a list or by iteration[2]. Grids that define an explicit sample point array must allow access to it, while grids that define sample points implicitly, such as regular grids, must have functionality to generate sample point information.
- Access to cells. In the same manner as sample points, cells must be accessible both independently and as a list or by iteration. Cells are the polygons that form a basis for the grid. Squares, rectangles, triangles, tetrahedra and cubes are some cell types. These cells are often represented as an index into the point array for each corner. Cells can be represented explicitly or implicitly depending on the structure of the grid.
- Approximation of points. For any point within the bounds of the grid, one should be able to locate the nearest sample point defined in the grid, as well as the cell that surrounds the point.
- Sample point extremities. It must be possible to query the grid for its extremities and to check if a given point is inside or outside the bounds of the grid.
- Access to field values. One must be able to access the field values for all sample points both independently and as a list or by iteration, in the same way as points and cells.
- Field value extremities. One must be able to extract the upper and lower bounds for field values.
- Approximation of values. For all points within the bounds of the grid, there must be functionality present for extracting the value at that point by approximating the value using the values in the surrounding cell and by using the value in the nearest grid point.

It is trivial to see that these abstract definitions form the basis for a single dataset. Another use for the abstract definition is a *container dataset*, that contains a collection of subdomain dataset. Such a container dataset would support the abstract definition by relying on the abstract definition of the subdomain datasets that it contains.

The simplest implementation is made possible by implementing a `Grid` class that is responsible for maintaining all the topology-related information and a `Field` class that is responsible for maintaining all value related information. A `Container`

[2]Iteration means that on can get all sample points one by one

`Grid` that maintains an array of `Grid` objects internally could be implemented to support the topology for subdomain grids. A similar `ContainerField` could be implemented to support the values for subdomain fields.

Container datasets can in this way be used to write implementations that pack multiple subdomain datasets in such a way that it shares the same interface as a single dataset. Given that the visualization algorithms operate on this interface only, the visualization techniques will work for parallel datasets and single datasets alike.

Note however, that the performance of each routine in the abstract definition may vary greatly depending on what type of grids the subdomains are. Some visualization techniques may perform slower as a result of that. Another issue that may reduce performance is that one cannot optimize the algorithms used in the visualization techniques based on specific implementations.

An example could be to generate a slice plane through a collection of parallel datasets. A slice plane is generated by defining a series of sampling points in a plane and extracting the field values for each sampling point. This can be achieved through approximation of values described above. The container field would for each sample point locate which subdomain contained it and extract the value from it. Overlapping boundaries would not affect the result since the both sides would contain the same value and either would do.

One can clearly see that searching through multiple datasets in order to find the correct one can slowdown the algorithm as well. Things that can be done to avoid this is to search through the collection of subdomains more intelligently. One improvement that can be done is to remember the subdomain that was last accessed and assume that it is likely that it will be accessed next. Another improvement is to build a list of neighbor info for the subdomains so that when searching in the last subdomain failed, one can try its neighbors, then finally the rest.

Chapter 4

Implementation of the PVis visualization system

This chapter covers the PVis visualization system, that has been written as a part of this thesis.

4.1 Overview

We start by giving an overview of the system and its requirements, then explain the details of the implementation. PVis has been written in Java and uses VisAD and Java 3D for visualization and rendering.

4.1.1 Design Requirements

The main functionality of PVis is to load a set of subdomain dataset, do some filtering and view them collectively as a single dataset. An assumption of PVis is that all the subdomain data in a parallel dataset are stored in Diffpack's simres format.

There should be more to a visualization system than only loading and displaying data. It should be able to use filtering modules on that data so that the information can be extracted and viewed in multiple ways. The user should be able to dynamically define the flow of data through filter modules. This is called pipeline based visualization and has been successfully adopted by products such as Iris Explorer[13] and The Visualization Toolkit (VTK)[1]. In the same way as Iris Explorer, we will through the graphical user interface let the user define which filter modules constitute the pipeline and how data flows through the filter modules.

PVis should, despite the extra complexity of parallel datasets, support the common filtering mechanisms available in conventional visualization software, as discussed in Chapter 3. This means it should for instance, be able to create iso surfaces and slice planes that span multiple subdomains.

Writing filter modules can be a time-consuming task, so supporting all possible modules is beyond the scope of this thesis. The system should however have a design that makes it easy to implement and incorporate new modules, so that users can extend the system with their own code.

4.1.2 Pipeline based visualization

A visualization pipeline describes a set of modules, and how data flow through these modules. This is similar to a directed acyclic graph [2], composed of nodes and edges. Modules define the nodes, which are capable of receiving input, processing it and passing the result to other modules. The flow of data between the modules is represented as edges. Each edge holds a module that passes down the data and a module that receives the data. Details on the PVis graph are covered in Section 4.2. The graph holds all the nodes and edges, and determines the correct order of invoking modules. The details of the PVis pipeline execution are covered in Section 4.3.

An example of a visualization pipeline is given in Figure 4.1. It contains three modules. A loader for importing datasets into the system, an boundary extraction module and a display. The output of the loader is connected to the boundary extraction module, whose output is connected to the display module. The display will with this pipeline, render the exterior of the data loaded.

Figure 4.1: Example of a pipeline

4.1.3 The PVis modules

Several modules have been written for PVis. The PVis modules are listed in Table 4.1. The table contains a short description of what each module does.

4.1.4 Representation of data

In order to process multiple datasets together we must find a means for passing sets of data through the modules, not only one at a time as is done in conventional software. Also, VisAD animation requires all time steps together, all time steps in each of the subdomain datasets have to be passed between the modules. This has been solved by passing a 2D array organized as $[I_{sd}][I_{ts}]$, where I_{sd} is the subdomain dataset index and I_{ts} is the time step index.

Since VisAD is being used for the implementation, it is natural to use VisAD's grid and field classes to represent the data in the PVis system as well. This reduces the development time and the amount of bugs. In addition it saves the system from

Module name	Description
`SimresSource`	Loads a single datasets.
`MultiSimresSource`	Loads a parallel datasets.
`PVisDisplay`	Displays and filters data.
`BoundaryExtractor`	Extracts the boundary of a dataset.
`Resampler`	Resamples datasets to a uniform grid.
`MyResampler`	Resample irregular datasets to a uniform grid.
`Combiner`	Merges multiple datasets to one dataset.
`Slicer`	Creates slice planes through volumes.
`Serializer`	Writes and loads serialized objects.

Table 4.1: The PVis modules

having to transform large amount of data from one format to another at run time, a process that could prove to be both time and memory consuming.

4.2 Class design of the PVis pipeline

This section covers the details of how the PVis pipeline is implemented, and how it executes during processing. Note that this implementation uses terminology from graph theory, so pipeline is called graph, modules are called nodes and flow of data between modules are called edges.

4.2.1 The Node class

Each module is represented in the system as a subclass of `Node`. Nodes have ingoing edges and outgoing edges that specify which other nodes are connected to it. A module's functionality is defined in its `process()` method. The method `dispose()` is used to release memory from a node once its processing is over. The method `getGetMethods()` returns the name of all the methods used to extract processed data from this object. The method `getSetMethods()` returns the name of all the methods used to set input data to this node. This is specified in more detail below. A pseudo implementation of `Node` is given here:

```
package pvis.graph;
class Node {
  Edge in[];
  Edge out[];

  boolean isProcessed();
  boolean isPreProcessed();
  boolean isPostProcessed();
  void process();
  void processNode();
  void dispose();
```

```
  String[] getGetMethods();
  String[] getSetMethods();
}
```

4.2.2 The Edge class

Connection between the modules in the PVis pipeline are described through the `Edge` objects. An edge refers to the node to start in, and a method used for extracting result from that node. In addition, the edge refers to the end node and method to call, used to put the result extracted from the start node. A pseudo implementation of the `Edge` class is given here, and the details of the calling is covered later in this section.

```
package pvis.graph;
class Edge {
  Node start;
  String getMethod;

  Node end;
  String setMethod;
}
```

4.2.3 The Graph class

The method for representing a graph that is used in the implementation of the PVis graph, is based on adjacency lists which is described in detail in e.g. [2]. The adjacency list representation of the graph is based on holding a list for each node, that contains that edges leading to and from that node.

The implementation used in PVis is an extension of the adjacency list representation, where each node holds a list of ingoing and outgoing edges. The `Graph` object holds a reference to all the nodes and all the edges in the graph. The only method of relevance here is `processGraph()`, which performs the graph execution, covered in Section 4.3

```
package pvis.graph;
class Graph {
  Map nodes;
  Map edges;

  void processGraph();
}
```

The type `Map` is a container for java objects. It is located in the `java.util` package in the standard library. It is an interface that is defined to map key objects to value objects for fast access. PVis uses the `HashMap` implementation, which stores objects in an array based on a hash function, described in e.g. [2], resulting in $\mathcal{O}(1)$ access times. Maps have been used in PVis to achieve fast and convenient lookup of objects.

4.3 PVis pipeline execution

This section describes in detail how the PVis pipeline executes. We start by discussing the method for determining the order in which nodes are processed, followed by the details of the calling mechanism, before finally showing an example.

4.3.1 Execution order

The modules in the pipeline are processed in an order corresponding to topological sort [2]. Topological sort is an ordering for directed acyclic graphs, that defines that one node can be processed only if all nodes prior to it are completed, or more precisely, all edges leading to this node, have starting nodes that are processed. Figure 4.2 shows an illustration of topological sort where four modules process in four passes. Pseudo code is given below:

```
for node in nodes
  if node.inedges is empty
     canProcess.add( node )
while canProcess not empty
  node.processNode()
  for outedges in node
    if edge.end.isPreProcessed()
       canProcess.add( end )
```

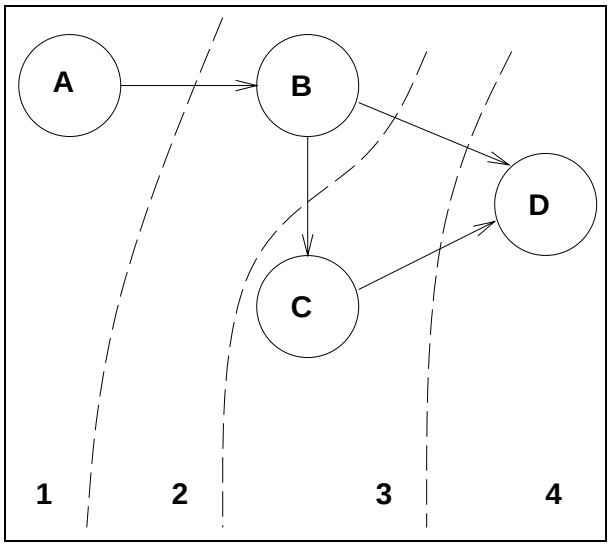


Figure 4.2: Illustration of topological sort, where four modules are processed in four passes.

As seen in the pseudo code, the graph is responsible for telling the nodes that they can process, but it does not call `process()` directly. This is done because,

before a node can process, it must fetch input data from its predecessors in the graph. How `processNode()` works is shown here:

```
for edge in inedges
  <sample input>
  if edge.start.isPostProcessed()
    edge.start.dispose()
process()
if no outedges
  dispose()
```

Nodes can exist in several states through the processing of the pipeline. These states are revealed by the `isXXXProcessed()` methods in the `Node` class, and are used to determine the order of events during pipeline processing. The first possible state a node can acquire is *preprocessed*, which means that all edges that end at this node, have starting nodes that are processed. This state is used to determine if processing can start for a node or not. Once a node has completed processing, it has state: *processed*. The last state a node can have is *postprocessed*, which means that all nodes that have ingoing edges starting at this node have are processed. This state is used to determine when node internal data can be disposed to save memory.

As stated in the beginning of this section, the graph that defines the pipeline must be acyclic. This is because a pipeline defines a one-way flow which in general starts at a source (a data loader), and ends up in a consumer (usually a renderer). Should the graph contain cycles, the nodes making up the cycle will always have a node prior to themselves that are not processed. These nodes will never be *preprocessed*, and will for that reason never be processed. Algorithms exist for detecting cycles in graphs [2], but that has been omitted in the implementation.

4.3.2 Calling mechanism

Edges in the graph, are composed of a start point, which consists of an object and a method, and a similar end point. Dynamically referencing methods, however is not trivial in Java, since it does not have the ability to pass references to methods as arguments, as discussed in Section 2.1.7

What can be done however, is to use Java language reflection, see Section 2.1.3 to inspect the `Node` in question. Reflection, can be used to search for methods that have a specific name and set of calling arguments. Methods are represented as an object of the class `Method`

```
package java.lang.reflect;
class Method {
  void invoke( Object instance, Object args[] );
}
```

Once the `Method` object has been acquired, it can be invoked.. Pseudo code for the procedure looks like this:

```
args = null
getMethod = nodeA.getMethod( getName, args )
result = getMethod.invoke( nodeA, args )

args = [result]
setMethod = nodeB.getMethod( setName, args )
setMethod.invoke( nodeB, args )
```

Output methods have been defined to have no arguments. Input methods have been defined to have one argument, the outcome of the get method that executed before it.

4.3.3 Example execution

To summarize the pipeline execution, we will give an example. Suppose node A is a module for loading fields, which are extracted through its method, $getField$. Node B is a module for viewing fields. Fields are added to the display through a method $addField$. To allow the output of A to be the input of B, an edge E_{AB} is created that has starting point at $(A, getField)$ and ends at $(B, addField)$, see Figure 4.3.

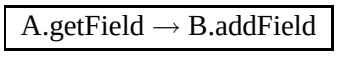


Figure 4.3: Pipeline used in the example execution

When the graph processes, it starts by locating the nodes that can be processed. In this case, A, because A does not have any incoming edges. In the next pass, B can be processed as its predecessor A has completed. B will use the edge AB to extract the field from A, using $getField$, and set the field to its own method $addField$. Then B can process, which means converting the field to geometric primitives and make them viewable to the user. B notifies A that it is done using A's data, and A reaches the postprocessed state, meaning that it can dispose itself, thus releasing the memory it has allocated.

4.4 The PVis Modules

For a module to be represented in the user interface, it must be a subclass of `Node`. In order to be of any use it must supply at least one input or output method.

Each module, must implement the abstract method `process()` which is called during pipeline processing, as specified in Section 4.3 Also, modules should implement the method `dispose()` to aid in memory recycling during processing.

The modules can also serve as an interface to a more complex structure such as the SimresSource loader, described in Section 4.4.1.

Any module that can be configured by the user must implement an ui component to interact with it. This ui component will be recognized by the ui and enable

the user to modify the settings for the module. Details on this is covered in Section 5.5

A module can be either a single class that does all the processing in its `process()` method, which is the case for the `Resampler` module in Section 4.4.3, or it can serve as an interface to a more complex hierarchy of classes such as the `MultiSimres Source` in Section 4.4.1, which uses the `process()` method to call a set of classes used to load datasets.

4.4.1 The SimresSource and MultiSimresSource modules

The implementation of the simres reader reflects the structure of the simres database on file. The simres file format is covered in Section 2.4.1. The base class is `SimresDB` located in the `pvis.diffpack` package, which servers as an interface to the simres reader. It is supplied a directory and name and uses that information to read the `.name.simres` file. This file is read using `FieldLineReader`, and produces an object of the class `FieldLine` for each line in it. The offset files are read in a class called `IXReader`. The `SimresDB` holds and instance of a `FieldReader` and a `GridReader` which are used to read grids and fields from the `.name.field` and `.name.grid` files upon request.

Grids exist in Diffpack as lattice grids which are rectangular, homogenous grids, and finite element grids composed of cells which can be of varying type. These are represented in the simres reader as `GridLattice` and `GridFE`, which both are subclasses of `Grid`. The grid classes have been structured in a representation close to those used in VisAD, meaning that point and element arrays are on the same form. This has been done to avoid an extra level of transformation which would be both time and memory consuming.

VisAD elements are composed of triangles and tetrahedra. The reader is therefore responsible for converting the elements on file to these element types while the file is being read. Squares are converted into two triangles and cubes are converted into six tetrahedra.

The grid and field files can be in ASCII format, binary format or both. Unfortunately, there is no class in the Java Standard Library that can read numerical values from an ASCII input stream[1]. Nor is there support for generally reading binary numbers. The standard library supplies a set of `DataInput` and `DataOutput` classes, but this class uses big-endian [2] form, which is Java's internal representation of binary values. The binary data produced by Diffpack however, can be both big-endian and little-endian[3] based on the platform it is running on.

As a result if this, a specialized reader, `DPInputStream` has been written that

[1] Although the functionality is present in other classes such as: java.lang.Integer, java.lang.Double, etc

[2] The bytes in a word (4 bytes) are stored with the most significant byte in the lowest address, the word is stored big-end-first.

[3] The bytes in a word (4 bytes) are stored with the least significant byte in the lowest address, the word is stored little-end-first.

has a set of ASCII read methods and corresponding binary read methods. The idea was to implement a subclass for big-endian binary and another for little-endian binaries, but only the little-endian version has been completed at present.

The Simres reader is represented in the graph as two different modules. `SimresSource` which is used to load a single Simres database into the pipeline. Internally it holds one `SimresDB` object whose read location can be configured through a simple user interface.

The other is `MultiSimresSource`, which loads a sequence of simres databases and holds one `SimresDB` internally for each of them. The user selects which subdomain datasets and time steps to load through a user interface.

The output of the `SimresSource` and `MultiSimresSource` modules is a 2D array of fields, organized as `[SimresIndex][timesteps]`.

4.4.2 The PVisDisplay module

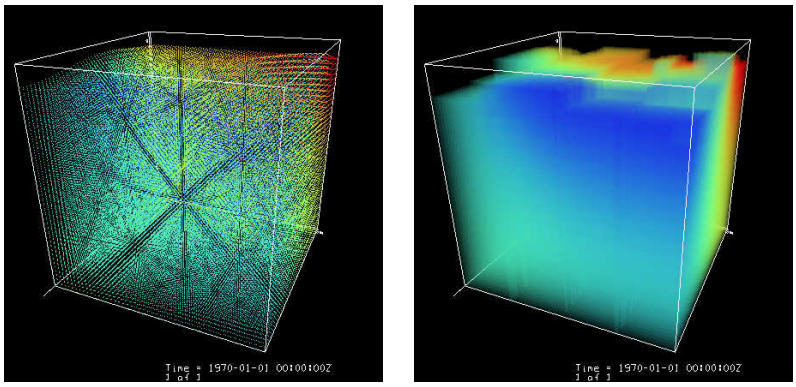
A fundamental module in any visualization system is the display and rendering module. In PVis, this is handled by using functionality already present in VisAD. The VisAD renderer is represented in the PVis system as a processor node, `PVisDisplay`, as any other module. It has one input method used to set fields that should be displayed.

Input fields are gathered in the `PVisDisplay` module until its `process()` method is called, when each field is added to the VisAD display module, with a given set of mappings.

As stated in Section 2.3, VisAD has all filtering functionality located in the display module as mappings of field values to display types, such as contouring and coloring, see Section 2.3.3. These settings as well as the display itself are a part of the displays user interface. By default, x coordinates are mapped to the `XAxis`, y to `YAxis`, and z to `ZAxis` for 3D data. For 2D data field values are mapped to `ZAxis` (indicating a height field). Through the graphical user interface of the `PVisDisplay` module, the user can activate or deactivate contouring, animation, color, alpha and a texturing property. This means that the user interface is responsible for setting up the maps that create the desired effects for the rendering part of the display. For each map that is activated its corresponding control widget is also opened so that the user can control the result of the map.

Changing display maps in the VisAD display and renderer is described in Section 2.3.3. In practical terms, this means that when the user turns off a map, like alpha, all fields and display maps must be removed. Then all fields and maps are added again.

As stated in Section 2.3, animation is achieved in VisAD by defining a 1D field containing all time steps. The `PVisDisplay` module handles this by creating one 1D field with all time steps of an animation for each subdomain dataset. What is being added to the display after this preprocess has been done is one 1D field per subdomain dataset that was originally loaded at the beginning of the pipeline.



a) b)

Figure 4.4: Visualization of a volumetric dataset. Image a) shows the dataset. Image b) shows the dataset with resampling.

4.4.3 The Resampler module

Volume rendering is available only to uniform data, as discussed in Section 2.3. Datasets however, are often represented as irregular grids in 3D, so the module `Resampler` was written to allow the user to resample irregular data on a uniform grid to enable the use of volume rendering functionality already present in VisAD. This was easily done since VisAD fields have a method for resampling fields on a new set.

The `resample()` method takes as input a `Set` object, an integer describing which interpolation mode to use and an integer describing error estimation mode. The `Set` object can be any VisAD set, but the `Resampler` module produces `Linear3DSet` since the goal of the `Resampler` modules is to enable volume rendering. The available interpolation modes are *weighted average* and *nearest neighbor* The error estimation mode is not used by the `Resampler` module.

For each sample point in the `Linear3DSet` the `resample()` method will calculate a value from the field using the desired interpolation mode. The output from the `resample()` method is a new field that has the `Linear3DSet` as topology.

The `Resampler` module creates a new `Linear3DSet` and uses it to call the `resample()` method for each input field it receives. The number of sample points in each direction and the interpolation mode can be configured through a simple user interface.

Images that show the result of resampling are shown in Figure 4.4, where a) shows a dataset as only points, and b) shows the dataset as a volume.

4.4.4 The MyResampler module

Resampling fields based on the `IrregularSet` class is extremely slow when using the standard functionality in VisAD. The `MyResampler` module was written to better the performance when working with datasets using this grid type. It is written

completely from scratch.

The `MyResampler` module is derived from the `Resampler` class and shares the same graphical user interface component. The difference is that in stead of relying on the `resample()` method present in the VisAD field implementation, it resamples the each point manually. This is done by creating a `LinearSet`, same as `Resampler` does, and extract all sample points from this new set. For each sample point the module locates the tetrahedron that contains the point in the irregular set, and calculates an average field value for the sample point based on the field values in the four corners of the tetrahedron.

Since the next sample point will in most cases lie close to the last sample point, it is likely that the next sample point will be inside the same tetrahedron as the previous or in one of the last tetrahedrons neighboring tetrahedra. Thus, by checking inside the last tetrahedron and its neighboring tetrahedra first the method will avoid traversing the entire set of tetrahedra for each sample point. If the search in the previous tetrahedron fails, standard a search is performed.

The following algorithm was used to determine if a point is within a tetrahedron or not. The tetrahedron is made up of four triangles. Each triangle is composed of three out of four of the tetrahedrons points. From the three points we can create a normal vector and from the normal vector we can create a plane equation for the triangle. The fourth point is inserted into the plane equation for the triangle. The same is done for the sample point. If the two points result in values that have equal signs, it means that they are located on the same side of the triangle. If this is true for all the four triangles, it means that the point is inside the tetrahedron.

4.4.5 The Slicer module

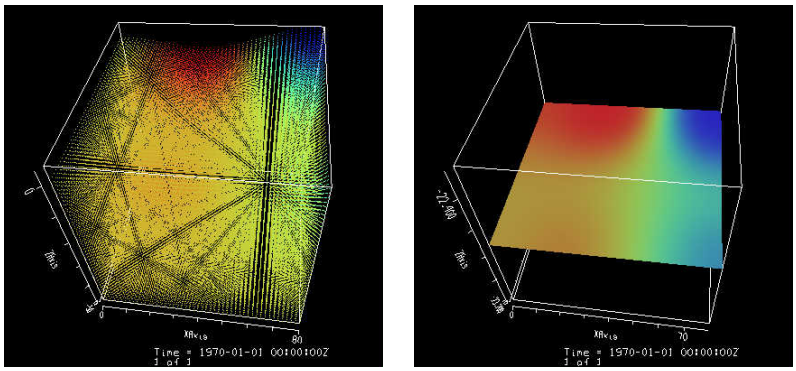
The `Slicer` module is used to create slice planes through volume datasets. This is done by defining a plane that passes through the volume, and resample the input dataset based on this plane.

The plane is created by first defining its normal vector, the `Slicer` module only allows slice planes that are normal to one of the three axis in the coordinate system. Positioning the plane in the direction of the normal vector is done by giving a weight value ranging from 0 to 1, where 0 is the lowest sample point along the normal vector in the dataset and 1 is the highest sample point along the normal vector in the dataset. The sample points are uniformly distributed between the minimum and maximum sample points in each direction.

The maximum and minimum values in each direction, x, y, z, are extracted from the dataset's grid using the `getHi()` and `getLow()` method in the class `SampledSet`.

The `Slicer` module has a simple graphical user interface component, where the user can set which axis to use, the number of sample points in each direction and the weight value, determining where the plane is resampled.

Illustrations of the result of the module can be seen in Figure 4.5



a) b)

Figure 4.5: Visualization of a volumetric dataset. Image a) shows the dataset. Image b) shows a slice plane through the dataset.

4.4.6 The BoundaryExtractor module

Another way of filtering 3D data to a format that can be viewed by VisAD is to extract the boundary of the field and represent it as a two dimensional surface. The `BoundaryExtractor` module was written to serve this purpose. Since various `Set` implementations differ in topology, different methods for extracting the boundary had to be implemented. These boundary extraction methods are derived from class named `BEProcessor` in the `pvis.visad` package. `BEProcessor` classes have been implemented to support `Gridded3DSet` named `BEGridded3DSet` and `Irregular3DSet` named `BEIrregular3DSet`. It is up to the `BoundaryExtractor` module to select the proper `BEProcessor` based on which `Set` the field is based on.

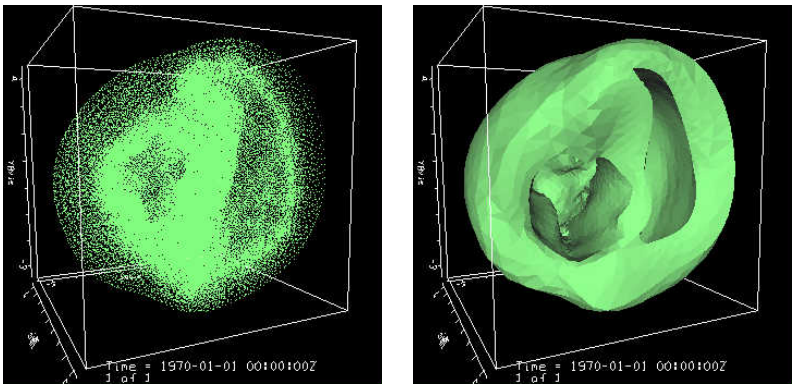
The `BoundaryExtractor` will create one boundary for each input field it receives as input, which means that if it receives overlapping subdomain datasets, the output will be boundaries that intersect.

The BEIrregular3DSet boundary extractor

The boundary of an irregular 3D geometry has in this implementation been assumed to be the surface composed of all triangles that are only part of one tetrahedron. This is a result of that each tetrahedron that has an edge to the interior of the geometry, will have a neighbor tetrahedron with which it shares a triangle. Thus, any triangle that is not part of two tetrahedron must be on the boundary of the geometry.

This information can be extracted by relying on the representation of irregular grids, the `Irregular3DSet` and `Delaunay` classes which is described in Section 2.3.1.

Using this information we can extract the boundary. This is done by searching through the triangles of each tetrahedron that has less than four neighbors. Each such tetrahedra has at least one triangle that is not shared with other tetrahedron



a) b)

Figure 4.6: Visualization of an irregular volumetric dataset. Image a) shows the dataset. Image b) shows the extracted boundary of the dataset.

$p_{top} = p_{bottom}$	=	$n_x n_y$
$p_{left} = p_{right}$	=	$n_y n_z$
$p_{front} = p_{rear}$	=	$n_x n_z$

Table 4.2: The number of points in the surfaces extracted from the boundary of a Gridded3DSet

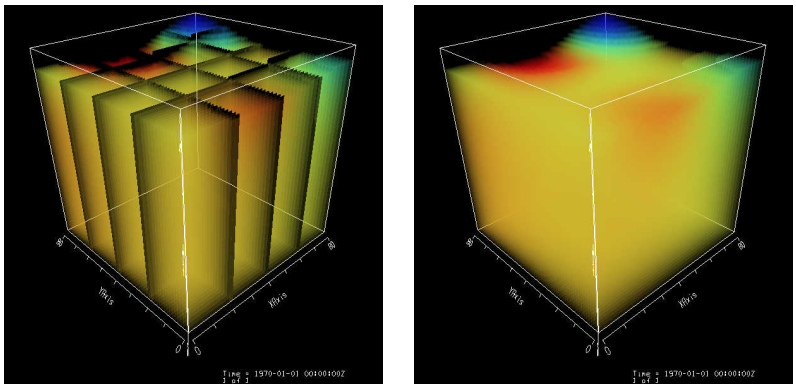
and these triangles are on the boundary. Once the points in all the triangles have been found, we can extract the field values for each point and create the 2D field which represents the boundary.

Images that show the result of boundary extraction are shown in Figure 4.6, where a) shows a dataset as only points, and b) shows the dataset as the boundary.

The BEGridded3DSet boundary extractor

The `Gridded3DSet` defines an ordered set of points as described in Section 2.3.1. The boundary of the a `Gridded3DSet` is composed of six surfaces that represent the top, bottom, front, rear, left and right sides of a uniformed or deformed cube. Using six separate loops, the sample points and field values for each of the surfaces are extracted from the set. The sample points are stored in six separate arrays and form the basis for six `Gridded2DSet` objects. These `Gridded2DSet` objects are joined using a `UnionSet`, described in Section 2.3.1. The field values are stored sequentially in one array in the same order as the sample points. The field values together with the `UnionSet` object composed of the sample points in the six surfaces are then used to create a new field that defines the boundary of the input field.

Given a `Gridded3DSet` object with n_x, n_y and n_z describing the number of points in each direction and p_{orient} describes the number of points in the surface orient, we have three pairs of surfaces where each pair has the same number of sample points, as can be seen in Table 4.2



a) b)

Figure 4.7: Visualization of a volumetric dataset. Image a) visualizes the dataset as multiple subdomains. Image b) visualizes the dataset combined to one domain.

4.4.7 The Combiner module

As stated in Section 2.2.5, Java 3D does not support multiple transparent surfaces stacked behind one another. The result is that polygons will be visible or invisible depending on which way the normal vector of the polygon is facing. When several subdomain datasets are resampled and one wishes to visualize the volume, the transparency issue results in poor images, as can be seen in Figure 4.7 a).

The `Combiner` module has been written as a workaround of this problem. Its purpose is to merge several subdomain datasets into one dataset. This is done by extracting the `Set` object from each of the field and join them in a `UnionSet`, described in Section 2.3.1. The field values are appended in one long array. The `UnionSet` object and the appended field values array are used to form the merged dataset.

Sadly, the `UnionSet` implementation is only partially finished in VisAD which means that not all functionality is present for datasets based on it. Resampling with the weighed average approximation method is the most important feature that is missing. For resampling a merged dataset one has to use the nearest neighbor approximation method.

The results of are shown in Figure 4.7, where the image in a) shows a volume (at its worst) as a resampled parallel dataset where the subdomain datasets are kept separate. The image in b) shows the same parallel dataset where the subdomain datasets have been merged using the combiner module before resampling.

4.4.8 The Serializer module

There are cases where one wish to load a dataset into memory and do some time-consuming processing on it and then display it. If this is done with the same data over and over, one would most likely prefer to save the processed data and load the processed data again and again instead of reprocessing the data each time. The

`Serializer` module was written to do this. It takes an input field and writes it to a file or loads a file and supplies it to other modules. The user can specify the file name through a simple graphical user interface.

The technique for writing and loading files is Java object serialization as described in Section 2.1.4, which provides a simple, yet flexible way of storing generic object structures to file.

4.4.9 Creating additional modules

The PVis system has been written with the intention that extending it with new modules should be easy, given that the user is familiar with the Java programming language.

All modules that are to be a part of the PVis pipeline must be derived from the class `Node`, which handles all the details on the execution of the PVis graph as discussed in Section 4.3.

There are two methods that the module must implement in order to work properly. The first is the `process()` method which is called during the execution of the PVis pipeline to perform the modules task. An example is the `Resampler` module which uses its `process()` method to create a resampled version of its input fields.

The second method that must be implemented in a `Node` is `dispose()`. Although it is not absolutely required that a module implements the `dispose()` method it is highly recommended since not doing so may lead to redundant memory massive objects existing in the PVis graph after they are no longer needed. The goal of the `dispose()` method is to reset all pointers that are not needed between calls to `process()`.

Methods that a module use to receive input must be have a name that starts with `in`. Methods that a module use to expose its processed output must have a name that starts with `out`. This is because the graphical user interface, described in Chapter 5, browses each modules methods and displays all the methods that have names starting with `in` as input methods and all methods that have names starting with `out` as output methods.

If the module is to be configured, then it must also implement the `Configurable` interface, which is described in Section 5.5

Below is listed the code for a simple module that receives an input object and exposes it without doing anything.

```
import pvis.visad.*;

public class PassThrough extends Node {

  private Object obj;

  /** Receives the input and stores it internally */
  public void inObject( Object input ) {
    obj = input;
  }
```

```
  /** Exposes the object to the listeners */
  public Object outObject() {
    return obj;
  }

  /** This module does not do anything */
  public void process() {
    // Do nothing
  }

  /** Release the memory referenced by this object */
  public void dispose() {
    obj = null;
  }
}
```

Chapter 5

The graphical user interface for the PVis visualization system

The previous chapter described the building blocks of PVis' pipeline architecture, and how they fit together. We now specify how this architecture is represented, so that the user can easily take advantage of its flexibility. It is important for user to see the nodes and edges present in the pipeline and that modifying the pipeline is simple. As stated earlier, we have chosen to implement a graphical user interface similar to that of Iris Explorer which has a visual, interactable representation of the pipeline, and that has the ability to open simple user interface components for all the modules that can be configured explicitly by the user. The user is also able to load and save the pipeline each time the PVis system is used.

5.1 Overview

The user interface is implemented as several Java classes, all of which are located in the `pvis.gui` package. The first is the `RootFrame` which defines the main window, as seen in Figure 5.1, and the structure of the user interface. Next is the `PipelineRenderer` which is responsible for the user interaction and visualization of the pipeline. The modules present in the pipeline are drawn using a third component, `NodeRenderer`. The last is the `Configurable` interface which is implemented by all modules that can be configured.

5.2 The RootFrame class

The `RootFrame` class is a Java window, derived from the standard library class `JFrame` in the `javax.swing` package. An object of the `RootFrame` class creates PVis' graphical user interface, which composed of several menu items, used to start the various actions that PVis can do. It also holds a list of all the modules that can be used in the pipeline.

5.2.1 Event management

The `RootFrame` maintains an event manager that registers all actions performed in the menu, the processor list and the start and stop buttons. The event manager is responsible for overseeing that the correct actions are taken, such as adding modules to the pipeline or starting processing of the pipeline.

The event management has been implemented by having the `RootFrame` class implement the `ActionListener`[1] interface. The menu items and buttons register the `RootFrame` object as listener and the `RootFrame` object is thus notified when an action happens. An action can be when the user clicks on a menu item or on a button.

5.2.2 Separate thread for pipeline processing

Pipeline processing is started in a separate thread than that for running the application. This is because the application thread is the virtual machines main event dispatcher, responsible for triggering all events for all windows, including input from keyboard and mouse as well as repaint of windows. It is recommended in [7] that only simple instructions and low cost tasks are done from this thread. Time consuming tasks should be done in separate threads. If the event dispatch thread is

[1]The `java.awt.event.ActionListener` interface defines one method `actionPerformed()` that is used to notify implementing classes that action has occurred.

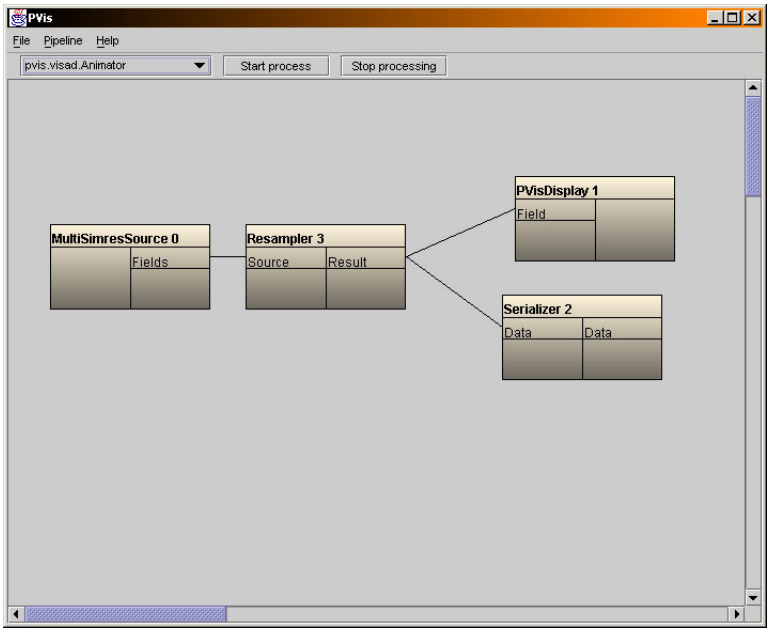


Figure 5.1: Snapshot of the graphical user interface of PVis

held up, the windows will not work properly.

In addition, the user might wish to stop the processing before it is completed, and that can only be achieved if it is running in a separate thread.

5.2.3 Loading available modules

All the modules that can be a part of the PVis pipeline must be explicitly specified for the graphical user interface. This is done in the file `processors.res` in the `pvis/resources` directory. This file contains the class names of all the modules. Dynamic class loading, see Section 2.1.2, is used to load these modules into the user interface.

5.3 The PipelineRenderer class

The `PipielineRenderer` class is responsible for drawing and interacting with the PVis pipeline. It draws each module using an object of the class `NodeRenderer` described in Section 5.4. Edges are drawn as simple lines using functionality in the Java standard library.

Interaction with the user is achieved through several event managers that have been implemented as inner classes in the `PipelineRenderer`. The event managers have been implemented by using the `MouseListener` interface in the `java.awt.event` package. The interface notifies objects of the implementing class on all mouse related events that occur, such as mouse motion and mouse clicks.

5.3.1 The MoverCraft event manager

The `MoverCraft` event manager is responsible for registering when the user drags modules from one location to another. This movement only affects the visual representation of the PVis pipeline, not the order in which it executes.

5.3.2 The ClickHandler event manager

The `ClickHandler` event manager is used to take appropriate action when the user clicks on a filter module. A double click opens a filter modules control widget if such a control widget is defined. Details on this are covered in Section 5.5. If the user clicks on the right button over a filter module, the `ClickHandler` will open a menu for that filter module. The menu can be used to delete the filter module or its edges.

5.3.3 The LinePainter event manager

The `LinePainter` event manager is used to create connections between modules. When the user drags the mouse from an output method of one module to an input method of another module, the `LinePainter` contiounsly draws a line. When the

user relases the mouse, the `LinePainter` creates an `Edge` object between the two modules.

5.4 The NodeRenderer class

The `NodeRenderer` is the simplest part of the user interface. It is derived from the `javax.swing.JPanel` class and is responsible for drawing a single module in the PVis pipeline with its name, input and output methods. It also contains functionality used to resolve which input or output method corresponds to a given point. This is used by the event managers in the `PipelineRenderer` to take appropriate action when the user clicks or drags the mouse over a `NodeRenderer` object.

Drawing is done by overriding the `paint()` method in `javax.swing.JPanel`. This method is called from the windowing toolkit in Java whenever the component needs to be repainted. The method is passed an object of the class `Graphics` from the `java.awt` package which encapsulates the graphics context of the current window. This object has several methods for paint such as `fillRect()`, `drawString()`, etc.

5.5 The Configurable interface

Some nodes can take user input. These nodes implement the `Configurable` interface (pseudo code below), which tells the node to define a simple user interface which can be used to modify and configure the node. It also tells the node to define what kind of data should be saved, so that the nodes which are loaded from file are configured in the same way as when they where saved.

There are no requirements to the user interface supplied by the `Configurable` interface, except that it is a subclass of `java.awt.Component` which is the object that all graphical user interface components in Java are derived from. The user interface is placed in a dialog box by the `ClickHandler` event manager as described above.

The methods `getSaveData()` and `setLoadData()` are used to store and load parameters that are configured in the user interface each time the PVis system is used. These methods operate on an `Object` object, but knowing that all non primitive types in Java are derived from this class, the object can actually be an array of new objects or any other form of Java object. What is located in the object is thus up to the module that implements the interface.

```
interface Configurable {

  Component getUI();
  void endUI();

  Object getSaveData();
  void setLoadData( Object o );
```

```
}
```

The modules that implement the `Configurable` interface and have a graphical user interface component associated with them are listed in Table 5.1.

Module	Graphical user interface component
MultiSimresSource	UIMultiSimresSource
SimresSource	UISimresSource
Resampler	UIResampler
Slicer	UISlicer
Serializer	UISerializer
PVisDisplay	UIPVisDisplay

Table 5.1: The modules implementing the Configurable interface, and their associated graphical user interface implementaiton.

All graphical user interface implementations are located in the `pvis.gui` package. The graphical user interface component `UIPVisDisplay` has been covered in detail in Section 5.6. The other graphical user interface components are trivial and the implementational details concerning them have been omitted.

5.6 The UIPVisDisplay class

The graphical user interface component for the `PVisDisplay` module is the part of the PVis system where the rendering happens. It is basically composed of two dialog boxes. One which contains only the VisAD display module that at all times displays the data and enables the user to interact with the mouse. The other dialog contains a menu with several options for modifying how the data is rendered among other things. These options and their implementational details are covered below.

The menu uses an event manager in the same manner as the `RootFrame` class.

The reason for two separate dialog boxes, one for control and one for display is that the display is a heavyweight[2] Java component and the menu is a lightweight[3] Java component. Heavyweight components are always drawn after lightweight components, thus overlapping heavyweight and lightweight components will not work properly.

5.6.1 Iso contour

This option is used to enable iso contours. Iso contours are made available in VisAD by creating a map from field values to the display type `IsoContour`, as

[2]Peer based, receives a graphic context from the OS and writes directly to it

[3]Draws to an offscreen buffer and flushes the buffer to the graphic context when the OS signals repainting

discussed in Section 2.3.3. This map is set to the VisAD display and a control widget is started to enable the user to change the threshold value of the iso contour.

Enabling or disabling contouring involves the replacement of a map and is subject to the issue on map replacement discussed in Section 2.3.3

5.6.2 3D Texturing

This option toggles the texturing property in the VisAD renderer. This option is enabled by default and is used to enable volume rendering as discussed in Section 2.3.5.

5.6.3 Color table editor

The color table editor option is used to launch a color table editor. The color table editor is a VisAD implementation which is used to modify the color table for the field values in the VisAD display.

The color table is coupled to the data in the display, in such a way that if the data is removed and added again the data will recreate a new default color table. There will be no connection between the old color table and the new data. To modify the new color table after a map change, one must thus open a new color table editor. Note that data has to be removed and added each time a map is changed, so changing another map, such as `IsoContour` described in Section 2.3.3, will reset the color table.

Changing the color table involves the replacement of a map and is subject to the issue on map replacement discussed in Section 2.3.3

5.6.4 Alpha

Alpha is an option which can be used to determine which display type, described in Section 2.3.3, is used to represent field values in the display. If alpha is enabled the display type `RGBA` is used. If alpha is disabled, the display type `RGB` is used. Once alpha is enabled the value can be changed in the color table editor described above.

Changing the alpha parameter involves the replacement of a map and is subject to the issue on map replacement discussed in Section 2.3.3

5.6.5 Polygon mode

There are three options available in the polygon mode menu. These are *filled*, *wireframe* and *points*. These options are provided by VisAD and can be set as the programmer wishes. VisAD relies on the Java3D rasterization mode which is set through the polygon attributes as described in Section 2.2.

5.6.6 Animation

Animation in VisAD is discussed in Section 2.3.3. Animation is controlled through an object of the `AnimationControl` class, which has methods for starting and stopping an animation, stepping forward and backward and setting the animation speed. The options in the menu of the `UIPVisDisplay` are used to trigger these methods.

5.6.7 Snapshot

Snapshot is the process of extracting the raster data in the VisAD display and creating an image from it. This is done by relying on the method `getImage()` which is available in the VisAD display, which returns an object of the `java.awt.Image` class. The image is written to a file using the `JpegEncoder` class located in the `ij.io` package, which is a part of the VisAD distribution.

The snapshot is named `snapshot_x.jpg`, where x starts at 0 and is incremented with one for each snapshot that is taken.

5.6.8 Subdomain dataset filter

The subdomain dataset filter was written to allow the user to enable or disable the viewing of individual subdomain datasets. This is done by creating a list containing all the subdomain datasets that are loaded into the `PVisDisplay` module and create a toggle button for each of them. When a subdomain dataset is disabled it is removed from the display and when it is enabled it is added again.

5.7 Pipeline storage

The PVis pipeline can be stored between each time it is used. The code for pipeline storage is located in the `pvis.io` package. The storage format is a self specified XML [15] format, and the file is read and written using an open source XML parser named Xerces [16]. When the pipeline is stored it is converted to a tree structure suited for XML and written using Xerces. When the pipeline is loaded, the XML file is read using Xerces and converted to a pipeline. An example of a pipeline file is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<pipeline>
 <nodes>
  <node class="pvis.visad.MultiSimresSource" key="2" x="59" y="65">
   <config>
    <array class="[Ljava.lang.Object;" length="4">
     <array class="[I" length="1">
      <object class="java.lang.Integer" value="3"/>
     </array>
     <array class="[I" length="2">
      <object class="java.lang.Integer" value="0"/>
      <object class="java.lang.Integer" value="3"/>
     </array>
     <object class="java.lang.String" value="data"/>
     <object class="java.lang.String" value="WA16"/>
    </array>
   </config>
  </node>
```

```
  <node class="pvis.visad.PVisDisplay" key="1" x="408" y="64">
   <config>
    <null/>
   </config>
  </node>
 </nodes>
 <edges>
  <edge class="pvis.visad.Edge" key="3">
   <incoming key="2" method="outFields"/>
   <outgoing key="1" method="inField"/>
  </edge>
 </edges>
</pipeline>
```

The tree structure is built by the an object of the `OutputHandler` class. This object is responsible creating a root element, named `pipeline` which contains all the nodes and edges in the graph. For each node it creates a new element that contains the class name, key and location. If the node implements the `Configurable` interface a block of save data is written out as well. Save data is written out recursively so although the `getSaveData()` method returns an object this object may be a nested structure. Using Java language reflection, see Section 2.1.3, this nested structure is converted into a subtree of elements.

The edges in the pipeline are converted to elements each containing an element for the module and method to extract data from and an element for the module and method to which the edge should pass the data.

Loading is done by using Xerces to load the XML file into a tree structure. The tree structure is the same as the one created when storing the file, and is decoded by an object of the `InputHandler` class. The nodes are then regenerated based on the class name supplied in the file. Any saved parameters are recreated and converted back into the same nested object structure as was returned by the `getSaveData()` when the pipeline was stored. The structure is passed to each module using the `setLoadData()` method. Edges are recreated using the specified location to extract data and pass data to. As the nodes and edges are created they are inserted into the `Graph` object, and will thus be a part of the PVis pipeline.

Chapter 6

Case studies

This chapter covers case studies that have been done using the PVis visualization system. The goal of these case studies is to measure the performance of the PVis system. The measurements concern processing times, memory usage, smoothness of animation and general usability of the PVis system. The results of the case studies are discussed together with some general issues in the next chapter.

6.1 Overview of the case studies

The specifications for the computer used to run the case studies are listed in Table 6.1.

6.1.1 Measuring time

The processing times have been measured using the Java method `currentTimeMillis()` in the class `java.lang.System`. According to [14] the method returns "the difference, measured in milliseconds, between the current time and midnight, January 1,

Hardware	
Processor	AMD Athlon XP1700+
Memory	512Mb (DDR)
Graphics card	NVidia GeForce3 Ti200, 64Mb
Software	
Platform	Microsoft Windows 2000
Java	Java HotSpot(TM) Client VM (build 1.3.1-24, mixed mode)
Java 3D	Java 3D(tm) 1.2.1_1 SDK (OpenGL) Version)
VisAD	Version 2.0

Table 6.1: Hardware and software specifications used to perform the case studies. The tests were run with the option `-mx500m` to the JVM, which means that the heap size is set to a maximum of 500Mb of memory (default is 64Mb).

1970 UTC". Time is measured by taking the difference in `currentTimeMillis()` between when a task starts and when a task stops. Time is measured in milliseconds.

This method gives rise to an estimation error, since MS Windows 2000 is a multi-tasking environment. This means that the number of milliseconds the task is allowed to actually run on the processor is less than the number of milliseconds that has elapsed since the task started, as a result of other tasks running as well. By running each task multiple times we hope to reduce this error. We will also have only the PVis system running on the computer so that as much processor time as possible can be spent on PVis.

6.1.2 Measuring memory usage

Memory usage has been measured in different ways. One method is to investigate a parallel dataset and estimate how much space it will take up, given the number of points and elements in each subdomain dataset. This estimate is based on the number of points, elements and field values and the number of bytes used to represent each datatype. The object representation of the data will in most cases include additional information such as object variables and tables to aid traversing irregular grids. This means that this estimate is only a lower bound estimate and will only give a pinpoint on the memory usage.

Another way that we use to measure memory usage is to monitor the allocated memory in the JVM. This is done by calling the methods `totalMemory()` and `freeMemory()` in the `Runtime` class in the `java.lang` package. These methods return the total amount of memory allocated by the JVM and the free memory. This is used to measure the memory allocated by each module by sampling the difference between total and free memory before and after a module processes. These methods are also used to extract the total memory that is used by the JVM when the pipeline has completed processing.

A third way of estimating how much memory is used is to serialize the objects involved and check the size of the file, see Section 2.1.4. This means that we can get a very accurate estimate on how much memory a specific object takes. The serialized object will however, contain some metadata information needed to deserialize it but in the case of numerical data, which largely consists of long data arrays, the overhead of metadata should be minimal.

The method `gc()` in the `java.lang.System` class can be used to hint to the garbage collector that it should run, but this method does not guaranty that GC will actually run. It is this method that has been used when we force garbage collection.

6.1.3 Measuring time and memory for the PVisDisplay module

The `PVisDisplay` module differs from the other modules in that one part of it is implemented using VisAD classes and the other part is a PVis module. As described in Section 4.4.2, the `process()` method of the `PVisDisplay` module only adds the

input fields to a VisAD display. The VisAD display will then render the data in a separate thread. Consequently the time and memory measurements obtained upon `PVisDisplay`'s completion of `process()` does only partially include the memory and time spent on rendering.

6.1.4 About the metric items

We will now give a short description of the metric items that have been used in the case studies, what they mean and where the numbers come from.

Estimated memory usage for field values per time step; The estimated amount of memory that the field value array will take for each time step. The field values are represented as an array or `float` values which means that the memory usage will be the number of points times 4 (which is the number of bytes used to represent a float).

Estimated memory usage for points per time step; The estimated amount of memory that the point array will take for each time step. For `LinearSet` this array is not present. For `GriddedSet` and `IrregularSet` this array is an array of `float` values organized as `[dim][numpts]`. For 3D data we then have that the amount of memory used will be 3 times 4 times the number of points.

Estimated memory usage for element representation per time step; The estimated amount of memory that the element representation will take. This value is only present for fields based on `IrregularSet`, since the other set implementations represent elements implicitly. The value is based on the size of the arrays in the `Delaunay` object described in Section 2.3.1.

Estimated memory usage per time step; The sum of the memory usage for field values, points and element representation for one time step.

Estimated total memory usage; The sum of the estimated memory usage for each time step.

6.1.5 Time and memory tables

For each case study we have measured the time and memory used. This has been done three times for each case study. Time and memory use is measured for each module individually, by sampling the values before and after the module processes. This value is measured immediately before and after a module processes so objects allocated during a module's `process()` method may not have had time be garbage collected. It also measures the time and memory used when the PVis pipeline has completed processing. This is done by sampling the values before the PVis pipeline starts and after it completes. Recalling that the graph processes in one thread, the graphical user interface in another and rendering in a third thread, we can assume that these numbers may vary depending on the time spent on the other threads. The final value is the memory use after garbage collection. This value is measured by waiting for the rendering to complete, then force garbage collection through the graphical user interface and finally displaying the memory usage in the JVM.

6.1.6 Description of the response times in the user interface

In addition to the various parameters that are measured and estimated, we will for each case study describe how the user interface component for the `PVisModule` responds to the user interaction. This includes navigating the dataset using the mouse for zoom and rotation, subdomain dataset filtering, switching between standard and wireframe view and updating the color table.

The responses are a description of how the user experiences the user interface. The values used are *immediate* which means that the response from a command is immediately visible in the display, *x seconds delay* where *x* represents the time the user experiences the delay. This can be a range *x to y*. Note that this is not a measured time, but rather a description of how the user experiences the visualization system. The last term *N/A*, means that the option is not available.

For animation the response time is based on the step forward and step backward options available in the user interface for the `PVisDisplay` module.

6.2 The 3D wave simulation

This case study involves a parallel dataset of 16 subdomain datasets created from a 3D wave simulation. The simulation is covered in detail in [27]. The field values in the datasets represent velocity potential, ϕ.

The subdomain datasets have an ordered topology which allows them to be represented through the `Gridded3DSet` class, described in Section 2.3.1. The topology of the parallel dataset is time dependent, which means that the sample points move from time step to time step. The metric items for the parallel dataset are listed in Table 6.2.

Number of subdomains:	16
Number of time steps:	33
Total number of fields:	528
Sample points per field:	Range from 6929 to 8036
Elements per field:	Range from 5760 to 6760
Total number of points per time step:	123.650
Total number of elements per time step:	103.340
Estimated memory usage for field values per time step:	494.600 bytes
Estimated memory usage for points per time step:	1.483.800 bytes
Estimated memory usage per time step:	1.988.400 bytes
Estimated total memory usage:	65.617.200 bytes

Table 6.2: Metric items for the wave simulation.

6.2.1 Measurement of memory usage

The purpose of this case study is to measure the amount of memory that the parallel dataset uses. This has been done by loading all the time steps for all subdomain datasets and serializing them. An illustration of the pipeline used for the test is showed in Figure 6.1. The test is run only once since the size of the serialized object cannot differ from test to test. The results from the test are listed in Table 6.3

When we compare the results in Table 6.3 with the estimated total memory usage in Table 6.2, we see that the values are close. The results are discussed further in Section 7.3.1.

Figure 6.1: Illustration of the pipeline used to measure memory usage

Type of serialized file	Size
16 subdomains and 33 time steps	69.9 Mb

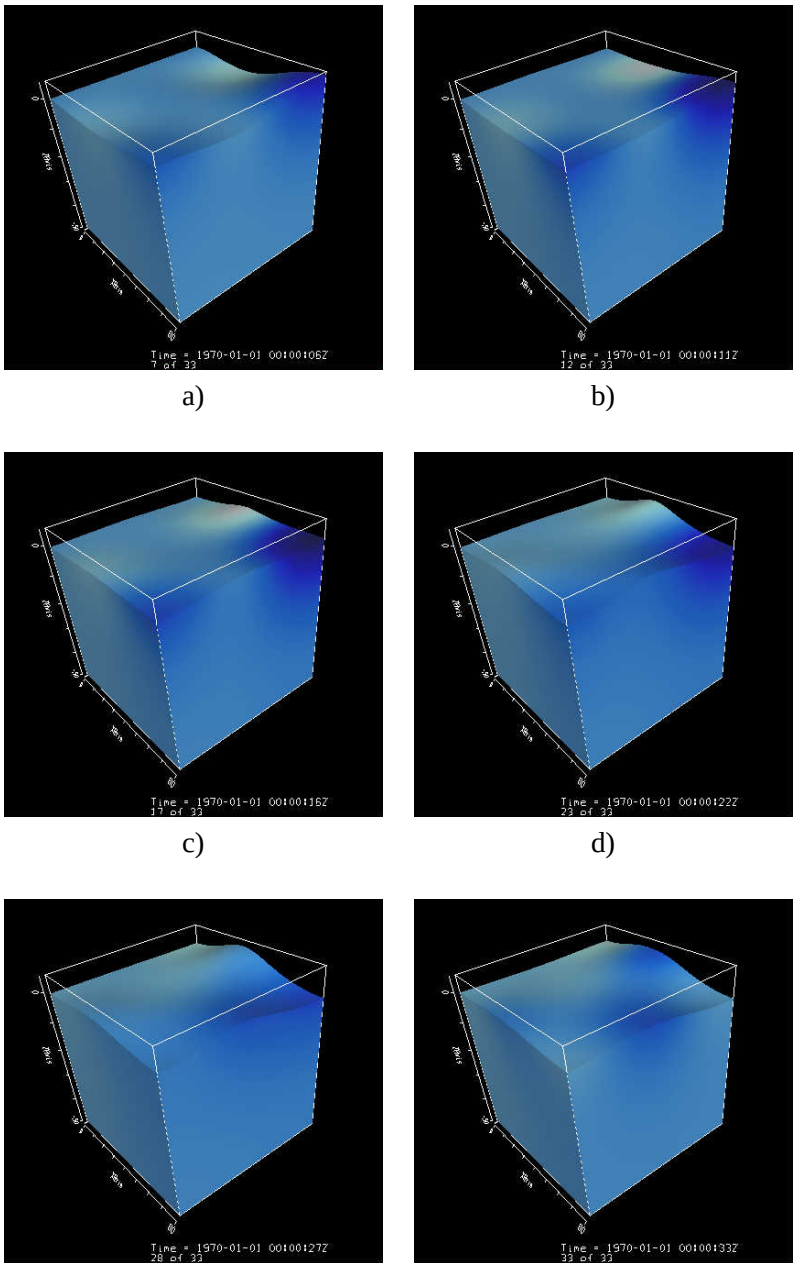
Table 6.3: Results of serializing the parallel dataset from the wave simulation to disk.

6.2.2 Visualization of the exterior

The purpose of this case study is to measure the time and memory used on loading the parallel dataset, filtering out the exterior and visualizing it. The exterior is filtered out using the `BoundaryExtractor` module described in Section 4.4.6. The case study has been divided into three parts. The first part visualizes the first time step only, the second part visualizes every other time step, and the third part studies the result of visualizing all the time steps. This is for checking how time and memory usage increase with the amount of data involved.

Snapshots from the visualization are shown in Figure 6.2. The dark color represent low field values and the bright color represent high field values. Figure 6.2.2 shows the last time step of the simulation, but with four of the subdomain datasets filtered out, using the subdomain dataset filter in the user interface component to the `PVisDisplay` module. This has been done to show how the parallel dataset is divided into subdomains.

Tables 6.4 - 6.10 contain the measurements that are associated with the three parts. An illustration of the pipeline used for the test is showed in Figure 6.4.

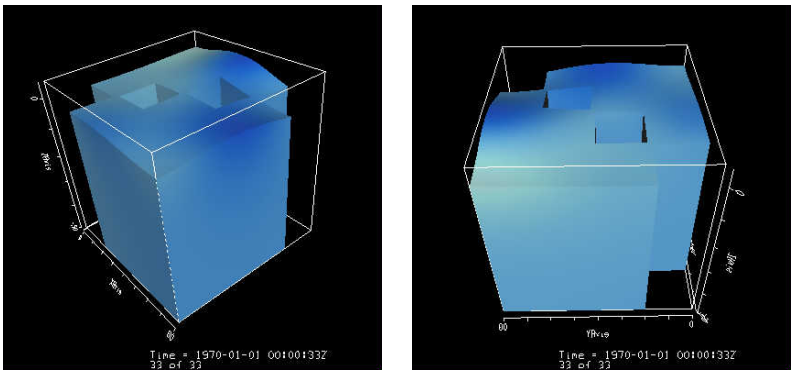


a) b)

c) d)

e) f)

Figure 6.2: Visualization of the exterior of the wave simulation at different time steps.



a) b)

Figure 6.3: Visualization of the exterior of the wave simulation, some subdomains are deliberately made invisible.

Visualization of the exterior of the wave simulation, one time step

We have visualized one time step of the exterior. The results of memory and time usage and are listed in Table 6.4. The results of user interaction is shown in Table 6.5.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	1.5 s	5.0 Mb	1.4 s	5.0 Mb	1.8 s	5.0 Mb
BoundaryExtractor	0.06 s	1.3 Mb	0.04 s	1.3 Mb	0.05 s	1.3 Mb
PVisDisplay	0.1 s	0.5 Mb	0.1 s	0.5 Mb	0.2 s	0.5 Mb
Graph completed	1.7 s	10.4 Mb	1.6 s	10.4 Mb	2.0 s	10.4 Mb
After GC		9.3 Mb		9.3 Mb		9.3 Mb

Table 6.4: Time and memory usage for visualization of the exterior of wave simulation, one time step.

Visualization of the exterior of the wave simulation, every other time step

This visualization is of the exterior of the wave equation and contains every other time step (1, 3, 5, etc), for a total of 17 time steps. The results of memory and time usage are listed in Table 6.6. The results of user interaction are shown in Table 6.7.

MultiSimresSource → BoundaryExtractor → PVisDisplay

Figure 6.4: Pipeline for loading the parallel dataset, filtering out the exterior of the dataset and visualize it.

Interaction type	Response
Iso Contours	Immediate
Color table updates	Immediate
Alpha	Immediate
Turn off subdomain	Immediate
Turn on subdomain	Immediate
Polygon mode	Immediate
Animation	N/A
Navigation	Immediate

Table 6.5: User interaction response for visualization of the exterior of wave equation, one time step.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	14.5 s	58.3 Mb	14.5 s	58.4 Mb	16.3 s	59.2 Mb
BoundaryExtractor	0.7 s	4.5 Mb	0.7 s	4.1 Mb	0.7 s	3.5 Mb
PVisDisplay	0.2 s	6.1 Mb	0.2 s	6.1 Mb	0.2 s	14.9 Mb
Graph completed	15.4 s	72.4 Mb	15.4 s	72.1 Mb	17.3 s	80.1 Mb
After GC		88.8 Mb		88.9 Mb		89.1 Mb

Table 6.6: Time and memory usage for visualization of the exterior of the wave simulation, every other time step.

Interaction type	Response
Iso Contours	2 to 15 seconds delay
Color table updates	Up to 5 seconds delay
Alpha	N/A
Turn off subdomain	Immediate
Turn on subdomain	2 to 5 seconds delay
Polygon mode	3 seconds delay
Animation	Immediate
Navigation	Immediate

Table 6.7: User interaction response for visualization of the exterior of the wave simulation, every other time step.

Visualization of the exterior of the wave simulation, all time steps

This visualization is of the exterior of the wave simulation for all 33 time steps. The results of memory and time usage are listed in Table 6.8. The results of user interaction are shown in Table 6.9.

It was observed that repeated calls to the heavy interaction types (those that took more than a few seconds) when operating on all 33 time steps caused the visualization system to slow down drastically. Knowing that drops in performance

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	34.0 s	95.6 Mb	31.8 s	100.1 Mb	33.3 s	100.1 Mb
BoundaryExtractor	1.0 s	31.0 Mb	1.0 s	30.1 Mb	1.0 s	30.4 Mb
PVisDisplay	0.3 s	12.2 Mb	0.6 s	-87.7 Mb	6.3 s	27.4 Mb
Graph completed	35.3 s	142.3 Mb	33.5 s	46.2 Mb	40.6 s	134.7 Mb
After GC		157.4 Mb		164.0 Mb		193.2 Mb

Table 6.8: Time and memory usage for visualizing the exterior of the wave simulation, all time steps.

Interaction type	Response
Iso Contours	5 to 20 seconds delay
Color table updates	10 to 15 seconds delay
Alpha	N/A
Turn off subdomain	Immediate
Turn on subdomain	5-20 seconds delay
Polygon mode	5-20 seconds delay
Animation	Immediate
Navigation	Immediate

Table 6.9: User interaction response for visualization of the exterior of the wave simulation, all time step

are often related to memory and garbage collection, we did another test where we removed subdomain datasets and added them again using the subdomain filter described in Section 5.6, while continuously monitoring the memory used by the system. The test was done with all subdomain datasets and all time steps loaded. The results are shown in Table 6.10. When we forced garbage collection the first time (350Mb $\rightarrow$ 200Mb), the user interface hung for 10 to 15 seconds, probably because the garbage collection consumed all processing resources on the computer.

The `MultiSimresSource` writes out one line of information for each field it loads from the parallel dataset. When all the 528 fields were loaded, we could observe that there were short delays (less than a second) on regular intervals. These delays occured once for every 10 to 15 field. Suspecting that this was the result of garbage collection, we started the JVM with the option `-verbose:gc` which writes out info each time the garbage collector runs. It could then be observed that each delay was followed by a release of memory as a result of garbage collection.

Summary of the visualization of the exterior of the wave simulation

Visualization of the exterior gives us a good indication on the behavior of the parallel dataset, since the movement of the sample points and field values are visible on the exterior. The displacement of the sample points on the surface of the parallel dataset is seen quite clearly in Figure 6.2. The images in Figure 6.2 also give in-

Action	Memory used	Memory used	Memory used
Initial (16 subdomains visible)	170.9 Mb	171.0 Mb	170.7 Mb
Remove 3 (13 visible)	171.8 Mb	170.6 Mb	171.6 Mb
Add 1 (14 visible)	277.9 Mb	276.4 Mb	275.4 Mb
Add 1 (15 visible)	341.6 Mb	325.3 Mb	332.5 Mb
Add 1 (16 visible)	356.3 Mb	355.8 Mb	354.7 Mb
Force GC	212.1 Mb	204.7 Mb	205.8 Mb
Force GC	178.8 Mb	177.9 Mb	178.7 Mb
Force GC	157.1 Mb	157.1 Mb	157.1 Mb
Force GC	157.2 Mb	157.1 Mb	157.1 Mb

Table 6.10: Results from memory monitoring when filtering subdomain datasets from the wave simulation.

sight into how the field values differ during the animation, since field values differ on the exterior of the parallel dataset, not only the interior.

The time it took to complete the visualization for all subdomain datasets and all time steps, was up to about 40 seconds. Considering the amount of data being processed, such processing times can be concidered acceptable. Once the visualization has been loaded into the VisAD display, navigating it with the mouse can be done without delay. When stepping through an animation, the response times are also immediate, which means that one can start animation and navigate the parallel dataset while the animation is still running.

When we change the maps in the display, described in Section 5.6, the response times drop to as much as 20 seconds and for repeated map changes, the computer begins swapping[1] and the system slows down to a degree where it is no longer usable.

Boundary extraction has an advantage over resampling in that it allows us to visualize the exterior surfaces using the sample points from the input grid, which means in this case, that the surface of the water is visualized using the same sample points from the parallel dataset, so we do not get loss of resolution. To achieve the same effect using resamping, one would have to resample on a very dense grid. Another advantage over resampling is that one can use lighting effects to increase the sense of depth in the visualization.

6.2.3 Volume visualization of the wave simulation

The purpose of this case study is to measure the time and memory spent on loading the parallel dataset, resampling it to a uniform grid and visualizing it using volume rendering. The case study has been divided into three parts. The first part visualizes the first time step only, the second part visualizes every other time step, and the

[1] Swapping is a term used to describe the process of shuffeling data between physical and virtual memory. The process and its concequences are described in Section 7.1.1

third part studies the result of visualizing all the time time steps. The case study has been divided into three parts to check how time and memory usage increases with the amount of data involved.

Snapshots from the visualization are shown in Figure 6.5. The color table used runs from blue for the low field values to white for intermediate values and red for high values. In Figure 6.6 the color table has the same shape, but the range from blue to white to red has been concentrated to the intermediate values. In addition alpha is set to follow the field values so it is transparent for blue and red and opaque for white. This has been done to create a form of contour volume.

The pipeline used to visualize the data is given in Figure 6.7. Notice that the `Combiner` module has been placed before the `Resampler` module. This is to avoid the transparency issue concerning Java 3D as discussed in Section 2.2.5. By combining the subdomain datasets before resampling, the result will be one dataset instead of 16 subdomain datasets. The combined parallel dataset is resampled on a uniform grid of size $64 \cdot 64 \cdot 64$.

The Tables 6.11-6.15 contain the measurements that are associated with the three parts.

Volume visualization of the wave simulation, one time step

This is a visualization of the resampled volume for one time step. The results of time and memory usage are listed in Table 6.11. The results of interaction are listed in Table 6.11.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	1.5 s	5.0 Mb	1.4 s	5.0 Mb	1.6 s	5.1 Mb
Combinder	0.0 s	1.0 Mb	0.0 s	1.0 Mb	0.0 s	0.9 Mb
Resampler	58.4 s	78.7 Mb	60.2 s	78.7 Mb	57.3 s	78.7 Mb
PVisDisplay	0.1 s	4.5 Mb	0.1 s	4.5 Mb	0.1 s	4.5 Mb
Graph completed	60.0 s	92.8 Mb	61.8 s	92.8 Mb	59.1 s	92.8 Mb
After GC		24.8 Mb		25.0 Mb		25.0 Mb

Table 6.11: Time and memory usage for volume visualization of the wave simulation, one time step.

Volume visualization of the wave simulation, every other time step

This is a visualization of the resampled volume for every other time step, for a total of 17 time steps in all. The results of time and memory usage are listed in Table 6.13. The results of interaction are listed in Table 6.14.

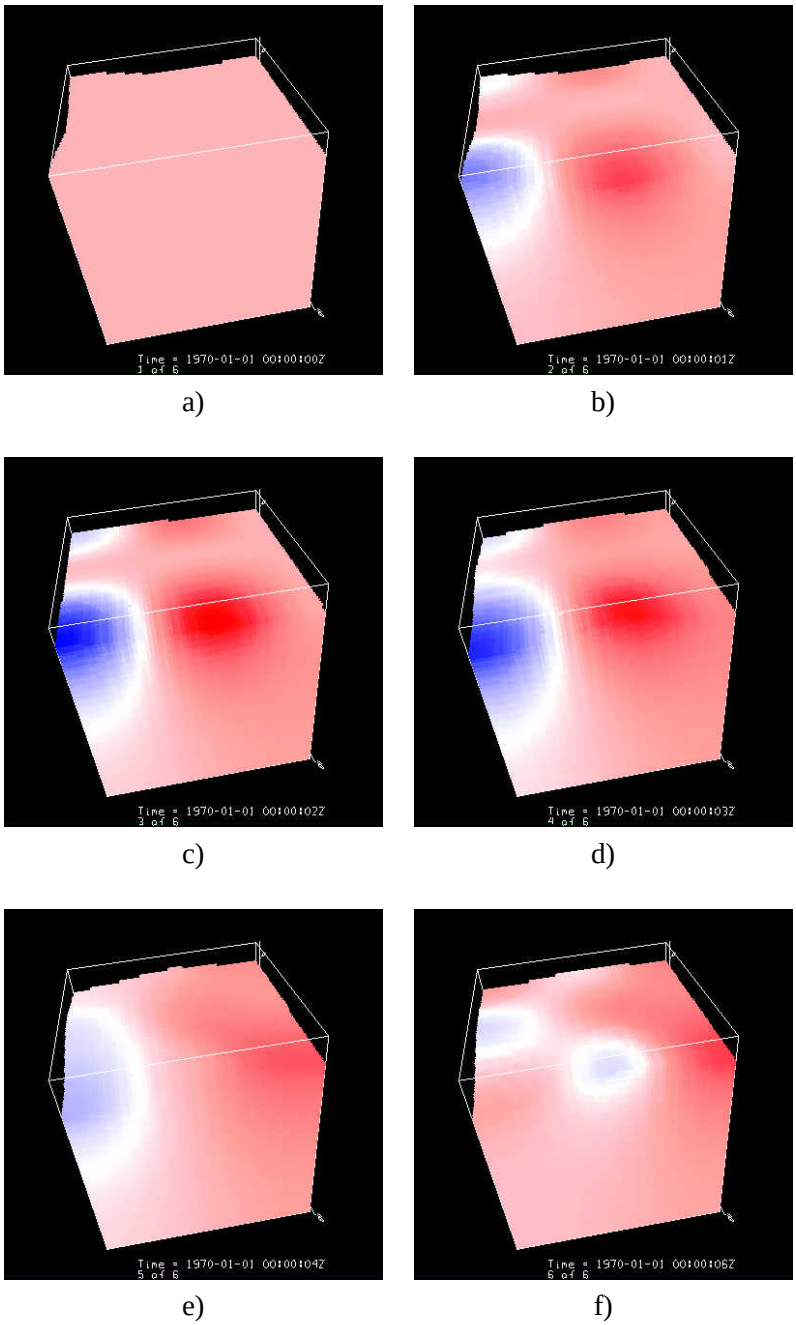


Figure 6.5: Volume visualization of the wave simulation, using a color table ranging from blue to white to red.

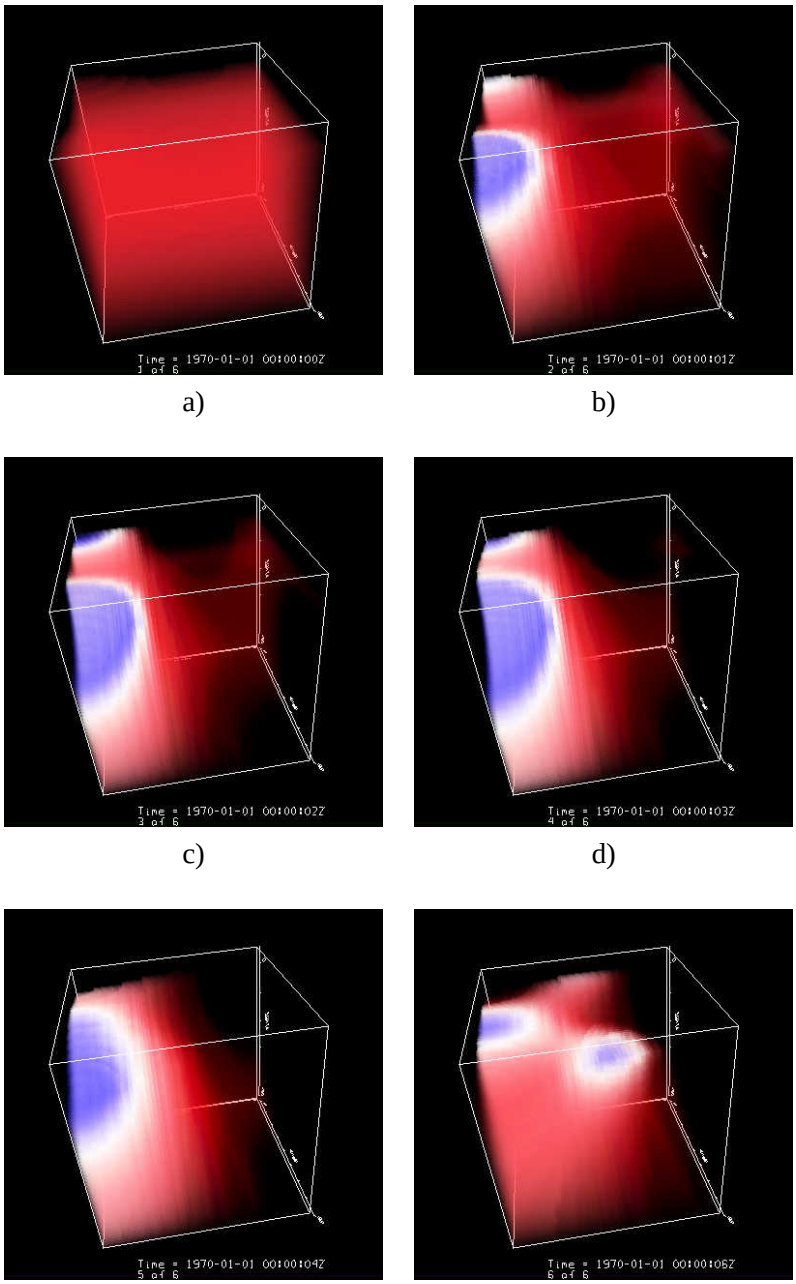


Figure 6.6: Volume visualization of wave simulation, using a color table ranging from transparent blue to opaque white to transparent red.

Interaction type	Response
Iso Contours	1 second delay
Color table updates	1 second delay
Alpha	1 second delay
Turn off subdomain	N/A
Turn on subdomain	N/A
Polygon mode	N/A
Animation	N/A
Navigation	Immediate

Table 6.12: User interaction response for volume visualization of the wave simulation, one time step.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	20.9 s	49.5 Mb	21.9 s	58.6 Mb	22.0 s	49.2 Mb
Combinder	0.2 s	15.1 Mb	0.3 s	39.1 Mb	0.2 s	14.8 Mb
Resampler	1005 s	132.6 Mb	1010 s	128.5 Mb	1001 s	186.8 Mb
PVisDisplay	0.7 s	88.5 Mb	0.7 s	88.2 Mb	1.2 s	-73.6 Mb
Graph completed	1026 s	289.3 Mb	1033 s	284.8 Mb	1025 s	180.8 Mb
After GC		297.0 Mb		297.9 Mb		316.7 Mb

Table 6.13: Time and memory usage for volume visualization of the wave simulation, every other time step.

Interaction type	Response
Iso Contours	More than 60 seconds
Color table updates	More than 60 seconds
Alpha	More than 60 seconds
Turn off subdomain	N/A
Turn on subdomain	N/A
Polygon mode	N/A
Animation	Some delay (see summary)
Navigation	Immediate

Table 6.14: User interaction response for volume visualization of the wave simulation, every other time step.

MultiSimresSource → Combiner → Resampler → PVisDisplay

Figure 6.7: Pipeline for loading the parallel dataset, combining the fields, resampling the combined fields and and visualizing the uniform dataset

Volume visualization of the wave simulation, all time step

This is a visualization of the resampled volume for all 33 time steps. The results of time and memory usage are listed in Table 6.15. It was however, not possible to complete the case study. After the graph had completed processing, during the rendering of the VisAD display, the PVis system crashed with the message out of memory, which means that the JVM has passed beyond the maximum memory limit and can no longer allocate memory for new objects. There is for that reason no listing of interaction results, nor of the memory usage *After GC*, since it is sampled after the VisAD has completed rendering. The tests have still been provided, though unsuccessful, since they also help to describe the behavior of the `Combiner` and `Resampler` module.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	40.3 s	95.8 Mb	40.1 s	107.0 Mb	38.9 s	103.1 Mb
Combinder	0.3 s	32.0 Mb	0.6 s	13.5 Mb	0.6 s	17.6 Mb
Resampler	2077 s	330.3 Mb	2177 s	196.9 Mb	2062 s	326.2 Mb
PVisDisplay	24.4 s	-116.0 Mb	11.7 s	172.3 Mb	10.5 s	-84.1 Mb
Graph completed	2142 s	345.6 Mb	2230 s	493.4 Mb	2112 s	366.4 Mb
After GC		N/A		N/A		N/A

Table 6.15: Time and memory usage for volume visualization of the wave simulation, all time steps.

Volume visualization of the wave simulation, all time steps, without combination

This is a visualization of the resampled volume for all 33 time steps without the use of the `Combiner` module before the `Resampler` module. The visualization is in this case only viewable from one direction as shown in Figure 4.7. The test is done to measure the increase in time created by the `Combiner` module. Each field is resampled on a uniform grid of size: $16 \cdot 16 \cdot 64$. The parallel dataset is divided into $4 \cdot 4 \cdot 1$ subdomain datasets, so the total of sample points will be $64 \cdot 64 \cdot 64$, the same number of sample points as used in the previous case study.

The results of time and memory usage are listed in Table 6.16. We see from the results in Table 6.15 compared with the results in Table 6.16, that resampling takes nearly twice as much time when processing fields based on the `UnionSet` class, created by the `Combiner` module.

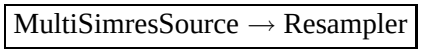


Figure 6.8: Pipeline used to load, resample and visualizing the parallel dataset.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	39.6 s	96.4 Mb	89.5 s	89.5 Mb	36.0 s	110.7 Mb
Resampler	1233 s	100.9 Mb	1229 s	211.5 Mb	1212 s	169.3 Mb
Graph completed	1273 s	201.5 Mb	1319 s	305.4 Mb	1249 s	283.9 Mb
After GC		67.3 Mb		45.5 Mb		48.5 Mb

Table 6.16: Time and memory usage for volume visualization of the wave simulation, all time steps, without combination.

Summary of volume visualization of the wave simulation

The images in Figure 6.5 show the wave simulation using a color table with all field values set to opaque. This image is not unlike that used in the visualization of the exterior in Figure 6.2, save the loss of resolution and depth. The images in Figure 6.6 however, show the wave simulation using a color table where the field values are only opaque for a small area of the color table, while the rest are transparent. By making parts of the volume transparent, we can more clearly see how the field values on the interior of the dataset change.

The processing times for resampling and volume rendering were more than 30 minuttes, as can be seen in Table 6.15, which means that doing the same visualization multiple times can be tedious. Note however, that if the same parallel dataset is to be resampled and visualized multiple times, much time could be saved by resampling the parallel dataset only once, then serialize it and load the serialized parallel dataset later. Loading a serialized dataset usually takes no more than a few seconds.

Volume visualization of the resampled parallel dataset worked acceptable when only one time step was involved. The response times for the second test, every other time step, were much slower for the types that involvled map changes, described in Section 5.6. At this point the computer started to access the hard disk intensively when we tried to change maps. For volume visualization of all time steps, the computer did not even manage to render the data, due to memory exhaustion. This will be furhter discussed in Chapter 7.

The response times for animation listed in Table 6.14, are measured by stepping one time step at a time. If this is done repeatedly with short delays in between we could however observe short delays, and occational hard disk access.

6.3 The 3D heart simulation

This case study involves a parallel dataset of 8 subdomain datasets produced by a 3D simulation of electric activity in the human heart. The simulation is covered in detail in [28]. The field values represent the electrical potential, u.

The subdomain datasets are irregular and use the `Irregular3DSet` class, described in Section 2.3.1, for representing the topology. The metric items for the

parallel dataset are listed in Table 6.17.

Number of subdomain dataset:	8
Number of time steps:	1
Total number of fields:	8
Sample points per field:	Range from 5164 to 5191
Elements per field:	Range from 24649 to 26557
Total number of points per time step:	41.453
Total number of elements per time step:	200.926
Estimated memory usage for field values per time step:	165.812 bytes
Estimated memory usage for points per time step:	497.426 bytes
Estimated memory usage for elements per time step	14.568.096
Estimated total memory usage:	15.231.334

Table 6.17: Metric items for the heart simulation.

6.3.1 Measurement of memory usage

The purpose of this case study is to measure the amount of memory that the parallel dataset uses. This has been done by loading all the time steps for all subdomain datasets and serializing it. An illustration of the pipeline used for the test is shown in Figure 6.1. The test is run only once since the size of the serialized object cannot differ from test to test. The results from the test are listed in Table 6.18.

When we compare the results in Table 6.18 with the estimated total memory usage in Table 6.17, we see that the measured memory usage is larger than the estimate. This is most likely the result of that the parallel dataset is based on an irregular grid. The results are discussed further in Section 7.3.1.

Type of serialized file	Size
8 Subdomain datasets	21.723.514 Mb

Table 6.18: Results of serializing the parallel dataset to disk

6.3.2 Visualization of the exterior of the heart simulation

The purpose of this case study is to measure the time and memory spent on loading the parallel dataset, filter out the exterior using the `BoundaryExtractor` module and visualize it.

Snapshots from the visualization are shown in Figure 6.9. The dark colors represent low field values and the bright colors represent high field values. The images in a) and b) show the parallel dataset with all subdomains, but from different angles. The images in c), d), e) and f) show the parallel dataset from different angles, where one or more subdomain datasets are filtered away using the subdomain filter described in 5.6.

The results from the visualization are listed in Tables 6.19 and 6.20. An illustration of the pipeline used for the test is showed in Figure 6.4.

Module	Time	Memory	Time	Memory	Time	Memory
MultiSimresSource	19.4 s	38.7 Mb	17.0 s	38.7 Mb	19.9 s	38.7 Mb
BoundaryExtractor	0.7 s	-0.1 Mb	0.7 s	-0.1 Mb	0.7 s	-0.1 Mb
PVisDisplay	0.1 s	0.8 Mb	0.1 s	1.2 Mb	0.1 s	0.1 Mb
Graph completed	20.1 s	42.7 Mb	17.8 s	43.2 Mb	20.8 s	43.2 Mb
After GC		42.6 Mb		42.5 Mb		42.9 Mb

Table 6.19: Time and memory usage for visualization of the exterior of the heart simulation.

Interaction type	Response
Iso Contours	Immediate
Color table updates	Immediate
Alpha	N/A
Turn off subdomain	Immediate
Turn on subdomain	Immediate
Polygon mode	Immediate
Animation	N/A
Navigation	Immediate

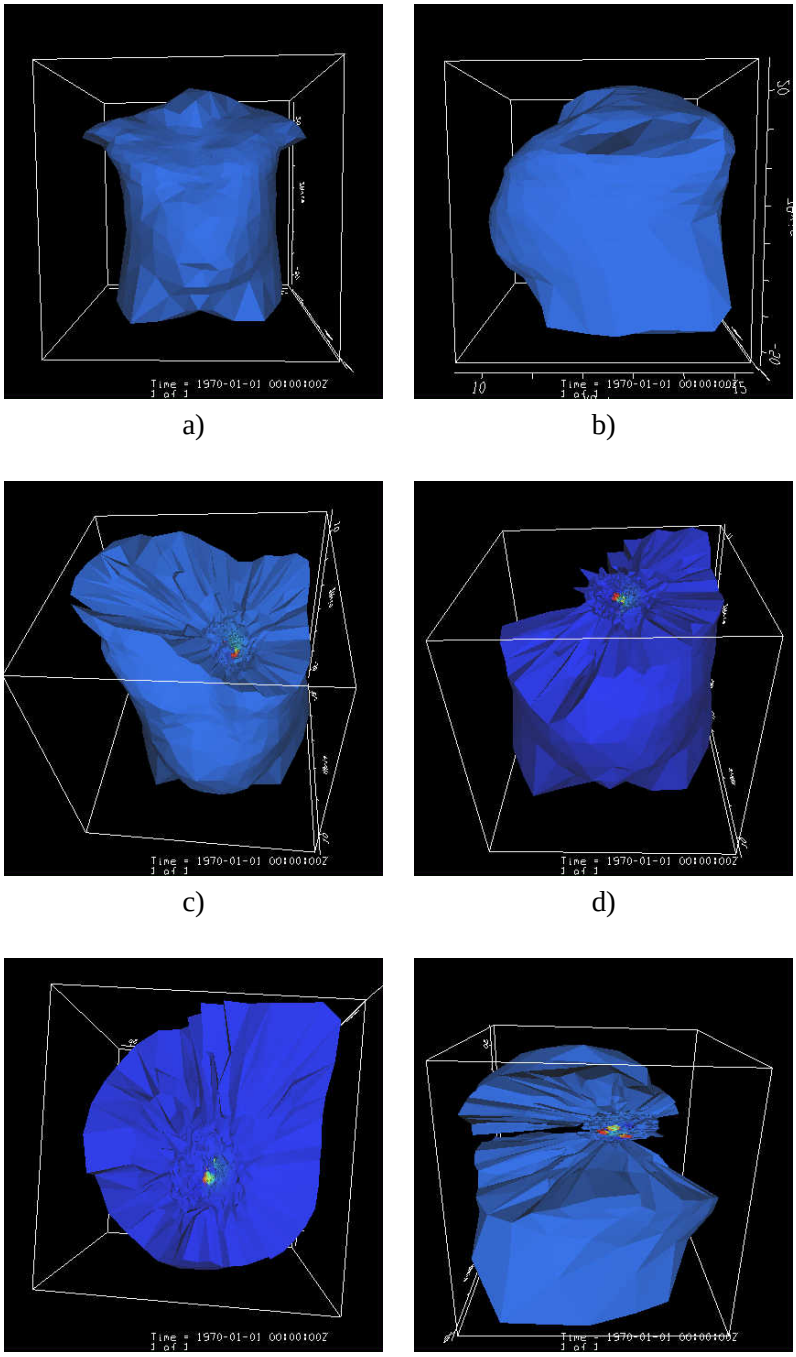
Table 6.20: User interaction response for visualization of the exterior of the heart simulation.

Visualization of the exterior does not give us the same insight as it did in the case study conducted in 6.2.2, since the field values located on the exterior are all the same value. By filtering away one or more of the subdomain datasets however, we can view the interior areas of relevance, as can be seen in the images c) through f) in Figure 6.9.

6.3.3 Comments on volume visualization of the heart simulation

Viewing the 3D heart simulation using volume rendering would most likely increase the information that could be extracted from it.

Resampling datasets based on `Irregular3DSet` however, was experienced to be very slow. In addition to that, the volume has to be combined in the `Combiner` module to avoid the transparency issue discussed in Section 4.4.7. Initial tests showed that resampling the combined dataset on a $64 \cdot 64 \cdot 64$ volume would take several days to complete. This is the result of extra complexity of the `UnionSet` produced from the `Combiner` module combined with the long processing times for resampling datasets based on irregular grids. Resampling on a coarser grid, would not produce an informative volume, so the test was dropped.



a) b)

c) d)

e) f)

Figure 6.9: Visualization of the exterior of the heart simulation. Some subdomain datasets have in images c)-f) been deliberately filtered out.

Chapter 7

Discussions

This chapter contains disucssions on three main topics. In the first section, we discuss the the issues concerning memory and performance that are to be expected from the software that the PVis system is based on, Java, Java 3D and VisAD. In the following section we go in depth of the performance and memory issues concerning each of the PVis modules. And finally we look at the case studies and see how the results measured here match the expected results.

7.1 The underlying software and tools

In this section we discuss the known factors of the software, that the PVis system is based on, where memory and performance are concerned. We start by looking at the Java runtime environment. Following that we discuss issues concerning Java 3D and VisAD.

7.1.1 The Java runtime environment

Array management in Java

As stated in Section 2.1.10, Java arrays are represented as objects, and multi-dimensional arrays are represented as recursive structures of arrays. There are three ways to represent multi dimensional arrays, and these are given in Table 7.1. Typically, the number of dimensions (N_{dims}) will be 1, 2 or 3, while the number of points (N_{pts}) can range from a few points up to millions of points for large data-sets. If we use method a) from Table 7.1, we get one array object that is used to represent all the data. The method described in b) uses one array object for each dimension, but since $N_{dims} \ll N_{pts}$ this method is not measurably worse in terms of memory usage than method a). Method c) however, will create one array object for each point in the dataset. Given that N_{dims} is small and N_{pts} is large, the result is a large amount of small array objects. Tests indicates that method c) uses 40% more memory than methods a) and b).

	Representation method	Result
a)	$[N_{dims} \cdot N_{pts}]$	Results in one array of length $N_{dims} \cdot N_{pts}$
b)	$[N_{dims}][N_{pts}]$	Results in N_{dims} arrays of length N_{pts}
c)	$[N_{pts}][N_{dims}]$	Results in N_{pts} arrays of length N_{dims}

Table 7.1: Representation of multi dimensional arrays in Java

Overhead of the Java Virtual Machine

As discussed in Section 2.1, Java programs execute within a Java virtual machine. The JVM executes programs by interpreting the bytecodes of the Java classes that make up the program. The bytecode of each individual class file is cached in memory to achieve better performance. In addition, some of the bytecode is at all times being compiled to native code (JIT technology described in Section 2.1.9). A class file rarely has a size of more than a few kilobytes, but many classes are required for the JVM and its runtime environment to run so it is in total a size to be considered. As a result of this, it is commonly observed that a trivial Java application without a graphical user interface will require a minimum of 4 megabytes of memory. If graphical user interfaces are used, this value increases to 10 to 15 megabytes easily.

Garbage collection

As mentioned in Section 2.1.8, Java uses a garbage collector (GC). The GC is a separate thread inside the JVM that is not accessible through the functionality of the standard library. It recognizes objects that are no longer referenced by the application and releases the memory allocated by these objects for reuse. Since the GC runs separately, it is not possible to signal the GC to immediately deallocate objects that are no longer in use.

Consider a loop that creates a new object as the first instruction. When the loop reaches the last instruction and starts over, the allocated object will be dereferenced. The second time the loop runs, the object allocated in the first run may not yet have been garbage collected and more memory has to be allocated when the second object is created, despite that the second object could have taken the same memory space as the first object, had the first been garbage collected.

One solution is to tune the maximum heap size parameter to the JVM (through the `-mx` option to the JVM) that sets the maximum amount of memory the JVM can allocate. Setting this value to high, will cause the use of virtual memory, a topic that is further discussed below. If the heap size is set close to the estimated memory limits, it is possible that unreferenced objects are not deallocated before new objects are created, which would mean that the application will run out of memory and not function properly.

This makes it very difficult to write applications in Java that fit within a desired memory limit. The general solution to this problem is to avoid object allocation as

much as possible and recycle objects instead, an approach that is described further in [26].

The topic of garbage collection will be further discussed in Section 7.3.2 where we study the results of a case study concerning garbage collection conducted in the previous chapter.

Physical and virtual memory

When an application is allocating memory in Java, there is no measurable border where physical memory ends and virtual memory begins. Virtual memory is based on storing parts of the physical memory which is not currently being used to disk and reload it into memory when it is needed again, a process often referred to as swapping.

The lack of a measurable border between physical and virtual memory, allows an application to transparently allocate memory far beyond the capacity of physical memory. Since read and write operations are much slower for hard disks than for memory, relying on virtual memory causes an application to slow down drastically.

It may be that even though the maximum memory set by the `-mx` parameter to the JVM is below the memory present on the computer, other processes such as the operating system, require parts of the physical memory for the computer to function properly. The actual physical memory available to an application is thus lower than the total amount of memory installed.

The effect of swapping can clearly be seen in the Table 6.14, where the use of virtual memory has caused the interaction times to increase drastically.

Interpreted code

As stated in Section 2.1.9, Java applications are based on interpreting bytecode, rather and executing machine code. This involves translating each virtual operation to one or more physical CPU operations, which results in slower execution. Several methods have been adopted to address this, such as just in time compilation (JIT) and dynamic optimization, and have proved to greatly the performance of applications written in Java.

The difference in performance from a statically compiled, such as C++, language to a dynamically compiled language, such as Java, can only be measured by a large scale application written identically in the two languages. The application has to be large scale for JIT compilation and dynamic optimazion to have any real effect. The difference could have been measured by rewriting the PVis application in the C++ programming language, but such work is beyond the scope of this thesis. We can therefore only state that runtime compilation will most likely slow down performance compared to the same code written in C++, but the actual amount of slowdown is not addressed.

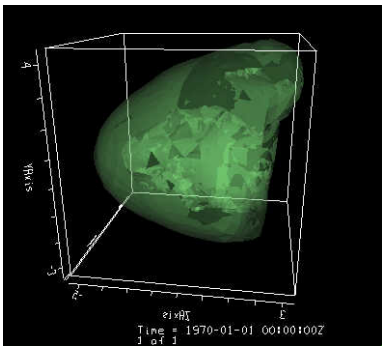


Figure 7.1: Visualization of a transparent surface, the strange result is due to the lack of depth sorting in Java 3D.

7.1.2 The Java 3D graphics library

The lack of depth sorting reduce the quality for several visualization techniques such as transparent surfaces and volume rendering. For transparent surfaces the result is a discontinuous surface which cannot be used to extract information. Figure 7.1 shows a transparent surface and how it is affected by the lack of depth sorting.

Volume visualization, as it is implemented in VisAD, is based on creating one set of slice planes for each viewing direction. Normally this could have been done by creating three sets of slice planes, but since Java 3D lacks depth sorting such a set of slice planes would only be visibly correct when viewed from one side. In VisAD this, has been solved by creating six sets of slice planes, one for each viewing direction.

When visualizing a parallel dataset composed of several fields, the lack of depth sorting results in faulty visualizations as can be seen in Figure 4.7.

Some visualization techniques can be made possible without loss of quality through the use of workarounds such as with volume rendering, but workarounds are often more time and memory consuming than straight forward methods and should in general be avoided if possible.

7.1.3 The VisAD visualization library

Irregular datasets

The `IrregularSet` class, that is described in Section 2.3.1 used to represent irregular datasets can only have triangles or tetrahedra as elements. For some datasets, such as an irregular box based grid that does not fit into the ordered topology of the `GriddedSet`, the elements have to be decomposed into tetrahedra. This means that each box has to be divided into six tetrahedron. Although the element size will be reduced from eight points to four points, we get an increase in the number of elements by a factor of six, which will result in more memory being used.

Member array	Array dimensions	Purpose
`Tri`	$[N_{elms}][S_{elm}]$	Describe the points in each element
`Vertices`	$[N_{pts}][V]$	Describe elements which share points
`Walk`	$[N_{elms}][S_{elm}]$	Describe elements which share sides
`Edges`	$[N_{elms}][N_{edges}]$	Describe elements which share lines

N_{elms}	The number of elements
N_{pts}	The number of points
N_{edges}	The number of line segments in an element (3 or 6)
S_{elm}	The number of points in each element (3 or 4)
V	The size is varying from index to index in the array

Table 7.2: The members of the `Delaunay` object used to represent the elements in an IrregularSet

The Delaunay class

As stated in Section 2.3.1, the `IrregularSet` class, which is used to represent irregular datasets in VisAD, uses an object of the `Delaunay` class to represent the elements in the topology. The `Delaunay` object is composed of four two dimensional arrays used to represent information about the elements. These arrays and their organization and purpose are listed in Table 7.2. Some of these arrays are represented using the memory inefficient method c) described in Section 7.1.

For the `Tri` array we have that $S_{elm} \ll N_{elms}$. Since VisAD does not support varying element types in an `IrregularSet`, the element size, S_{elm} is constant. The array could thus be represented using either method a) or method b) from Table 7.1. The same applies to the `Walk` array. This is because each index in the `Walk` array is of length S_{elm} even if it does not have a complete set of neighbors, (see Section 4.4.6 for further details on neighbors), in which case it is filled with negative values. The `Edges` array also falls into this category since we have that $N_{edges} \ll N_{elms}$. Similar to the `Walk` array, it is filled with negative values if the neighboring edge set is incomplete.

The VisAD renderer

The VisAD renderer is based on maps from metadata descriptions of the field to different rendering modes, as described in Sections 2.3.2 and 2.3.3. It is also stated there that maps cannot be changed when there are fields in the display. To change the set of maps in a renderer, one must therefore remove all fields from the renderer and add the fields to the renderer again after the set of maps have been changed. VisAD is based on Java 3D version 1.1, which does not allow reference to geometric primitives, which means that to change an object in the scene graph, it has to be completely recreated. This means that changing a map results in that all the Java 3D based geometry information such as polygons and attributes has to be recreated based on the new set of mappings. This will have significant impact on both

memory usage and rendering time.

By using version 1.2 of Java 3D and a mutable scene graph this impact could have been reduced significantly. This is because version 1.2 of Java 3D supports referenced geometry arrays, which means that one can update individual parts of a visual object, such as its color table or polygon attributes, without replacing the entire object.

7.2 The PVis modules

This section discusses the performance and the memory usage expected from each of the PVis modules. This is done by inspecting the code for each of the PVis modules and discuss how they will use memory and how they will perform.

7.2.1 The MultiSimresSource and SimresSource modules

The `SimresSource` and the `MultiSimresSource` are covered as one module, since they rely on the same implementation, as described in Section 4.4.1.

The class `DPInputStream` read ASCII based input using temporary objects. This is true for all types except integers, which are read and generated character by character. Each field and grid located in the simres databases, include a text based header that contain information about the data. This means that for all grids and fields that are loaded there are several created several objects. These objects have no value once the field has been loaded, and are discarded, resulting in garbage collection.

In addition to the objects generated by the `DPInputStream`, there are also data structures like arrays and lists that are created when data are loaded, used to organize the data while it is being loaded. These data structures are also temporary and are discarded when a field has been loaded. There is however, no object generation on the loops that are used to read the binary data arrays, points, elements and field values.

To give an example of the amount of temporal objects that are generated, consider loading an irregular dataset from a binary file. The `GridReader` would in this case create 11 temporary objects and the `FieldReader` would create 12 temporary object. For the wave simulation in the case study in Section 6.2 which is composed of 528 fields and grids, this would lead to a total of 12.144 temporary objects.

When reading text based simres files, there should be a lot of temporary objects created to read the float values, which would result in many short lived objects that need to garbage collected. This would slow down the reading process drastically.

For complete details, one would have to read the source code for the simres reader, which is located in the `pvis.diffpack` package.

There are no particular algorithms involved in the processing of the `SimresSource` and the `MultiSimresSource` modules. The time spent during the processing of these modules spent in most part on loading the data arrays, which is considered to

be of linear complexity $\mathcal{O}(n)$ relative to the number of field values, sameple points and elements in the dataset.

7.2.2 The PVisDisplay module

The `PVisDisplay` module, described in 4.4.2, is used to render and display datasets in the PVis system. Recalling that the `PVisDisplay` module is only responsible for passing the input fields into the VisAD renderer, it should follow that the `process()` method of the `PVisDisplay` does not need to allocate any objects, nor should it take any considerable time before it completes.

7.2.3 The BoundaryExtractor module

The `BoundaryExtractor` module, described in Section 4.4.6, uses different processor subclasses to extract the boundary from different grid types.

The simplest processor class `BEGridded3DSet` is based on extracting the six surfaces forming the exterior of the grid. The number of points in each surface are listed in Table 4.2, and can be used to predetermine the number of points in each of the six fields that are created. The only memory that is allocated during the processing of the `BEGridded3DSet` is the data arrays used to describe the sample points and field values for the output field. The complexity of the algorithm is $\mathcal{O}(N)$ where N is the sum of the points in all the six surfaces.

The processor class `BEIrregular3DSet` is a more complex since the number of triangles on the boundary cannot be predetermined and since the sample points on the boundary has to be found through searching, not accessed directly. Each triangle that is found to be a part of the boundary is placed as three indices in an array. Since the number of triangles are not pre determinable, the array will dynamically expand, should the number of triangles surpass the arrays capacity. Increasing an array's capacity is done by allocating a new larger array, and copy the contents of the old into the new array. Array copying will result in that the old array is discarded and has to be garbage collected.

The complexity of the `BEIrregular3DSet` is $\mathcal{O}(N_{tri})$, where N_{tri} is the total number of triangles in the grid. N_{tri} is $4 \cdot N_{elms}$, the number of elements in the `Delaunay` object used to represent the irregular grid.

7.2.4 The Resampler module

The `Resampler` module is, as described in Section 4.4.3, implemented using VisAD functionality, which means that we have little control over the amount of memory it uses or the time it spends processing, but we can make a few assumptions.

The memory required to perform the resampling should in general not be more than the data arrays from the input dataset and output dataset. It may be however, that the resample method uses temporary objects to achieve better performance, but such details are unknown.

Since the `resample()` method is implemented specifically for each `Set` implementation, the complexity of the resample method is dependent on the topology of the data it resamples. In general the complexity will be $\mathcal{O}(N_{elms} \cdot N_{pts})$, where N_{elms} is the number of elements in the field that is to be resampled and N_{pts} is the number of sample points to resample.

For `Set` implementations where the elements are implicitly defined, such as the `LinearSet` described in Section 2.3.1, location of each cell is of complexity $\mathcal{O}(1)$ which will result in the complexity of $\mathcal{O}(N_{pts})$ for the resample method.

For the `UnionSet` implementation, which is a combination of multiple `Set` objects, the number of elements, N_{elms}, is the sum of elements for all sets.

7.2.5 The MyResampler module

The `MyResampler` module, described in 4.4.4, is in terms of memory and performance very similar to the `Resampler` module discussed in above. The difference is that the code is known, and the assumptions made above can be confirmed.

No temporary objects are used during the processing of the `MyResampler` module, which means that the only memory that is being allocated is the array used to represent the field values of the output dataset.

The complexity of the `MyResampler` module will be $\mathcal{O}(N_{elms} \cdot N_{pts})$, same as for the `Resampler` module, but the optimization that has been added, described in detail in Section 4.4.4, should result in processing times that are better than what the $\mathcal{O}$ estimate should indicate.

Note that if the sample points are so dense that the a sample point will always fit into the same element as the last sample point or a neighboring element of the last sample point, searching through the elements will not be needed and the complexity of the algorithm will be $\mathcal{O}(N_{pts})$.

7.2.6 The Slicer module

The `Slicer` module, described in Section 4.4.5, is implemented based on resampling functionality already present in VisAD, same as the `Resampler` module described above, we cannot with certainty determining the complexity of the algorithm involved, nor the amount of temporary objects are used. We can however make the same assumptions as above and state that the complexity of the algorithm is $\mathcal{O}(N_{elms} \cdot N_{pts})$, where N_{elms} is the number of elements in the input field and N_{pts} is the number of sample points in grid used for resampling.

In addition to calling up the functionality of VisAD, the `Slicer` module generates the grid of the plane it resamples on, which has complexity relative to the number of sample points in the generated slice plane, $\mathcal{O}(N_{pts})$.

There is no allocation of temporary objects during the processing of the `Slicer` module.

7.2.7 The Combiner module

The `Combiner` module, described in Section 4.4.7, is used to merge subdomain datasets into a single dataset.

The `process()` method of the `Combiner` module extracts the grid and field values for each of the input fields. The grids are joined in a `UnionSet` and the field values are joined in one array. The output field is created from the combined field value array and the grids combined in the `UnionSet`. There are no temporary objects created, and the only memory allocated is the combined field value array and the `UnionSet` object. The `UnionSet` is based already existing objects so memory increase will in its case be minimal, which means that the only relevant memory is the combined field value array. The complexity of the algorithm is $\mathcal{O}(N_{subdomains})$, where $N_{subdomains}$ is the number of subdomains.

7.2.8 The Serializer module

The `Serializer` module, described in Section 4.4.8, is used to read and write serialized objects such as fields in the PVis pipeline. It relies on functionality in Java to enable this, so we can not say for certain what the memory usage and complexity will be, but it is not likely that large amount of temporary objects are created during the serialization process, since its only the content of the existing objects that are involved.

We can also assume that the serialization process is a recursive routine that traverses the object structure one object at a time, and that objects are written out only once. The time spent on processing will thus be relative to the number of objects in the object structure and their size, since larger objects will take more time to read or write.

7.3 The case studies

7.3.1 The time and memory measurements

In this section we discuss what can be determined by the results in the time and memory tables for each of the case studies conducted in Chapter 6.

Memory estimates compared with serialized objects

Serialized objects are used in the case studies to measure how well our memory estimates of the fields correspond to the amount of memory the field objects actually use as described in Section 6.1.2.

The 3D wave simulation described in Section 6.2 is represented using a `Gridded-3DSet` as grid. This class has an array containing all the points, with the elements represented implicitly based on the organization of the point array. Based on these facts, we estimated the size of all the fields in the dataset to be 65.6 Mb, as shown in Table 6.2. The size of the serialized dataset is 69.9 Mb as shown in Table 6.3. The

serialized dataset is 6.5% larger than our estimate. This is not bad considering that the serialized object contains a little information, such as which classes are being used. In addition to the information, we must consider the other members of the field or grid objects such as the `MathType` objects used to describe the fields and grids.

The heart simulation described in Section 6.3 is represented using a `Irregular 3DSet`, which uses a point array to represent the points and a `Delaunay` object to represent its elements. Based on these facts, we estimated all the fields in the parallel dataset to be 15.2 Mb, as shown in Table 6.17. The size of the serialized dataset is 21.7 Mb as shown in Table 6.18. The serialized object is in this case 42.8% larger than the estimate. As with the `Gridded3DSet` described above, we have an overhead of class information in the serialized file and other members of the field and grid objects. In addition to these two factors, the `Irregular3DSet` represents elements using a `Delaunay` object and we saw in Section 7.1.3 that the organization of some of the arrays were of the memory inefficient sort, type c) from Table 7.1.

Time usage

It is clearly seen from Tables 6.4 and 6.19 that the performance depend on the grid used to represent the dataset. The dataset from the wave simulation contains about 3 times as many points for one time step than the dataset from the heart simulation. Despite this, the dataset from the wave simulation is loaded several times faster.

This is partially the result that the module must read the element array for the heart dataset as well. In addition to that the elements in the irregular grid has to be loaded, the `Delaunay` object, used to represent the elements in a `Irregular3DSet`, is only given the array `Tri` used to describe which points make up the elements. The other three element arrays, described in Table 7.2, must be computed before the `Delaunay` object is valid. The computation is done internally by the `Delaunay` object.

The `Resampler` module had the longest processing times in the case studies. This is the result of that it has complexity $\mathcal{O}(N_{elms} \cdot N_{pts})$. It is not surprising that resampling the heart dataset is more time consuming considering that the heart dataset has almost twice as many elements than the wave dataset, as can be seen from Tables 6.2 and 6.17. In addition to that, the heart dataset is represented using `Irregular3DSet` class which makes searching through elements slow, compared with the wave dataset which is represented using the `Gridded3DSet` where the elements are implicitly defined, and searching through them is faster.

We could see in Tables 6.8 and 6.13 that the processing times for the `PVisDisplay` could vary from almost immediate completion, as seen in e.g. the first listing in Table 6.8, to several second, as seen in e.g. the last listing in Table 6.8. This is the result of that the module passes the input fields to the VisAD renderer module, which starts processing the fields into geometric primitives in a separate thread. The rendering thread is prioritized and delays the thread executing the

PVis pipieline for several seconds.

Memory usage

It could be seen in Table 6.8 that memory usage of the `MultiSimresSource` is a bit larger than what is estimated in Table 6.2 and what is the result of the serialzied file shown in Table 6.3. This is most likely the result of temporary objects that have not yet been deallocated. The same results can be observed for the dataset from the heart simulation. The estimate is given in Table 6.17, the size of the serialized dataset in Table 6.18 and the memory used in reality in Table 6.19. The difference for the heart dataset is larger, but that is most likely the result of more temporary objects being generated during the completion of the `Irregular3DSet` and the `Delaunay` object.

Some modules, such as the `PVisDisplay` module in the second test shown in Table 6.8, appear with negative memory usage. This is the result of that garbage collection has been performed during the processing of the module, and since the memory measurement is based on the usage in the JVM and not for a specific thread independently, this number will be affected by garbage collection. In the case of negative memory usage, more memory has been deallocated from previous modules than the module itself allocated.

7.3.2 Garbage collection

Expectations of garbage collection are discussed in Section 7.1.1. We now see how these expectations fit together with the measurements obtained in the case studies.

In the summary of Section 6.2.2 we reported that during the processing of the `MultiSimresSource` module, there were occasional short time delays. By starting the JVM with the option `-verbose:gc` we learned that the delays were caused by garbage collection. These delays were short, but that is most likely the result of that the temporary objects created in the module are relatively few and small. Had these delays occured while an application was performing a time critical process, such as displaying video at 30 frames per second, the delay of garbage collection would cause the frame rate to become unsteady.

The drastic slowdown seen in Table 6.13 compared with Table 6.11, is a result of that the JVM allocated memory beyond the limits of physical memory and allocated virtual memory instead, which is significantly slower. The reason that the JVM allocated this much is that the JVM was set up with a heap size of 500 Mb, and that less than 500 Mb of physical memory is available. The windows task manager[1] reported that Windows 2000 with its applications and services use over 120 Mb of memory when the computer is not doing anything. Recalling that the computer that ran the case studies had 512 Mb of physical memory, this means that the border between virtual memory and physical memory is, for the JVM and the PVis system, less than 400 Mb.

[1] A monitoring application much like `top` in UNIX

We could try to minimize the heap size, to reduce the chance of the JVM allocating virtual memory, but this could result in that the JVM ran out of memory, since it did not have time to garbage collect dereferenced objects before new objects were created.

A surprise from the results listed in Table 6.10 is that we have to force garbage collection 3 times before the allocated memory stabilizes. It was expected that one pass of the garbage collector recognized the unreferenced objects and deallocated them. This could be the result of massive nested object structures, where the garbage collector is unable to determine if an object is dereferenced or not on the first pass. On the second pass, parts of the nested object structure has been deallocated, revealing other unreferenced objects that can be collected. Thus one part of the nested object structure is collected for each pass.

Another reason that could cause this is that the garbage collector is limited to only collecting a certain amount of unreferenced objects on each pass. Recall from the tests associated with the results in Table 6.10 that the JVM did nothing other than garbage collection for up to 15 seconds. To avoid even longer delays, it limits the amount of objects to be deallocated so that other threads are allowed to run as well.

7.3.3 Interaction with the PVis system

Fast, but limited animation

As can be seen from the tables concerning interaction in Chapter 6, animation is smooth for all datasets that are successfully loaded. This is without doubt a result from VisAD's approach to animation which is based on loading all time steps of a simulation into memory at once. The reason that this results in faster animation is that all fields for all time steps are loaded, filtered and processed into geometric primitives, before any rendering starts.

If the parallel dataset is too large, it will not fit into memory when one tries to load all time steps for all subdomain datasets together. The result was seen in Section 6.2.3 where the application crashed due to memory exhaustion when we tried to visualize all time steps using volume rendering.

The alternative is to iterate through the animation and load, filter and process one time step at a time into geometric primitives. This method would be more time consuming for each time step, but would enable visualization of animations that range over many time steps. The PVis graph however, does not support this kind of iteration.

Change of display maps

Consider the Tables 6.12 and 6.14. It is interesting to see that even though only one time step is visible in the display, the time it took to change display maps, such as the color table, increased significantly from Table 6.12 to Table 6.14. This is a direct result of the way animation is handled. When a map is changed, the slice

planes used for volume rendering has to be recreated, not only for the time step that is currently visible, but for all the time steps in the renderer.

This increase in allocated memory from Table 6.10 indicate that whenever fields are removed from the display and added again, they are completely discarded and totally new objects are created to replace them. Recalling from Section 2.3.3 that changes to the set of display maps must take place when no fields are in the renderer, change of display maps result in that all fields in the renderer are removed, added and reprocessed into geometric primitives again.

The vast amount of memory involved when fields are discarded and regenerated will eventually cause the JVM to run out of physical memory and the JVM will use virtual memory in stead. This is what happened in Table 6.14 when the response times were over 60 seconds, which exceeds linear times compared to the results from Table 6.12, where each interaction type was delayed only for a second.

Navigating the visualization

Navigation the dataset, by means of rotation, zoom and pan, was observed to have immediate response times, as can be seen e.g. in Table 6.14. The observations indicate that once the fields has been processed into geometric primitives and been made a part of the Java 3D scene graph, the navigation, such as rotation, is only transformation matrices that are accelerated by the graphics hardware. As long as the size of the dataset is small enough to fit into the graphics memory, 64 Mb on the computer used for the case studies, navigation is smooth.

We can estimate which limits this enforces on the system. Recall that volume rendering in Java 3D and VisAD uses 6 sets of slice planes. Each slice plane is textured using RGBA[2] color. The size of each texture is determined by the number of sample points in the grid.

With 64 Mb of graphics memory this should limit us to a field of size $128 \cdot 128 \cdot 128$ for one time step, or a field of size $64 \cdot 64 \cdot 64$ for 10 time steps. If the visualization contains fields that exceed these limits, physical memory (RAM) would have to be used as well, and we would most likely experience slowdowns since much time would be spent copying the geometry between physical memory and graphical memory. The effects of this can be seen in Table 6.14 where we try to visualize 17 time steps of a dataset of size $64 \cdot 64 \cdot 64$. Recalling from Section 7.1.1 that there is no border between physical memory and virtual memory, the use of memory may result in massive swapping and drastic slowdowns in performance.

[2] Red, Green, Blue, Alpha, 8 bits per channel

Chapter 8

Concluding remarks

In this chapter we try to summarize the work that is done in this thesis and draw conclusions from the discussions in Chapter 7. Following that we look at future work that can be done.

8.1 The PVis visualization system

In this section we summarize the most relevant things we have learned from the various parts of the PVis system. We describe the key points we learned from Java, Java 3D, VisAD and the PVis system.

8.1.1 Using Java in a visualization system

During code development and while performing the case studies, we were continuously looking for signs that would indicate that performance suffered greatly as a result of PVis being written in Java. We could not determine if Java executes PVis significantly slower than a similar application written in other languages, as discussed in Section 7.1.1. Our experience was that performance of the PVis system implemented in Java was sufficient.

As was seen in Sections 7.1.1 and 7.3.2, a major issue for concern is garbage collection. The results discussed there lead us to think that it is difficult and in some cases impossible to write memory tight applications in Java. With the term memory tight we mean applications that stay within a predefined memory limit. Another result of garbage collection is that it may have significant impacts on performance, should it start during a time critical part of the application, as discussed in 7.3.2. The solution to both cases is to avoid the use of temporary objects, and rely on the reuse of long lived objects instead.

The extensive standard library, with well tested code for nearly all application level purposes, gives the programmer a good starting point. The presence of a garbage collector avoids direct memory leaks, meaning that dereferenced objects will be collected sooner or later. Multi threading is integrated into the runtime

environment. These facts, among others, lead us to conclude that when developing software in Java, one can expect a gain in productivity over languages such as C++.

8.1.2 Concerning Java 3D

The results from the case studies in Chapter 6, on navigation and animation, indicate that once the datasets have been processed to geometry and been made a part of the Java 3D scene graph, graphics are rendered quickly, which indicate that Java 3D, as a graphics library, performs at an acceptable level. This also means that for Java 3D to be an efficient graphics library, it requires an efficient graphics processor that can translate datasets into the Java 3D scene graph efficiently.

The lack of depth sorting in Java 3D reduces Java 3D's quality as a graphics library for visualization. We saw in Section 7.1.2 that it is not possible to draw transparent surfaces properly and volume rendering uses twice the required memory, 6 sets of slice planes instead of 3.

8.1.3 Concerning VisAD

VisAD is a fully functional Java 3D based visualization library, but contains some points, as mentioned on several occasions earlier, that make it unfitting to some degree for the work that was done in this thesis.

The first thing that makes VisAD unsuitable for the PVis system is that all its functionality is located in the display and rendering module. PVis is a pipeline based system built for interconnecting filter modules to achieve maximum flexibility. VisAD is based on loading datasets into the display and filtering them using one method only. Although the functionality of filtering is present in VisAD, such as iso surfaces, it is not present in such a way that it can be extracted into a filter module. It is integrated into the renderer. The fact that filtering functionality is located in the display and rendering module, also limits the flexibility of the visualizations that can be done in VisAD. One can for instance not combine a iso surface with volume rendering, since the map that triggers iso surfaces would render the volume as an iso surface as well.

VisAD is based upon Java 3D version 1.1, as stated in 7.1.3. In this version, Java 3D did not allow reference to geometry array by reference. Each time geometric primitives are updated, either by display map changes, subdomain dataset filtering or switching between solid and wire frame, all geometry arrays have to be regenerated. This means allocation of new memory and deallocation of the old through garbage collection. This is a cause for significant slow downs when interacting with the dataset in the display module in the PVis system.

As stated in Section 7.3.3 VisAD achieves animation by loading all time steps into memory and view them. This approach results in fast animation since all processing is complete when animation starts, but limits the number of time steps to the amount that can fit in memory. If the time dependent dataset is to large,

animation is not possible. This result is especially bad for the PVis system, which is targeted at loading large parallel datasets, potentially with many time steps.

8.1.4 Conclusion of the PVis visualization system

The goal of this thesis was to implement a system that could be used to visualize parallel datasets created by simulators written using Diffpack. Standard visualization techniques should be available despite the extra complexity of parallel datasets. It should also have the ability to filter out one or more subdomain datasets from the visualization.

The PVis system matches most of these criteria. It can load both single datasets and parallel datasets from Diffpack's simres format. It currently supports a only few filter modules for various visualization techniques, but the PVis system was written such that new modules can be added easily. In the display module of the PVis system one can view and navigate the data in addition to several other filtering options. The PVis system is based around a visualization pipeline, where the user dynamically builds a pipeline of modules and connections between them. This approach gives the user full flexibility to configure how a dataset is filtered before it is rendered.

There is also functionality that is missing from the PVis system. As stated earlier, PVis does not have the ability to animate time dependent datasets that span more time steps than would fit in memory at once. In addition to this, the PVis system only support datasets that have scalar field values. Vector fields and tensor fields are not supported.

8.2 Future work

In this section we try to cover what could be done to increase the quality and performance of the PVis system.

8.2.1 Loops in the PVis graph

One of the most serious drawbacks of the PVis system in its current version is that it relies on VisAD type animation which loads all time steps of the datasets before rendering, which may lead to memory problems. If we could simulate the effect of a `for` loop in the graph, it would be possible to iterate through the animation and loading one time step, filter it and display it, then do the same for the next time step.

Note that the introduction of loop modules would cause the pipeline to become cyclic since the loop's end module would be connected to the loop's start module, so partial re-implementation would be required in the pipeline execution as well.

With the introduction of loops in the graph, one could also introduce automatic screenshot generation. A module could be placed after the display in the graph that

extracted the image displayed there, and write it to a file. This would be far more useful than the manual snapshot that is currently present in the PVis system.

8.2.2 Pipeline based visualization library

The visualization library VisAD, on which the PVis system has been based is not pipeline based. This is a drawback, since it has caused the display module in PVis to be far more complex than it would otherwise have been. One could imagine that visualization library used in the modules were flexible enough to allow a simple display module and yet keep all the functionality. This can be done in two ways.

The first is to completely replace VisAD as visualization library and use a more pipeline based library instead, such as for instance VTK [1, 29]. Doing so would increase the functionality of the PVis system with all the functionality of the new visualization library. The drawback however, is that the PVis system is based on VisAD's dataset representation of fields and grids, and so are all the currently existing modules. Should VisAD be replaced by another visualization library, all the current modules would have to be modified to accommodate this.

The second method is to use VisAD for rendering only, and discard the fact that it has filtering functionality in its display and renderer module. By not relying on the display maps for rendering, one can also combine different rendering techniques, such as combining an iso surface with a volume. Note also that implementing all the functionality would be a time consuming process.

8.2.3 Vector and Tensor support

It is a weakness of the PVis system that it does not support vector and tensor fields. These were originally omitted as a result of that VisAD does not support them in the renderer. If VisAD is replaced however, as proposed in Section 8.2.2, vector and tensor support should be integrated into the PVis system.

8.2.4 Virtual global domain

Visualization techniques that are written to operate on parallel datasets, are more complex than those that operate on single dataset. In Section 3.2.3 we proposed an implementation of a virtual global domain, that abstracted the complexity of multiple subdomains in the parallel dataset to appear as a single dataset for all operations on the parallel dataset.

In PVis, this could have been made possible by extending the field and grid implementations of VisAD, or perhaps another visualization library, to match the abstract defintions described in 3.2.3. From these extended implementations one could derive container grids and container fields used to represent the virtual global domain.

Bibliography

[1] W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit, An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1998.

[2] M. A. Weiss, *Data Structures and Algorithm Analasys*, The Benjamin / Cummings Publishing Company, Inc, 1994

[3] *java.sun.com - The Source for Java^{TM} Technology*, http://java.sun.com/

[4] C. S. Horstmann, G. Cornell, *Core Java 2, Volume II - Advanced Features*, Sun Microsystems Press, 2000.

[5] T. Lindholm, F. Yellin, *The Java^{tm} Virtual Machine Specification*, http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[6] *The Java Tutorial, A practical guide for programmers*, http://java.sun.com/docs/books/tutorial/

[7] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification, Second Edition*, http://java.sun.com/docs/books/jls/second_edition/html/

[8] *Java 3D^{tm} Official Website*, http://java.sun.com/products/java-media/3D

[9] H. Sowizral, K. Rushforth, M. Deering, *Java 3D^{tm} API Specification*, JavaSoft, 2000

[10] *VisAD Home Page*, http://www.ssec.wisc.edu/ billh/visad.html

[11] *VisAD Generated Javadoc*, http://www.ssec.wisc.edu/ dglo/visad

[12] *Java Coding Conventions*, http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[13] *Iris Explorer Official website*, http://www.nag.co.uk/Welcome_IEC.html

[14] *Java 2 Platform API Specification, Generated Javadoc*, http://java.sun.com/j2se/1.3/docs/index.html

[15] *Extensible Markup Language (XML)*, http://www.w3c.org/XML.

[16] *Xerces, XML parser for Java and C++*, http://xml.apahce.org

[17] K. W. Brodlie, *Scientific Visualization Techniques and Applications*, Springer-Verlag, 1992.

[18] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[19] X. Cai, *An object-oriented model for developing parallel PDE software*, Preprint 1998-4, Department of Informatics, University of Oslo.

[20] H. P. Langtangen, *Computational Partial Differental Equations, Numerical Methods and Diffpack Programming* Springer-Verlag, 1999.

[21] *Diffpack World Wide Web homepage*, http://www.nobjects.com/Diffpack.

[22] Ø. Hjelle, *Triangulations and Triangle-Based Surfaces, Lecture notes for INF-TT, University of Oslo*, http://www.ifi.uio.no/inftt/Div/contentsTriang.html

[23] *OpenGL WWW Homepage*, http://www.opengl.org/

[24] *DirectX WWW Homepage*, http://www.microsoft.com/directx/

[25] Ø. Andreasen, *Seminar in data-visualization, UNIKI-VAVD, Lecture notes*, ftp://ftp.ffi.no/spub/stsk/oya/2000/oversikt.html.

[26] *The Java^{TM} 2 Enterprise Edition Developer's Guide*, http://java.sun.com/j2ee/

[27] X. Cai, *Numerical simulation of 3D fully nonlinear water waves on parallel computers*, In Kågstrøm et al (eds), Applied Parallel Computing, PARA'98, Lecture Notes in Computer Science, No. 1541, 1998, pp 48-55.

[28] X. Cai and G.T. Lines, *Enabling Numerical and Software Technologies for Studying the Electrical Activity in Human Heart*, To appear in proceedings of the PARA'02 Conference.

[29] *Kitware Inc. official website*, http://www.kitware.com/

[30] *Silicon Graphics official website*, http://www.sgi.com/