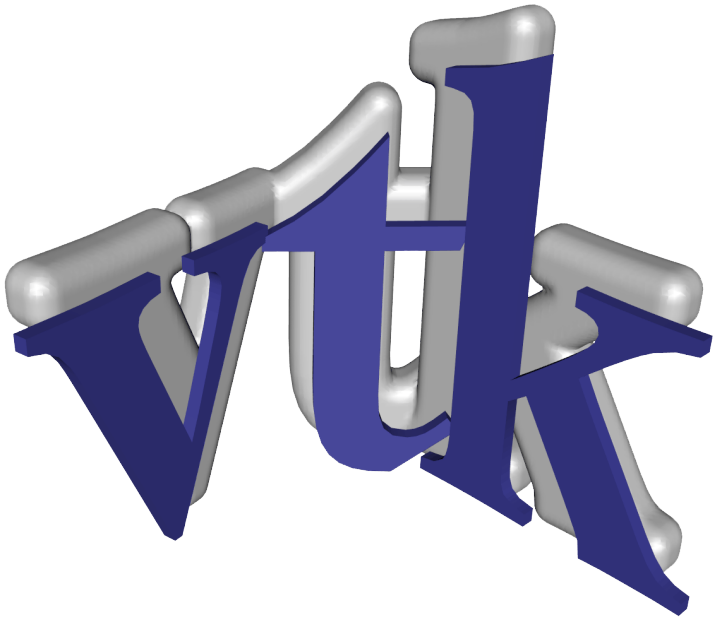
The Visualization Toolkit
An Object-Oriented Approach To 3D Graphics
Fourth Edition



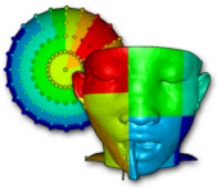
Will Schroeder, Ken Martin, Bill Lorensen [1]

2006 Revised 26 April 2005

[1] with special contributors: Lisa Sobierajski Avila, Rick Avila, C. Charles Law

Contents

Preface



Visualization is a great field to work in these days. Advances in computer hardware and software have brought this technology into the reach of nearly every computer system. Even the ubiquitous personal computer now offers specialized 3D graphics hardware at discount prices. And with recent releases of the Windows operating systems such as XP, OpenGL has become the de facto standard API for 3D graphics.

We view visualization and visual computing as nothing less than a new form of communication. All of us have long known the power of images to convey information, ideas, and feelings. Recent trends have brought us 2D images and graphics as evidenced by the variety of graphical user interfaces and business plotting software. But 3D images have been used sparingly, and often by specialists using specialized systems. Now this is changing. We believe we are entering a new era where 3D images, visualizations, and animations will begin to extend, and in some cases, replace the current communication paradigm based on words, mathematical symbols, and 2D images. Our hope is that along the way the human imagination will be freed like never before.

This text and companion software offers one view of visualization. The field is broad, including elements of computer graphics, imaging, computer science, computational geometry, numerical analysis, statistical methods, data analysis, and studies in human perception. We certainly do not pretend to cover the field in its entirety. However, we feel that this text does offer you a great opportunity to learn about the fundamentals of visualization. Not only can you learn from the written word and companion images, but the included software will allow you to *practice* visualization. You can start by using the sample data we have provided here, and then move on to your own data and applications. We believe that you will soon appreciate visualization as much as we do.

In this, the third edition of *Visualization Toolkit* textbook, we have added several new features since the first and second editions. Volume rendering is now extensively supported, including the ability to combine opaque surface graphics with volumes. We have added an extensive image processing pipeline that integrates conventional 3D visualization and graphics with imaging. Besides several new filters such as clipping, smoothing, 2D/3D Delaunay triangulation, and new decimation algorithms, we have added several readers and writers, and better support net-based tools such as Java and VRML. VTK now supports cell attributes, and attributes have been generalized into data arrays that are labeled as being scalars, vectors, and so on. Parallel processing, both shared-memory and distributed models, is a major addition. For example, VTK has been used on a large 1024-processor computer at the US National Labs to process nearly a pit-a-pat of data. A suite of 3D widgets is now available in VTK, enabling powerful data interaction techniques. Finally,

VTK's cross-platform support has greatly improved with the addition of CMake—a very nice tool for managing the compile process (CMake).

The additions of these features required the support of three special contributors to the text: Lisa Sobierajski Avila, Rick Avila, and C. Charles Law. Rick and Lisa worked hard to create an object-oriented design for volume rendering, and to insure that the design and software is fully compatible with the surface-based rendering system. Charles is the principle architect and implementer for the imaging pipeline. We are proud of the streaming and caching capability of the architecture: It allows us to handle large data sets despite limited memory resources.

Especially satisfying has been the response from users of the text and software. Not only have we received a warm welcome from these wonderful people, but many of them have contributed code, bug fixes, data, and ideas that greatly improved the system. In fact, it would be best to categorize these people as co-developers rather than users of the system. We would like to encourage anyone else who is interested in sharing their ideas, code, or data to contact the VTK user community at VTK, or one of the authors. We would very much welcome any contributions you have to make. Contact us at Kitware.

Acknowledgments

During the creation of the *Visualization Toolkit* we were fortunate to have the help of many people. Without their aid this book and the associated software might never have existed. Their contributions included performing book reviews, discussing software ideas, creating a supportive environment, and providing key suggestions for some of our algorithms and software implementations.

We would like to first thank our management at the General Electric Corporate R&D Center who allowed us to pursue this project and utilize company facilities: Peter Meenan, Manager of the Computer Graphics and Systems Program, and Kirby Vosburgh, Manager of the Electronic Systems Laboratory. We would also like to thank management at GE Medical Systems who worked with us on the public versus proprietary software issues: John Lalonde, John Heinen, and Steve Roehm.

We thank our co-workers at the R&D Center who have all been supportive: Matt Turek, for proof reading much of the second edition; also Majeid Alyassin, Russell Blue, Jeanette Bruno, Shane Chang, Nelson Corby, Rich Hammond, Margaret Kelliher, Tim Kelliher, Joyce Langan, Paul Miller, Chris Nafis, Bob Tatar, Chris Volpe, Boris Yamrom, Bill Hoffman (now at Kitware), Harvey Cline and Siegwalt Ludke. We thank former co-workers Skip Montanaro (who created a FAQ for us), Dan McLachlan and Michelle Barry. We'd also like to thank our friends and co-workers at GE Medical Systems: Ted Hudacko (who managed the first VTK users mailing list), Darin Okerlund, and John Skinner. Many ideas, helpful hints, and suggestions to improve the system came from this delightful group of people.

The third edition is now published by Kitware, Inc. We very much appreciate the efforts of the many contributors at Kitware who have helped make VTK one of the leading visualization systems in the world today. Sébastien Barré, Andy Cedilnik, Berk Geveci, Amy Henderson, and Brad King have each made significant contributions. Thank also to the folks at GE Global Research such as Jim Miller who continue to push the quality of the system, particularly with the creation of the DART system for regression testing. The US National Labs, led by Jim Ahrens of Los Alamos, has been instrumental in adding parallel processing support to VTK. An additional special thanks to Kitware for accepting the challenge of publishing this book.

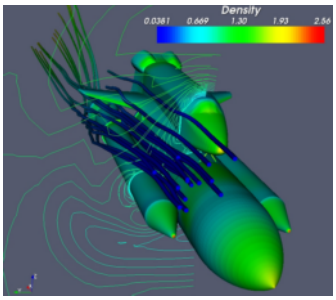
Many of the bug fixes and improvements found in the second and third editions came from talented people located around the world. Some of these people are acknowledged in the software and elsewhere in the text, but most of them have contributed their time, knowledge, code, and data without regard for recognition and acknowledgment. It is this exchange of ideas and information with people like this that makes the *Visualization Toolkit* such a fun and exciting project to work on. In particular we would like to thank John Biddiscombe, Charl P. Botha, David Gobbi, Tim Hutton, Dean Inglis, and Prabhu Ramachandran. Thank you very much.

A special thanks to the software and text reviewers who spent their own time to track down some nasty bugs, provide examples, and offer suggestions and improvements. Thank you Tom Citriniti, Mark Miller, George Petras, Hansong Zhang, Penny Rheingans, Paul Hinker, Richard Ellson, and Roger Crawfis. We'd also like to mention that Tom Citriniti at Rensselaer, and Penny Rheingans at the University of Mississippi (now at the University of Maryland Baltimore County)

were the first faculty members to teach from early versions of this text. Thank you Penny and Tom for your feedback and extra effort.

Most importantly we would like to thank our friends and loved ones who supported us patiently during this project. We know that you shouldered extra load for us. You certainly saw a lot less of us! But we're happy to say that we're back. Thank you.

1.0
Introduction




```
0266240 137055 062276 161755 115466 005322 132567 073216 002655
0266260 171466 126161 117155 065763 016177 014460 112765 055527
0266300 003767 175367 104754 036436 172172 150750 043643 145410
0266320 072074 000007 040627 070652 173011 002151 125132 140214
0266340 060115 014356 015164 067027 120206 070242 033065 131334
0266360 170601 170106 040437 127277 124446 136631 041462 116321
0266400 020243 005602 004146 121574 124651 006634 071331 102070
0266420 157504 160307 166330 074251 024520 114433 167273 030635
0266440 133614 106171 144160 010652 007365 026416 160716 100413
0266460 026630 007210 000630 121224 076033 140764 000737 003276
0266500 114060 042647 104475 110537 066716 104754 075447 112254
0266520 030374 144251 077734 015157 002513 173526 035531 150003
0266540 146207 015135 024446 130101 072457 040764 165513 156412
0266560 166410 067251 156160 106406 136770 030516 064740 022032
0266600 142166 123707 175121 071170 076357 037233 031136 015232
0266620 075074 016744 044055 102230 110063 033350 052765 172463
```

Visualization transforms numbers to images.

*V*isualization 2: the act or process of interpreting in visual terms or of putting into visual form, *Websters Ninth New Collegiate Dictionary.*"

1.1 What Is Visualization?

Visualization is a part of our everyday life. From weather maps to the exciting computer graphics of the entertainment industry, examples of visualization abound. But what is visualization? Informally, visualization is the transformation of data or information into pictures. Visualization engages the primary human sensory apparatus, vision, as well as the processing power of the human mind. The result is a simple and effective medium for communicating complex and/or voluminous information

1.1.1 Terminology

Different terminology is used to describe visualization. Scientific visualization is the formal name given to the field in computer science that ncompasses user interface, data representation and processing algorithms, visual representations, and other sensory presentation such as sound or touch

[3]. The term data visualization is another phrase used to describe visualization. Data visualization is generally interpreted to be more general than scientific visualization, since it implies treatment of data sources beyond the sciences and engineering. Such data sources include financial, marketing, or business data. In addition, the term data visualization is broad enough to include application of statistical methods and other standard data analysis techniques [2]. Another recently emerging term is information visualization. This field endeavors to visualize abstract information such as hyper-text documents on the World Wide Web, directory/ file structures on a computer, or abstract data structures [15]. A major challenge facing information visualization researchers is to develop coordinate systems, transformation methods, or structures that meaningfully organize and represent data.

Another way to classify visualization technology is to examine the context in which the data exists. If the data is spatial-temporal in nature (up to three spatial coordinates and the time dimension) then typically methods from scientific visualization are used. If the data exists in higher dimensional spaces, or abstract spaces, then methods from information visualization are used. This distinction is important, because the human perceptual system is highly tuned to space-time relationships. Data expressed in this coordinate system is inherently understood with little need for explanation. Visualization of abstract data typically requires extensive explanations as to what is being viewed. This is not to say that there is no overlap between scientific and information visualization often the first step in the information visualization process is to project abstract data into the spatial-temporal domain, and then use the methods of scientific visualization to view the results. The projection process can be quite complex, involving methods of statistical graphics, data mining, and other techniques, or it may be as simple as selecting a lower-dimensional subset of the original data.

In this text we use the term data visualization instead of the more specific terms scientific visualization or information visualization. We feel that scientific visualization is too narrow a description of the field, since visualization techniques have moved beyond the scientific domain and into areas of business, social science, demographics, and information management in general. We also feel that the term data visualization is broad enough to encompass the term information visualization.

1.1.2 Examples of Visualization

Perhaps the best definition of visualization is offered by example. In many cases visualization is influencing peoples' lives and performing feats that a few years ago would have been unimaginable. A prime example of this is its application to modern medicine.

Computer imaging techniques have become an important diagnostic tool in the practice of modern medicine. These include techniques such as X-ray *Computed Tomography* (CT) and *Magnetic Resonance Imaging* (MRI). These techniques use a sampling or data acquisition process to capture information about the internal anatomy of a living patient. This information is in the form of *slice-planes* or cross-sectional images of a patient, similar to conventional photographic X-rays. CT imaging uses many pencil thin X-rays to acquire the data, while MRI combines large magnetic fields with pulsed radio waves. Sophisticated mathematical techniques are used to reconstruct the slice-planes. Typically, many such closely spaced slices are gathered together into a *volume* of data to complete the study.

As acquired from the imaging system, a slice is a series of numbers representing the attenuation of X-rays (CT) or the relaxation of nuclear spin magnetization (MRI) [8]. On any given slice these numbers are arranged in a matrix, or regular array. The amount of data is large, so large that it is not possible to understand the data in its raw form. However, by assigning to these numbers a gray scale value, and then displaying the data on a computer screen, structure emerges. This structure results from the interaction of the human visual system with the spatial organization of the data

and the gray-scale values we have chosen. What the computer represents as a series of numbers, we see as a cross section through the human body: skin, bone, and muscle. Even more impressive results are possible when we extend these techniques into three dimensions. Image slices can be gathered into volumes and the volumes can be processed to reveal complete anatomical structures. Using modern techniques, we can view the entire brain, skeletal system, and vascular system on a living patient without interventional surgery. Such capability has revolutionized modern medical diagnostics, and will increase in importance as imaging and visualization technology matures.

Another everyday application of visualization is in the entertainment industry. Movie and television producers routinely use computer graphics and visualization to create entire worlds that we could never visit in our physical bodies. In these cases we are visualizing other worlds as we imagine them, or past worlds we suppose existed. It's hard to watch the movies such as *JurassicPark* and *Toy Story* and not gain a deeper appreciation for the awesome Tyrannosaurus Rex, or to be charmed by Toy Story's heroic Buzz Lightyear.

Morphing is another popular visualization technique widely used in the entertainment industry. Morphing is a smooth blending of one object into another. One common application is to morph between two faces. Morphing has also been used effectively to illustrate car design changes from one year to the next. While this may seem like an esoteric application, visualization techniques are used routinely to present the daily weather report. The use of isovalue, or contour, lines to display areas of constant temperature, rainfall, and barometric pressure has become a standard tool in the daily weather report.

Many early uses of visualization were in the engineering and scientific community. From its inception the computer has been used as a tool to simulate physical processes such as ballistic trajectories, fluid flow, and structural mechanics. As the size of the computer simulations grew, it became necessary to transform the resulting calculations into pictures. The amount of data overwhelmed the ability of the human to assimilate and understand it. In fact, pictures were so important that early visualizations were created by manually plotting data. Today, we can take advantage of advances in computer graphics and computer hardware. But, whatever the technology, the application of visualization is the same: to display the results of simulations, experiments, measured data, and fantasy; and to use these pictures to communicate, understand, and entertain.

1.2 Why Visualize?

Visualization is a necessary tool to make sense of the flood of information in today's world of computers. Satellites, supercomputers, laser digitizing systems, and digital data acquisition systems acquire, generate, and transmit data at prodigious rates. The Earth-Orbiting Satellite (EOS) transmits terabytes of data every day. Laser scanning systems generate over 500,000 points in a 15 second scan [19]. Supercomputers model weather patterns over the entire earth [5]. In the first four months of 1995, the New York Stock Exchange processed, on average, 333 million transactions per day [16]. Without visualization, most of this data would sit unseen on computer disks and tapes. Visualization offers some hope that we can extract the important information hidden within the data.

There is another important element to visualization: It takes advantage of the natural abilities of the human vision system. Our vision system is a complex and powerful part of our bodies. We use it and rely on it in almost everything we do. Given the environment in which our ancestors lived, it is not surprising that certain senses developed to help them survive. As we described earlier in the example of a 2D MRI scan, visual representations are easier to work with. Not only do we have strong 2D visual abilities, but also we are adept at integrating different viewpoints and other visual clues into a mental image of a 3D object or plot. This leads to interactive visualization, where we can manipulate our viewpoint. Rotating about the object helps to achieve a better understanding. Likewise, we have a talent for recognizing temporal changes in an image. Given an animation

consisting of hundreds of frames, we have an uncanny ability to recognize trends and spot areas of rapid change.

With the introduction of computers and the ability to generate enormous amounts of data, visualization offers the technology to make the best use of our highly developed visual senses. Certainly other technologies such as statistical analysis, artificial intelligence, mathematical filtering, and sampling theory will play a role in large-scale data processing. However, because visualization directly engages the vision system and human brain, it remains an unequaled technology for understanding and communicating data.

Visualization offers significant financial advantages as well. In today's competitive markets, computer simulation teamed with visualization can reduce product cost and improve time to market. A large cost of product design has been the expense and time required to create and test design prototypes. Current design methods strive to eliminate these physical prototypes, and replace them with digital equivalents. This digital prototyping requires the ability to create and manipulate product geometry, simulate the design under a variety of operating conditions, develop manufacturing techniques, demonstrate product maintenance and service procedures, and even train operators on the proper use of the product before it is built. Visualization plays a role in each case. Already CAD systems are used routinely to model product geometry and design manufacturing procedures. Visualization enables us to view the geometry, and see special characteristics such as surface curvature. For instance, analysis techniques such as finite element, finite difference, and boundary element techniques are used to simulate product performance; and visualization is used to view the results. Recently, human ergonomics and anthropometry are being analyzed using computer techniques in combination with visualization [9]. Three-dimensional graphics and visualization are being used to create training sequences. Often these are incorporated into a hypertext document or World Wide Web (WWW) pages. Another practical use of graphics and visualization has been in-flight simulators. This has been shown to be a significant cost savings as compared to flying real airplanes and is an effective training method.

1.3 Imaging, Computer Graphics, and Visualization

There is confusion surrounding the difference between imaging, computer graphics, and visualization. We offer these definitions.

- *Imaging*, or image processing, is the study of 2D pictures, or images. This includes techniques to transform (e.g., rotate, scale, shear), extract information from, analyze, and enhance images.

  The resulting data is mapped to a graphics system for display.

- *Computer graphics* is the process of creating images using a computer. This includes both 2D paint-and-draw techniques as well as more sophisticated 3D drawing (or rendering) techniques.

- *Visualization* is the process of exploring, transforming, and viewing data as images (or other sensory forms) to gain understanding and insight into the data.

Based on these definitions we see that there is overlap between these fields. The output of computer graphics is an image, while the output of visualization is often produced using computer graphics. Sometimes visualization data is in the form of an image, or we wish to visualize object geometry using realistic rendering techniques from computer graphics.

Generally speaking we distinguish visualization from computer graphics and image processing in three ways.

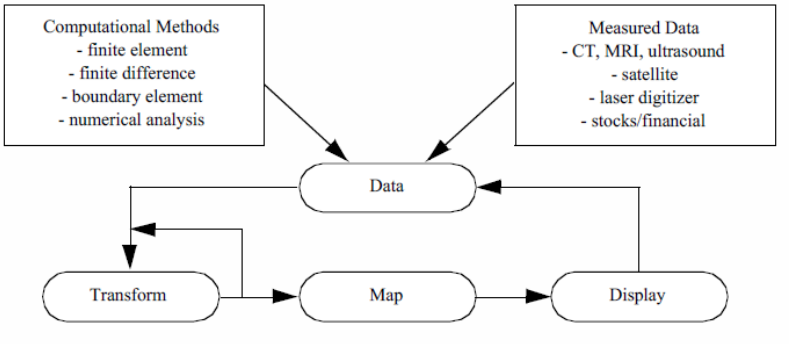


Figure 1.1: The visualization process. Data from various sources is repeatedly transformed to extract, derive, and enhance information. The resulting data is mapped to a graphics system for display.

1. The dimensionality of data is three dimensions or greater. Many well-known methods are available for data of two dimensions or less; visualization serves best when applied to data of higher dimension.
2. Visualization concerns itself with data transformation. That is, information is repeatedly created and modified to enhance the meaning of the data.
3. Visualization is naturally interactive, including the human directly in the process of creating, transforming, and viewing data.

Another perspective is that visualization is an activity that encompasses the process of exploring and understanding data. This includes both imaging and computer graphics as well as data processing and filtering, user interface methodology, computational techniques, and software design. **Figure 1.1** depicts this process.

As this figure illustrates we see that the visualization process focuses on data. In the first step data is acquired from some source. Next, the data is transformed by various methods, and then mapped to a form appropriate for presentation to the user. Finally, the data is rendered or displayed, completing the process. Often, the process repeats as the data is better understood or new models are developed. Sometimes the results of the visualization can directly control the generation of the data. This is often referred to as *analysis steering*. Analysis steering is an important goal of visualization because it enhances the interactivity of the overall process.

1.4 Origins of Data Visualization

The origin of visualization as a formal discipline dates to the 1987 NSF report *Visualization in Scientific Computing* [3]. That report coined the term *scientific visualization*. Since then the field has grown rapidly with major conferences, such as IEEE Visualization, becoming well established. Many large computer graphics conferences, for example ACM SIGGRAPH, devote large portions of their program to visualization technology.

Of course, data visualization technology had existed for many years before the 1987 report referenced [18]. The first practitioners recognized the value of presenting data as images. Early pictorial data representations were created during the eighteenth century with the arrival of statistical graphics. It was only with the arrival of the digital computer and the development of the field of computer graphics, that visualization became a practicable discipline.

The future of data visualization and graphics appears to be explosive. Just a few decades ago, the field of data visualization did not exist and computer graphics was viewed as an offshoot of the more formal discipline of computer science. As techniques were created and computer power increased, engineers, scientists, and other researchers began to use graphics to understand and communicate data. At the same time, user interface tools were being developed. These forces have now converged to the point where we expect computers to adapt to humans rather than the other way around. As such, computer graphics and data visualization serve as the window into the computer, and more importantly, into the data that computers manipulate. Now, with the visualization window, we can extract information from data and analyze, understand, and manage more complex systems than ever before.

Dr. Fred Brooks, Kenan Professor of Computer Science at the University of North Carolina at Chapel Hill and recipient of the John von Neumann Medal of the IEEE, puts it another way. At the award presentation at the ACM SIGGRAPH '94, Dr. Brooks stated that computer graphics and visualization offer "intelligence amplification" (IA) as compared to artificial intelligence (AI). Besides the deeper philosophical issues surrounding this issue (e.g., human before computer), it is a pragmatic observation. While the long-term goal of AI has been to develop computer systems that could replace humans in certain applications, the lack of real progress in this area has lead some researchers to view the role of computers as amplifiers and assistants to humans. In this view, computer graphics and visualization play a significant role, since arguably the most effective human/computer interface is visual. Recent gains in computer power and memory are only accelerating this trend, since it is the interface between the human and the computer that often is the obstacle to the effective application of the computer.

1.5 Purpose of This Book

There currently exist texts that define and describe data visualization, many of them using case studies to illustrate techniques and typical applications. Some provide high-level descriptions of algorithms or visualization system architectures. Detailed descriptions are left to academic journals or conference proceedings. What these texts lack is a way to *practice* visualization. Our aim in this text is to go beyond descriptions and provide tools to learn about and apply visualization to your own application area. In short, the purpose of the book is fourfold.

1. Describe visualization algorithms and architectures in detail.
2. Demonstrate the application of data visualization to a broad selection of case studies.
3. Provide a working architecture and software design for application of data visualization to real-world problems.
4. Provide effective software tools packaged in a C++ class library. We also provide language bindings for the interpreted languages Tcl, Python, and Java.

Taken together, we refer to the text and software as the *Visualization Toolkit*, or VTK for short. Our hope is that you can use the text to learn about the fundamental concepts of visualization, and then adapt the computer code to your own applications and data.

1.6 What This Book Is Not

The purpose of this book is not to provide a rigorous academic treatise on data visualization. Nor do we intend to include an exhaustive survey of visualization technology. Our goal is to bridge

the formal discipline of data visualization with practical application, and to provide a solid technical overview of this emerging technology. In many cases we refer you to the included software to understand implementation details. You may also wish to refer to the appropriate references for further information.

1.7 Intended Audience

Our primary audience is computer users who create, analyze, quantify, and/or process data. We assume a minimal level of programming skill. If you can write simple computer code to import data and know how to run a computer program, you can practice data visualization with the software accompanying this book.

As we wrote this book we also had in mind educators and students of introductory computer graphics and visualization courses. In more advanced courses this text may not be rigorous enough to serve as sole reference. In these instances, this book will serve well as a companion text, and the software is well suited as a foundation for programming projects and class exercises.

Educators and students in other disciplines may also find the text and software to be valuable tools for presenting results. Courses in numerical analysis, computer science, business simulation, chemistry, dynamic systems, and engineering simulations, to name a few, often require large-scale programming projects that create large amounts of data. The software tools provided here are easy to learn and readily adapted to different data sources. Students can incorporate this software into their work to display and analyze their results.

1.8 How to Use This Book

There are a number of approaches you can take to make effective use of this book. The particular approach depends on your skill level and goals. Three likely paths are as follows:

Novice. You're a novice if you lack basic knowledge of graphics, visualization, or object-oriented principles. Start by reading Chapter 2 if you are unfamiliar with object-oriented principles, Chapter 3 if you are unfamiliar with computer graphics, and Chapter 4 if you are unfamiliar with visualization. Continue by reading the application studies in Chapter 12. You can then move on to the CD-ROM and try out some programming examples. Leave the more detailed treatment of algorithms and data representation until you are familiar with the basics and plan to develop your own applications.

Hacker. You're a hacker if you are comfortable writing your own code and editing other's. Review the examples in Chapter 3, Chapter 4, and Chapter 12. At this point you will want to acquire the companion software guide to this text (*The VTK User's Guide*) or become familiar with the programming resources at VTK. Then retrieve the examples from the CD-ROM and start practicing.

Researcher/Educator. You're a researcher if you develop computer graphics and/or visualization algorithms or if you are actively involved in using and evaluating such systems. You're an educator if you cover aspects of computer graphics and/or visualization within your courses. Start by reading Chapter 2, Chapter 3, and Chapter 4. Select appropriate algorithms from the text and examine the associated source code. If you wish to extend the system, we recommend that you acquire the companion software guide to this text (*The VTK User's Guide*) or become familiar with the programming resources at VTK.

1.9 Software Considerations and Example Code

In writing this book we have attempted to strike a balance between practice and theory. We did not want the book to become a user manual, yet we did want a strong correspondence between algorithmic presentation and software implementation. (Note: *The VTK User's Guide* published by Kitware, Inc. http://www.kitware.com is recommended as a companion text to this book.) As a result of this philosophy, we have adopted the following approach:

Application versus Design. The book's focus is the application of visualization techniques to real-world problems. We devote less attention to software design issues. Some of these important design issues include: memory management, deriving new classes, shallow versus deep object copy, single versus multiple inheritance, and interfaces to other graphics libraries. Software issues are covered in the companion text The *VTK User's Guide* published by Kitware, Inc.

Theory versus Implementation. Whenever possible, we separate the theory of data visualization from our implementation of it. We felt that the book would serve best as a reference tool if the theory sections were independent of software issues and terminology. Toward the end of each chapter there are separate implementation or example sections that are implementation specific. Earlier sections are implementation free.

Documentation. This text contains documentation considered essential to understanding the software architecture, including object diagrams and condensed object descriptions. More extensive documentation of object methods and data members is embedded in the software (in the.h header files) and on CD-ROM or online at VTK. In particular, the Doxygen generated manual pages contain detailed descriptions of class relationships, methods, and other attributes.

We use a number of conventions in this text. Imported computer code is denoted with a typewriter font, as are external programs and computer files. To avoid conflict with other C++ class libraries, all class names in VTK begin with the " vtk" prefix. Methods are differentiated from variables with the addition of the " ()" postfix. (Other conventions are listed in *VTK User's Guide*.)

All images in this text have been created using the *Visualization Toolkit* software and data found on the included CD-ROM or from the Web site http://www.vtk.org. In addition, every image has source code (sometimes in C++ and sometimes a Tcl script). We decided against using images from other researchers because we wanted you to be able to practice visualization with every example we present. Each computer generated image indicates the originating file. Files ending in.cxx are C++ code, files ending in.tcl are Tcl scripts. Hopefully these examples can serve as a starting point for you to create your own applications.

To find the example code you will want to search in one of three areas. The standard VTK distribution includes an VTK/Examples directory where many well-documented examples are found. The VTK testing directories VTK/Testing, for example, VTK/Graphics/Testing/Tcl, contain some of the example code used in this text. These examples use the data found in the VTKData distribution. Finally, a separate software distribution, the VTKTextbook distribution, contains examples and data that do not exist in the standard VTK distribution. The VTK, VTKData, and VTKTextbook distributions are found on the included CD-ROM and/or on the web site at VTK.

1.10 Chapter-by-Chapter Overview

Chapter 2: Object-Oriented Design

This chapter discusses some of the problems with developing large and/or complex software systems and describes how object-oriented design addresses many of these problems. This chapter defines the key terms used in object-oriented modelling and design and works through a real-world

example. The chapter concludes with a brief look at some object-oriented languages and some of the issues associated with object-oriented visualization.

Chapter 3: Computer Graphics Primer

Computer graphics is the means by which our visualizations are created. This chapter covers the fundamental concepts of computer graphics from an application viewpoint. Common graphical entities such as cameras, lights, and geometric primitives are described along with some of the underlying physical equations that govern lighting and image generation. Issues related to currently available graphics hardware are presented, as they affect how and what we choose to render. Methods for interacting with data are introduced.

Chapter 4: The Visualization Pipeline

This chapter explains our methodology for transforming raw data into a meaningful representation that can than be rendered by the graphics system. We introduce the notion of a visualization pipeline, which is similar to a data flow diagram from software engineering. The differences between process objects and data objects are covered, as well as how we resolved issues between performance and memory usage. We explain the advantages to a pipeline network topology regarding execution ordering, result caching, and reference counting.

Chapter 5: Basic Data Representation

There are many types of data produced by the variety of fields that apply visualization. This chapter describes the data objects that we use to represent and access such data. A flexible design is introduced where the programmer can interact with most any type of data using one consistent interface. The three high level components of data (structure, cells, and data attributes) are introduced, and their specific subclasses and components are discussed.

Chapter 6: Fundamental Algorithms

Where the preceding chapter deals with data objects, this one introduces process objects. These objects encompass the algorithms that transform and manipulate data. This chapter looks at commonly used techniques for isocontour extraction, scalar generation, color mapping, and vector field display, among others. The emphasis of this chapter is to provide the reader with a basic understanding of the more common and important visualization algorithms.

Chapter 7: Advanced Computer Graphics

This chapter covers advanced topics in computer graphics. The chapter begins by introducing transparency and texture mapping, two topics important to the main thrust of the chapter: volume rendering. Volume rendering is a powerful technique to see inside of 3D objects, and is used to visualize volumetric data. We conclude the chapter with other advanced topics such as stereoscopic rendering, special camera effects, and 3D widgets.

Chapter 8: Advanced Data Representation

Part of the function of a data object is to store the data. The first chapter on data representation discusses this aspect of data objects. This chapter focuses on basic geometric and topological access methods, and computational operations implemented by the various data objects. The chapter

covers such methods as coordinate transformations for data sets, interpolation functions, derivative calculations, topological adjacency operations, and geometric operations such as line intersection and searching.

Chapter 9: Advanced Algorithms

This chapter is a continuation of *Fundamental Algorithms* and covers algorithms that are either more complex or less widely used. Scalar algorithms such as dividing cubes are covered along with vector algorithms such as stream ribbons. A large collection of modelling algorithms is discussed, including triangle strip generation, polygon decimation, feature extraction, and implicit modelling. We conclude with a look at some visualization algorithms that utilize texture mapping.

Chapter 10: Image Processing

While 3D graphics and visualization is the focus of the book, image processing is an important tool for preprocessing and manipulating data. In this chapter we focus on several important image processing algorithms, as well as describe how we use a streaming data representation to process large datasets.

Chapter 11: Visualization on the Web

The Web is one of the best places to share your visualizations. In this chapter we show you how to write Java-based visualization applications, and how to create VRML (Virtual Reality Modelling Language) data files for inclusion in your own Web content.

Chapter 12: Applications

In this chapter we tie the previous chapters together by working through a series of case studies from a variety of application areas. For each case, we briefly describe the application and what information we expect to obtain through the use of visualization. Then, we walk through the design and resulting source code to demonstrate the use of the tools described earlier in the text.

1.11 Legal Considerations

We make no warranties, expressly or implied, that the computer code contained in this text is free of error or will meet your requirements for any particular application. Do not use this code in any application where coding errors could result in injury to a person or loss of property. If you do use the code in this way, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of this code.

The computer code contained in this text is copyrighted. We grant permission for you to use, copy, and distribute this software for any purpose. However, you may not modify and then redistribute the software. Some of the algorithms presented here are implementations of patented software. If you plan to use this software for commercial purposes, please insure that applicable patent laws are observed.

Some of the data on the CD-ROM may be freely distributed or used (with appropriate acknowledgment). Refer to the local README files or other documentation for details.

Several registered trademarks are used in this text. UNIX is a trademark of UNIX System Laboratories. Sun Workstation and XGL are trademarks of Sun Microsystems, Inc. Microsoft, MS, MS-DOS, and Windows are trademarks of Microsoft Corporation. The X Window System

is a trademark of the Massachusetts Institute of Technology. Starbase and HP are trademarks of Hewlett-Packard Inc. Silicon Graphics and OpenGL, are trademarks of Silicon Graphics, Inc. Macintosh is a trademark of Apple Computer. RenderMan is a trademark of Pixar.

1.12 Bibliographic Notes

A number of visualization texts are available. The first six texts listed in the reference section are good general references ([10], [11], [1], [21], [2], and[6]). Gallagher [6] is particularly valuable if you are from a computational background. Wolff and Yaeger [21] contains many beautiful images and is oriented towards Apple Macintosh users. The text includes a CD-ROM with images and software.

You may also wish to learn more about computer graphics and imaging. Foley and van Dam [7] is the basic reference for computer graphics.

Another recommended text is [4]. Suggested reference books on computer imaging are [12] and [20].

Two texts by Tufte [18] [17] are particularly impressive. Not only are the graphics superbly done, but the fundamental philosophy of data visualization is articulated. He also describes the essence of good and bad visualization techniques.

Another interesting text is available from Siemens, a large company offering medical imaging systems [8]. This text describes the basic concepts of imaging technology, including MRI and CT. This text is only for those users with a strong mathematical background. A less mathematical overview of MRI is available from [14].

To learn more about programming with Visualization Toolkit, we recommend the text The VTK Users Guide [13]. This text has an extensive example suite as well as descriptions of the internals of the software. Programming resources including a detailed description of APIs, VTK file formats, and class descriptions are provided.

Bibliography

[1] K. W. Brodlie et al. *Scientific Visualization Techniques and Applications*. Springer-Verlag, Berlin, 1992.

[2] L. Rosenblum et al. *Scientific Visualization Advances and Challenges*. Harcourt Brace & Company, London, 1994.

[3] B. H. McCormick, T. A. DeFanti, and M. D. Brown. *Visualization in Scientific Computing*. Report of the NSF Advisory Panel on Graphics, Image Processing and Workstations, 1987.

[4] P. Burger and D. Gillies. *Interactive Computer Graphics Functional, Procedural and Device-Level Methods*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[5] P. C. Chen. "A Climate Simulation Case Study". In: *Proceedings of Visualization '93*. IEEE Computer Society Press, Los Alamitos, 1993, pp. 391–401.

[6] R. S. Gallagher, ed. *Computer Visualization Graphics Techniques for Scientific and Engineering Analysis*. CRC Press, Boca Raton, FL, 1995.

[7] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice (2d Ed)*. Addison-Wesley, Reading, MA, 1990.

[8] E. Krestel, ed. *Imaging Systems for Medical Diagnostics*. Siemens-Aktienges, Munich, 1990.

[9] *McDonnell Douglas Human Modeling System Reference Manual*. Version 2.1. Report MDC 93K0281. Mc-Donnell Douglas Corporation, Human Factors Technology. July 1993.

[10] G. M. Nielson and B. Shriver, eds. *Visualization in Scientific Computing*. IEEE Computer Society Press, Los Alamitos, 1990.

[11] N. M. Patrikalakis, ed. *Scientific Visualization of Physical Phenomena*. Springer-Verlag, Berlin, 1991.

[12] T. Pavlidis. *Graphics and Image Processing*. Computer Science Press, Rockville, MD, 1982.

[13] Will Schroeder, Ken Martin, and Bill Lorensen. *The VTK Users Guide*. Ed. by W. Schroeder. Kitware, 2006. ISBN: 1-930934-19-X. URL: https://www.amazon.com/Visualization-Toolkit-Object-Oriented-Approach-Graphics/dp/193093419X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=193093419X.

[14] H. J. Smith and F. N. Ranallo. *A Non-Mathematical Approach to Basic MRI*. Medical Physics Publishing Corporation, Madison, WI, 1989.

[15] *The First Information Visualization Symposium*. IEEE Computer Society Press, 1995.

[16] *The New York Times Business Day, Tuesday, May 2* (1990).

[17] E. R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1999.

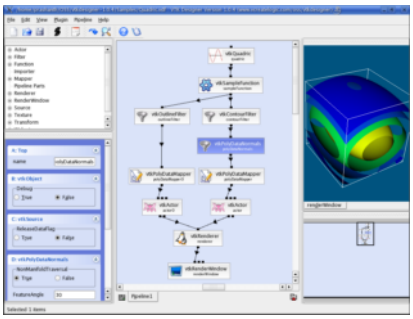
[18] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1999.

[19] K. Waters and D. Terzopoulos. “Modeling and Animating Faces Using Scanned Data”. In: *Visualization and Computer Animation*. 2. 1991, pp. 123–128.

[20] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[21] R. S. Wolff and L. Yaeger. *Visualization of Natural Phenomena TELOS*. Springer-Verlag, Santa Clara, CA, 1993.

2.0
Object-Oriented Design



VTK Designer image courtesy of vcreatelogic.com.

Object-oriented systems are becoming widespread in the computer industry for good reason.

Object-oriented systems are more modular, easier to maintain, and easier to describe than traditional procedural systems. Since the Visualization Toolkit has been designed and implemented using object-oriented design, we devote this chapter to summarizing the concepts and practice of object-oriented design and implementation.

2.1 Introduction

Todays software systems try to solve complex, real-world problems. A rigorous software design and implementation methodology can ease the burden of this complexity. Without such a methodology, software developers can find it difficult to meet a systems specifications. Furthermore, as specifications change and grow, a software system that does not have a solid, underlying architecture and design will have difficulty adapting to these expanding requirements.

2.2 Bibliographic Notes

There are several excellent textbooks on object-oriented design. Both [12] and [10] present language-independent design methodologies. Both books emphasize modelling and diagramming as key aspects of design. [15] also describes the OO design process in the context of Eiffel, an OO language. Another popular book has been authored by Booch [6].

Anyone who wants to be a serious user of object-oriented design and implementation should read the books on Smalltalk [4] [11] by the developers of Smalltalk at Xerox Parc. In another early object-oriented programming book, [7] describes OO techniques and the programming language

Objective-C. Objective-C is a mix of C and Smalltalk and was used by Next Computer in the implementation of their operating system and user interface.

There are many texts on object-oriented languages. CLOS [14] describes the Common List Object System. Eiffel, a strongly typed OO language is described by [15] . Objective-C [7] is a weakly typed language.

Since C++ has become a popular programming language, there now many class libraries available for use in applications. [13] describes an extensive class library for collections and arrays modeled after the Smalltalk classes described in [4] . [18] and [16] describe the Standard Template Library, a framework of data structures and algorithms that is now a part of the ANSI C++ standard. Open Inventor [1] is a C++ library supporting interactive 3D computer graphics. The Insight Segmentation and Registration Toolkit (ITK) is a relatively newclass library often used in combination with VTK [21] for medical data processing. VXL is a C++library for computer vision research and implementation [22]. Several mathematical libraries such as VNL (a part of VXL) and Blitz++ [2] are also available. A wide variety of other C++ toolkits are available, Google searches [3] are the best way to find them.

C++ texts abound. The original description by the author of C++ [20] is a must for any serious C++ programmer. Another book [8] describes standard extensions to the language. These days the UML book seriesof which [9] and [19] are quite popularare highly recommended resources. Several books on generic programming [5] and STL [17] are also useful. Check with your colleagues for their favourite C++ book. To keep in touch with new developments there are conferences, journals, and Web sites. The strongest technical conference on object-oriented topics is the annual Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) conference. This is where researchers in the field describe, teach and debate the latest techniques in object-oriented technology. The bimonthly Journal of Object-Oriented Programming (JOOP) published by SIGS Publications, NY, presents technical papers, columns, and tutorials on the field. Resources on the World Wide Web include the Usenet newsgroups comp.object and comp.lang.c++.

Bibliography

[1] URL: http://oss.sgi.com/projects/inventor/.

[2] URL: http://www.oonumerics.org/blitz/.

[3] URL: https://www.google.com/.

[4] A. Goldberg, D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[5] M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[6] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Co., Redwood City, CA, 1991.

[7] B. J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA, 1986.

[8] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[9] J. Rumbaugh G. Booch I. Jacobson. *The Unified Modeling Language User Guide (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1998. ISBN: 0201571684. URL: https://www.amazon.com/Unified-Modeling-Language-Addison-Wesley-Technology/dp/0201571684?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201571684.

[10] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, England. 1979.

[11] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[13] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming*. John Wiley & Sons, Ltd., Chichester, England, 1990.

[14] S. Keene. *Object-Oriented Programming in Common Lisp: A Programmers Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, Hertfordshire, England, 1988.

[16] D. Musser and A. Stepanov. "Algorithm-Oriented Generic Libraries". In: *Software Practice and Experience* 24.7 (July 1994), pp. 623–642.

[17] David R. Musser and Atul Saini. *Stl Tutorial & Reference Guide: C++ Programming With the Standard Template Library (Addison-Wesley Professional Computing Series)*. Addison-Wesley, 1996. ISBN: 0201633981. URL: https://www.amazon.com/Stl-Tutorial-Reference-Guide-Addison-Wesley/dp/0201633981?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201633981.

[18] P.J. Plauger et al. *The C++ Standard Template Library*. Prentice Hall, 2000. ISBN: 978-0134376332. URL: https://www.amazon.com/C-Standard-Template-Library/dp/0134376331?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0134376331.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional. ISBN: 020130998X. URL: https://www.amazon.com/Unified-Modeling-Language-Reference-Manual/dp/020130998X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=020130998X.

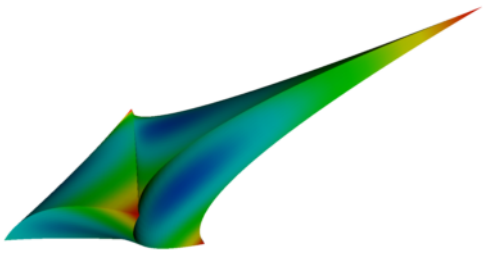
[20] Bjarne Stroustrup. *The C++ Programming Language: Special Edition (3rd Edition)*. Addison-Wesley Professional, 2000. ISBN: 0-201-70073-5. URL: https://www.amazon.com/Programming-Language-Special-3rd/dp/0201700735?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201700735.

[21] *The Insight Software Consortium*. URL: https://www.itk.org.

[22] *VXL*. URL: http://vxl.sourceforge.net/.

3.0

Computer Graphics Primer



Rendering quadratic tetrahedra.

Computer graphics is the foundation of data visualization. Practically speaking, visualization is the process that transforms data into a set of graphics primitives. The methods of computer graphics are then used to convert these primitives into pictures or animations. This chapter discusses basic computer graphics principles. We begin by describing how lights and physical objects interact to form what we see. Next we examine how to simulate these interactions using computer graphics techniques. Hardware issues play an important role here since modern computers have built-in hardware support for graphics. The chapter concludes with a series of examples that illustrate our object-oriented model for 3D computer graphics.

3.1 Introduction

Computer graphics is the process of generating images using computers. We call this process *rendering*. There are many types of rendering processes, ranging from 2D paint programs to sophisticated 3D techniques. In this chapter we focus on basic 3D techniques for visualization.

We can view rendering as the process of converting graphical data into an image. In data visualization our goal is to transform data into graphical data, or *graphics primitives*, that are then rendered. The goal of our rendering is not so much photo realism as it is information content. We also strive for interactive graphical displays with which it is possible to directly manipulate the underlying data. This chapter explains the process of rendering an image from graphical data. We begin by looking at the way lights, cameras, and objects (or actors) interact in the world around us. From this foundation we explain how to simulate this process on a computer.

3.2 A Physical Description of Rendering

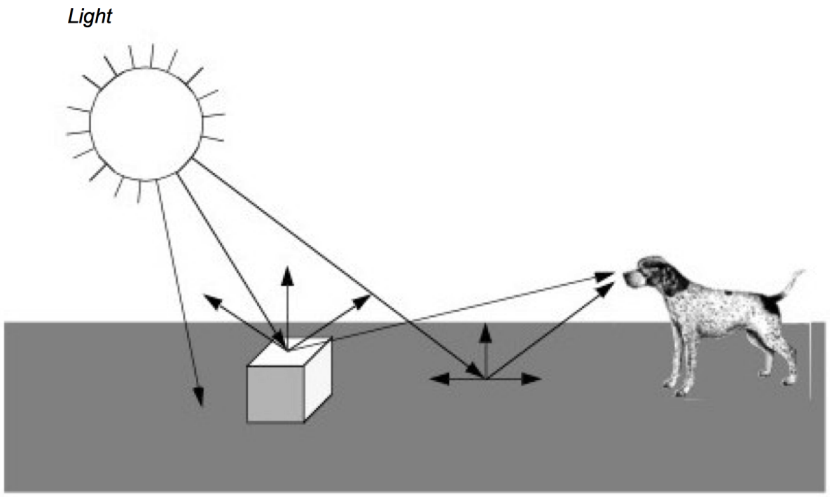


Figure 3.1: Physical generation of an image.

Figure 3.1 presents a simplified view of what happens when we look at an object, in this case a cube. Rays of light are emitted from a light source in all directions. (In this example we assume that the light source is the sun.) Some of these rays happen to strike the cube whose surface absorbs some of the incident light and reflects the rest of it. Some of this reflected light may head towards us and enter our eyes. If this happens, then we "see" the object. Likewise, some of the light from the sun will strike the ground and some small percentage of it will be reflected into our eyes.

As you can imagine, the chances of a ray of light travelling from the sun through space to hit a small object on a relatively small planet are low. This is compounded by the slim odds that the ray of light will reflect off the object and into our eyes. The only reason we can see is that the sun produces such an enormous amount of light that it overwhelms the odds. While this may work in real life, trying to simulate it with a computer can be difficult. Fortunately, there are other ways to look at this problem.

A common and effective technique for 3D computer graphics is called *ray-tracing* or *ray-casting*. Ray-tracing simulates the interaction of light with objects by following the path of each light ray. Typically, we follow the ray backwards from the viewer's eyes and into the world to determine what the ray strikes. The direction of the ray is in the direction we are looking (i.e., the view direction) including effects of perspective (if desired). When a ray intersects an object, we can determine if that point is being lit by our light source. This is done by tracing a ray from the point of intersection towards the light. If the ray intersects the light, then the point is being lit. If the ray intersects something else before it gets to the light, then that light will not contribute to illuminating the point. For multiple light sources we just repeat this process for each light source. The total contributions from all the light sources, plus any ambient scattered light, will determine the total lighting or shadow for that point. By following the light's path backwards, ray tracing only looks at rays that end up entering the viewer's eyes. This dramatically reduces the number of rays that must be computed by a simulation program.

Having described ray tracing as a rendering process, it may be surprising that many members of the graphics community do not use it. This is because ray tracing is a relatively slow image generation method since it is typically implemented in software. Other graphics techniques have

been developed that generate images using dedicated computer hardware. To understand why this situation has emerged, it is instructive to briefly examine the taxonomy and history of computer graphics.

3.2.1 Image-Order and Object-Order Methods

Rendering processes can be broken into two categories: *image-order* and *object-order*. Ray tracing is an image-order process. It works by determining what happens to each ray of light, one at a time. An object-order process works by rendering each object, one at a time. In the above example, an object-order technique would proceed by first rendering the ground and then the cube.

To look at it another way consider painting a picture of a barn. Using an image-order algorithm you would start at the upper left corner of the canvas and put down a drop of the correct color paint. (Each paint drop is called a picture element or *pixel*.) Then you would move a little to the right and put down another drop of paint. You would continue until you reached the right edge of the canvas, then you would move down a little and start on the next row. Each time you put down a drop of paint you make certain it is the correct color for each pixel on the canvas. When you are done you will have a painting of a barn.

An alternative approach is based on the more natural (at least for many people) object-order process. We work by painting the different objects in our scene, independent of where the objects actually are located on the scene. We may paint from back to front, front-to-back, or in arbitrary order. For example, we could start by painting the sky and then add in the ground. After these two objects were painted we would then add in the barn. In the image-order process we worked on the canvas in a very orderly fashion; left to right, top to bottom. With an object-order process we tend to jump from one part of the canvas to another, depending on what object we are drawing.

The field of computer graphics started out using object-order processes. Much of the early work was closely tied to the hardware display device, initially a vector display. This was little more than an oscilloscope, but it encouraged graphical data to be drawn as a series of line segments. As the original vector displays gave way to the currently ubiquitous raster displays, the notion of representing graphical data as a series of objects to be drawn was preserved. Much of the early work pioneered by Bresenham [1] at IBM focused on how to properly convert line segments into a form that would be suitable for line plotters. The same work was applied to the task of rendering lines onto the raster displays that replaced the oscilloscope. Since then the hardware has become more powerful and capable of displaying much more complex primitives than lines.

It wasn't until the early 1980s that a paper by Turner Whitted [7] prompted many people to look at rendering from a more physical perspective. Eventually ray tracing became a serious competitor to the traditional object-order rendering techniques, due in part to the highly realistic images it can produce. Object-order rendering has maintained its popularity because there is a wealth of graphics hardware designed to quickly render objects. Ray tracing tends to be done without any specialized hardware and therefore is a time-consuming process.

3.2.2 Surface versus Volume Rendering

The discussion to this point in the text has tacitly assumed that when we render an object, we are viewing the surfaces of objects and their interactions with light. However, common objects such as clouds, water, and fog, are translucent, or scatter light that passes through them. Such objects cannot be rendered using a model based exclusively on surface interactions. Instead, we need to consider the changing properties inside the object to properly render them. We refer to these two rendering models as *surface rendering* (i.e., render the surfaces of an object) and *volume rendering* (i.e., render the surface and interior of an object).

Generally speaking, when we render an object using surface rendering techniques, we mathematically model the object with a surface description such as points, lines, triangles, polygons, or 2D and 3D splines. The interior of the object is not described, or only implicitly represented from the surface representation (i.e., surface is the boundary of the volume). Although techniques do exist that allow us to make the surface transparent or translucent, there are still many phenomena that cannot be simulated using surface rendering techniques alone (e.g., scattering or light emission). This is particularly true if we are trying to render data interior to an object, such as X-ray intensity from a CT scan.

Volume rendering techniques allow us to see the inhomogeneity inside objects. In the prior CT example, we can realistically reproduce X-ray images by considering the intensity values from both the surface and interior of the data. Although it is premature to describe this process at this point in the text, you can imagine extending our ray tracing example from the previous section. Thus rays not only interact with the surface of an object, they also interact with the interior.

In this chapter we focus on surface rendering techniques. While not as powerful as volume rendering, surface rendering is widely used because it is relatively fast compared to volumetric techniques, and allows us to create images for a wide variety of data and objects. Chapter 7 describes volume rendering in more detail.

3.2.3 Visualization Not Graphics

Although the authors would enjoy providing a thorough treatise on computer graphics, such a discourse is beyond the scope of this text. Instead we make the distinction between visualization (exploring, transforming, and mapping data) and computer graphics (mapping and rendering). The focus will be on the principles and practice of visualization, and not on 3D computer graphics. In this chapter and Chapter 7 we introduce basic concepts and provide a working knowledge of 3D computer graphics. For those more interested in this field, we refer you to the texts recommended in the "Bibliographic Notes" on page 70 at the end of this chapter.

One of the regrets we have regarding this posture is that certain rendering techniques are essentially visualization techniques. We see this hinted at in the previous paragraph, where we use the term "mapping" to describe both visualization and computer graphics. There is not currently and will likely never be a firm distinction between visualization and graphics. For example, many researchers consider volume rendering to be squarely in the field of visualization because it addresses one of the most important forms of visualization data. Our distinction is mostly for our own convenience, and offers us the opportunity to finish this text. We recommend that a serious student of visualization supplement the material presented here with deeper books on computer graphics and volume rendering.

In the next few pages we describe the rendering process in more detail. We start by describing several color models. Next we examine the primary components of the rendering process. There are sources of light such as the sun, objects we wish to render such as a cube or sphere (we refer to these objects as actors), and there is a camera that looks out into the world. These terms are taken from the movie industry and tend to be familiar to most people. Actors represent graphical data or objects, lights illuminate the actors, and the camera constructs a picture by projecting the actors onto a view plane. We call the combination of lights, camera, and actors the scene, and refer to the rendering process as rendering the scene.

3.3 Color

The electromagnetic spectrum visible to humans contains wavelengths ranging from about 400 to 700 nanometers. The light that enters our eyes consists of different *intensities* of these

wavelengths, an example of which is shown in Figure 3.2. This intensity plot defines the color of the light, therefore a different plot results in a different color. Unfortunately, we may not notice the difference since the human eye throws out most of this information. There are three types of color receptors in the human eye called *cones*. Each type responds to a subset of the 400 to 700 nanometer wave-length range as shown in Figure 3.3. Any color we see is encoded by our eyes into these three overlapping responses. This is a great reduction from the amount of information that actually comes into our eyes. As a result, the human eye is incapable of recognizing differences in any colors whose intensity curves, when applied to the human eye's response curves, result in the same triplet of responses. This also implies that we can store and represent colors in a computer using a simplified form without the human eye being able to recognize the difference.

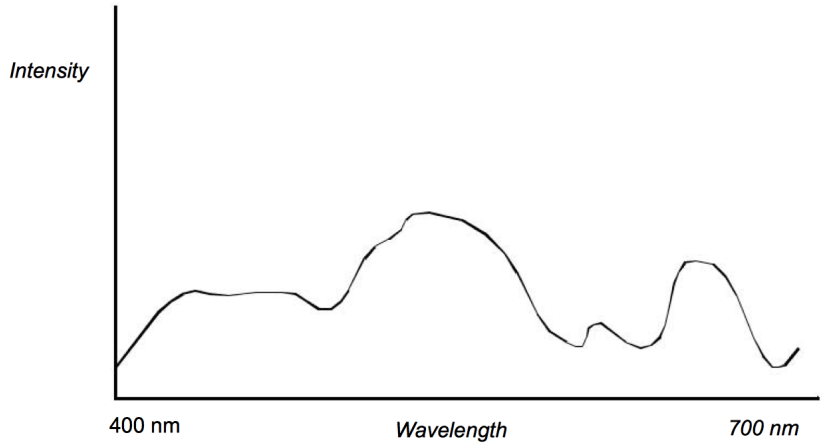


Figure 3.2: Wavelength versus Intensity plot.

The two simplified component systems that we use to describe colors are RGB and HSV color systems. The RGB system represents colors based on their red, green, and blue intensities. This can be thought of as a three dimensional space with the axes being red, green, and blue. Some common colors and their RGB components are shown in Table 3.1.

The HSV system represents colors based on their hue, saturation, and value. The value component is also known as the brightness or intensity component, and represents how much light is in the color. A value of 0.0 will always give you black and a value of 1.0 will give you something bright. The hue represents the dominant wavelength of the color and is often illustrated using a circle as in Figure 3.4. Each location on the circumference of this circle represents a different hue and can be specified using an angle. When we specify a hue we use the range from zero to one, where zero corresponds to zero degrees on the hue circle and one corresponds to 360 degrees. The saturation indicates how much of the hue is mixed into the color. For example, we can set the value to one, which gives us a bright color, and the hue to 0.66, to give us a dominant wavelength of blue. Now if we set the saturation to one, the color will be a bright primary blue. If we set the saturation to 0.5, the color will be sky blue, a blue with more white mixed in. If we set the saturation to zero, this indicates that there is no more of the dominant wavelength (hue) in the color than any other wavelength. As a result, the final color will be white (regardless of hue value). Table 3.1 lists HSV values for some common colors.

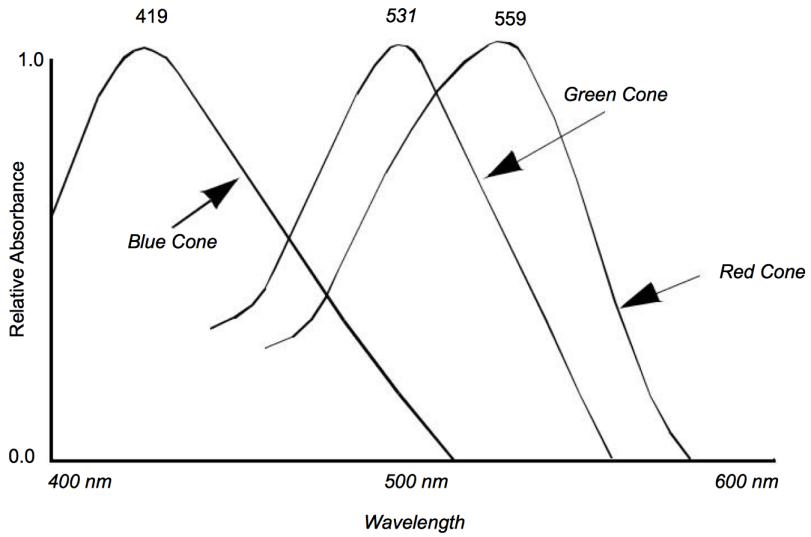


Figure 3.3: Relative absorbance of light by the three types of cones in the human retina.

3.4 Lights

One of the major factors controlling the rendering process is the interaction of light with the actors in the scene. If there are no lights, the resulting image will be black and rather uninformative. To a great extent it is the interaction between the emitted light and the surface (and in some cases the interior) of the actors in the scene that defines what we see. Once rays of light interact with the actors in a scene, we have something for our camera to view.

Of the many different types of lights used in computer graphics, we will discuss the simplest, the infinitely distant, point light source. This is a simplified model compared to the lights we use at home and work. The light sources that we are accustomed to typically radiate from a region in space (a filament in an incandescent bulb, or a light-emitting gas in a fluorescent light). The point source lighting model assumes that the light is emitted in all directions from a single point

Color	RGB	HSV
Black	$0, 0, 0$	$*, *, 0$
White	$1, 1, 1$	$*, 0, 1$
Red	$1, 0, 0$	$0, 1, 1$
Green	$0, 1, 0$	$1/3, 1, 1$
Blue	$0, 0, 1$	$2/3, 1, 1$
Cyan	$0, 1, 1$	$1/2, 1, 1$
Magenta	$1, 0, 1$	$5/6, 1, 1$
Yellow	$1, 1, 0$	$1/6, 1, 1$
Sky Blue	$1/2, 1/2, 1$	$2/3, 1/2, 1$

Table 3.1: Common colors in RGB and HSV space

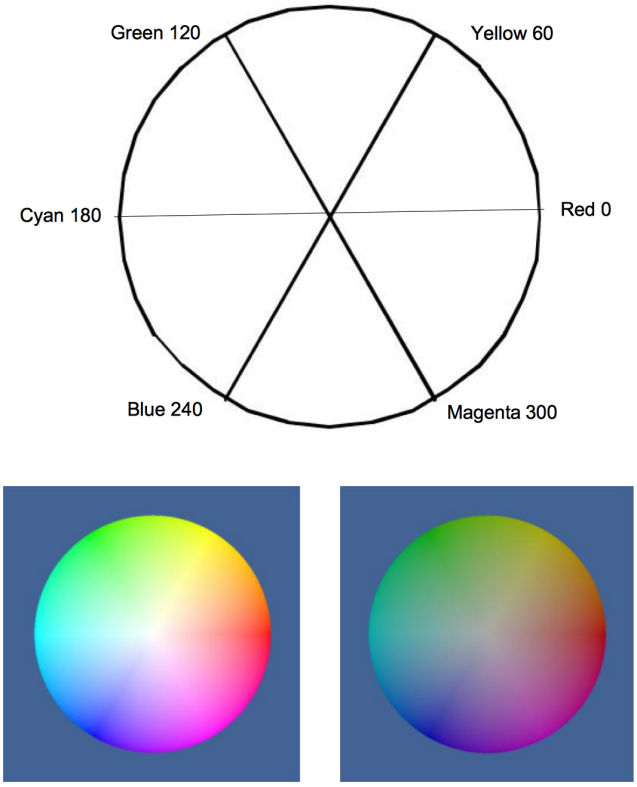


Figure 3.4: On the top, circular representation of hue. The other two images on the bottom are slices through the HSV color space. The first slice has a value of 1.0, the other has a value of 0.5.

in space. For an infinite light source, we assume that it is positioned infinitely far away from what it is illuminating. This is significant because it implies that the incoming rays from such a source will be parallel to each other. The emissions of a local light source, such as a lamp in a room, are not parallel. Figure 3.5 illustrates the differences between a local light source with a finite volume, versus an infinite point light source. The intensity of the light emitted by our infinite light sources also remains constant as it travels, in contrast to the actual $1/distance^2$ relationship physical lights obey. As you can see this is a great simplification, which later will allow us to use less complex lighting equations.

3.5 Surface Properties

As rays of light travel through space, some of them intersect our actors. When this happens, the rays of light interact with the surface of the actor to produce a color. Part of this resulting color is actually not due to direct light, but rather from *ambient* light that is being reflected or scattered from other objects. An ambient lighting model accounts for this and is a simple approximation of the complex scattering of light that occurs in the real world. It applies the intensity curve of the light source to the color of the object, also expressed as an intensity curve. The result is the color of the light we see when we look at that object. With such a model, it is important to realize that a

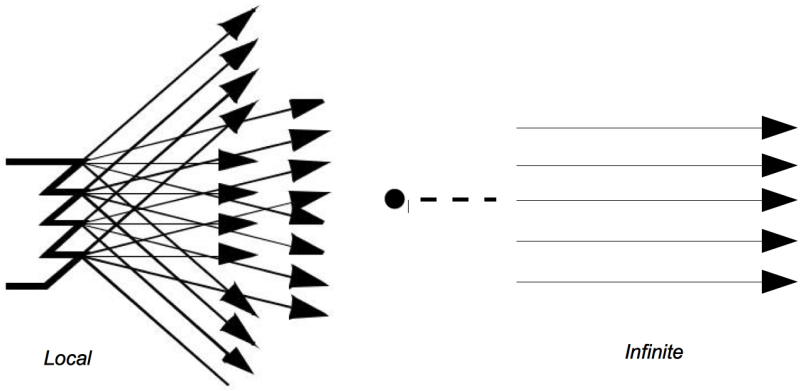


Figure 3.5: Local light source with a finite volume versus an infinite point light source.

white light shining on a blue ball is indistinguishable from a blue light shining on a white ball. The ambient lighting equation is

$$R_a = L_c \cdot O_a \tag{3.1}$$

where R_a is the resulting intensity curve due to ambient lighting, L_c is the intensity curve of the ambient light, and O_a is the color curve of the object. To help keep the equations simple we assume that all of the direction vectors are normalized (i.e., have a magnitude of one).

Two components of the resulting color depend on direct lighting. *Diffuse lighting*, which is also known as Lambertian reflection, takes into account the angle of incidence of the light onto an object. Figure 3.6 shows the image of a cylinder that becomes darker as you move laterally from its center. The cylinder's color is constant; the amount of light hitting the surface of the cylinder changes. At the center, where the incoming light is nearly perpendicular to the surface of the cylinder, it receives more rays of light per surface area. As we move towards the side, this drops until finally the incoming light is parallel to the side of the cylinder and the resulting intensity is zero.

The contribution from diffuse lighting is expressed in Equation 3.2 and illustrated in Figure 3.7.

$$R_d = L_c O_d[\overrightarrow{O}_n \cdot (-\overrightarrow{L}_n)] \tag{3.2}$$

where R_d is the resulting intensity curve due to diffuse lighting, L_c is the intensity curve for the light, and O_c is the color curve for the object. Notice that the diffuse light is a function of the relative angle between incident light vector and $\overrightarrow{L}_n$ and the surface normal of the object $\overrightarrow{O}_n$. As a result diffuse lighting is independent of viewer position.

Specular lighting represents direct reflections of a light source off a shiny object. Figure 3.9 shows a diffusely lit ball with varying specular reflection. The specular intensity (which varies between the top and bottom rows) controls the intensity of the specular lighting. The specular power, $O_s p$, indicates how shiny an object is, more specifically it indicates how quickly specular reflections diminish as the reflection angles deviate from a perfect reflection. Higher values indicate a faster drop off, and therefore a shinier surface. Referring to Figure 3.8, the equation for specular lighting is

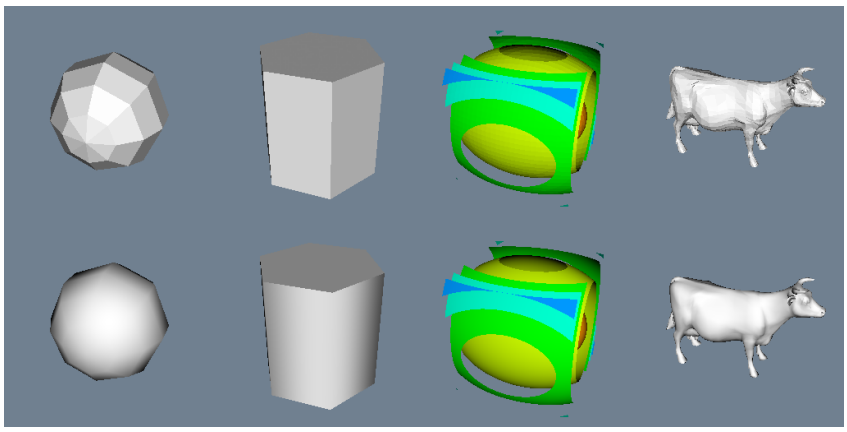


Figure 3.6: Flat and Gouraud shading. Different shading methods can dramatically improve the look of an object represented with polygons. On the top, flat shading uses a constant surface normal across each polygon. On the bottom, Gouraud shading interpolates normals from polygon vertices to give a smoother look.

$$R_s. = L_c O_s[\vec{S} \cdot (-\vec{C}_n)]^{O_{sp}} \vec{S} = 2[\vec{O}_n \cdot (-\vec{L}_n)]\vec{O}_n + \vec{L}_n \tag{3.3}$$

where $\vec{C}_n$ is the direction of projection for the camera and $\vec{S}$ is the direction of specular reflection.

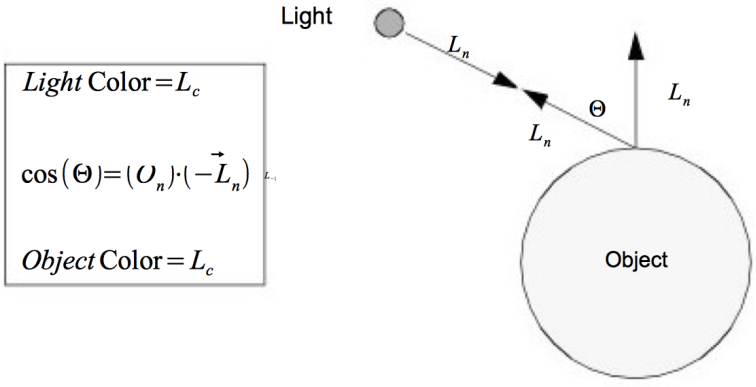


Figure 3.7: Diffuse lighting

We have presented the equations for the different lighting models independently. We can apply all lighting models simultaneously or in combination. Equation 3.4 combines ambient, diffuse and specular lighting into one equation.

$$R_c = O_{ai} O_{ac} L_c - O_{di} O_{dc} L_c (\vec{O}_n \cdot \vec{L}_n) + O_{si} O_{sc} L_c [\vec{S} \cdot (-\vec{C}_n)]^{O_{sp}} \tag{3.4}$$

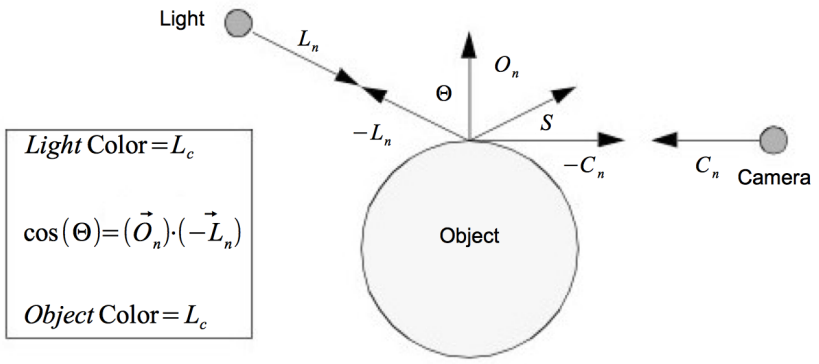


Figure 3.8: Specular lighting.

The result is a color at a point on the surface of the object. The constants O_{ai}, O_{di}, and O_{si} control the relative amounts of ambient, diffuse and specular lighting for an object. The constants O_{ac} O_{dc} and O_{sc} specify the colors to be used for each type of lighting. These six constants along with the specular power are part of the surface material properties. (Other properties such as transparency will be covered in later sections of the text.) Different combinations of these property values can simulate dull plastic and polished metal. The equation assumes an infinite point light source as described in Lights on 3.4. However the equation can be easily modified to incorporate other types of directional lighting.

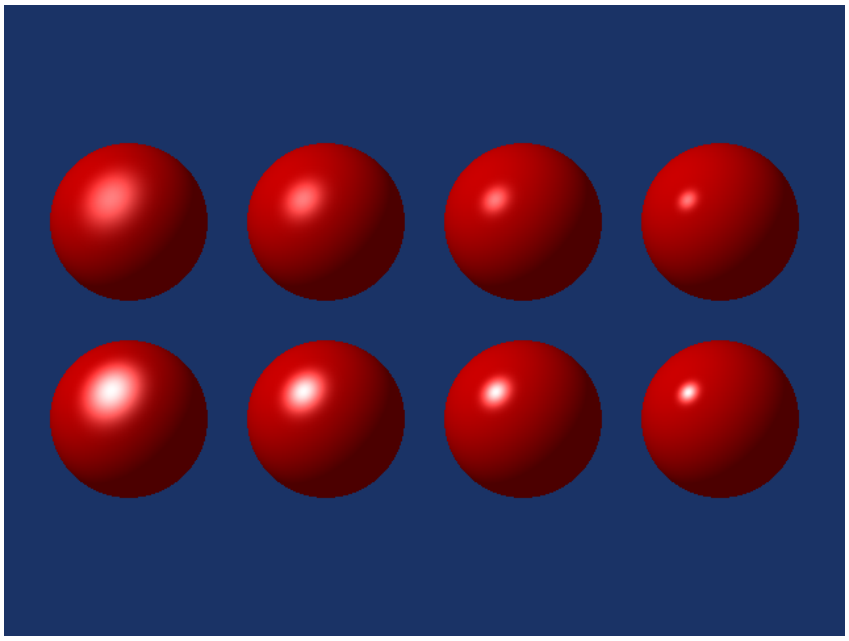


Figure 3.9: Effects of specular coefficients. Specular coefficients control the apparent "shininess" of objects. The top row has a specular intensity value of 0.5; the bottom row 1.0. Along the horizontal direction the specular power changes. The values (from left to right) are 5, 10, 20, and 40.

3.6 Cameras

We have light sources that are emitting rays of light and actors with surface properties. At every point on the surface of our actors this interaction results in some composite color (i.e., combined color from light, object surface, specular, and ambient effects). All we need now to render the scene is a camera. There are a number of important factors that determine how a 3D scene gets projected onto a plane to form a 2D image (see Figure 3.10). These are the position, orientation, and focal point of the camera, the method of camera projection , and the location of the camera clipping planes.

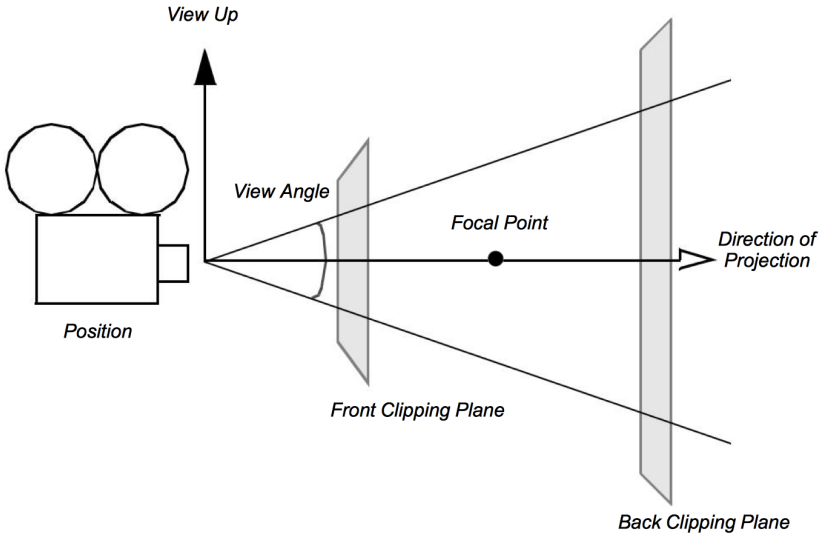


Figure 3.10: Camera attributes.

The position and focal point of the camera define the location of the camera and where it points. The vector defined from the camera position to the focal point is called the direction of projection . The camera image plane is located at the focal point and is typically perpendicular to the projection vector. The camera orientation is controlled by the position and focal point plus the camera view-up vector. Together these completely define the camera view.

The method of projection controls how the actors are mapped to the image plane. *Orthographic projection* is a parallel mapping process. In orthographic projection (or parallel projection) all rays of light entering the camera are parallel to the projection vector. *Perspective projection* occurs when all light rays go through a common point (i.e., the viewpoint or center of projection). To apply perspective projection we must specify a perspective angle or camera view angle.

The front and back clipping planes intersect the projection vector, and are usually perpendicular to it. The clipping planes are used to eliminate data either too close to the camera or too far away. As a result only actors or portions of actors within the clipping planes are (potentially) visible. Clipping planes are typically perpendicular to the direction of projection. Their locations can be set using the cameras clipping range. The location of the planes are measured from the cameras position along the direction of projection. The front clipping plane is at the minimum range value, and the back clipping plane is at the maximum range value. Later on in Chapter 7 , when we discuss stereo rendering, we will see examples of clipping planes that are not perpendicular to the direction of projection.

Taken together these camera parameters define a rectangular pyramid, with its apex at the

cameras position and extending along the direction of projection. The pyramid is truncated at the top with the front clipping plane and at the bottom by the back clipping plane. The resulting view frustum defines the region of 3D space visible to the camera.

While a camera can be manipulated by directly setting the attributes mentioned above, there are some common operations that make the job easier. Figure 3.11 and Figure 3.12 will help illustrate these operations. Changing the azimuth of a camera rotates its position around its view up vector, centered at the focal point . Think of this as moving the camera to the left or right while always keeping the distance to the focal point constant. Changing a cameras elevation rotates its position around the cross product of its direction of projection and view up centered at the focal point. This corresponds to moving the camera up and down. To roll the camera, we rotate the view up vector about the view plane normal. Roll is sometimes called twist.

The next two motions keep the cameras position constant and instead modify the focal point. Changing the yaw rotates the focal point about the view up centered at the cameras position. This is like an azimuth, except that the focal point moves instead of the position. Changes in pitch rotate the focal point about the cross product of the direction of projection and view up centered at the cameras position. Dollying in and out moves the cameras position along the direction of projection, either closer or farther from the focal point. This operation is specified as the ratio of its current distance to its new distance. A value greater than one will dolly in, while a value less than one will dolly out. Finally, zooming changes the cameras view angle, so that more or less of the scene falls within the view frustum.

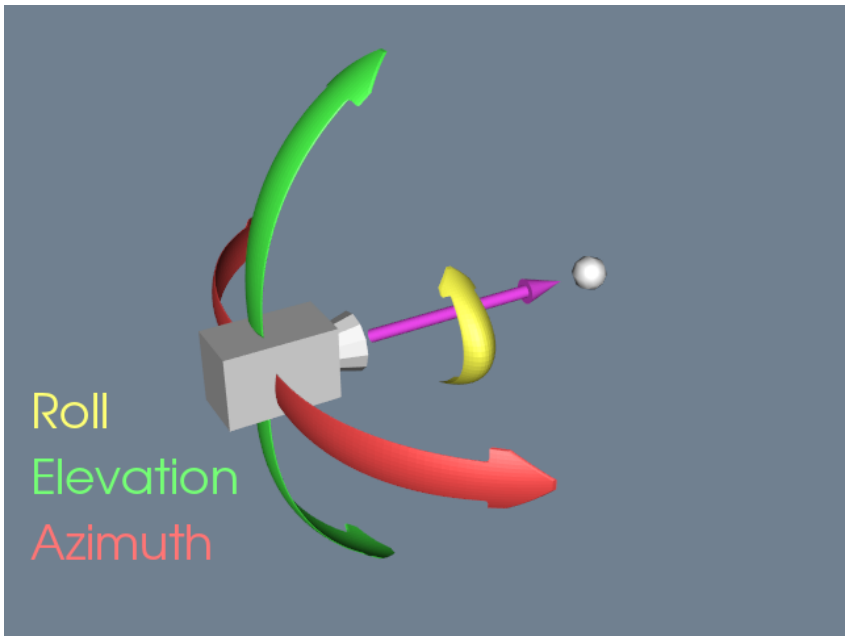


Figure 3.11: Camera movements around the focal point.

Once we have the camera situated, we can generate our 2D image. Some of the rays of light traveling through our 3D space will pass through the lens on the camera. These rays then strike a flat surface to produce an image. This effectively projects our 3D scene into a 2D image. The cameras position and other properties determine which rays of light get captured and projected. More specifically, only rays of light that intersect the cameras position, and are within its viewing frustum, will affect the resulting 2D image.

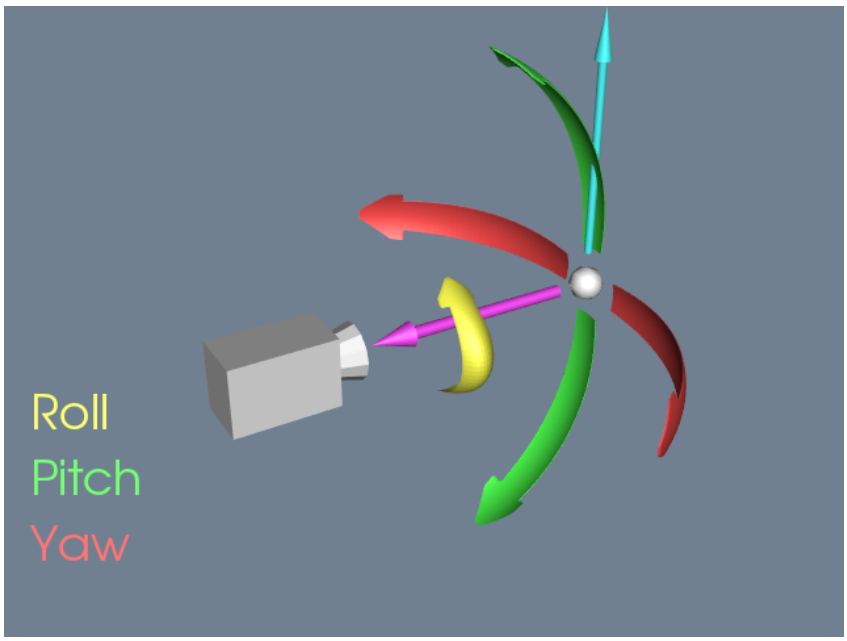


Figure 3.12: Camera movements around the camera position.

This concludes our brief rendering overview. The light has traveled from its sources to the actors, where it is reflected and scattered. Some of this light gets captured by the camera and produces a 2D image. Now we will look at some of the details of this process.

3.7 Coordinate Systems

There are four coordinate systems commonly used in computer graphics and two different ways of representing points within them (Figure 3.13). While this may seem excessive, each one serves a purpose. The four coordinate systems we use are: *model*, *world* , *view*, and *display*.

The model coordinate system is the coordinate system in which the model is defined, typically a local Cartesian coordinate system. If one of our actors represents a football, it will be based on a coordinate system natural to the football geometry (e.g., a cylindrical system). This model has an inherent coordinate system determined by the decisions of whoever generated it. They may have used inches or meters as their units, and the football may have been modelled with any arbitrary axis as its major axis.

The world coordinate system is the 3D space in which the actors are positioned. One of the actors responsibilities is to convert from the models coordinates into world coordinates. Each model may have its own coordinate system but there is only one world coordinate system. Each actor must scale, rotate, and translate its model into the world coordinate system. (It may also be necessary for the modeller to transform from its natural coordinate system into a local Cartesian system. This is because actors typically assume that the model coordinate system is a local Cartesian system.) The world coordinate system is also the system in which the position and orientation of cameras and lights are specified.

The view coordinate system represents what is visible to the camera. This consists of a pair of x and y values, ranging between (-1,1), and a z depth coordinate. The x, y values specify location in the image plane, while the z coordinate represents the distance, or range, from the camera. The

cameras properties are represented by a four by four transformation matrix (to be described shortly), which is used to convert from world coordinates into view coordinates. This is where the perspective effects of a camera are introduced.

The display coordinate system uses the same basis as the view coordinate system, but instead of using negative one to one as the range, the coordinates are actual x, y pixel locations on the image plane. Factors such as the windows size on the display determine how the view coordinate range of (-1,1) is mapped into pixel locations. This is also where the viewport comes into effect.

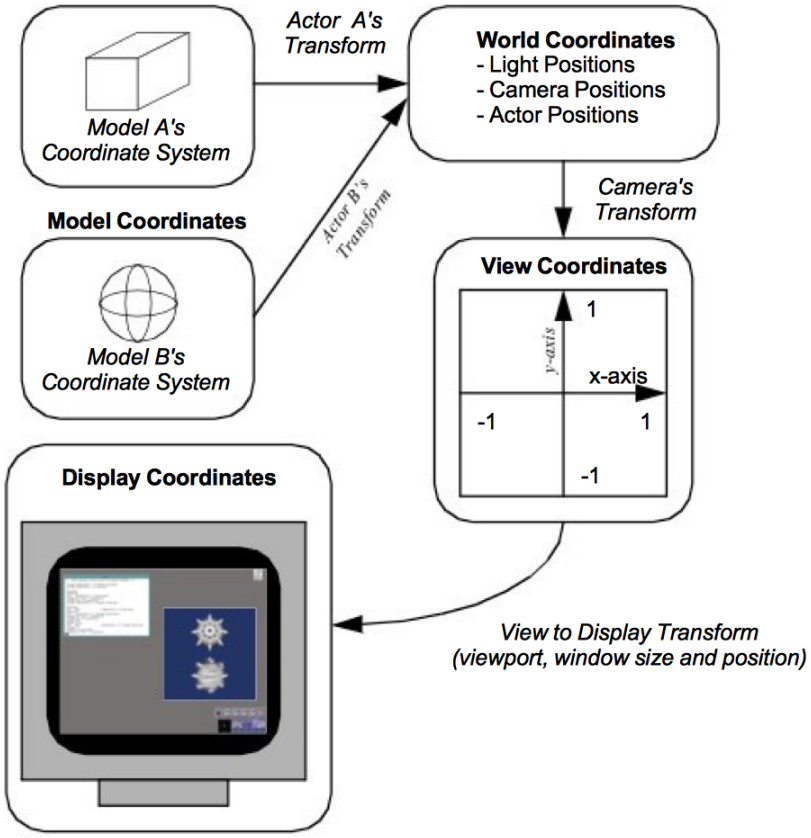


Figure 3.13: Modelling, world, view and display coordinate system.

You may want to render two different scenes, but display them in the same window. This can be done by dividing the window into rectangular viewports. Then, each renderer can be told what portion of the window it should use for rendering. The viewport ranges from (0,1) in both the x and y axis. Similar to the view coordinate system, the z-value in the display coordinate system also represents depth into the window. The meaning of this z-value will be further described in the section titled Z-Buffer on page 49.

3.8 Coordinate Transformation

When we create images with computer graphics, we project objects defined in three dimensions onto a two-dimensional image plane. As we saw earlier, this projection naturally includes

perspective. To include projection effects such as vanishing points we use a special coordinate system called homogeneous coordinates.

The usual way of representing a point in 3D is the three element Cartesian vector (x, y, z). Homogeneous coordinates are represented by a four element vector $(x_h h, y_h, z_h, w_h)$. The conversion between Cartesian coordinates and homogeneous coordinates is given by:

$$x = \frac{x_h}{w_h} \quad x = \frac{y_h}{w_h} \quad x = \frac{z_h}{w_h} \tag{3.5}$$

Using homogeneous coordinates we can represent an infinite point by setting w h to zero. This capability is used by the camera for perspective transformations. The transformations are applied by using a 4x4 transformation matrix . Transformation matrices are widely used in computer graphics because they allow us to perform translation, scaling, and rotation of objects by repeated matrix multiplication. Not all of these operations can be performed using a 3x3 matrix.

For example, suppose we wanted to create a transformation matrix that translates a point (x, y, z) in Cartesian space by the vector (t_x, t_y, t_z). We need only construct the translation matrix given by

$$T_T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

and then postmultiply it with the homogeneous coordinate (x_h, y_h, z_h, w_h). To carry this example through, we construct the homogeneous coordinate from the Cartesian coordinate (x, y, z) by setting $w_h = 1$ to yield $(x, y, z, 1)$. Then to determine the translated point (x', y', z') we premultiply current position by the transformation matrix T_T to yield the translated coordinate. Substituting into Equation 3.6 we have the result

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.7}$$

Converting back to Cartesian coordinates via Equation 3.5 we have the expected solution

$$x' = x + t_x y' = y + t_y z' = z + t_z \tag{3.8}$$

The same procedure is used to scale or rotate an object. To scale an object we use the transformation matrix

$$T_s = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

where the parameters s_x, s_y, and s_z are scale factors along the x, y, z axes. Similarly we can rotate an object around the x axes by angle θ using the matrix

$$T_{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

Around the y axis we use

$$TT_{R_y} = \begin{bmatrix} \cos\theta & 0 & \sin\theta 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

and around the z axis we use

$$T_{R_z} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

Another useful rotation matrix is used to transform one coordinate axes $x - y - z$ to another coordinate axes $x' - y' - z'$. To derive the transformation matrix we assume that the unit x'. To derive the transformation matrix we assume that the unit x' axis make the angles $(\theta_{x'x}, \theta_{x'y}, \theta_{x'z})$ around the x-y-z axes (these are called direction cosines). Similarly, the unit y' axis makes the angles $(\theta_{y'x}, \theta_{y'y}, \theta_{y'z})$ and the unit z' axis makes the angles $(\theta_{z'x}, \theta_{z'y}, \theta_{z'z})$. The resulting rotation matrix is formed by placing the direction cosines along the rows of the transformation matrix as follows

$$T_R = \begin{bmatrix} \cos\theta_{x'x} & \cos\theta_{x'y} & \cos\theta_{x'z} & 0 \\ \cos\theta_{y'x} & \cos\theta_{z'y} & \cos\theta_{y'z} & 0 \\ \cos\theta_{z'x} & \cos\theta_{z'y} & \cos\theta_{z'z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

Rotations occur about the coordinate origin. It is often more convenient to rotate around the center of the object (or a user-specified point). Assume that we call this point the objects center . To rotate around we Oc must first translate the object to the Oorigin, apply rotations, and then translate the object back.

Transformation matrices can be combined by matrix multiplication to achieve combinations of translation, rotation, and scaling. It is possible for a single transformation matrix to represent all types of transformation simultaneously. This matrix is the result of repeated matrix multiplications. A word of warning: The order of the multiplication is important. For example, multiplying a translation matrix by a rotation matrix will not yield the same result as multiplying the rotation matrix by the translation matrix.

3.9 Actor Geometry

We have seen how lighting properties control the appearance of an actor, and how the camera in combination with transformation matrices is used to project an actor to the image plane. What is left to define is the geometry of the actor, and how we position it in the world coordinate system.

3.9.1 Modelling

A major topic in the study of computer graphics is modelling or representing the geometry of physical objects. Various mathematical techniques have been applied including combinations of points, lines, polygons, curves, and splines of various forms, and even implicit mathematical functions.

This topic is beyond the scope of the text. The important point here is that there is an underlying geometric model that specifies what the object's shape is and where it is located in the model coordinate system.

In data visualization, modelling takes a different role. Instead of directly creating geometry to represent an object, visualization algorithms *compute* these forms. Often the geometry is abstract (like a contour line) and has little relationship to real world geometry. We will see how these models are computed when we describe visualization algorithms in Chapter 6 and Chapter 9.

The representation of geometry for data visualization tends to be simple, even though computing the representations is not. These forms are most often primitives like points, lines, and polygons, or visualization data such as volume data. We use simple forms because we desire high performance and interactive systems. Thus we take advantage of computer hardware (to be covered in "Graphics Hardware" on 3.10) or special rendering techniques like volume rendering (see "Volume Rendering" on 7.3).

3.9.2 Actor Location and Orientation

Every actor has a transformation matrix that controls its location and scaling in world space. The actor's geometry is defined by a model in model coordinates. We specify the actor's location using orientation, position, and scale factors along the coordinate axes. In addition, we can define an origin around which the actor rotates. This feature is useful because we can rotate the actor around its center or some other meaningful point.

The orientation of an actor is determined by rotations stored in an orientation vector (Ox,Oy, Oz). This vector defines a series of rotational transformation matrices. As we saw in the previous section on transformation matrices, the order of application of the transformations is not arbitrary. We have chosen a fixed order based on what we think is natural to users. The order of transformation is a rotation by O y around the y axis, then by around Ox the x axis, and finally by O z around the z axis. This ordering is arbitrary and is based on the standard camera operations. These operations (in order) are a camera azimuth, followed by an elevation, and then a roll (Figure3.14).

All of these rotations take place around the origin of the actor. Typically this is set to the center of its bounding box, but it can be set to any convenient point. There are many different methods for changing an actors orientation. RotateX(), RotateY(), and RotateZ() are common methods that rotate about their respective axes. Many systems also include a method to rotate about a userdefined axis. In the Visualization Toolkit the RotateXYZ() method is used to rotate around an arbitrary vector passing through the origin.

3.10 Graphics Hardware

Earlier we mentioned that advances in graphics hardware have had a large impact on how rendering is performed. Now that we have covered the fundamentals of rendering a scene, we look at some of the hardware issues. First, we discuss raster devices that have replaced vector displays as the primary output device. Then, we look at how our programs communicate to the graphics hardware. We also examine the different coordinate systems used in computer graphics, hidden line/surface removal, and z-buffering.

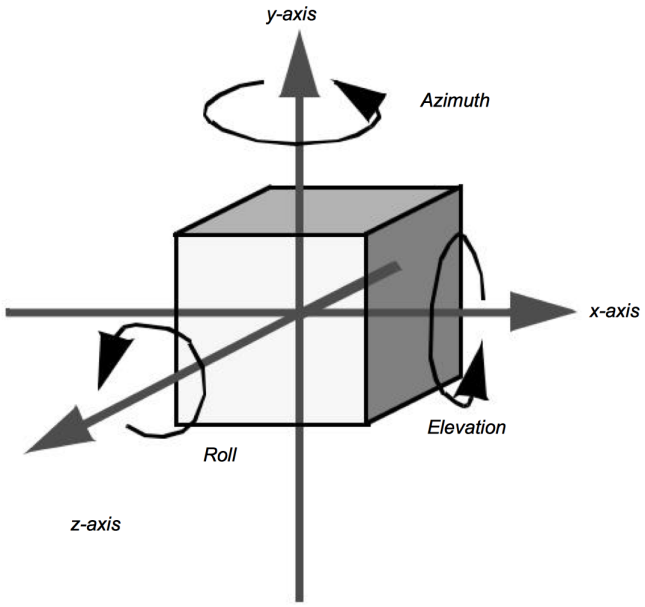


Figure 3.14: Actor coordinate system.

3.10.1 Raster Devices

The results of computer graphics is pervasive in today's world—digital images (generated with computer graphics) may be found on cell phones, displayed on computer monitors, broadcast on TV, shown at the movie theatre and presented on electronic billboards. All of these, and many more, display mediums are raster devices. A raster device represents an image using a two dimensional array of picture elements called pixels. For example, the word "hello" can be represented as an array of pixels. as shown in Figure 3.15. Here the word "hello" is written within a pixel array that is twenty-five pixels wide and ten pixels high. Each pixel stores one bit of information, whether it is black or white. This is how a black and white laser printer works, for each point on the paper it either prints a black dot or leaves it the color of the paper. Due to hardware limitations, raster devices such as laser printers and computer monitors do not actually draw accurate square pixels like those in Figure 3.15. Instead, they tend to be slightly blurred and overlapping. Another hardware limitation of raster devices is their resolution. This is what causes a 300 dpi (dots per inch) laser printer to produce more detailed output than a nine pin dot matrix printer. A 300 dpi laser printer has a resolution of 300 pixels per inch compared to roughly 50 dpi for the dot matrix printer.

Color computer monitors typically have a resolution of about 80 pixels per inch, making the screen a pixel array roughly one thousand pixels in width and height. This results in over one million pixels, each with a value that indicates what color it should be. Since the hardware in color monitors uses the RGB system, it makes sense to use that to describe the colors in the pixels. Unfortunately, having over one million pixels, each with a red, green, and blue component, can take up a lot of memory. This is part of what differentiates the variety of graphics hardware on the market. Some companies use 24 bits of storage per pixel, others use eight, some advanced systems use more than 100 bits of storage per pixel. Typically, the more bits per pixel the more accurate the colors will be.

One way to work around color limitations in the graphics hardware is by using a technique called *dithering*. Say, for example, that you want to use some different shades of gray, but your graphics hardware only supports black and white. Dithering lets you approximate shades of gray by

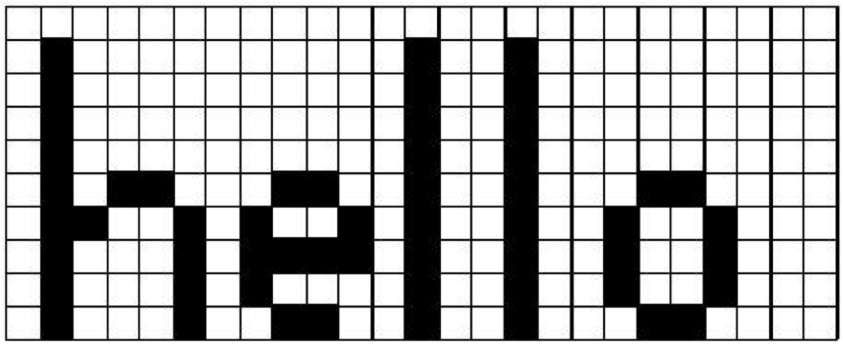


Figure 3.15: A pixel array for the word "hello".

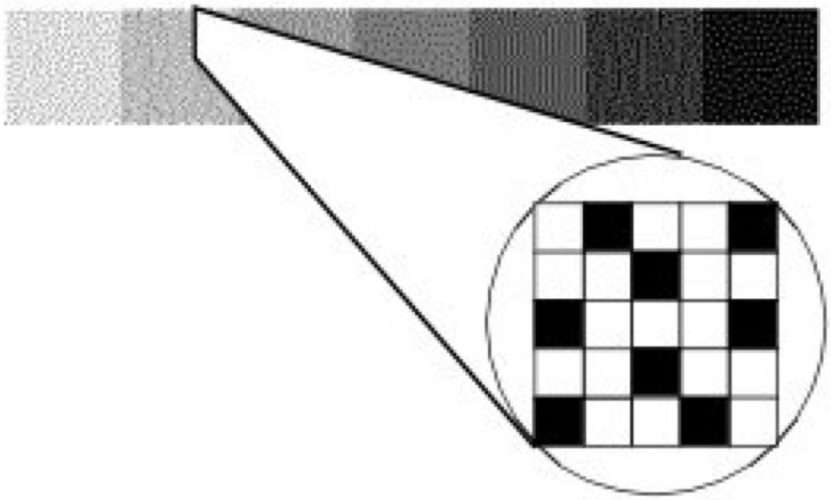


Figure 3.16: Black and white dithering.

using a mixture of both black and white pixels. In Figure 3.16, seven gray squares are drawn using a mixture of black and white pixels. From a distance the seven squares look like different shades of gray even though up close, it's clear that they are just different mixtures of black and white pixels. This same technique works just as well for other colors. For example, if your graphics hardware supports primary blue, primary green, and white but not a pastel sea green, you can approximate this color by dithering the green, blue, and white that the hardware does support.

3.10.2 Interfacing to the Hardware

Now that we have covered the basics of display hardware, the good news is that you rarely need to worry about them. Most graphics programming is done using higher-level primitives than individual pixels. Figure 3.17 shows a typical arrangement for a visualization program. At the bottom of the hierarchy is the display hardware that we already discussed; chances are your programs will not interact directly with it. The top three layers above the hardware are the layers you may need to be concerned with.

Many programs take advantage of application libraries as a high-level interface to the graphics capabilities of a system. The *Visualization Toolkit* accompanying this book is a prime example of

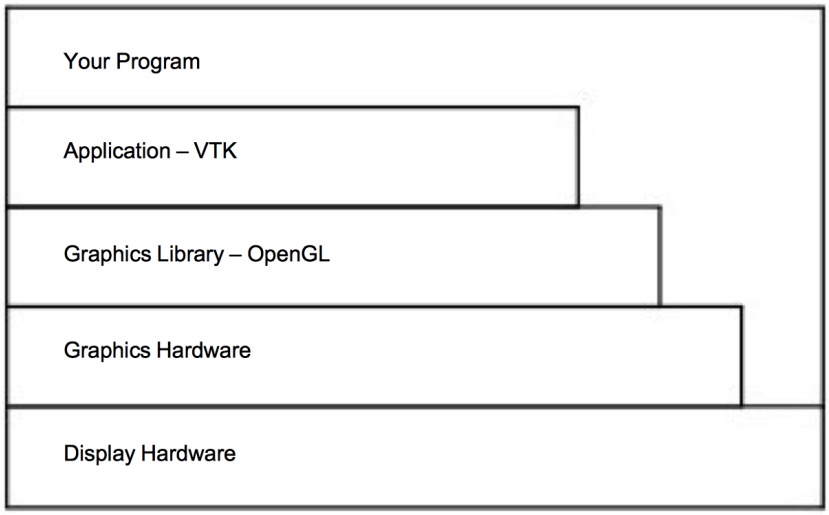


Figure 3.17: Typical graphics interface hierarchy.

this. It allows you to display a complex object or graph using just a few commands. It is also possible to interface to a number of different graphics libraries, since different libraries may be supported on different hardware platforms.

The graphics library and graphics hardware layers both perform similar functions. They are responsible for taking high-level commands from an application library or program, and executing them. This makes programming much easier by providing more complex primitives to work with. Instead of drawing pixels one at a time, we can draw primitives like polygons, triangles, and lines, without worrying about the details of which pixels are being set to which colors. Figure 3.18 illustrates some high-level primitives that all mainstream graphics libraries support.

This functionality is broken into two different layers because different machines may have vastly different graphics hardware. If you write a program that draws a red polygon, either the graphics library or the graphics hardware must be able to execute that command. On high-end systems, this may be done in the graphics hardware, on others it will be done by the graphics library in software. So the same commands can be used with a wide variety of machines, without worrying about the underlying graphics hardware.

The fundamental building block of the primitives in Figure 3.18 is a point (or vertex). A vertex has a position, normal, and color, each of which is a three element vector. The position specifies where the vertex is located, its normal specifies which direction the vertex is facing, and its color specifies the vertex's red, green, and blue components.

A polygon is built by connecting a series of points or vertices as shown in Figure 3.19. You may be wondering why each vertex has a normal, instead of having just one normal for the entire polygon. A planar polygon can only be facing one direction regardless of what the normals of its vertices indicate. The reason is that sometimes a polygon is used as an approximation of something else, like a curve. Figure 3.20 shows a top-down view of a cylinder. As you can see, it's not really a cylinder but rather a polygonal approximation of the cylinder drawn in gray. Each vertex is shared by two polygons and the correct normal for the vertex is not the same as the normal for the polygon. Similar logic explains why each vertex has a color instead of just having one color for an entire polygon.

When you limit yourself to the types of primitives described above, there are some additional properties that many graphics systems support. Edge color and edge visibility can be used to

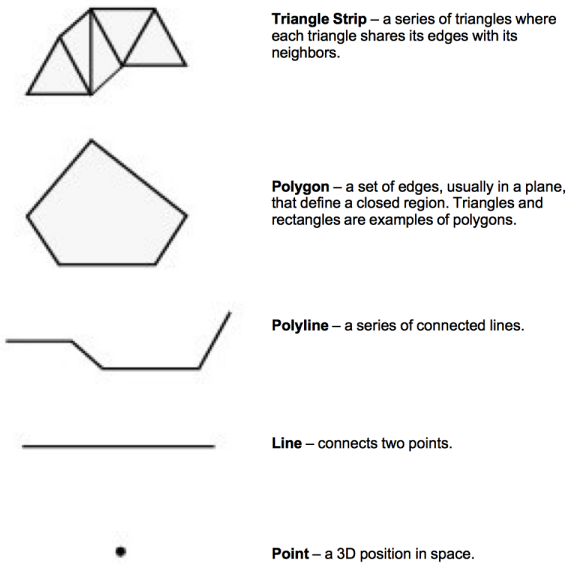


Figure 3.18: Graphics primitives.

highlight the polygon primitives that make up an actor. Another way to do this is by adjusting the representation from *surface* to *wireframe* or *points*. This replaces surfaces such as polygons with either their boundary edges or points respectively. While this may not make much sense from a physical perspective, it can help in some illustrations. Using edge visibility when rendering a CAD model can help to show the different pieces that comprise the model.

3.10.3 Rasterization

At this point in the text we have described how to represent graphics data using rendering primitives, and we have described how to represent images using raster display devices. The question remains, how do we convert graphics primitives into a raster image? This is the topic we address in this section. Although a thorough treatise on this topic is beyond the scope of this text, we will do our best to provide a high-level overview.

The process of converting a geometric representation into a raster image is called *rasterization* or *scan conversion*. In the description that follows we assume that the graphics primitives are triangle polygons. This is not as limiting as you might think, because any general polygon can be tessellated into a set of triangles. Moreover, other surface representations such as splines are usually tessellated by the graphics system into triangles or polygons. (The method described here is actually applicable to convex polygons.)

Most of today's hardware is based on object-order rasterization techniques. As we saw earlier in this chapter, this means processing our actors in order. And since our actors are represented by polygon primitives, we process polygons one at a time. So although we describe the processing of one polygon, bear in mind that many polygons and possibly many actors are processed.

The first step is to transform the polygon using the appropriate transformation matrix. We also project the polygon to the image plane using either parallel or orthographic projection. Part of this process involves clipping the polygons. Not only do we use the front and back clipping planes

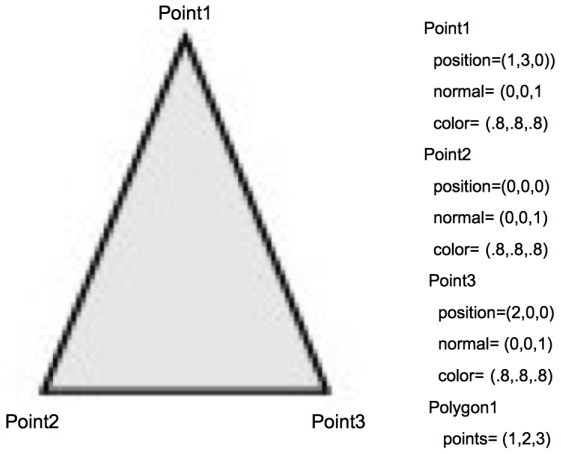


Figure 3.19: An example polygon.

to clip polygons too close or too far, but we must also clip polygons crossing the boundaries of the image plane. Clipping polygons that cross the boundary of the view frustum means we have to generate new polygonal boundaries.

With the polygon clipped and projected to the image plane, we can begin scan-line processing (Figure 3.21). The first step identifies the initial scan-line intersected by the projected polygon. This is found by sorting the vertices' y values. We then find the two edges joining the vertex on the left and right sides. Using the slopes of the edges along with the data values we compute delta data values. These data are typically the R, G, and B color components. Other data values include transparency values and z depth values. (The z values are necessary if we are using a z-buffer, described in the next section.) The row of pixels within the polygon (i.e., starting at the left and right edges) is called a *span*. Data values are interpolated from the edges on either side of the span to compute the internal pixel values. This process continues span-by-span, until the entire polygon is filled. Note that as new vertices are encountered, it is necessary to recompute the delta data values.

The shading of the polygon (i.e., color interpolation across the polygon) varies depending on the actor's interpolation attribute. There are three possibilities: *flat*, *Gouraud*, or *Phong shading*. Figure 3.6 illustrates the difference between flat and Gouraud interpolation. Flat shading calculates the color of a polygon by applying the lighting equations to just one normal (typically the surface normal) of the polygon. Gouraud shading calculates the color of a polygon at all of its vertices using the vertices' normals and the standard lighting equations. The interior and edges of the polygon are then filled in by applying the scan-line interpolation process. Phong shading is the most realistic of the three. It calculates a normal at every location on the polygon by interpolating the vertex normals. These are then used in the lighting equations to determine the resulting pixel colors. Both flat and Gouraud shading are commonly used methods. The complexity of Phong shading has prevented it from being widely supported in hardware.

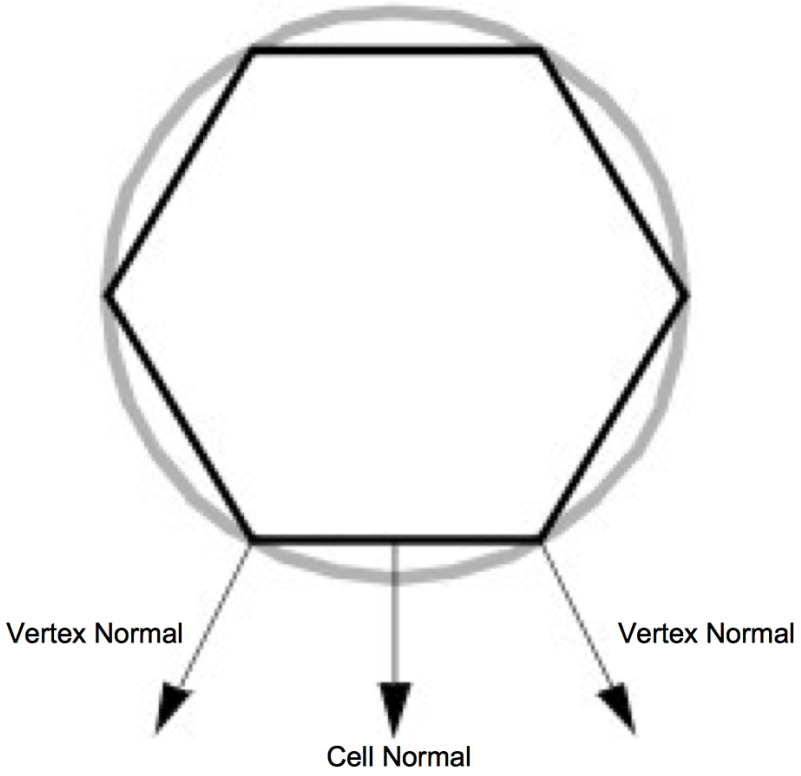


Figure 3.20: Vertex and polygon normals.

3.10.4 Z-Buffer

In our earlier description of the rendering process, we followed rays of light from our eye through a pixel in the image plane to the actors and back to the light source. A nice side effect of ray tracing is that viewing rays strike the first actor they encounter and ignore any actors that are hidden behind it. When rendering actors using the polygonal methods described above, we have no such method of computing which polygons are hidden and which are not. We cannot generally count on the polygons being ordered correctly. Instead, we can use a number of hidden-surface methods for polygon rendering.

One method is to sort all of our polygons from back to front (along the cameras view vector) and then render them in that order. This is called the painters algorithm or painters sort, and has one major weakness illustrated in Figure 3.22. Regardless of the order in which we draw these three triangles, we cannot obtain the desired result, since each triangle is both in front of, and behind, another triangle. There are algorithms that sort and split polygons as necessary to treat such a situation [Carlson85]. This requires more initial processing to perform the sorting and splitting. If the geometric primitives change between images or the camera view changes, then this processing must be performed before each render.

Another hidden surface algorithm, z-buffering, takes care of this problem and does not require sorting. Z-buffering takes advantage of the z-value (i.e., depth value along direction of projection) in the view coordinate system. Before a new pixel is drawn, its z-value is compared against the current z-value for that pixel location. If the new pixel would be in front of the current pixel, then

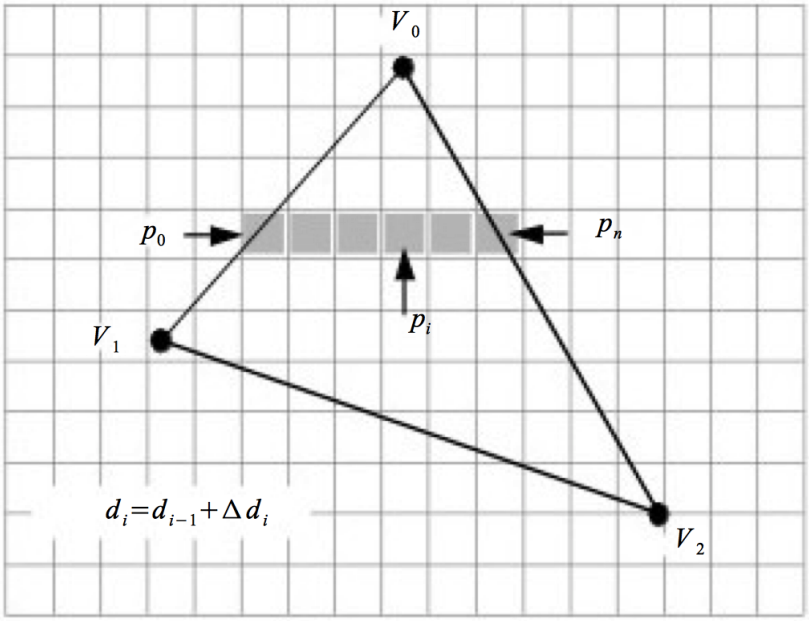


Figure 3.21: Rasterizing a convex polygon. Pixels are processed in horizontal spans (or scan-lines) in the image plane. Data values d_i at point p_i are interpolated along the edges and then along the scan-line using delta data values. Typical data values are RGB components of color.

it is drawn and the z-value for that pixel location is updated. Otherwise the current pixel remains and the new pixel is ignored. Z-buffering has been widely implemented in hardware because of its simplicity and robustness. The downside to z-buffering is that it requires a large amount of memory, called a z-buffer, to store a z-value of every pixel. Most systems use a z-buffer with a depth of 24 or 32 bits. For a 1000 by 1000 display that translates into three to four megabytes just for the z-buffer. Another problem with z-buffering is that its accuracy is limited depending on its depth. A 24-bit z-buffer yields a precision of one part in 16,777,216 over the height of the viewing frustum. This resolution is often insufficient if objects are close together. If you do run into situations with z-buffering accuracy, make sure that the front and back clipping planes are as close to the visible geometry as possible.

3.11 Putting It All Together

This section provides an overview of the graphics objects and how to use them in VTK.

3.11.1 The Graphics Model

We have discussed many of the objects that play a part in the rendering of a scene. Now it's time to put them together into a comprehensive object model for graphics and visualization.

In the *Visualization Toolkit* there are seven basic objects that we use to render a scene. There are many more objects behind the scenes, but these seven are the ones we use most frequently. The objects are listed in the following and illustrated in Figure 3.23.

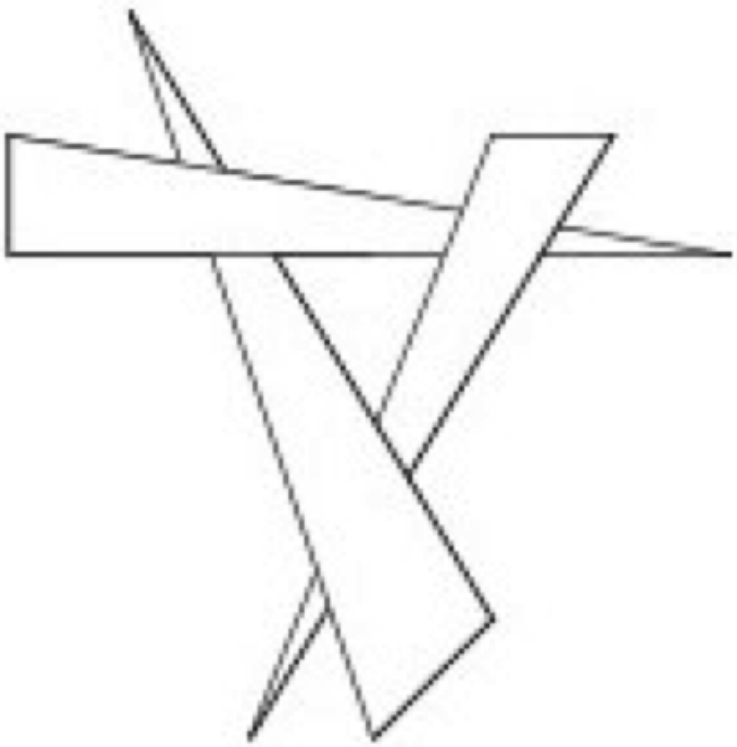


Figure 3.22: Problem with Painter's algorithm.

- vtkRenderWindow — manages a window on the display device; one or more renderers draw into an instance of vtkRenderWindow.
- vtkRenderer — coordinates the rendering process involving lights, cameras, and actors.
- vtkLight — a source of light to illuminate the scene.
- vtkCamera — defines the view position, focal point, and other viewing properties of the scene.
- vtkActor — represents an object rendered in the scene, including its properties and position in the world coordinate system. (*Note*: vtkActor is a subclass of vtkProp. vtkProp is a more general form of actor that includes annotation and 2D drawing classes. See "Assemblies and Other Types of vtkProp" on page74 for more information.)
- vtkProperty — defines the appearance properties of an actor including color, transparency,

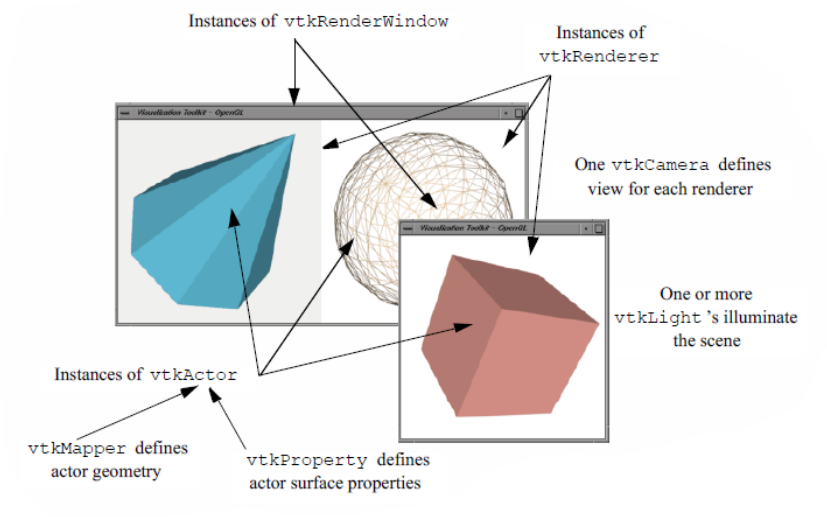


Figure 3.23: Illustrative diagram of graphics objects (Model.cxx) and (Model.py).

and lighting properties such as specular and diffuse. Also representational properties like wireframe and solid surface.

- vtkMapper — the geometric representation for an actor. More than one actor may refer to the same mapper.

The class vtkRenderWindow ties the rendering process together. It is responsible for managing a window on the display device. For PCs running Windows, this will be a Microsoft display window, for Linux and UNIX systems this will be an X window, and on the Mac (OSX) a Quartz window. In VTK, instances of vtkRenderWindow are device independent. This means that you do not need to be concerned about what underlying graphics hardware or software is being used, the software automatically adapts to your computer as instances of vtkRenderWindow are created. (See "Achieving Device Independence" on page 54 for more information.)

In addition to window management, vtkRenderWindow objects are used to manage renderers and store graphics specific characteristics of the display window such as size, position, window title, *window depth*, and the d*ouble buffering* flag. The depth of a window indicates how many bits are allocated per pixel. Double buffering is a technique where a window is logically divided into two buffers. At any given time one buffer is currently visible to the user. Meanwhile, the second buffer can be used to draw the next image in an animation. Once the rendering is complete, the two buffers can be swapped so that the new image is visible. This common technique allows animations to be displayed without the user seeing the actual rendering of the primitives. High-end graphics systems perform double buffering in hardware. A typical system would have a rendering window with a depth of 72 bits. The first 24 bits are used to store the red, green, and blue (RGB) pixel components for the front buffer. The next 24 bits store the RGB values for the back buffer. The last 24 bits are used as a z-buffer.

The class vtkRenderer is responsible for coordinating its lights, camera, and actors to produce an image. Each instance maintains a list of the actors, lights, and an active camera in a particular scene. At least one actor must be defined, but if lights and a camera are not defined, they will be created automatically by the renderer. In such a case the actors are centered in the image and the default camera view is down the z-axis. Instances of the class vtkRenderer also provide methods to

specify the background and ambient lighting colors. Methods are also available to convert to and from world, view, and display coordinate systems.

One important aspect of a renderer is that it must be associated with an instance of the vtkRenderWindow class into which it is to draw, and the area in the render window into which it draws must be defined by a rectangular *viewport*. The viewport is defined by normalized coordinates (0,1) in both the *x* and *y* image coordinate axes. By default, the renderer draws into the full extent of the rendering window (viewpoint coordinates (0,0,1,1)). It is possible to specify a smaller viewport. and to have more than one renderer draw into the same rendering window.

Instances of the class vtkLight illuminate the scene. Various instance variables for orienting and positioning the light are available. It is also possible to turn on/off lights as well as setting their color. Normally at least one light is "on" to illuminate the scene. If no lights are defined and turned on, the renderer constructs a light automatically. Lights in VTK can be either positional or infinite. Positional lights have an associated cone angle and attenuation factors. Infinite lights project light rays parallel to one another.

Cameras are constructed by the class vtkCamera. Important parameters include camera position, focal point, location of front and back clipping planes, view up vector, and field of view. Cameras also have special methods to simplify manipulation as described previously in this chapter.

These include elevation, azimuth, zoom, and roll. Similar to vtkLight, an instance of vtkCamera will be created automatically by the renderer if none is defined.

Instances of the class vtkActor represent objects in the scene. In particular, vtkActor combines object properties (color, shading type, etc.), geometric definition, and orientation in the world coordinate system. This is implemented behind the scenes by maintaining instance variables that refer to instances of vtkProperty, vtkMapper, and vtkTransform. Normally you need not create properties or transformations explicitly, since these are automatically created and manipulated using vtkActor's methods. You do need to create an instance of vtkMapper (or one of its subclasses). The mapper ties the data visualization pipeline to the graphics device. (We will say more about the pipeline in the next chapter.)

In VTK, actors are actually subclasses of vtkProp (arbitrary props) and vtkProp3D (those that can be transformed in 3D space. (The word prop is derived from the stage, where a props is an object in the scene.) There are other subclasses of props and actors with specialized behavior (see “Assemblies and Other Types of vtkProp” on page74 for more information). One example is vtkFollower. Instances of this class always face the active camera. This is useful when designing signs or text that must be readable from any camera position in the scene.

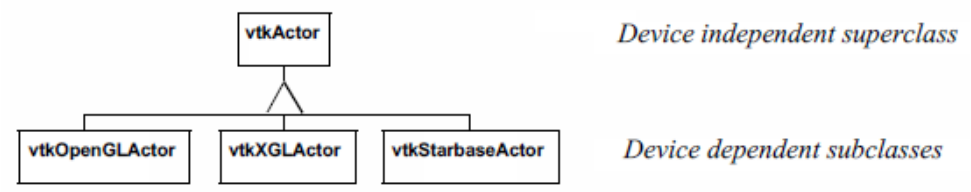
Instances of the class vtkProperty affect the rendered appearance of an actor. When actors are created, a property instance is automatically created with them. It is also possible to create property objects directly and then associate the property object with one or more actors. In this way actors can share common properties.

Finally, vtkMapper (and its subclasses) defines object geometry and, optionally, vertex colors. In addition, vtkMapper refers to a table of colors (i.e., vtkLookupTable) that are used to color the geometry. (We discuss mapping of data to colors in “Color Mapping” on page163.) We will examine the mapping process in more detail in “Mapper Design” on page195. For now assume that vtkMapper is an object that represents geometry and other types of visualization data.

There is another important object, vtkRenderWindowInteractor, that captures events (such as mouse clicks and mouse motion) for a renderer in the rendering window. vtkRenderWindowInteractor captures these events and then triggers certain operations like camera dolly, pan, and rotate, actor picking, into/out of stereo mode, and so on. Instances of this class are associated with a rendering window using the SetRenderWindow() method.

3.11.2 Achieving Device Independence

A desirable property of applications built with VTK is that they are device independent. This means that computer code that runs on one operating system with a particular software/hardware configuration runs unchanged on a different operating system and software/hardware configuration. The advantage of this is that the programmer does not need to expend effort porting an application between different computer systems. Also, existing applications do not need to be rewritten to take advantage of new developments in hardware or software technology. Instead, VTK handles this transparently by a combination of inheritance and a technique known as *object factories*.



(a) Inheritance of device classes.

```
1    vtkActor *vtkActor::New()
2    {
3    vtkObject* ret = vtkGraphicsFactory::CreateInstance("vtkActor");
4    return (vtkActor*)ret;
5    }
```

(b) Code fragment from vtkActor::New().

```
1    if (!strcmp("OpenGL",rl) || !strcmp("Win32OpenGL",rl) ||
2    !strcmp("CarbonOpenGL",rl) || !strcmp("CocoaOpenGL",rl))
3    {
4    if(strcmp(vtkclassname, "vtkActor") == 0)
5    {
6    return vtkOpenGLActor::New();
7    }
```

(c) Code fragment from vtkGraphicsFactory::CreateInstance(vtkclassname).

Figure 3.24: Achieving device independence using (a) inheritance and object factories (b) and (c)..

Figure 3.24 (a) illustrates the use of inheritance to achieve device independence. Certain classes like vtkActor are broken into two parts: a device independent superclass and a device dependent subclass.

The trick here is that the user creates a device dependent subclass by invoking the special constructor New() in the device independent superclass. For example we would use (in C++)

```
1  vtkActor *anActor = vtkActor::New()
```

to create a device dependent instance of vtkActor. The user sees no device dependent code, but in actuality anActor is a pointer to a device dependent subclass of vtkActor. Figure 3.24 (b) is a code fragment of the constructor method New() which uses VTK's object factory mechanism. In turn,

the vtkGraphicsFactory (used to instantiate graphical classes) produces the appropriate concrete subclass when requested to instantiate an actor as shown in Figure 3.24.

The use of object factories as implemented using the New() method allows us to create device independent code that can move from computer to computer and adapt to changing technology. For example, if a new graphics library became available, we would only have to create a new device dependent subclass, and then modify the graphics factory to instantiate the appropriate sub-class based on environment variables or other system information. This extension would be localized and only done once, and all applications based on these object factories would be automatically ported without change.

This section works through some simple applications implemented with VTK graphics objects. The focus is on the basics: how to create renderers, lights, cameras, and actors. Later chapters tie together these basic principles to create applications for data visualization.

3.11.3 Examples

Render a Cone.

The following C++ code uses most of the objects introduced in this section to create an image of a cone. The vtkConeSource generates a polygonal representation of a cone and vtkPolyDataMapper maps the geometry (in conjunction with the actor) to the underlying graphics library. (The source code to this example can be found in Cone.cxx or Cone.py. The source code contains additional documentation as well.)

Listing 3.1: Cone.cxx

```
1  #include "vtkConeSource.h"
2  #include "vtkPolyDataMapper.h"
3  #include "vtkRenderWindow.h"
4  #include "vtkCamera.h"
5  #include "vtkActor.h"
6  #include "vtkRenderer.h"
7
8  int main( int argc, char *argv[] )
9  {
10   vtkConeSource *cone = vtkConeSource::New();
11     cone->SetHeight( 3.0 );
12     cone->SetRadius( 1.0 );
13     cone->SetResolution( 10 );
14
15   vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
16     coneMapper->SetInputConnection( cone->GetOutputPort() );
17   vtkActor *coneActor = vtkActor::New();
18     coneActor->SetMapper( coneMapper );
19
20   vtkRenderer *ren1= vtkRenderer::New();
21     ren1->AddActor( coneActor );
22     ren1->SetBackground( 0.1, 0.2, 0.4 );
23
24   vtkRenderWindow *renWin = vtkRenderWindow::New();
25     renWin->AddRenderer( ren1 ); renWin->SetSize( 300, 300 );
26
```

```
27    int i;
28    for (i = 0; i < 360; ++i)
29      {
30  // render the image
31       renWin->Render();
32  // rotate the active camera by one degree
33       ren1->GetActiveCamera()->Azimuth( 1 );
34      }
35    cone->Delete();
36    coneMapper->Delete();
37    coneActor->Delete();
38  // cleanup
39    ren1->Delete();
40    renWin->Delete();
41
42    return 0;
43  }
```

Some comments about this example. The include files vtk*.h include class definitions for the objects in VTK necessary to compile this example. We use the constructor New() to create the objects in this example, and the method Delete() to destroy the objects. In VTK the use of New() and Delete() is mandatory to insure device independence and properly manage reference counting. (See VTK Users Guide for details.) In this example the use of Delete() is really not necessary because the objects are automatically deleted upon program termination. But generally speaking, you should always use a Delete() for every invocation of New(). (Future examples will not show the Delete() methods in the scope of the main() program to conserve space, nor show the required #include statements.)

The data representing the cone (a set of polygons) in this example is created by linking together a series of objects into a pipeline (which is the topic of the next chapter). First a polygonal representation of the cone is created with a vtkConeSource and serves as input to the data mapper as specified with the SetInput() method. The SetMapper() method associates the mapper's data with the coneActor. The next line adds coneActor to the renderer's list of actors. The cone is rendered in a loop running over 360°. Since there are no cameras or lights defined in the above example, VTK automatically generates a default light and camera as a convenience to the user. The camera is accessed through the GetActiveCamera() method, and a one degree azimuth is applied as shown. Each time a change is made to any objects a Render() method is invoked to produce the corresponding image. Once the loop is complete all allocated objects are destroyed and the program exits.

There are many different types of source objects in VTK similar to vtkConeSource as shown in Figure 3.25. In the next chapter we will learn more about source and other types of filters.

Events and Observers.

A visualization toolkit like VTK is frequently used in interactive applications or may be required to provide status during operation. In addition, integration with other packages such as GUI toolkits is a common task. Supporting such features requires a mechanism for inserting user functionality into the software. In VTK, the *command/observer* design pattern [4] is used for this purpose.

Fundamental to this design pattern as implemented in VTK is the concept of *events*. An event signals that an important operation has occurred in the software. For example, if the user presses the left mouse button in the render window, VTK will invoke the LeftButtonPressEvent.

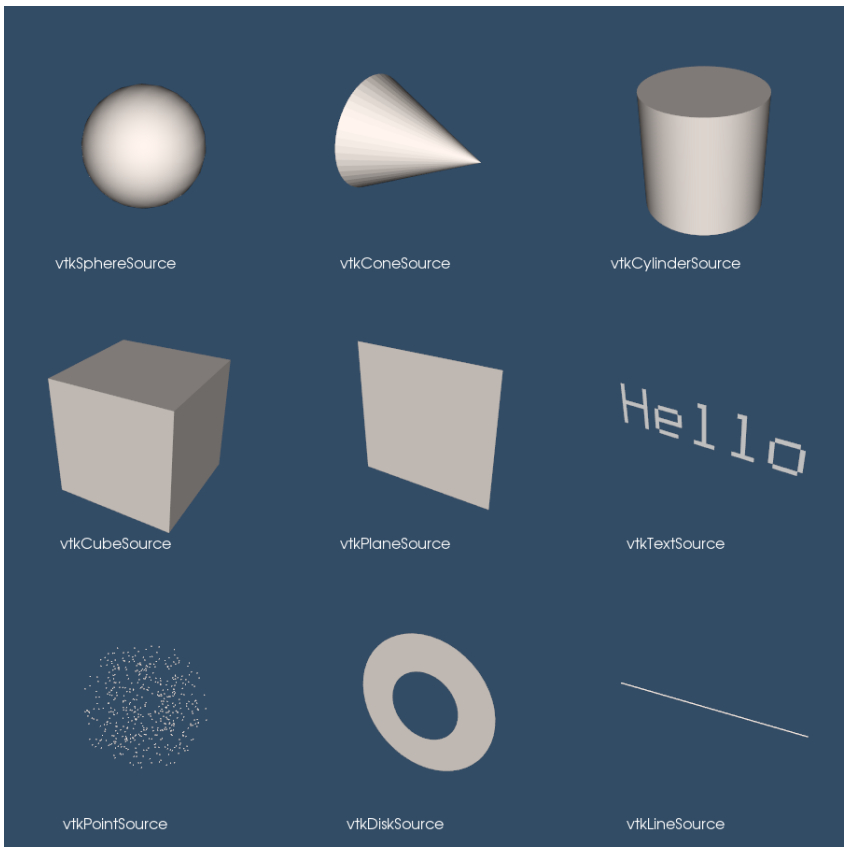


Figure 3.25: Examples of source objects that procedurally generate polygonal models. These nine images represent just some of the capability of VTK. From upper left in reading order: sphere, cone, cylinder, cube, plane, text, random point cloud, disk (with or without hole), and line source. Other polygonal source objects are available; check the subclasses of vtkPolyDataAlgorithm. See SourceObjectsDemo.cxx or SourceObjectsDemo.py

Observers are objects that register their interest in a particular event or events. When one of these events is invoked, the observer receives notification and may perform any valid operation at that point; that is, execute the command associated with the observer. The benefit of the command/observer design pattern is that is simple in concept and implementation, yet provides significant power to the user. However it does require the software implementation to invoke events as it operates.

In the next example, an observer watches for the StartEvent invoked by the renderer just as it begins the rendering process. The observer in turn executes its associated command which simply prints out the camera's current position.

```
1 #include "vtkCommand.h"
2 // Callback for the interaction
3 class vtkMyCallback : public vtkCommand
4 {
5 public:
6   static vtkMyCallback *New()
7     { return new vtkMyCallback; }
```

```
 8   virtual void Execute(vtkObject *caller, unsigned long, void*)
 9     {
10       vtkRenderer *ren =
11               reinterpret_cast<vtkRenderer*>(caller);
12       cout << ren->GetActiveCamera()->GetPosition()[0] << " "
13       ren->GetActiveCamera()->GetPosition()[1] << " "
14       ren->GetActiveCamera()->GetPosition()[2] << "n";
15     }
16 };
17
18 int main( int argc, char *argv[] )
19 {
20   vtkConeSource *cone = vtkConeSource::New();
21     cone->SetHeight( 3.0 );
22     cone->SetRadius( 1.0 );
23     cone->SetResolution( 10 );
24
25   vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
26    coneMapper->SetInputConnection(cone->GetOutputPort());
27   vtkActor *coneActor = vtkActor::New();
28   coneActor->SetMapper( coneMapper );
29
30   vtkRenderer *ren1= vtkRenderer::New();
31     ren1->AddActor( coneActor );
32     ren1->SetBackground( 0.1, 0.2, 0.4 );
33
34   vtkRenderWindow *renWin = vtkRenderWindow::New();
35     renWin->AddRenderer( ren1 ); renWin->SetSize( 300, 300 );
36
37   vtkMyCallback *mo1 = vtkMyCallback::New();
38     ren1->AddObserver(vtkCommand::StartEvent,mo1); mo1->Delete();
39
40   int i;
41   for (i = 0; i < 360; ++i)
42   {
43   //   render the image
44     renWin->Render();
45   // rotate the active camera by one degree
46     ren1->GetActiveCamera()->Azimuth( 1 );
47   }
48
49   cone->Delete();
50   coneMapper->Delete();
51   coneActor->Delete();
52   ren1->Delete();
53   renWin->Delete();
54   return 0;
55 }
```

The observer is created by deriving from the class vtkCommand. The Execute() method is

required to be implemented by any concrete subclass of vtkCommand (i.e., the method is pure virtual). The resulting subclass, vtkMyCommand, is instantiated and registered with the renderer instance ren1 using the AddObserver() method. In this case the StartEvent is the observed event.

This simple example does not demonstrate the true power of the command/observer design pattern. Later in this chapter (“Interpreted Code” on page 62) we will see how this functionality is used to integrate a simple GUI into VTK. In Chapter 7 three-dimensional interaction widgets will be introduced (“3D Widgets and User Interaction” on page 132).

Creating Multiple Renderers.

The next example is a bit more complex and uses multiple renderers that share a single rendering window. We use viewports to define where the renderers should draw in the render window. (This C++ code can be found in Cone3.cxx.)

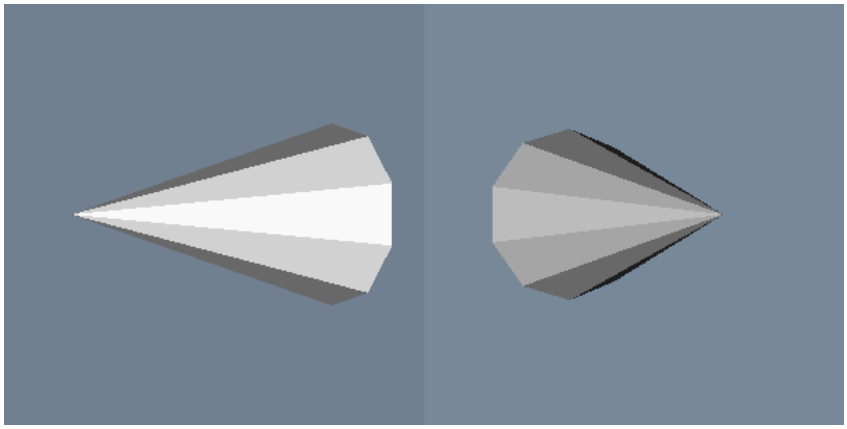


Figure 3.26: Two frames of output from Cone3.cxx. See Cone3.cxx or Cone3.py

Listing 3.2: Cone3.cxx

```
1  vtkRenderer *ren1= vtkRenderer::New();
2    ren1->AddActor( coneActor );
3    ren1->SetBackground( 0.1, 0.2, 0.4 );
4    ren1->SetViewport(0.0, 0.0, 0.5, 1.0);
5
6  vtkRenderer *ren2= vtkRenderer::New();
7    ren2->AddActor( coneActor );
8    ren2->SetBackground( 0.2, 0.3, 0.5 );
9    ren2->SetViewport(0.5, 0.0, 1.0, 1.0);
10
11 vtkRenderWindow *renWin = vtkRenderWindow::New();
12   renWin->AddRenderer( ren1 );
13   renWin->AddRenderer( ren2 );
14   renWin->SetSize( 600, 300 );
15
16 ren1->GetActiveCamera()->Azimuth(90);
17
18 int i;
```

```
19 for (i = 0; i < 360; ++i)
20   {
21   // render the image renWin->Render();
22   // rotate the active camera by one degree
23     ren1->GetActiveCamera()->Azimuth( 1 );
24     ren2->GetActiveCamera()->Azimuth( 1 );
25   }
```

As you can see, much of the code is the same as the previous example. The first difference is that we create two renderers instead of one. We assign the same actor to both renderers, but set each renderer's background to a different color. We set the viewport of the two renderers so that one is on the left half of the rendering window and the other is on the right. The rendering window's size is specified as 600 by 300 pixels, which results in each renderer drawing into a viewport of 300 by 300 pixels.

A good application of multiple renderers is to display different views of the same world as demonstrated in this example. Here we adjust the first renderer's camera with a 90 degree azimuth. We then start a loop that rotates the two cameras around the cone. Figure 3.26 shows two frames from this animation.

Properties and Transformations.

The previous examples did not explicitly create property or transformation objects or apply actor methods that affect these objects. Instead, we accepted default instance variable values. This procedure is typical of VTK applications. Most instance variables have been preset to generate acceptable results, but methods are always available for you to overide the default values.

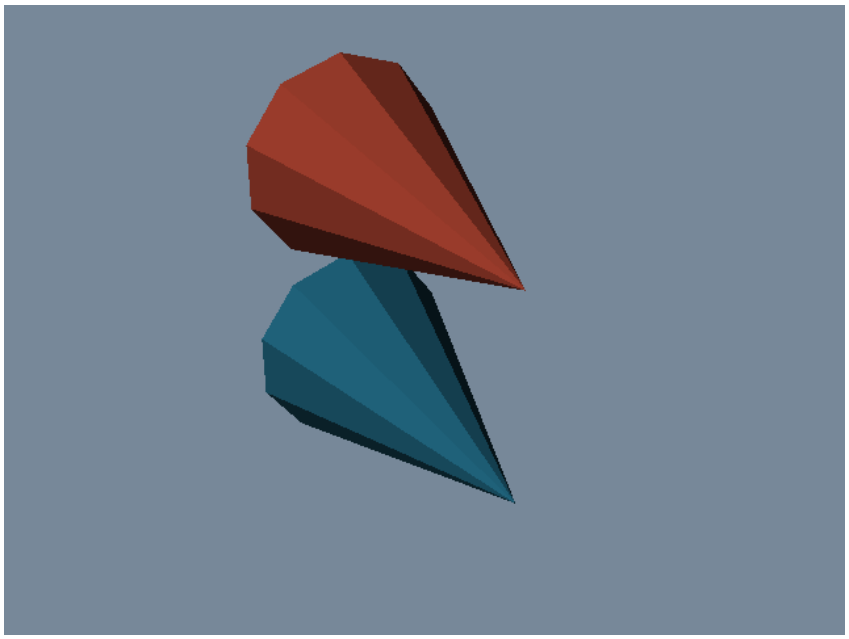


Figure 3.27: Modifying properties and transformation matrix. See Cone4.cxx or Cone4.py

This example creates an image of two cones of different colors and specular properties. In addition, we transform one of the objects to lay next to the other. The C++ source code for this

example can be found in Cone4.cxx.

Listing 3.3: Cone4.cxx

```
vtkActor *coneActor = vtkActor::New();
  coneActor->SetMapper(coneMapper);
  coneActor->GetProperty()->SetColor(0.2, 0.63, 0.79);
  coneActor->GetProperty()->SetDiffuse(0.7);
  coneActor->GetProperty()->SetSpecular(0.4);
  coneActor->GetProperty()->SetSpecularPower(20);

vtkProperty *property = vtkProperty::New();
  property->SetColor(1.0, 0.3882, 0.2784);
  property->SetDiffuse(0.7);
  property->SetSpecular(0.4);
  property->SetSpecularPower(20);

vtkActor *coneActor2 = vtkActor::New();
  coneActor2->SetMapper(coneMapper);
  coneActor2->GetProperty()->SetColor(0.2, 0.63, 0.79);
  coneActor2->SetProperty(property);
  coneActor2->SetPosition(0, 2, 0);

vtkRenderer *ren1= vtkRenderer::New();
  ren1->AddActor( coneActor );
  ren1->AddActor( coneActor2 );
  ren1->SetBackground( 0.1, 0.2, 0.4 );
```

We set the actor coneActor properties by modifying the property object automatically created by the actor. This differs from actor coneActor2, where we create a property directly and then assign it to the actor. ConeActor2 is moved from its default position by applying the SetPosition() method. This method affects the transformation matrix that is an instance variable of the actor. The resulting image is shown in Figure 3.27.

Introducing vtkRenderWindowInteractor.

The previous examples are not interactive. That is, it is not possible to directly interact with the data without modifying and recompiling the C++ code. One common type of interaction is to change camera position so that we can view our scene from different vantage points. In the *Visualization Toolkit* we have provided a suite of convenient objects to do this: vtkRenderWindowInteractor, vtkInteractorStyle and their derived classes.

Instances of the class vtkRenderWindowInteractor capture windowing system specific mouse and keyboard events in the rendering window, and then translate these events into VTK events. For example, mouse motion in an X11 or Windows application (occurring in a render window) would be translated by vtkRenderWindowInteractor into VTK's MouseMoveEvent. Any observers registerered for this event would be notified (see "Events and Observers" on 56). Typically an instance of vtkInteractorStyle is used in combination with vtkRenderWindowInteractor to define a behavior associated with particular events. For example, we can perform camera dolly, pan, and rotation by using different mouse button and motion combinations. The following code fragment shows how to instantiate and use these objects. This example is the same as our first example with the addition of the interactor and interactor style. The complete example C++ code is in Cone5.cxx.

Listing 3.4: Cone5.cxx

```
vtkRenderWindowInteractor *iren =
  vtkRenderWindowInteractor::New();

  iren->SetRenderWindow(renWin);

vtkInteractorStyleTrackballCamera *style =
 vtkInteractorStyleTrackballCamera::New();

  iren->SetInteractorStyle(style);
  iren->Initialize();
  iren->Start();
```

After the interactor is created using its New() method, we must tell it what render window to capture events in using the SetRenderWindow() method. In order to use the interactor we have to initialize and start the event loop using the Initialize() and Start() methods, which works with the event loop of the windowing system to begin to catch events. Some of the more useful events include the “w” key, which draws all actors in wireframe; the “s” key, which draws the actors in surface form; the “3” key, which toggles in and out of 3D stereo for those systems that support this; the “r” key, which resets camera view; and the " e" key, which exits the application. In addition, the mouse buttons rotate, pan, and dolly about the camera’s focal point. Two advanced features are the “u” key, which executes a user-defined function; and the “p” key, which picks the actor under the mouse pointer.

Interpreted Code.

In the previous example we saw how to create an interactor style object in conjunction with vtkRenderWindowInteractor to enable us to manipulate the camera by mousing in the render window. Although this provides flexibility and interactivity for a large number of applications, there are examples throughout this text where we want to modify other parameters. These parameters range from actor properties, such as color, to the name of an input file. Of course we can always write or modify C++ code to do this, but in many cases the turn-around time between making the change and seeing the result is too long. One way to improve the overall interactivity of the system is to use an interpreted interface. Interpreted systems allow us to modify objects and immediately see the result, without the need to recompile and relink source code. Interpreted languages also provide many tools, such as GUI (Graphical User Interface) tools, that simplify the creation of applications.

The Visualization Toolkit has built into its compilation process the ability to automatically generate language bindings to the [Ousterhout94]. This so-called wrapping process automatically creates a layer between the C++ VTK library and the interpreter as illustrated in Figure 3.28. There is a one-to-one mapping between C++ methods and Tcl C++ functions for most objects and methods in the system. To demonstrate this, the following example repeats the previous C++ example except that it is implemented with a Tcl script. (The script can be found in Cone5.tcl).

Listing 3.5: Cone5.tcl

```
package require vtk
package require vtkinteraction
vtkConeSource cone
  cone SetHeight 3.0
  cone SetRadius 1.0
```

```
6    cone SetResolution 10
7
8  vtkPolyDataMapper coneMapper
9    coneMapper SetInputConnection [cone GetOutputPort]
10
11 vtkActor coneActor
12   coneActor SetMapper coneMapper
13
14 vtkRenderer ren1
15   ren1 AddActor coneActor
16   ren1 SetBackground 0.1 0.2 0.4
17
18 vtkRenderWindow renWin
19   renWin AddRenderer ren1
20   renWin SetSize 300 300
21
22 vtkRenderWindowInteractor iren
23   iren SetRenderWindow renWin
24
25 vtkInteractorStyleTrackballCamera style
26   iren SetInteractorStyle style
27   iren AddObserver UserEvent {wm deiconify .vtkInteract}
28   iren Initialize
29
30 wm withdraw .
```

Figure 3.28: In VTK the C++ library is automatically wrapped with the interpreted languages Tcl, Python, and Java.

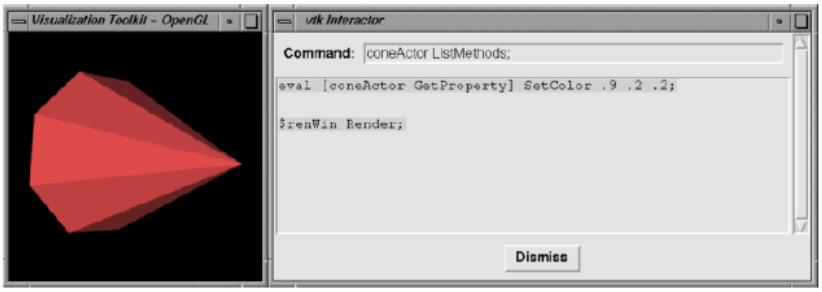


Figure 3.29: Using Tcl and Tk to build an interpreted application (Cone5.tcl).

The example begins by loading some shared libraries defining various VTK classes. Next the standard visualization pipeline is created from the vtkConeSource and vtkPolyDataMapper. The rendering classes are created exactly the same as with the C++ example. One major addition is an observer to watch for a UserEvent in the rendering window (by default a “keypress-u”). The observer triggers the invocation of a Tcl script to raise a Tk interactor GUI widget called.vtkInteract. This GUI, which allows the direct typing of Tcl statements, is shown in Figure 3.29 and is defined by the Tcl command package require vtkinteraction which was executed earlier in the script. (Note: Tk is a popular GUI toolkit for interpreted languages and is distributed as part of Tcl.)

As we can see from this example, the number of lines of code is less for the Tcl example than for equivalent C++ code. Also, many of the complexities of C++ are hidden using the interpreted language. Using this user-interface GUI we can create, modify, and delete objects, and modify their

instance variables. The resulting changes appear as soon as a Render() method is applied or mouse events in the rendering window cause a render to occur. We encourage you to use Tcl (or one of the other interpreters) for rapid creation of graphics and visualization examples. C++ is best used when you desire higher performing applications.

Transform Matrices

Transformation matrices are used throughout the Visualization Toolkit. Actors (subclasses of vtkProp3D — see "Assemblies and Other Types of vtkProp" on page 67) use them to position and orient themselves. Various filters, including vtkGlyph3D and vtkTransformFilter, use transformation matrices to implement their own functionality. As a user you may never use transformation matrices directly, but understanding them is important to successful use of many VTK classes.

The most important aspect to applying transformation matrices is to understand the order in which the transformations are applied. If you break down a complex series of transformations into simple combinations of translation, scaling, and rotation, and keep careful track of the order of application, you will have gone a long way to mastering their use.

A good demonstration example of transformation matrices is to examine how vtkActor uses its internal matrix. vtkActor has an internal instance variable Transform to which it delegates many of its methods or uses the matrix to implement its methods. For example, the RotateX(), RotateY(),and RotateZ() methods are all delegated to Transform. The method SetOrientation() uses Transform to orient the actor. The vtkActor class applies transformations in an order that we feel is natural to most users.

The vtkActor class applies transformations in an order that we feel is natural to most users. As a convenience, we have created instance variables that abstract the transformation matrices. The he Origin (o_x, o_y, o_z) specifies the point that is the center of rotation and scaling. The Position (p_x, p_y, p_z)p specifies a final translation of the object. Orientation (r_x, r_y, r_z) defines the rotation about the x, y and z axes. Scale (s_x, s_y, s_z) defines scale factors for the x, y, and z axes. Internally, the actor uses these instance variables to create the following sequence of transformations (see Equation (3.6), Equation (3.9), Equation (3.13).

$$T = T_T(p_x + o_{x'} p_y + o_{y'} p_y + o_z) T_{R_z} T_{R_x} T_{R_z} T_S T_T(-o_x, -o_y, -o_z) \tag{3.14}$$

The term $T_T(x, y, z)$ denotes the translations in the x, y and z direction. Recall that we premultiply the transformation matrix times the position vector. This means the transformations are read from right to left. In other words, Equation (3.14) proceeds as follows:

1. Translate the actor to its origin. Scaling and rotation will occur about this point. The initial translation will be countered by a translation in the opposite direction after scaling and rotations are applied.

2. Scale the geometry.

3. Rotate the actor about the *y*, then *x*, and then *z* axes.

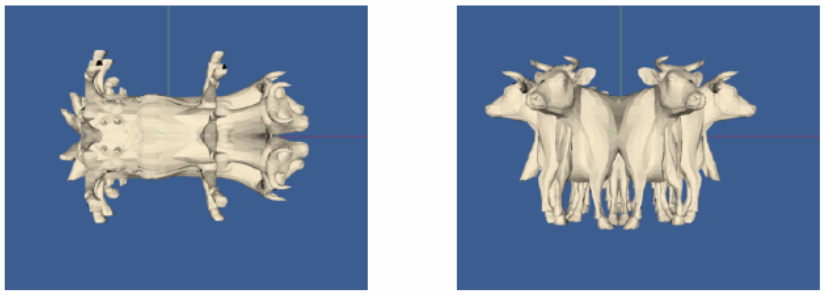
4. Undo the translation of step 1 and move the actor to its final location.

The order of the transformations is important. In VTK the rotations are ordered to what is natural in most cases. We recommend that you spend some time with the software to learn how these transformations work with your own data.

Probably the most confusing aspect of transformations are rotations and their effect on the Orientation instance variable. Generally orientations are not set directly by the user, and most

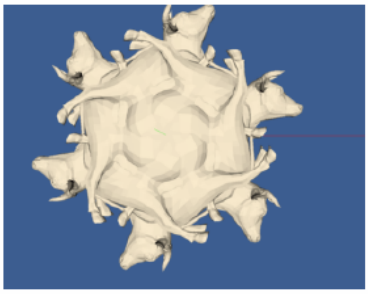
users will prefer to specify rotations with the RotateX(), RotateY(), and RotateZ() methods. These methods perform rotations about the x, y, and z axes in an order specified by the user. New rotations are applied to the right of the rotation transformation. If you need to rotate your actor about a single axis, the actor will rotate exactly as you expect it will, and the resulting orientation vector will be as expected. For example, the operation RotateY(20) will produce an orientation of (0,20,0) and a RotateZ(20) will produce (0,0,20). However, a RotateY(20) followed by a RotateZ(20) will not produce (0,20,20) but produce an orientation of (6.71771, 18.8817, 18.8817)! This is because the rotation portion of Equation (3.14) is built from the rotation order z, then x, and then y. To verify this, a RotateZ(20) followed by a RotateY(20) does produce an orientation of (0,20,20). Adding a third rotation can be even more confusing.

A good rule of thumb is to only use the SetOrientation() method to either reset the orientation to (0,0,0) or to set just one of the rotations. The RotateX(), RotateY(), and RotateZ() methods are preferred to SetOrientation() when multiple angles are needed. Remember that these rotations are applied in reverse order. Figure 3.30 illustrates the use of the rotation methods. We turn off the erase between frames using the render windows EraseOff() method so we can see the effects of the rotations. Note that in the fourth image the cow still rotates about her own y axis even though an x axis rotation preceded the y rotation.

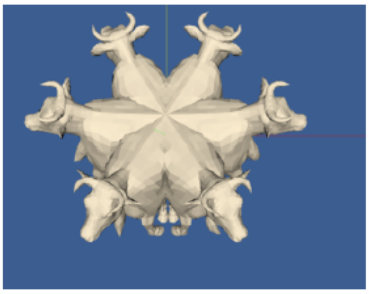


(a) Six rotations about the x axis.

(b) Six rotations about the y axis.



(c) Six rotations about the z axis.



(d) First a rotation about the x axis, then six rotations about the y axis.

Figure 3.30: Rotations of a cow about her axes. In this model, the x axis is from the left to right; the y axis is from bottom to top; and the z axis emerges from the image. The camera location is the same in all four images.

We have seen that VTK hides some of the complexities of matrix transformations by using instance variables that are more natural than a transformation matrix. But there will be times when the predefined order of transformations performed by the actor will not be sufficient. vtkActor has an instance variable UserMatrix that contains a 4 x 4 transformation matrix. This matrix is applied before the transformation composed by the actor. As you become more comfortable with 4 x 4 transformation matrices you may want to build your own matrix. The object vtkTransform creates

and manipulates these matrices. Unlike an actor, an instance of vtkTransform does not have an instance variable for position, scale, origin, etc. You control the composition of the matrix directly.

The following statements create an identical 4 x 4 matrix that the actor creates:

```
vtkTransform *myTrans = vtkTransform::New ();
  myTrans->Translate(position[0],position[1],position[2]);
  myTrans->Translate(origin[0],origin[1],origin[2]);
  myTrans->RotateZ(orientation[2]);
  myTrans->RotateX (orientation[0]);
  myTrans->RotateZ(orientation[1];
  myTrans->Scale (scale[0],scale[1],scale[2]);
  myTrans->Translate (-origin[0],-origin[1],-origin[2]);
```

Compare this sequence of transform operations with the transformation in Equation (3.14).

Our final example shows how the transform built with vtkTransform compares with a transform built by vtkActor. In this example, we will transform our cow so that she rotates about the world coordinate origin $(0,0,0)$. She will appear to be walking around the origin. We accomplish this in two ways: one using vtkTransform and the actor's UserMatrix, then using the actor's instance variables.

First, we will move the cow five feet along the z axis then rotate her about the origin. We always specify transformations in the reverse order of their application:

```
vtkTransform *walk = vtkTransform::New(); walk->RotateY(0,20,0);
  walk->Translate(0,0,5);

vtkActor *cow=vtkActor::New();
  cow->SetUserMatrix(walk->GetMatrix());
```

These operations produce the transformation sequence:

$$T = T_T(0,0,5-(-5))T_{R_y}T_S T_T(0,0,-(-5)) \tag{3.15}$$

Now we do the same using the cow's instance variables:

```
vtkActor *cow=vtkActor::New();
  cow->SetOrigin(0,0,-5);
  cow->RotateY(20);
  cow->SetPosition(0,0,5);
```

These operations produce the transformation sequence:

$$T = T_T(0,0,5-(-5))T_{R_y}T_S T_T(0,0,-(-5)) \tag{3.16}$$

Cancelling the minus signs in the right-most translation matrix and combining the position and origin translation produce the equivalent transform that we built with vtkTransform. Figure 3.30 shows the cow rotating with the specified transformation order. Your preference is a matter of taste and how comfortable you are with matrix transformations. As you become more skilled (and your demands are greater) you may prefer to always build your transformations. VTK gives you the choice.

There is one final and powerful operation that affects an actor's orientation. You can rotate an actor about an arbitrary vector positioned at the actor's origin. This is done with the actor's (and transform's) RotateWXYZ() method. The first argument of the operation specifies the number of degrees to rotate about the vector specified by the next three arguments. Figure 3.31 shows how to

rotate the cow about a vector passing through her nose. At first, we leave the origin at $(0, 0, 0)$. This is obviously not what we wanted. The second figure shows the rotation when we change the cow's rotation origin to the tip of her nose.

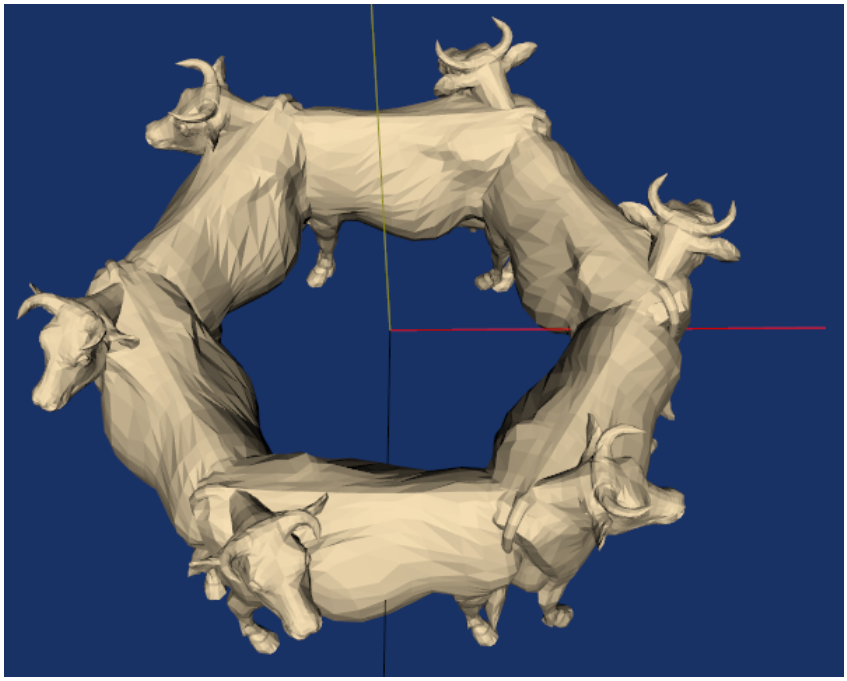


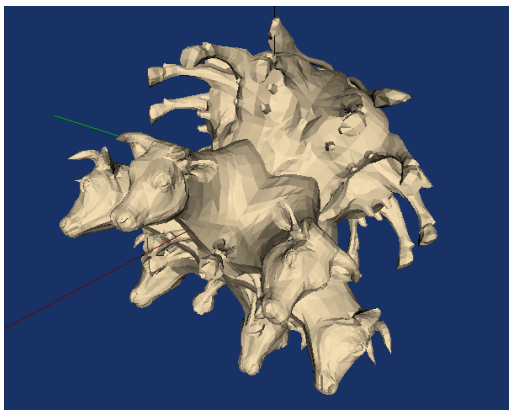
Figure 3.31: Modifying properties and transformation matrix. See WalkCow.cxx or WalkCow.py

Assemblies and Other Types of vtkProp

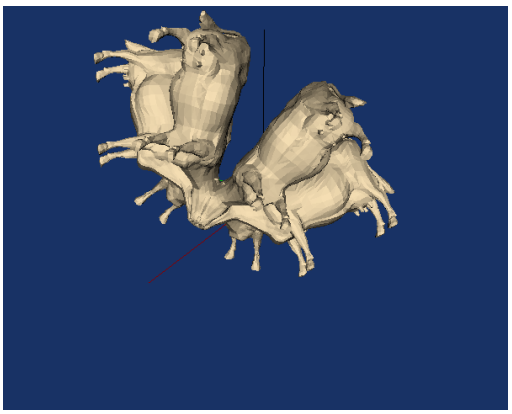
Often it is desirable to collect actors into a hierarchy of transform-dependent groups. For example, a robot arm may be represented by rigid links connected at joints such as the shoulder joint, upper arm, elbow, lower arm, wrist joint, and hand. In such a configuration, when the shoulder joint rotates, the expected behavior is that the entire arm rotates since the links are connected together. This is an example of what is referred to as an *assembly* in VTK. vtkAssembly is just one of many actor-like classes in VTK. As Figure 3.33 shows, these classes are arranged into a hierarchy of vtkProps. (In stage and film terminology, a prop is something that appears or is used on stage.) Assemblies are formed in VTK by instantiating a vtkAssembly and then adding *parts* to it. A part is any instance of vtkProp3D —including other assemblies. This means that assemblies can be formed into hierarchies (as long as they do not contain self-referencing loops). Assemblies obey the rules of transformation concatenation illustrated in the previous section (see "Transformation Matrices" on 64). Here is an example of how to create a simple assembly hierarchy (from assembly.tcl).

Listing 3.6: Part of assembly.tcl

```
1 vtkSphereSource sphere
2 vtkPolyDataMapper sphereMapper
3   sphereMapper SetInputConnection [sphere GetOutputPort] vtkActor
4 sphereActor
5   sphereActor SetMapper sphereMapper
```



(a) With origin $(0, 0, 0)$.(WalkCowA.cxx or WalkCowA.py)



(b) With origin at $(6.1, 1.3, .02)$.(WalkCowB.cxx or WalkCowB.py)

Figure 3.32: The cow rotating about a vector passing through her nose.

```
6    sphereActor SetOrigin 2 1 3
7    sphereActor RotateY 6
8    sphereActor SetPosition 2.25 0 0
9    [sphereActor GetProperty] SetColor 1 0 1
10
11 vtkCubeSource cube
12 vtkPolyDataMapper cubeMapper
13   cubeMapper SetInputConnection [cube GetOutputPort] vtkActor
14 cubeActor
15   cubeActor SetMapper cubeMapper
16   cubeActor SetPosition 0.0 .25 0
17   [cubeActor GetProperty] SetColor 0 0 1
18 vtkConeSource cone
19 vtkPolyDataMapper coneMapper
20   coneMapper SetInputConnection [cone GetOutputPort] vtkActor
21 coneActor
22   coneActor SetMapper coneMapper
23   coneActor SetPosition 0 0 .25
24   [coneActor GetProperty] SetColor 0 1 0
25 vtkCylinderSource cylinder
26 vtkPolyDataMapper cylinderMapper
27   cylinderMapper SetInputConnection [cylinder GetOutputPort]
28   cylinderMapper SetResolveCoincidentTopologyToPolygonOffset
29
30 vtkActor cylinderActor
31   cylinderActor SetMapper cylinderMapper
32   [cylinderActor GetProperty] SetColor 1 0 0
33
34 vtkAssembly assembly
35   assembly AddPart cylinderActor
36   assembly AddPart sphereActor
```

```
37    assembly AddPart cubeActor
38    assembly AddPart coneActor
39    assembly SetOrigin 5 10 15
40
41  // allows faces a specified camera and is used for billboards.
42    assembly AddPosition 5 0 0
43    assembly RotateX 15
44    ren1 AddActor assembly
45    ren1 AddActor coneActor
46  }
```

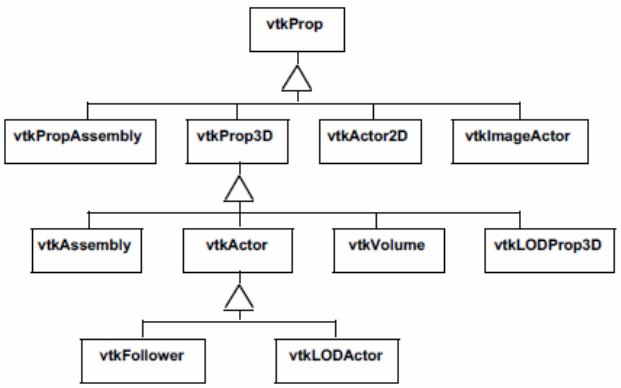


Figure 3.33: The vtkProp hierarchy. Props that can be transformed in 3D space are a subclass of vtkProp3D. Images can be drawn effectively with vtkImageActor. Overlay text and graphics use vtkActor2D. Hierarchical groups of vtkProps are gathered into a vtkPropAssembly. Volume rendering uses vtkVolume. Collections of transformable props create a vtkAssembly. Level-of-detail rendering uses vtkLODProp3D and vtkLODActor. A vtkFollower allows faces a specified camera and is used for billboards.

Note that in this example various actors are added to the assembly with the AddPart() method. The top-level element of the assembly is the only prop in the hierarchy added to the renderer (with AddActor()). Note also that the coneActor appears twice: once as a part of the assembly, and once as a separate actor added to the renderer with AddActor(). As you might imagine, this means that the rendering of assemblies requires concatenation of transformation matrices to insure the correct positioning of each vtkProp3D. Furthermore, hierarchical assemblies require special treatment during picking (i.e., graphically selecting props) since a vtkProp can appear more than once in different assembly hierarchies. Picking issues are discussed in more detail in “Picking” on page 134.

As Figure 3.33 indicates, there are other types of vtkProp as well. Most of these will be informally described in the many examples found in this book. In particular, extensive coverage is given to vtkVolume when we describe volume rendering (see “Volume Rendering” on page 132).

3.12 Chapter Summary

Rendering is the process of generating an image using a computer. Computer graphics is the field of study that encompasses rendering techniques, and forms the foundation of data visualization.

Three-dimensional rendering techniques simulate the interaction of lights and cameras with objects, or actors, to generate images. A scene consists of a combination of lights, cameras, and actors. Object-order rendering techniques generate images by rendering actors in a scene in order. Image-order techniques render the image one pixel at a time. Polygon based graphics hardware is based on object-order techniques. Ray tracing or ray-casting is an image-order technique.

Lighting models require a specification of color. We saw both the RGB (red-green-blue) and HSV (hue-saturation-value) color models. The HSV model is a more natural model than the RGB model for most users. Lighting models also include effects due to ambient, diffuse, and specular lighting.

There are four important coordinate systems in computer graphics. The model system is the 3D coordinate system where our geometry is defined. The world system is the global Cartesian system. All modelled data is eventually transformed into the world system. The view coordinate system represents what is visible to the camera. It is a 2D system scaled from $(-1, 1)$. The display coordinate system uses actual pixel locations on the computer display.

Homogeneous coordinates are a 4D coordinate system in which we can include the effects of perspective transformation. Transformation matrices are 4×4 matrices that operate on homogeneous coordinates. Transformation matrices can represent the effects of translation, scaling, and rotation of an actor. These matrices can be multiplied together to give combined transformations.

Graphics programming is usually implemented using higher-level graphics libraries and specialized hardware systems. These dedicated systems offer better performance and easier implementation of graphics applications. Common techniques implemented in these systems include dithering and z-buffering. Dithering is a technique to simulate colors by mixing combinations of available colors. Z-buffering is a technique to perform hidden-line and hidden-surface removal.

3.13 Bibliographic Notes

This chapter provides the reader with enough information to understand the basic issues and terms used in computer graphics. There are a number of good text books that cover computer graphics in more detail and are recommended to readers who would like a more thorough understanding. The bible of computer graphics is [5]. For those wishing for less intimidating books [2] and [6] are also useful references. You also may wish to peruse proceedings of the ACM SIGGRAPH conferences. These include papers and references to other papers for some of the most important work in computer graphics. [3] provides a good introduction for those who wish to learn more about the human vision system.

Bibliography

[1] J. E. Bresenham. "Algorithm for Computer Control of a Digital Plotter". In: *IBM Systems Journal* 4.1 (Jan. 1965), pp. 25–30.

[2] P. Burger and D. Gillies. *Interactive Computer Graphics Functional, Procedural and Device-Level Methods*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[3] N. R. Carlson. *Physiology of Behaviour (3d Edition)*. Allyn and Bacon Inc., Newton, MA, 1985.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0-201-63361-2. URL: https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201633612.

[5] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice (2d Ed)*. Addison-Wesley, Reading, MA, 1990.

[6] A. Watt. *3D Computer Graphics (2d Edition)*. Addison-Wesley, Reading, MA, 1993.

[7] T. Whitted. "An Improved Illumination Model for Shaded Display". In: *Communications of the ACM* 23.6 (1980), pp. 343–349.

3.14 Exercises

1. Estimate the odds of a ray of light being emitted from the sun, traveling to earth and hitting a one meter square picnic blanket. You can assume that the sun is a point light source that emits light uniformly in all directions. The approximate distance from the sun to the earth is 150,000,000km.

   (a) What are the odds when the sun is directly overhead?

   (b) What are the odds when the sun is inclined 45 degrees relative to the surface normal of the picnic blanket?

   (c) What assumptions or approximations did you make?

2. Proceeding from your result of Exercise 3.1, what are the difficulties in determining the odds of a ray of light traveling from the sun to hit the picnic blanket and then entering a viewer's eye?

3. The color cyan can be represented in both the HSV and RGB color spaces as shown in Table 3.1. These two representations for cyan do not yield the same wavelength intensity plots. How do they differ?

4. The vtkSphereSource class generates a polygonal model of a sphere. Using the examples at the end of this chapter as starting points, create a program to display a white sphere. Set the ambient and diffuse intensities to 0.5. Then add a for-loop to this program that adjusts the ambient and diffuse color of this sphere so that as the loop progresses, the diffuse color goes from red to blue, and the ambient color goes from blue to green. You might also try adjusting other lighting parameters such as specular color, ambient, diffuse, and specular intensity.

5. Using the vtkSphereSource as described in Exercise 3.4, create a program to display the sphere with a light source positioned at $(1, 1, 1)$. Then extend this program by adding a for loop that will adjust the active cameras clipping range so that increasing portions of the interior of the sphere can be seen. By increasing the first value of the clipping range, you will be adjusting the position of the front clipping plane. Once the front clipping plane starts intersecting the sphere, you should be able to see inside of it. The default radius of the vtkSphereSource is 0.5, so make sure that you adjust the clipping range in increments less than 1.0.

6. Modify the program presented in "Render a Cone" on page 55 so that the user can enter in a world coordinate in homogenous coordinates and the program will print out the resulting display coordinate. Refer to the reference page for vtkRenderer for some useful methods.

   (a) Are there any world coordinates that you would expect to be undefined in display coordinates?

   (b) What happens when the world coordinates are behind the camera?

7. Consider rasterizing a ten by ten pixel square. Contrast the approximate difference in the number of arithmetic operations that would need to be done for the cases where it is flat, Gouraud, or Phong shaded.

8. When using a z-buffer, we must also interpolate the z-values (or depth) when rasterizing a primitive. Working from Exercise 3.7, what is the additional burden of computing z-buffer values while rasterizing our square?

9. vtkTransform has a method GetOrientation() that looks at the resulting transformation matrix built from a series of rotations and provides the single x, y, and z rotations that will reproduce the matrix. Specify a series of rotations in a variety of orders and request the orientation with GetOrientation(). Then apply the rotations in the same order that vtkActor does and verify that the resulting 4 x 4 transformation matrix is the same.

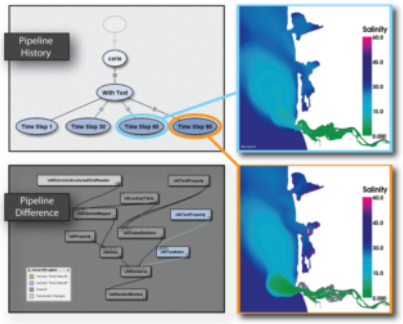
10. vtkTransform, by default, applies new transformations at the right of the current transformation. The method PostMultiply() changes the behavior so that the transformations are applied to the left.

    (a) Use vtkTransform to create a transform using a variety of transformation operators including Scale(), RotateXYZ(), and Translate(). Then create the same matrix with PostMultiplyOn().

    (b) Applying rotations at the right of a series of transformations in effect rotates the object about its own coordinate system. Use the rotations.tcl script to verify this. Can you explain this?

    (c) Applying rotations at the left of a series of transformations in effect rotates the object about the world coordinate system. Modify the rotations.tcl script to illustrate this. (Hint: you will have to create an explicit transform with vtkTransform and set the actor's transform with SetUserMatrix().)

4.0
The Visualization Pipeline



The VisTrails multi-view visualization system. VisTrials enables interactive creation of visualization pipelines, maintaining the history of their evolution, optimizing their execution and allowing multiple pipelines to be compared in a spreadsheet-style layout. Image courtesy of SCI Institute University of Utah.

IN the previous chapter we created graphical images using simple mathematical models for lighting, viewing, and geometry. The lighting model included ambient, diffuse, and specular effects. Viewing included the effects of perspective and projection. Geometry was defined as a static collection of graphics primitives such as points and polygons. In order to describe the process of visualization we need to extend our understanding of geometry to include more complex forms. We will see that the visualization process transforms data into graphics primitives. This chapter examines the process of data transformation and develops a model of data flow for visualization systems.

4.1 Overview

Visualization transforms data into images that efficiently and accurately convey information about the data. Thus, visualization addresses the issues of *transformation* and *representation*.

Transformation is the process of converting data from its original form into graphics primitives, and eventually into computer images. This is our working definition of the visualization process. An example of such a transformation is the process of extracting stock prices and creating an *x-y* plot depicting stock price as a function of time.

Representation includes both the internal data structures used to depict the data and the graphics primitives used to display the data. For example, an array of stock prices and an array of times

are the computational representation of the data, while the x-y plot is the graphical representation. Visualization transforms a computational form into a graphical form.

From an object-oriented viewpoint, transformations are processes in the functional model, while representations are the objects in the object model. Therefore, we characterize the visualization model with both functional models and object models.

4.1.1 A Data Visualization Example

A simple mathematical function for a quadric will clarify these concepts. The function

$$F(x, y, z) = a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a9 \tag{4.1}$$

is the mathematical representation of a quadric. Figure 4.1a shows a visualization of Equation 4.1 in the region $-1 \leqslant x, y, z \leqslant 1$. The visualization process is as follows. We sample the data on a regular grid at a resolution of $50x50x50$. Three different visualization techniques are then used. On the left, we generate 3D surfaces corresponding to the function $F(x, y, z) = c$ where c is an arbitrary constant (i.e., the isosurface value). In the center, we show three different planes that cut through the data and are colored by function value. On the right we show the same three planes that have been contoured with constant valued lines. Around each we place a wireframe outline.

4.1.2 The Functional Model

The functional model in Figure 4.1b illustrates the steps to create the visualization. The oval blocks indicate operations (processes) we performed on the data, and the rectangular blocks represent data stores (objects) that represent and provide access to data. Arrows indicate the direction of data movement. Arrows that point into a block are inputs; data flowing out of a block indicate outputs. The blocks also may have local parameters that serve as additional input. Processes that create data with no input are called data *source* objects, or simply sources. Processes that consume data with no output are called *sinks* (the are also called *mappers* because these processes map data to a final image or output). Processes with both an input and an output are called *filters*.

The functional model shows how data flows through the system. It also describes the dependency of the various parts upon one another. For any given process to execute correctly, all the inputs must be up to date. This suggests that functional models require a synchronization mechanism to insure that the correct output will be generated.

4.1.3 The Visualization Model

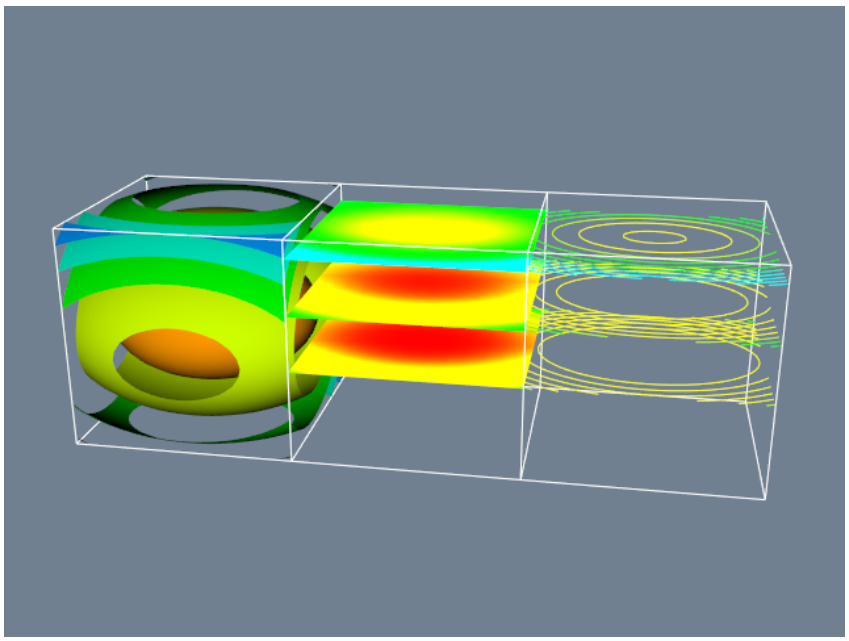
In the examples that follow we will frequently use a simplified representation of the functional model to describe visualization processes (Figure 4.1c). We will not explicitly distinguish between sources, sinks, data stores, and process objects. Sources and sinks are implied based on the number of inputs or outputs. Sources will be process objects with no input. Sinks will be process objects with no output. Filters will be process objects with at least one input and one output. Intermediate data stores will not be represented. Instead we will assume that they exist as necessary to support the data flow. Thus, as Figure 4.1c shows, the *Lines* data store that the *Outline* object generates (Figure 4.1b) are combined into the single object *Outline*. We use oval shapes to represent objects in the visualization model.

4.2 The Visualization Pipeline

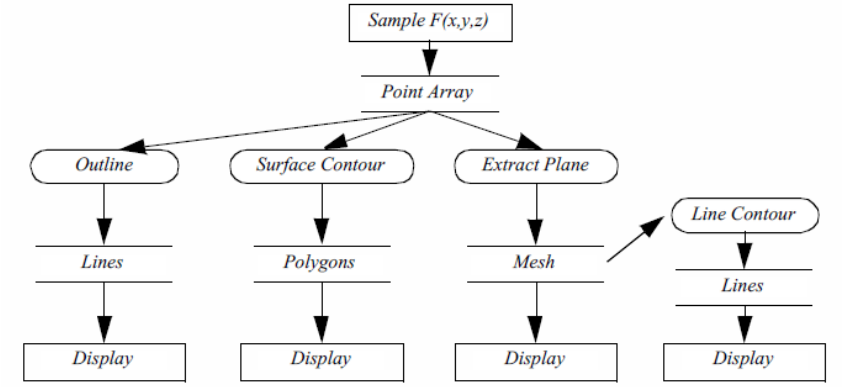
4.2.1 Pipeline Design and Implementation

4.3 Putting it All Together

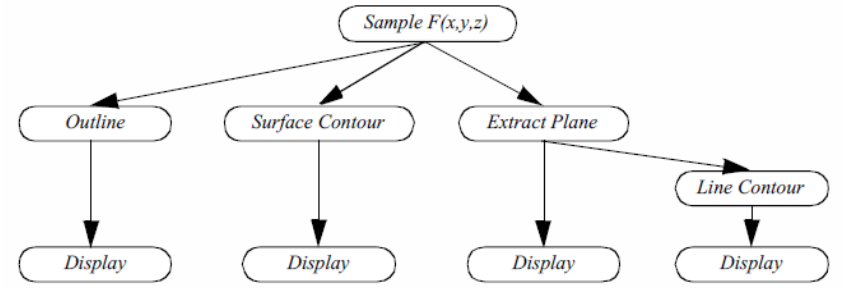
4.3.1 Warped Sphere



(a) Quadric visualization. (QuadricVisualization.cxx or QuadricVisualization.py)

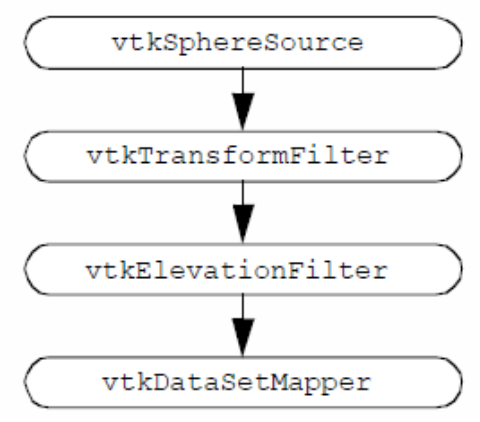


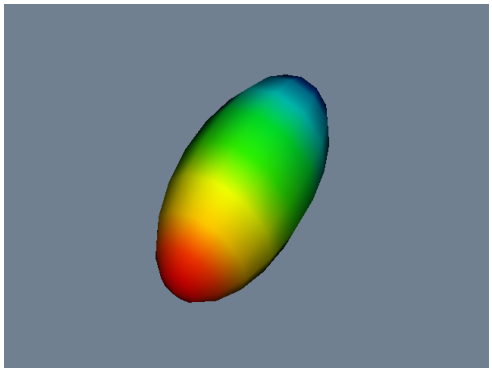
(b) Functional model.



(c) Visualization network.

Figure 4.1: Visualizing a quadric function $F(x, y, z) = c$.





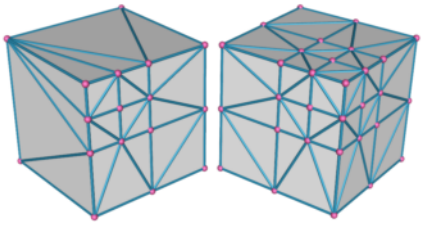
(TransformSphere.cxx or TransformSphere.py)

Listing (4.1) Warped Sphere.

```
1   vtkSphereSource *sphere = vtkSphereSource::New();
2     sphere->SetThetaResolution(12);
3     sphere->SetPhiResolution(12);
4
5   vtkTransform *aTransform = vtkTransform::New();
6     aTransform->Scale(1,1.5,2);
7
8   vtkTransformFilter *transFilter = vtkTransformFilter::New();
9     transFilter->SetInputConnection(sphere->GetOutputPort());
10    transFilter->SetTransform(aTransform);
11
12  vtkElevationFilter *colorIt = vtkElevationFilter::New();
13    colorIt->SetInputConnection(transFilter->GetOutputPort());
14    colorIt->SetLowPoint(0,0,-1);
15    colorIt->SetHighPoint(0,0,1);
16
17  vtkLookupTable *lut = vtkLookupTable::New();
18    lut->SetHueRange(0.667,0.0); lut->SetSaturationRange(1,1);
19    lut->SetValueRange(1,1);
20
21  vtkDataSetMapper *mapper = vtkDataSetMapper::New();
22    mapper->SetLookupTable(lut);
23    mapper->SetInputConnection(colorIt->GetOutputPort());
24
25  vtkActor *actor = vtkActor::New();
26    actor->SetMapper(mapper);
```

Figure 4.2: The addition of a transform filter to the previous example.

5.0
Basic Data Representation



Compatible tessellations.

IN Chapter 4 we developed a pragmatic definition of the visualization process: mapping information into graphics primitives. We saw how this mapping proceeds through one or more steps, each step transforming data from one form, or data representation, into another. In this chapter we examine common data forms for visualization. The goal is to familiarize you with these forms, so that you can visualize your own data using the tools and techniques provided in this text.

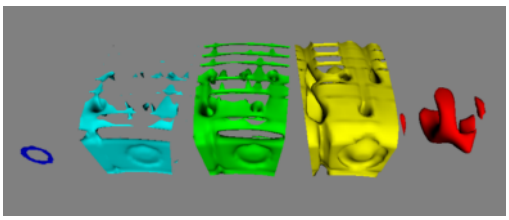
5.1 Introduction

To design representational schemes for data we need to know something about the data we might encounter. We also need to keep in mind design goals, so that we can design efficient data structures and access methods. The next two sections address these issues.

5.1.1 Characterizing Visualization Data

Since our aim is to visualize data, clearly we need to know something about the character of the data. This knowledge will help us create useful data models and powerful visualization systems.

6.0
Fundamental Algorithms



Isosurfaces of a combustor dataset computed at multiple values..

We have seen how to represent basic types of visualization data such as image data, structured grids, unstructured grids, and polygonal data. This chapter explores methods to transform this data to and from these various representations, eventually generating graphics primitives that we can render. These methods are called algorithms, and are of special interest to those working in the field of visualization. Algorithms are the verbs that allow us to express our data in visual form. By combining these verbs appropriately, we can reduce complex data into simple, readily comprehensible sentences that are the power of data visualization.

6.1 Introduction

The algorithms that transform data are the heart of data visualization. To describe the various transformations available, we need to categorize algorithms according to the structure and type of transformation. By structure we mean the effects that transformation has on the topology and geometry of the dataset. By type we mean the type of dataset that the algorithm operates on.

Structural transformations can be classified in four ways, depending on how they affect the geometry, topology, and attributes of a dataset.

- *Geometric transformations* alter input geometry but do not changed the topology of the dataset. For example, if we translate, rotate, and/or scale the points of a polygonal dataset, the topology does not change, but the point coordinates, and therefore the geometry, does.

- *Topological transformations* alter input topology but do not change geometry and attribute data. Converting a dataset type from polygonal data to unstructured grid data, or from image data to unstructured grid, changes the topology but not the geometry. More often, however, the geometry changes whenever the topology does, so topological transformation is uncommon.

- *Attribute transformations* convert data attributes from one form to another, or create new attributes from the input data. The structure of the dataset remains unaffected. Computing vector magnitude or creating scalars based on elevation are data attribute transformations.

- *Combined transformations* change both dataset structure and attribute data. For example, computing contour lines or surfaces is a combined transformation.

We also may classify algorithms according to the type of data they operate on, or the type of data they generate. By type, we most often mean the type of attribute data, such as scalars or vectors. Typical categories include:

- *Scalar algorithms* operate on scalar data. For example, the generation of contour lines of temperature on a weather map.

- *Vector algorithms* operate on vector data. Showing oriented arrows of airflow (direction and magnitude) is an example of vector visualization.

- *Tensor algorithms* operate on tensor matrices. An example of a tensor algorithm is to show the components of stress or strain in a material using oriented icons.

- *Modelling algorithms* generate dataset topology or geometry, or surface normals or texture data. Modelling algorithms tend to be the catch-all category for many algorithms, since some do not fit neatly into any single category mentioned above. For example, generating glyphs oriented according to the vector direction and then scaled according to the scalar value, is a combined scalar/vector algorithm. For convenience we classify such an algorithm as a modelling algorithm, because it does not fit squarely into any other category.

Algorithms also can be classified according to the type of data they process. This is the most common scheme found in the visualization literature. However, this scheme is not without its problems. Often the categories overlap, resulting in confusion. For example, a category (not mentioned above) is *volume visualization*, which refers to the visualization of volume data (or in our terminology, image data). This category was initially created to describe the visualization of scalar data arranged on a volume, but more recently, vector (and even tensor) data has been visualized on a volume. Hence, we have to qualify our techniques to *volume vector visualization*, or other potentially confusing combinations.

In the text that follows, we will use the attribute type classification scheme: scalar, vector, tensor, and modelling. In cases where the algorithms operate on a particular dataset type, we place them in the appropriate category according to our best judgment. Be forewarned, though, that alternative classification schemes do exist, and may be better suited to describing the true nature of the algorithm.

6.1.1 Generality Versus Efficiency

Most algorithms can be written specifically for a particular dataset type, or more generally, treating any dataset type. The advantage of a specific algorithm is that it is usually faster than a comparable general algorithm. (See "Other Data Abstractions" on page138 where we discussed the tradeoff between abstract and concrete forms.) An implementation of a specific algorithm also may be more memory efficient and its implementation may better reflect the relationship between the algorithm and the dataset type it operates on.

One example of this is contour surface creation. Algorithms for extracting contour surfaces were originally developed for volume data, mainly for medical applications. The regularity of volumes lends itself to efficient algorithms. However, the specialization of volume-based algorithms

$$s_i = min, i = 0$$
$$s_i > max, i = n - 1$$
$$i = n\left(\frac{s_i - min}{max - min}\right)$$

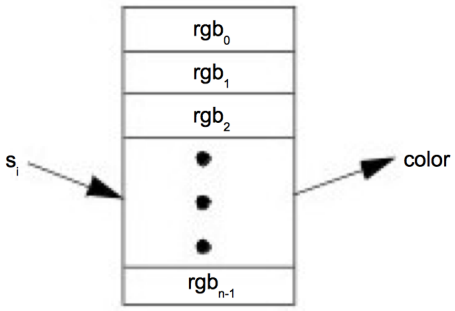


Figure 6.1: Mapping scalars to colors via a lookup table.

precludes their use for more general datasets such as structured or unstructured grids. Although the contour algorithms can be adapted to these other dataset types, they are less efficient than those for volume datasets.

Our presentation of algorithms favors the more general implementations. In some special cases we will describe performance improving techniques for particular dataset types. Refer to the bibliography at the end of each chapter for detailed descriptions of specialized algorithms.

6.2 Scalar Algorithms

Scalars are single data values associated with each point and/or cell of a dataset. (Recall that in the *Visualization Toolkit* we associate data with points.) Because scalar data is commonly found in real-world applications, and because scalar data is so easy to work with, there are many different algorithms to visualize it.

6.2.1 Color Mapping

Color mapping is a common scalar visualization technique that maps scalar data to colors, and displays the colors on the computer system. The scalar mapping is implemented by indexing into a *color lookup table*. Scalar values serve as indices into the lookup table.

The mapping proceeds as follows. The lookup table holds an array of colors (e.g., red, green, blue components or other comparable representations). Associated with the table is a minimum and maximum *scalar range (min, max)* into which the scalar values are mapped. Scalar values greater than the maximum range are clamped to the maximum color, scalar values less than the minimum range are clamped to the minimum color value. Then, for each scalar value x_i, the index i into the color table with n entries (and 0-offset) is given by Figure 6.1.

A more general form of the lookup table is called a transfer function. A transfer function is any expression that maps scalar values into a color specification. For example, Figure 6.2 maps scalar values into separate intensity values for the red, green, and blue color components. We can also use transfer functions to map scalar data into other information such as local transparency. (Transfer functions are discussed in more detail in "Transparency and Alpha Values" on page 131 and "Volume Rendering" on page 132). A lookup table is a discrete sampling of a transfer function. We can create a lookup table from any transfer function by sampling the transfer function at a set of discrete points.

Color mapping is a one-dimensional visualization technique. It maps one piece of information (i.e., a scalar value) into a color specification. However, the display of color information is not limited to one-dimensional displays. Often we use color information mapped onto 1D, 2D, or 3D objects. This is a simple way to increase the information content of our visualizations.

The key to color mapping for scalar visualization is to choose the lookup table entries carefully. Figure 6.3 shows four different lookup tables used to visualize gas density as fluid flows through a combustion chamber. The first lookup table is gray-scale. Grayscale tables often provide better structural detail to the eye. The other three images in Figure 6.3 uses different colored lookup tables. The second uses rainbow hues from blue to red. The third uses rainbow hues arranged from red to blue. The last table uses a table designed to enhance contrast. Careful use of colors can often enhance important features of a dataset. However, any type of lookup table can exaggerate unimportant details or create visual artifacts because of unforeseen interactions between data, color choice, and human physiology.

Designing lookup tables is as much art as it is science. From a practical point of view, tables should accentuate important features, while minimizing less important or extraneous details. It is also desirable to use palettes that inherently contain scaling information. For example, a color rainbow scale from blue to red is often used to represent temperature scale, since many people associate "blue" with cold temperatures, and "red" with hot temperatures. However, even this scale is problematic: a physicist would say that blue is hotter than red, since hotter objects emit more blue light (i.e., shorter wavelength) than red. Also, there is no need to limit ourselves to "linear" lookup tables. Even though the mapping of scalars into colors has been presented as a linear operation (Figure 6.1, the table itself need not be linear. That is, tables can be designed to enhance small variations in scalar value using logarithmic or other schemes, improving the comfort level and engaging the human observer more deeply in the presentation of data improves the effectiveness of communication.

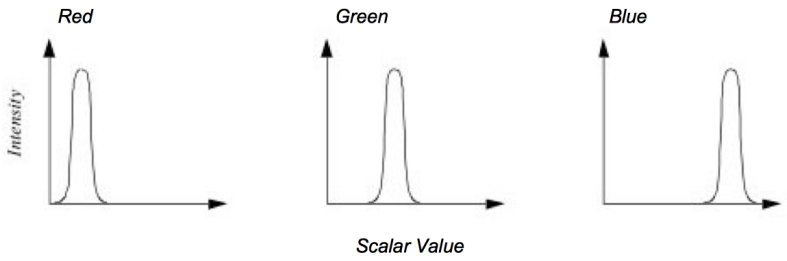


Figure 6.2: Transfer function for color components red, green and blue as a function of scalar value.

6.2.2 Contouring

A natural extension to color mapping is *contouring*. When we see a surface colored with data values, the eye often separates similarly colored areas into distinct regions. When we contour data, we are effectively constructing the boundary between these regions. These boundaries correspond to contour lines (2D) or surfaces (3D) of constant scalar value.

Examples of 2D contour displays include weather maps annotated with lines of constant temperature (isotherms), or topological maps drawn with lines of constant elevation. Three-dimensional contours are called *isosurfaces*, and can be approximated by many polygonal primitives. Examples of isosurfaces include constant medical image intensity corresponding to body tissues such as skin,

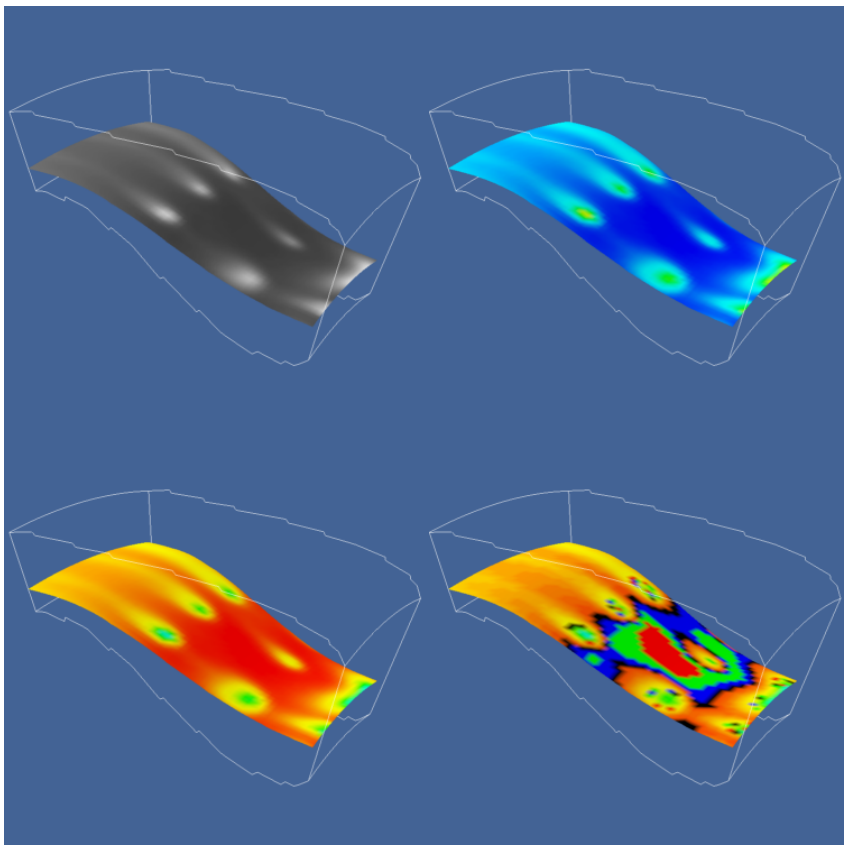


Figure 6.3: Flow density colored with different lookup tables. Top-left: grayscale; Top-right rainbow (blue to red); lower-left rainbow (red to blue); lower-right large contrast. (Rainbow.cxx) and (Rainbow.py)

bone, or other organs. Other abstract isosurfaces such as surfaces of constant pressure or temperature in fluid flow also may be created.

Consider the 2D structured grid shown in Figure 6.4. Scalar values are shown next to the points that define the grid. Contouring always begins by selecting a scalar value, or contour value, that corresponds to the contour lines or surfaces generated. To generate the contours, some form of interpolation must be used. This is because we have scalar values at a finite set of points in the dataset, and our contour value may lie between the point values. Since the most common interpolation technique is linear, we generate points on the contour surface by linear interpolation along the edges. If an edge has scalar values 10 and 0 at its two endpoints, and if we are trying to generate a contour line of value 5, then edge interpolation computes that the contour passes through the midpoint of the edge.

Once the points on cell edges are generated, we can connect these points into contours using a few different approaches. One approach detects an edge intersection (i.e., the contour passes through an edge) and then "tracks" this contour as it moves across cell boundaries. We know that if a contour edge enters a cell, it must exit a cell as well. The contour is tracked until it closes back on itself, or exits a dataset boundary. If it is known that only a single contour exists, then the process stops. Otherwise, every edge in the dataset must be checked to see whether other contour lines exist. Another approach uses a divide and conquer technique, treating cells independently. This is

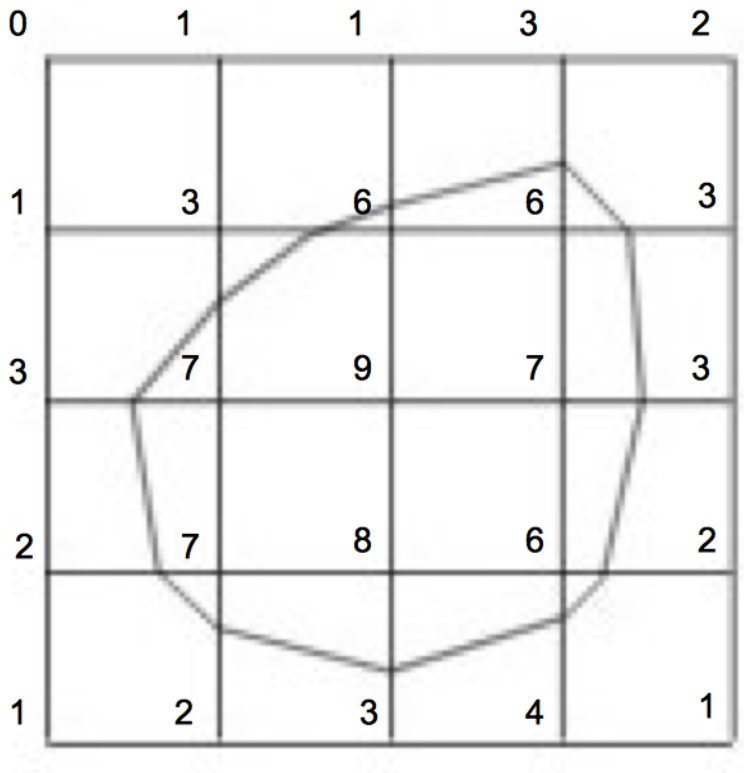


Figure 6.4: Contouring a 2D stuctured grid with contour line value = 5.

the *marching squares* algorithm in 2D, and *marching cubes* [16] in 3D. The basic assumption of these techniques is that a contour can only pass through a cell in a finite number of ways. A case table is constructed that enumerates all possible topological *states* of a cell, given combinations of scalar values at the cell points. The number of topological states depends on the number of cell vertices, and the number of inside / outside relationships a vertex can have with respect to the contour value. A vertex is considered inside a contour if its scalar value is larger than the scalar value of the contour line. Vertices with scalar values less than the contour value are said to be outside the contour. For example, if a cell has four vertices and each vertex can be either inside or outside the contour, there are $2^4 = 16$ possible ways that the contour passes through the cell. In the case table we are not interested in where the contour passes through the cell (e.g., geometric intersection), just how it passes through the cell (i.e., topology of the contour in the cell).

Figure 6.5 shows the sixteen combinations for a square cell. An index into the case table can be computed by encoding the state of each vertex as a binary digit. For 2D data represented on a rectangular grid, we can represent the 16 cases with 4 bit index. Once the proper case is selected, the location of the contour line / cell edge intersection can be calculated using interpolation. The algorithm processes a cell and then moves, or *marches t* o the next cell. After all cells are visited, the contour will be completed. In summary, the marching algorithms proceed as follows:

1. Select a cell.
2. Calculate the inside / outside state of each vertex of the cell.
3. Create an index by storing the binary state of each vertex in a separate bit.

4. Use the index to look up the topological state of the cell in a case table.

5. Calculate the contour location (via interpolation) for each edge in the case table.

This procedure will construct independent geometric primitives in each cell. At the cell boundaries duplicate vertices and edges may be created. These duplicates can be eliminated by using a special coincident point merging operation. Note that interpolation along each edge should be done in the same direction. If not, numerical roundoff will likely cause points to be generated that are not precisely coincident, and will not merge properly.

There are advantages and disadvantages to both the edge-tracking and marching cubes approaches. The marching squares algorithm is easy to implement. This is particularly important when we extend the technique into three dimensions, where isosurface tracking becomes much more difficult. On the other hand, the algorithm creates disconnected line segments and points, and the required merging operation requires extra computation resources. The tracking algorithm can be implemented to generate a single polyline per contour line, avoiding the need to merge coincident points.

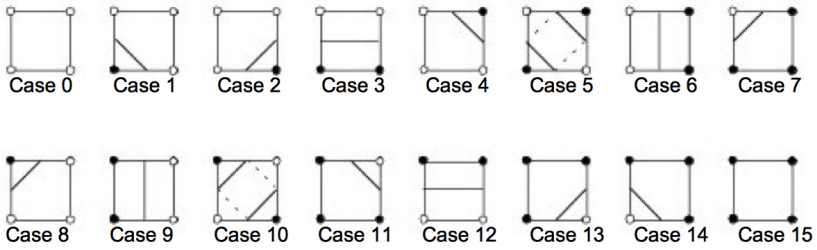


Figure 6.5: Sixteen different marching squares cases. Dark vertices indicate scalar value is above contour value. Cases 5 and 10 are ambiguous.

As mentioned previously, the 3D analogy of marching squares is marching cubes. Here, there are 256 different combinations of scalar value, given that there are eight points in a cubical cell (i.e., 2^8 combinations). Figure 6.6 shows these combinations reduced to 15 cases by using arguments of symmetry. We use combinations of rotation and mirroring to produce topologically equivalent cases.

In two dimensions, contour ambiguity is simple to treat: for each ambiguous case we implement one of the two possible cases. The choice for a particular case is independent of all other choices. Depending on the choice, the contour may either extend or break the current contour as illustrated in Figure 6.9. Either choice is acceptable since the resulting contour lines will be continuous and closed (or will end at the dataset boundary).

In three dimensions the problem is more complex. We cannot simply choose an ambiguous case independent of all other ambiguous cases. For example Figure 6.9 shows what happens if we carelessly implement two cases independent of one another. In this figure we have used the usual case 3 but replaced case 6 with its *complementary* case. Complementary cases are formed by exchanging the "dark" vertices with "light" vertices. (This is equivalent to swapping vertex scalar value from above the isosurface value to below the isosurface value, and vice versa.) The result of pairing these two cases is that a hole is left in the isosurface.

Several different approaches have been taken to remedy this problem. One approach tessellates the cubes with tetrahedron, and uses a *marching tetrahedra* technique. This works because the marching tetrahedra exhibit no ambiguous cases. Unfortunately, the marching tetrahedra algorithm generates isosurfaces consisting of more triangles, and the tessellation of a cube with tetrahedra

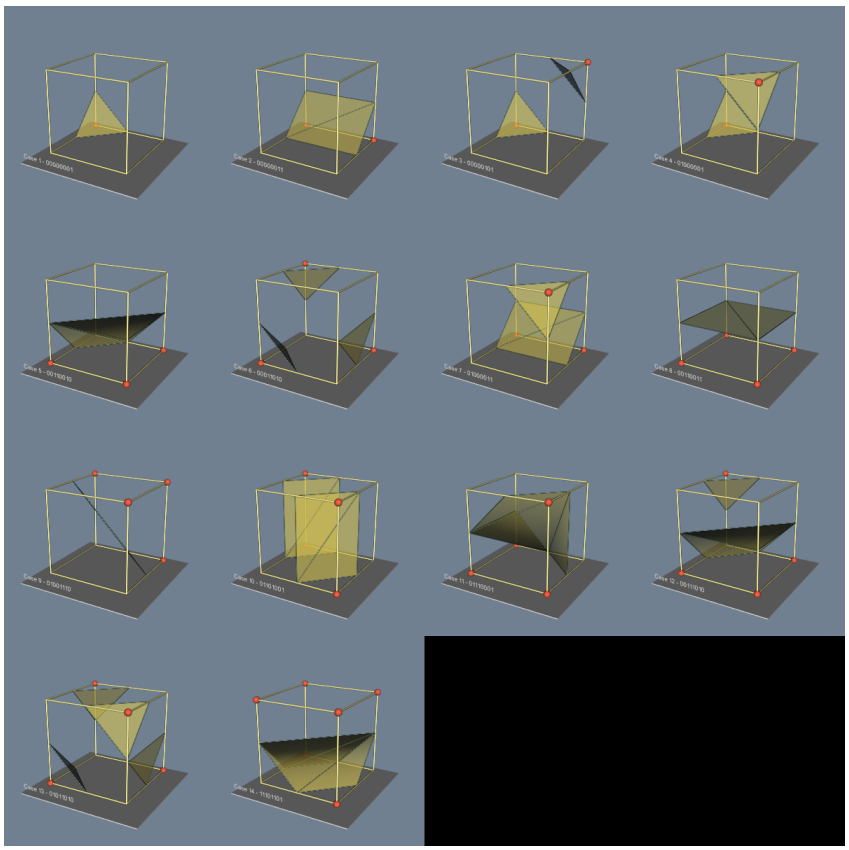


Figure 6.6: Marching Cubes cases for 3D isosurface generation. The 256 possible cases have been reduced to 15 cases using symmetry. Red vertices are greater than the selected isosurface value. (MarchingCasesA.cxx) and (MarchingCasesA.py)

requires making a choice regarding the orientation of the tetrahedra. This choice may result in artificial "bumps" in the isosurface because of interpolation along the face diagonals as shown in Figure 6.7. Another approach evaluates the asymptotic behavior of the surface, and then chooses the cases to either join or break the contour. Nielson and Hamann [19] have developed a technique based on this approach they call the *asymptotic decider*. It is based on an analysis of the variation of the scalar variable across an ambiguous face. The analysis determines how the edges of isosurface polygons should be connected.

A simple and effective solution extends the original 15 marching cubes cases by adding additional complementary cases. These cases are designed to be compatible with neighboring cases and prevent the creation of holes in the isosurface. There are six complementary cases required, corresponding to the marching cubes cases 3, 6, 7, 10, 12, and 13. The complementary marching cubes cases are shown in Figure 6.10.

We can extend the general approach of marching squares and marching cubes to other topological types. In VTK we use marching lines, triangles, and tetrahedra to contour cells of these types (or composite cells that are composed of these types). In addition, although we speak of regular types such as squares and cubes, marching cubes can be applied to any cell type topologically equivalent to a cube (e.g., hexahedron or non-cubical voxel).

Figure 6.11 shows four applications of contouring. In Figure 6.11a we see 2D contour lines of

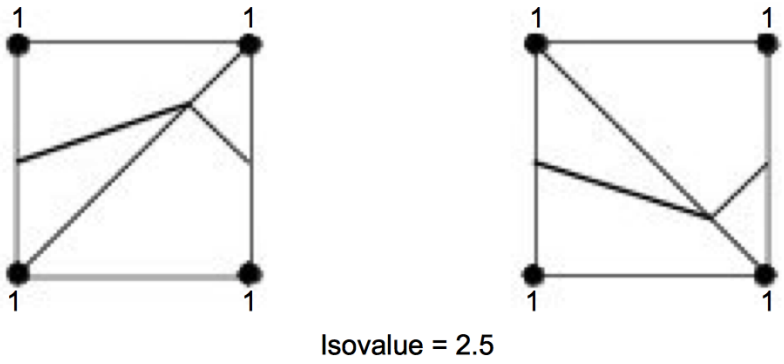


Figure 6.7: Using marching triangles or marching tetrahedra to resolve ambiguous cases on rectangular lattice (only face of cube is shown). Choice of diagonal orientation may result in "bumps" in contour surface. In 2D, diagonal orientation can be chosen arbitrarily, but in 3D diagonal is constrained by neighbor.

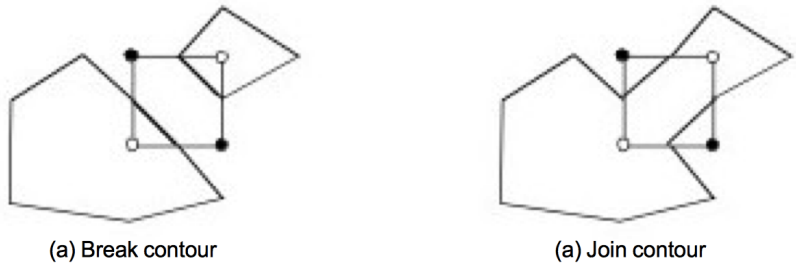


Figure 6.8: Choosing a particular contour case will break (a) or join (b) the current contour. Case shown is marching squares case 10.

CT density value corresponding to different tissue types. These lines were generated using marching squares. Figure 6.11b through Figure 6.11d are isosurfaces created by marching cubes. Figure 6.11b is a surface of constant image intensity from a computed tomography (CT) Xray imaging system. (Figure 6.11a is a 2D subset of this data.) The intensity level corresponds to human bone. Figure 6.11c is an isosurface of constant flow density. Figure 6.11d is an isosurface of electron potential of an iron protein molecule. The image shown in Figure 6.11b is immediately recognizable because of our familiarity with human anatomy. However, for those practitioners in the fields of computational fluid dynamics and molecular biology, Figure 6.11c and Figure 6.11d are equally familiar. As these examples show, methods for contouring are powerful yet general techniques for visualizing data from a variety of fields.

6.2.3 Scalar Generation

The two visualization techniques presented thus far, color mapping and contouring, are simple, effective methods to display scalar information. It is natural to turn to these techniques first when visualizing data. However, often our data is not in a form convenient to these techniques. The data may not be single-valued (i.e., a scalar), or it may be a mathematical or other complex rela-

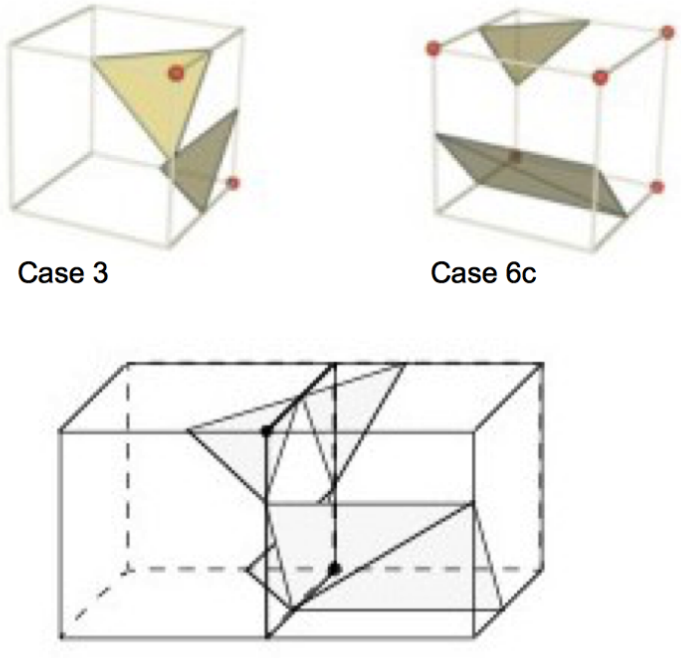


Figure 6.9: Arbitrarily choosing marching cubes cases leads to holes in the isosurface.

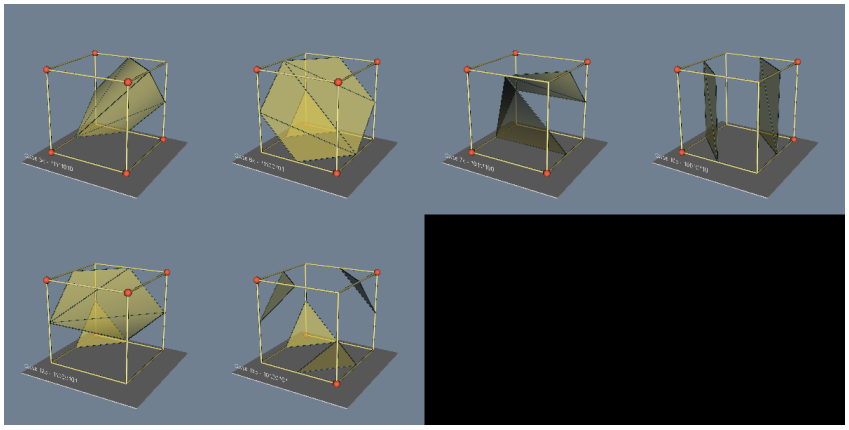
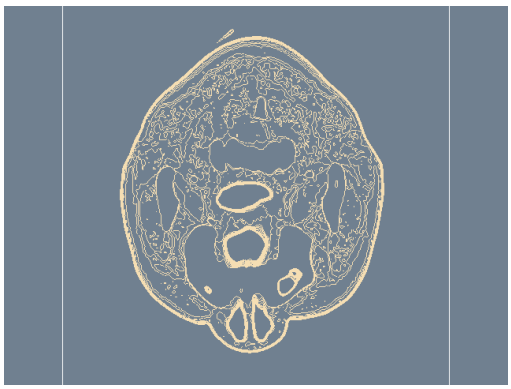


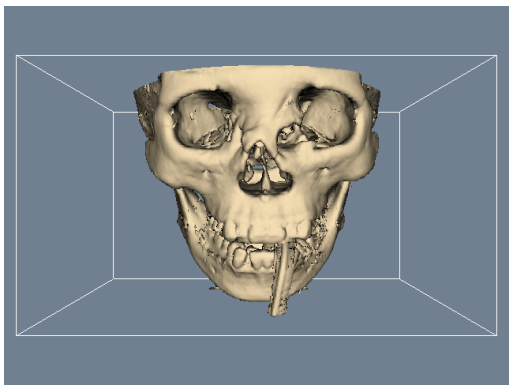
Figure 6.10: Marching cubes complementary cases.(MarchingCasesB.cxx) and (MarchingCasesB.py)

tionship. That is part of the fun and creative challenge of visualization: We must tap our creative resources to convert data into a form we can visualize.

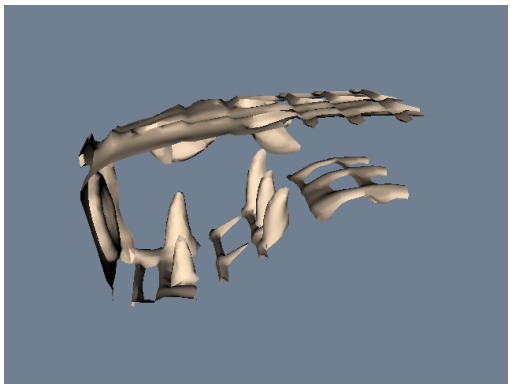
For example, consider terrain data. We assume that the data is *xyz* coordinates, where *x* and *y* represent the coordinates in the plane, and *z* represents the elevation above sea level. Our desired visualization is to color the terrain according to elevation. This requires creating a color map — possibly using white for high altitudes, blue for sea level and below, and various shades of green



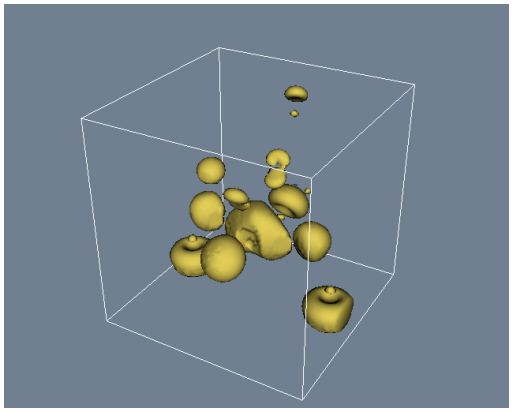
(a) Marching squares used to generate contour lines.(HeadSlice.cxx or HeadSlice.py)



(b) Marching Cubes surface of human bone.(HeadBone.cxx or HeadBone.py)



(c) Marching squares used to generate contour lines.(CombustorIsosurface.cxx or CombustorIsosurface.py)



(d) Marching Cubes surface of human bone.(IronIsoSurface.cxx or IronIsoSurface.py)

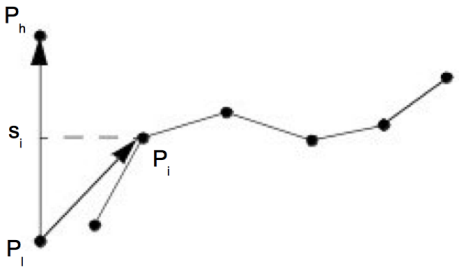
Figure 6.11: Contouring examples.

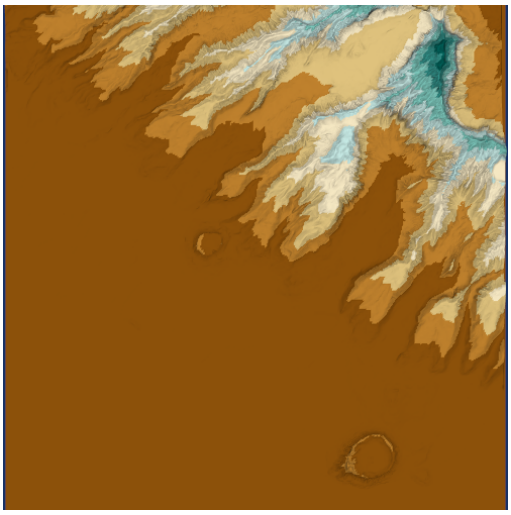
and brown corresponding to elevation between sea level and high altitude. We also need scalars to index into the color map. The obvious choice here is to extract the z coordinate. That is, scalars are simply the z-coordinate value.

This example can be made more interesting by generalizing the problem. Although we could easily create a filter to extract the z-coordinate, we can create a filter that produces elevation scalar values where the elevation is measured along any axis. Given an oriented line starting at the (low) point p_l (e.g., sea level) and ending at the (high) point p_h (e.g., mountain top), we compute the elevation scalar s_i at point $p_i = (x_i, y_i, z_i)$ using the dot product as shown in Figure 6.12. The scalar is normalized using the magnitude of the oriented line, and may be clamped between minimum and maximum scalar values (if necessary). The bottom half of this figure shows the results of applying this technique to a terrain model of Honolulu, Hawaii. A lookup table of 256 colors ranging from deep brown (water) to dark turquoise (mountain top) is used to color map this figure.

Part of the creative practice of visualization is selecting the best technique for given data from the palette of available techniques. Often this requires creative mapping by the user of the visualization system. In particular, to use scalar visualization techniques we need only to create a relationship to generate a unique scalar value. Other examples of scalar mapping include an index value into a list of data, computing vector magnitude or matrix determinate, evaluating surface

$$s_i = \frac{(p_i - p_j) \cdot (p_h - p_l)}{|p_h - p_l|^2}$$





Applying the technique to terrain data from Honolulu, Hawaii.(Hawaii.cxx or Hawaii.py)

Figure 6.12: Computing scalars using normalized dot product.

curvature, or determining distance between points. Scalar generation, when coupled with color mapping or contouring, is a simple, yet effective, technique for visualizing many types of data.

6.3 Vector Algorithms

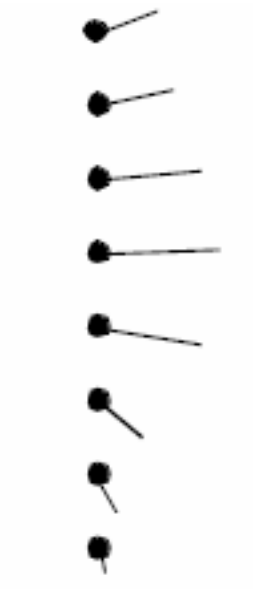
Vector data is a three-dimensional representation of direction and magnitude. Vector data often results from the study of fluid flow, or when examining derivatives (i.e., rate of change) of some quantity.

6.3.1 Hedgehogs and Oriented Glyphs

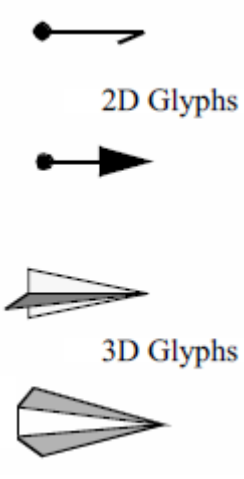
A natural vector visualization technique is to draw an oriented, scaled line for each vector (Figure 6.13a). The line begins at the point with which the vector is associated and is oriented in the direction of the vector components (v_x, v_y, v_z). Typically, the resulting line must be scaled up or down to control the size of its visual representation. This technique is often referred to as a *hedgehog* because of the bristly result.

There are many variations of this technique Figure 6.13b). Arrows may be added to indicate the direction of the line. The lines may be colored according to vector magnitude, or some other

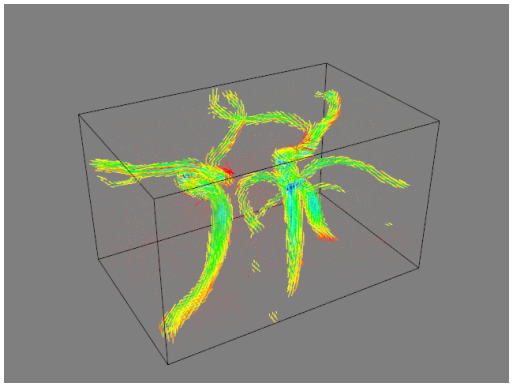
scalar quantity (e.g., pressure or temperature). Also, instead of using a line, oriented "glyphs" can be used. By glyph we mean any 2D or 3D geometric representation such as an oriented triangle or cone.



(a) Oriented lines.



(b) Glyphs



(c) Complex vector visualization. (TestComplexV.cxx or TestComplexV.py)

Figure 6.13: Vector visualization techniques.

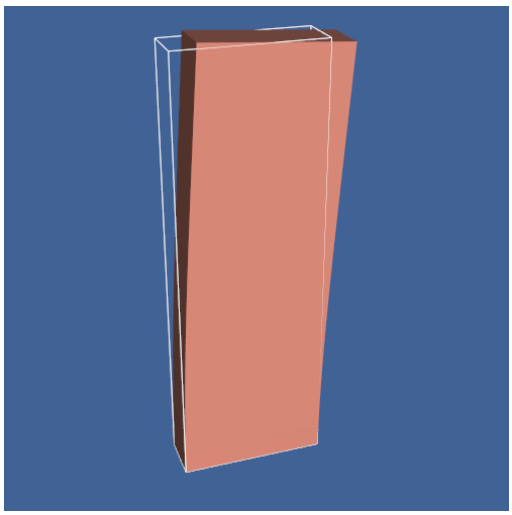
Care should be used in applying these techniques. In 3D it is often difficult to understand the position and orientation of a vector because of its projection into a 2D image. Also, using large numbers of vectors can clutter the display to the point where the visualization becomes meaningless. Figure 6.13c shows 167,000 3D vectors (using oriented and scaled lines) in the region of the human carotid artery. The larger vectors lie inside the arteries, the smaller vectors lie outside the arteries and are randomly oriented (measurement error) but small in magnitude. Clearly the details of the vector field are not discernible from this image.

Scaling glyphs also poses interesting problems. In what Tufte has termed a "visualization lie", [24] scaling a 2D or 3D glyph results in nonlinear differences in appearance. The surface area of an object increases with the square of its scale factor, so two vectors differing by a factor of two in magnitude may appear up to four times different based on surface area. Such scaling issues are common in data visualization, and great care must be taken to avoiding misleading viewers.

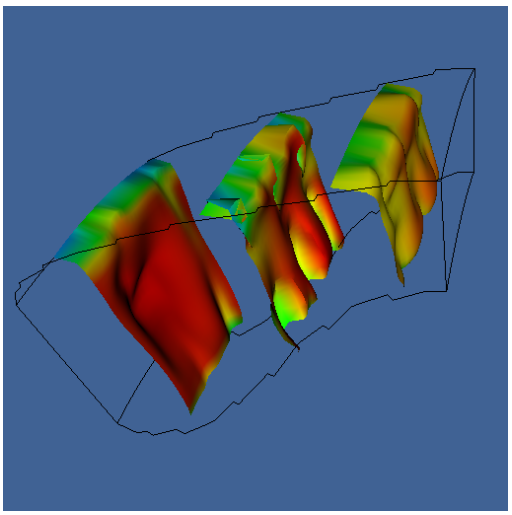
6.3.2 Warping

Vector data is often associated with "motion". The motion is in the form of velocity or displacement. An effective technique for displaying such vector data is to "warp" or deform geometry according to the vector field. For example, imagine representing the displacement of a structure under load by deforming the structure. Or if we are visualizing the flow of fluid, we can create a flow profile by distorting a straight line inserted perpendicular to the flow.

Figure 6.14 shows two examples of vector warping. In the first example the motion of a vibrating beam is shown. The original undeformed outline is shown in wireframe. The second example shows warped planes in a structured grid dataset. The planes are warped according to flow



(a) Vibration of beam. (PlateVibration.cxx or PlateVibration.py)



(b) Momentum profiles. (VelocityProfile.cxx or VelocityProfile.py)

Figure 6.14: Warping geometry to show vector field.

momentum. The relative back and forward flow are clearly visible in the deformation of the planes.

Typically, we must scale the vector field to control geometric distortion. Too small a distortion may not be visible, while too large a distortion can cause the structure to turn inside out or selfintersect. In such a case the viewer of the visualization is likely to lose context, and the visualization will become ineffective.

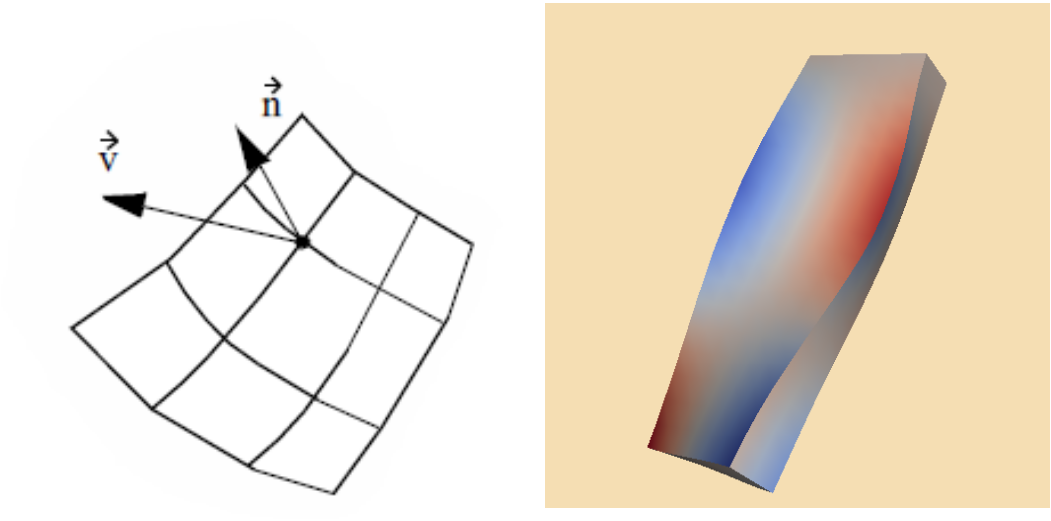
6.3.3 Displacement Plots

Vector displacement on the surface of an object can be visualized with displacement plots. A displacement plot shows the motion of an object in the direction perpendicular to its surface. The object motion is caused by an applied vector field. In a typical application the vector field is a displacement or strain field.

Vector displacement plots draw on the ideas in "Scalar Generation" on page 89. Vectors are converted to scalars by computing the dot product between the surface normal and vector at each point (Figure 6.15a). If positive values result, the motion at the point is in the direction of the surface normal (i.e., positive displacement). Negative values indicate that the motion is opposite the surface normal (i.e., negative displacement).

A useful application of this technique is the study of vibration. In vibration analysis, we are interested in the eigenvalues (i.e., natural resonant frequencies) and eigenvectors (i.e., mode shapes) of a structure. To understand mode shapes we can use displacement plots to indicate regions of motion. There are special regions in the structure where positive displacement changes to negative displacement. These are regions of zero displacement. When plotted on the surface of the structure, these regions appear as the so-called *modal* lines of vibration. The study of modal lines has long been an important visualization tool for understanding mode shapes.

Figure 6.15b shows modal lines for a vibrating rectangular beam. The vibration mode in this figure is the second torsional mode, clearly indicated by the crossing modal lines. (The aliasing in the figure is because of the coarseness of the analysis mesh.) To create the figure we combined the procedure of Figure 6.15a with a special lookup table. The lookup table was arranged with white



(a) Scalar computation $S = \vec{v} \cdot \vec{n}$.

(b) Displacement plot. (DisplacementPlot.cxx or DisplacementPlot.py)

Figure 6.15: Vector displacement plots. (a) Vector converted to scalar via dot product computation; (b) Surface plot of vibrating plate. Dark areas show nodal lines. Bright areas show maximum motion.

areas in the center (i.e., corresponds to zero dot product) and dark areas at the beginning and end of the table (corresponds to 1 or −1 dot product). As a result, regions of large normal displacement are dark and regions near the modal lines are light.

6.3.4 Time Animation

Some of the techniques described so far can be thought of as moving a point or object over a small time step. The hedgehog line is an approximation of a point's motion over a time period whose duration is given by the scale factor. In other words, if velocity $\vec{V} = dx/dt$, then displacement of a point is

$$dx = \vec{V}\, dt \tag{6.1}$$

This suggests an extension to our previous techniques: repeatedly displace points over many time steps. Figure 6.16 shows such an approach. Beginning with a sphere S centered about some point C, we move S repeatedly to generate the bubbles shown. The eye tends to trace out a path by connecting the bubbles, giving the observer a qualitative understanding of the fluid flow in that area. The bubbles may be displayed as an animation over time (giving the illusion of motion) or as a multiple exposure sequence (giving the appearance of a path).

Such an approach can be misused. For one thing, the velocity at a point is instantaneous.

Once we move away from the point the velocity is likely to change. Using Equation 6.1 assumes that the velocity is constant over the entire step. By taking large steps we are likely to jump over changes in the velocity. Using smaller steps we will end in a different position. Thus the choice of step size is a critical parameter in constructing accurate visualization of particle paths in a vector field.

To evaluate Equation 6.1 we can express it as an integral:

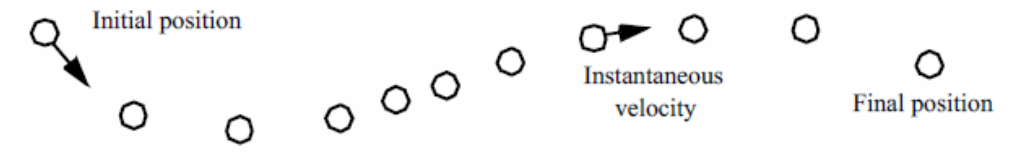


Figure 6.16: Time animation of a point C. Although the spacing between points varies, the time increment between each point is constant.

$$\vec{x}(t) = \int_t \vec{V}\, dt \tag{6.2}$$

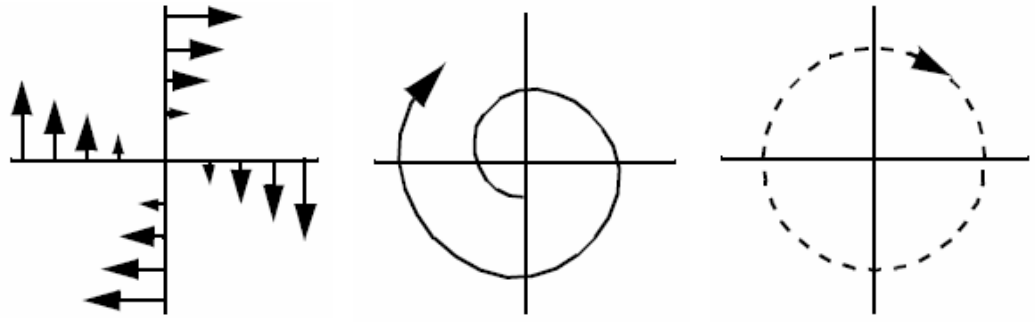
Although this form cannot be solved analytically for most real world data, its solution can be approximated using numerical integration techniques. Accurate numerical integration is a topic beyond the scope of this book, but it is known that the accuracy of the integration is a function of the step size dt. Since the path is an integration throughout the dataset, the accuracy of the cell interpolation functions, as well as the accuracy of the original vector data, plays an important role in realizing accurate solutions. No definitive study is yet available that relates cell size or interpolation function characteristics to visualization error. But the lesson is clear: the result of numerical integration must be examined carefully, especially in regions of large vector field gradient. However, as with many other visualization algorithms, the insight gained by using vector integration techniques is qualitatively beneficial, despite the unavoidable numerical errors.

The simplest form of numerical integration is Euler's method,

$$\vec{x}_{i+1} = \vec{x}_i + \vec{V}_i \Delta t \tag{6.3}$$

where the position at time is the $\vec{x}_{i+1}$ vector sum of the previous position plus the instantaneous velocity times the incremental time step Δt.

Euler's method has error on the order of $O(\Delta t^2)$, which is not accurate enough for some applications. One such example is shown in Figure 6.17. The velocity field describes perfect rotation about a central point. Using Euler's method we find that we will always diverge and, instead of generating circles, will generate spirals instead.



(a) Rotational vector field. (b) Eulers method. (c) Runge-Kutta..

Figure 6.17: Eulers integration (b) and Runge-Kutta integration of order 2 (c) applied to uniform rotational vector field (a). Eulers method will always diverge.

In this text we will use the Runge-Kutta technique of order 2 [7]. This is given by the expression

$$\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta t}{2}(\vec{V}_i + \vec{V}_{i+1}) \tag{6.4}$$

One final note about accuracy concerns. The errors involved in either perception or computation of visualizations is an open research area. The discussion in the preceding paragraph is a good example of this. There we characterized the error in streamline integration using conventional numerical integration arguments. But there is a problem with this argument. In visualization applications, we are integrating across cells whose function values are continuous, but whose derivatives are not. As the streamline crosses the cell boundary, subtle effects may occur that are not treated by the standard numerical analysis. Thus the standard arguments need to be extended for visualization applications.

Integration formulas require repeated transformation from global to local coordinates. Consider moving a point through a dataset under the influence of a vector field. The first step is to identify the cell that contains the point. This operation is a search (see "Searching" on page 134), plus a conversion to local coordinates. Once the cell is found, then the next step is to compute the velocity at that point by interpolating the velocity from the cell points. The point is then incrementally repositioned (using the integration formula Equation 6.4). The process is then repeated until the point exits the dataset or the distance or time traversed exceeds some specified value.

This process can be computationally demanding. There are two important steps we can take to improve performance.

1. *Improving search procedures.* There are two distinct types of searches. Initially, the starting location of the particle must be determined by a global search procedure. Once the initial location of the point is determined in the dataset, an incremental search procedure can then be used. Incremental searching is efficient because the motion of the point is limited within a single cell, or at most across a cell boundary. Thus, the search space is greatly limited, and the incremental search is faster relative to the global search.

2. *Coordinate transformation.* The cost of a coordinate transformation from global to local coordinates can be reduced if either of the following conditions are true: the local and global coordinate systems are identical with one another (or vary by *xyz* translation), or if the vector field is transformed from global space to local coordinate space. The image data coordinate system is an example of a local coordinates which are parallel to global coordinates, hence global to local coordinate transformation can be greatly accelerated. If the vector field is transformed into local coordinates (either as a preprocessing step or on a cell by cell basis), then the integration can proceed completely in local space. Once the integration path is computed, selected points along the path can be transformed into global space for the sake of visualization.

6.3.5 Streamlines

A natural extension of the previous time animation techniques is to connect the point position over many time steps. The result is a numerical approximation to a particle trace represented as a line.

Borrowing terminology from the study of fluid flow, we can define three related line representation schemes for vector fields.

- *Particle traces* are trajectories traced by fluid particles over time.

- *Streaklines* are the set of particle traces at a particular time t_i through a specified point x_i.
- *Streamlines* are integral curves along a curve s satisfying the equation

$$s = \int_t \overrightarrow{V}\, ds, \text{ with } s = (x, t) \tag{6.5}$$

for a particular time t.

Streamlines, streaklines, and particle traces are equivalent to one another if the flow is steady. In time–varying flow, a given streamline exists only at one moment in time. Visualization systems generally provide facilities to compute particle traces. However, if time is fixed, the same facility can be used to compute streamlines. In general, we will use the term streamline to refer to the method of tracing trajectories in a vector field. Please bear in mind the differences in these representations if the flow is time–varying.

Figure 6.18 shows forty streamlines in a small kitchen. The room has two windows, a door (with air leakage), and a cooking area with a hot stove. The air leakage and temperature variation combine to produce air convection currents throughout the kitchen. The starting positions of the streamlines were defined by creating a *rake*, or curve (and its associated points). Here the rake was a straight line. These streamlines clearly show features of the flow field. By releasing many streamlines simultaneously we obtain even more information, as the eye tends to assemble nearby streamlines into a "global" understanding of flow field features.

Many enhancements of streamline visualization exist. Lines can be colored according to velocity magnitude to indicate speed of flow. Other scalar quantities such as temperature or pressure also may be used to color the lines. We also may create constant time dashed lines. Each dash represents a constant time increment. Thus, in areas of high velocity, the length of the dash will be greater relative to regions of lower velocity. These techniques are illustrated in Figure 6.19 for airflow around a blunt fin. This example consists of a wall with half a rounded fin projecting into the fluid flow. (Using arguments of symmetry, only half of the domain was modeled.) Twenty five streamlines are released upstream of the fin. The boundary layer effects near the junction of the fin and wall are clearly evident from the streamlines. In this area, flow recirculation is apparent, as well as the reduced flow speed.

6.4 Tensor Algorithms

As we mentioned earlier, tensor visualization is an active area of research. However there are a few simple techniques that we can use to visualize real 3×3 symmetric tensors. Such tensors are used to describe the state of displacement or stress in a 3D material. The stress and strain tensors for an elastic material are shown in Figure 6.20*.

In these tensors the diagonal coefficients are the so-called normal stresses and strains, and the off-diagonal terms are the shear stresses and strains. Normal stresses and strains act perpendicular to a specified surface, while shear stresses and strains act tangentially to the surface. Normal stress is either compression or tension, depending on the sign of the coefficient.

A 3×3 real symmetric matrix can be characterized by three vectors in 3D called the eigenvectors, and three numbers called the eigenvalues of the matrix. The eigenvectors form a 3D coordinate system whose axes are mutually perpendicular. In some applications, particularly the study of materials, these axes also are referred to as the principle axes of the tensor and are physically significant. For example, if the tensor is a stress tensor, then the principle axes are the directions of normal stress and no shear stress. Associated with each eigenvector is an eigenvalue. The eigenvalues are

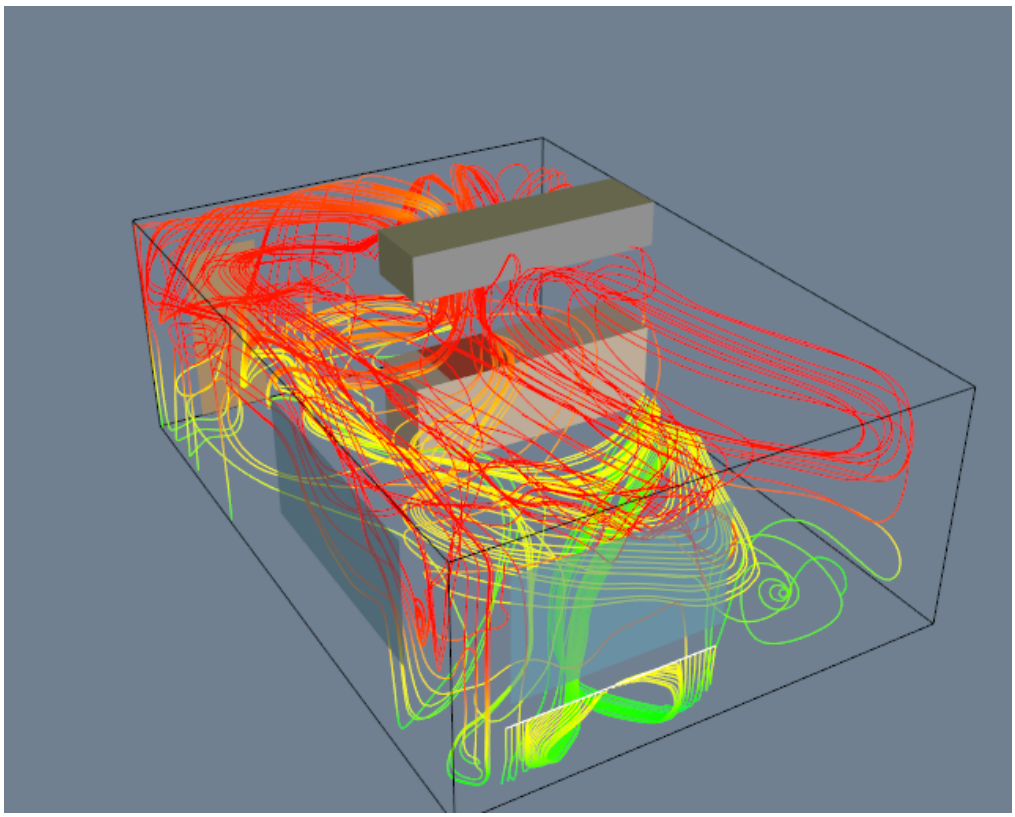


Figure 6.18: Flow velocity computed for a small kitchen. Forty streamlines start along the rake positioned under the window. Some eventually travel over the hot stove and are convected upwards. (Kitchen.cxx or Kitchen.py)

often physically significant as well. In the study of vibration, eigenvalues correspond to the resonant frequencies of a structure, and the eigenvectors are the associated mode shapes.

Mathematically we can represent eigenvalues and eigenvectors as follows. Given a matrix A the eigenvector $\overrightarrow{x}$ and eigenvalue λ must satisfy the relation

$$A \cdot x = \lambda \overrightarrow{x} \tag{6.6}$$

For Equation 6.6 to hold, the matrix determinant must satisfy

$$det|A - \lambda I| = 0 \tag{6.7}$$

Expanding this equation yields a n^{th} degree polynomial in λ whose roots are the eigenvalues. Thus, there are always n eigenvalues, although they may not be distinct. In general, Equation 6.7 is not solved using polynomial root searching because of poor computational performance. (For matrices of order 3 root searching is acceptable because we can solve for the eigenvalues analytically.) Once we determine the eigenvalues, we can substitute each into Equation 6.7 to solve for the associated eigenvectors.

We can express the eigenvectors of the 3×3 system as

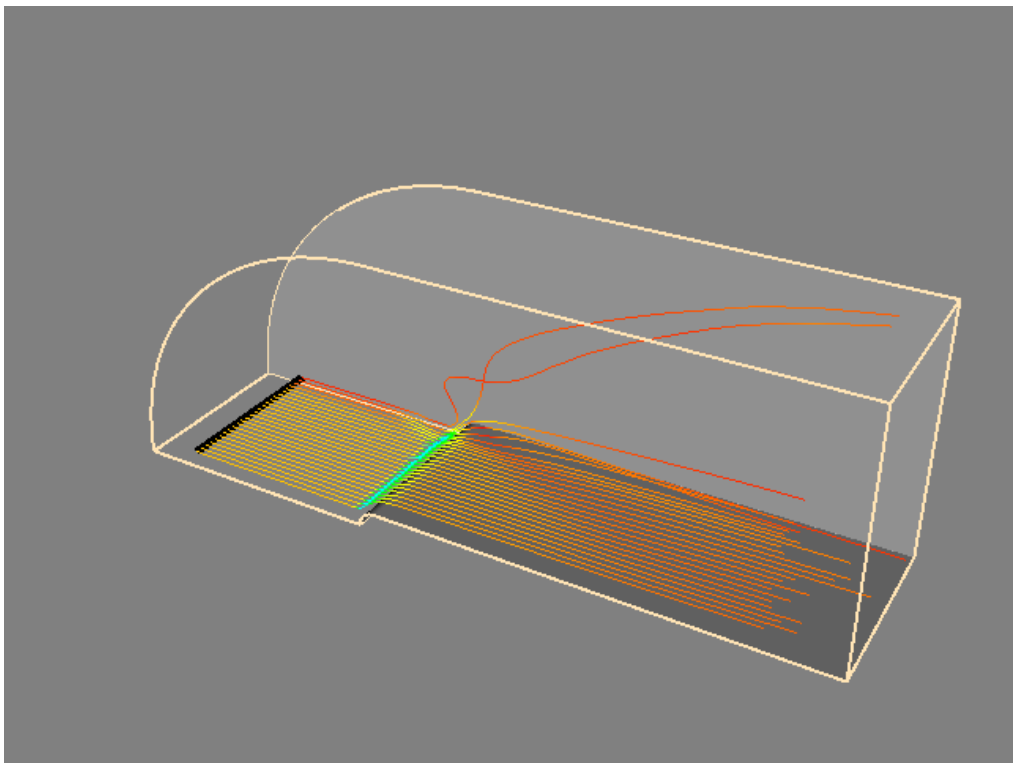


Figure 6.19: airflow around a blunt fin. (BluntStreamlines.cxx or BluntStreamlines.py)

$$\vec{v_i} = \lambda_i \vec{e_i} \; with \; i = 1, 2, 3 \tag{6.8}$$

with $\vec{e_i}$ a unit vector in the direction of the eigenvalue, and λ_i the eigenvalues of the system. If we order eigenvalues such that

$$\lambda_1 \geq \lambda_2 \geq \lambda_3 \tag{6.9}$$

then we refer to the corresponding eigenvectors $\vec{v_1}$, $\vec{v_2}$ and $\vec{v_1}$ as the *major*, *medium* and *minor* eigenvectors.

6.4.1 Tensor Ellipsoids

This leads us to the tensor ellipsoid technique for the visualization of real, symmetric 3×3 matrices. The first step is to extract eigenvalues and eigenvectors as described in the previous section. Since eigenvectors are known to be orthogonal, the eigenvectors form a local coordinate system. These axes can be taken as the *minor*, *medium* and *major* axes of an ellipsoid. Thus, the shape and orientation of the ellipsoid represents the relative size of the eigenvalues and the orientation of the eigenvectors.

To form the ellipsoid we begin by positioning a sphere at the tensor location. The sphere is then rotated around its origin using the eigenvectors, which in the form of Equation 6.8 are direction cosines. The eigenvalues are used to scale the sphere. Using 4×4 transformation matrices and

$$\begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{bmatrix}$$

(a) Stress tensor.

$$\begin{bmatrix} \frac{\partial u}{\partial x} & (\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}) & (\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}) \\ (\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}) & \frac{\partial v}{\partial y} & (\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}) \\ (\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}) & (\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}) & \frac{\partial w}{\partial z} \end{bmatrix}$$

(b) Strain tensor

Figure 6.20: Stress and strain tensors. Normal stresses in the $x - y - z$ coordinate directions indicated as $(\sigma_x, \sigma_y, \sigma_z)$, shear stresses indicated as t_{ij}. Material displacement represented by (u, v, w) components.

referring to Equation 3.6, Equation 3.9 and Equation 3.13, we form the ellipsoid by transforming the sphere centered at the origin using the matrix T

$$T = T_T \cdot T_R \cdot T_S \tag{6.10}$$

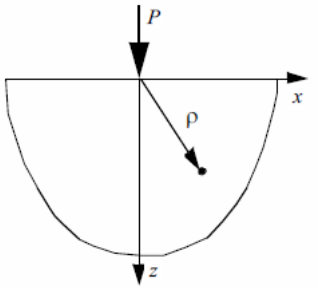
(remember to read right to left). The eigenvectors can be directly plugged in to create the rotation matrix, while the point coordinates $x-y-z$ and eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3$ are inserted into the translation and scaling matrices. A concatenation of these matrices forms the final transformation matrix T.

Figure 6.21a depicts the tensor ellipsoid technique. In Figure 6.21b) we show this technique to visualize material stress near a point load on the surface of a semiinfinite domain. (This is the so-called Boussinesq's problem.) From Saada [22] we have the analytic expression for the stress components in Cartesian coordinates shown in Figure 6.21c. Note that the z-direction is defined as the axis originating at the point of application of the force P. The variable ρ is the distance from the point of load application to a point $x - y - z$. The orientation of the x and y axes are in the plane perpendicular to the z axis. (The rotation in the plane of these axes is unimportant since the solution is symmetric around the z axis.) (The parameter ν is Poisson's ratio which is a property of the material. Poisson's ratio relates the lateral contraction of a material to axial elongation under a uniaxial stress condition. See [22] or [23] for more information.)

In Figure 6.21a we visualize the analytical results of Boussinesq's problem from Saada. The top portion of the figure shows the results by displaying the scaled and oriented principal axes of the stress tensor. (These are called *tensor axes*.) In the bottom portion we use tensor ellipsoids to show the same result. Tensor ellipsoids and tensor axes are a form of *glyph* (see "Glyphs" on page 108) specialized to tensor visualization.

A certain amount of care must be taken to visualize this result since there is a stress singularity at the point of contact of the load. In a real application loads are applied over a small area and not at a single point. Also, plastic behavior prevents stress levels from exceeding a certain point. The results of the visualization, as with any computer process, are only as good as the underlying model.

(a) Tensor ellipsoid.



(b) Point load on semifinite domain.

$$
\begin{aligned}
\sigma_x &= -\frac{P}{2\pi\rho^2}\left(\frac{3zx^2}{\rho^3} - (1-2v)\left(\left(\frac{z}{\rho} - \frac{\rho}{\rho+z} + \frac{x^2(2\rho+z)}{\rho(\rho+z)^2}\right)\right)\right) \\
\sigma_y &= -\frac{P}{2\pi\rho^2}\left(\frac{3zy^2}{\rho^3} - (1-2v)\left(\left(\frac{z}{\rho} - \frac{\rho}{\rho+z} + \frac{y^2(2\rho+z)}{\rho(\rho+z)^2}\right)\right)\right) \\
\sigma_z &= \frac{3Pz^3}{2\pi\rho^5} \\
\tau_{xy} &= \tau_{yx} = -\frac{P}{2\pi\rho^2}\left(\frac{3xyz}{\rho^3} - (1-2v)\left(\frac{xy(2\rho+z)}{\rho(\rho+z)^2}\right)\right) \\
\tau_{xz} &= \tau_{zx} = -\frac{3Pxz^2}{2\pi\rho^5} \\
\tau_{yz} &= \tau_{zy} = -\frac{3Pyz^2}{2\pi\rho^5}
\end{aligned}
$$

(c) Analytic solution

Figure 6.21: Stress and strain tensors. Normal stresses in the $x-y-z$ coordinate directions indicated as $(\sigma_x, \sigma_y, \sigma_z)$, shear stresses indicated as t_{ij}. Material displacement represented by (u, v, w) components.

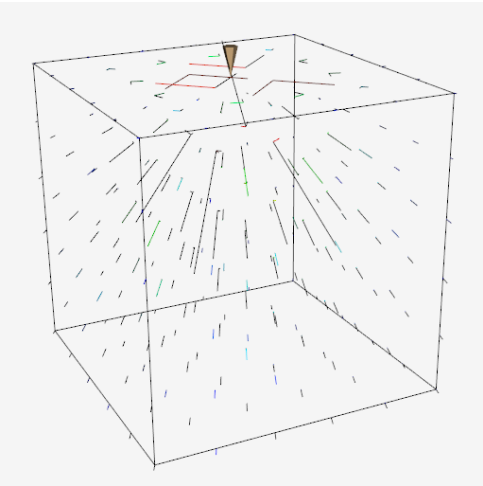
6.5 Modelling Algorithms

Modelling algorithms are the catchall category for our taxonomy of visualization techniques. Modelling algorithms have one thing in common: They create or change dataset geometry or topology.

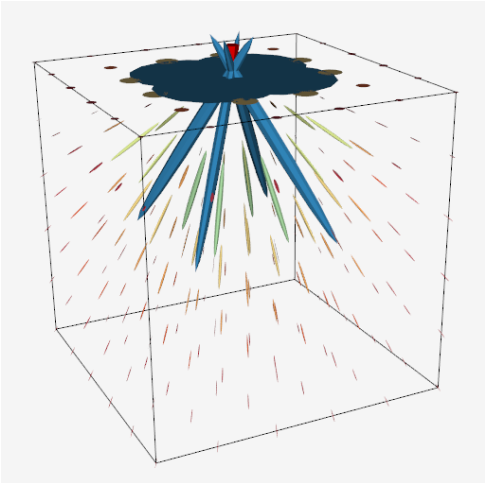
6.5.1 Source Objects

As we have seen in previous examples, source objects begin the visualization pipeline. Source objects are used to create geometry such as spheres, cones, or cubes to support visualization context or are used to read in data files. Source objects also may be used to create dataset attributes. Some examples of source objects and their use are as follows.

Modelling Simple Geometry. Spheres, cones, cubes, and other simple geometric objects can be used alone or in combination to model geometry. Often we visualize real-world ap-



(a) Tensor axes. (TensorAxes.cxx or TensorAxes.py)



(b) Tensor ellipsiods. (TensorEllipsoids.cxx or TensorEllipsoids.py)

Figure 6.22: Tensor visualization techniques; (a) Tensor axes, (b) Tensor ellipsiods.

plications such as air flow in a room and need to show real-world objects such as furniture, windows, or doors. Real-world objects often can be represented using these simple geometric representations. These source objects generate their data procedurally. Alternatively, we may use reader objects to access geometric data defined in data files. These data files may contain more complex geometry such as that produced by a 3D CAD (ComputerAided Design) system.

Supporting Geometry. During the visualization process we may use source objects to create supporting geometry. This may be as simple as three lines to represent a coordinate axis or as complex as tubes wrapped around line segments to thicken and enhance their appearance. Another common use is as supplemental input to objects such as streamlines or probe filters. These filters take a second input that defines a set of points. For streamlines, the points determine the initial positions for generating the streamlines. The probe filter uses the points as the position to compute attribute values such as scalars, vectors, or tensors.

Data Attribute Creation. Source objects can be used as procedures to create data attributes. For example, we can procedurally create textures and texture coordinates. Another use is to create scalar values over a uniform grid. If the scalar values are generated from a mathematical function, then we can use the visualization techniques described here to visualize the function. In fact, this leads us to a very important class of source objects: implicit functions.

6.5.2 Implicit Functions

Implicit functions are functions of the form

$$F(x, y, z) = c \tag{6.11}$$

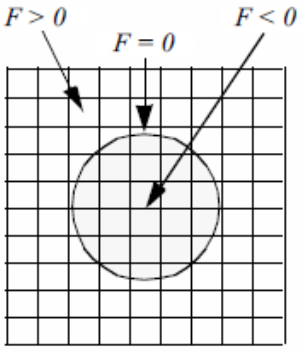
where c is an arbitrary constant. Implicit functions have three important properties.

- *Simple geometric description.* Implicit functions are convenient tools to describe common geometric shapes. This includes planes, spheres, cylinders, cones, ellipsoids, and quadrics.
- *Region separation.* Implicit functions separate 3D Euclidean space into three distinct regions. These regions are inside, on, and outside the implicit function. These regions are defined as $F(x, y, z) < 0$, $F(x, y, z) = 0$ and $F(x, y, z) > 0$, respectively.
- *Scalar generation.* Implicit functions convert a position in space into a scalar value. That is, given an implicit function we can sample it at a point (x_i, y_i, z_i) to generate a scalar value c_i.

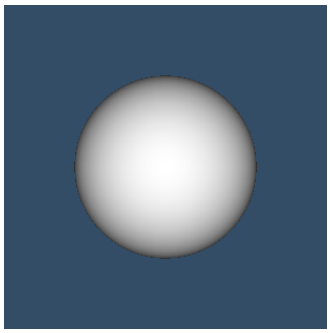
  An example of an implicit function is the equation for a sphere of radius R.

$$F(x, y, z) = x^2 + y^2 + z^2 - R^2 \tag{6.12}$$

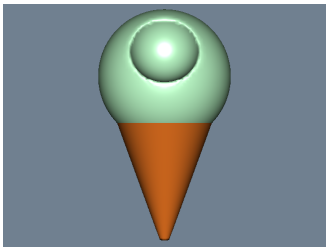
This simple relationship defines the three regions (on $F(x, y, z = 0)$ on the sphere), $F(x, y, z) < 0$ (inside the sphere), and $F(x, y, z) > 0$ (outside the sphere). Any point maybe be classified as inside, on, or outside the sphere simply by evaluating Equation 6.12.



(a) 2D depiction of sphere sampling.



(b) Isosurface of a sphere. (ImplicitSphere.cxx or ImplicitSphere.py)



(c) Boolean combination of two spheres. (IceCream.cxx or IceCream.py)

Figure 6.23: Sampling functions: (a) 2D depiction of sphere sampling] (b) Isosurface of sampled sphere; (c) Boolean combination of two spheres, a cone, and two planes. (One sphere intersects the other, the planes clip the cone).

Implicit functions have a variety of uses. This includes geometric modelling, selecting data, and visualizing complex mathematical descriptions.

6.5.3 Modelling Objects.

Implicit functions can be used alone or in combination to model geometric objects. For example, to model a surface described by an implicit function, we sample F on a dataset and generate an isosurface at a contour value c_i. The result is a polygonal representation of the function. Figure 6.23b shows an isosurface for a sphere of radius=1 sampled on a volume. Note that we can choose nonzero contour values to generate a family of offset surfaces. This is useful for creating blending functions and other special effects.

Implicit functions can be combined to create complex objects using the boolean operators union, intersection, and difference. The union operation between $F \cup G$ between two functions $F(x, y, z)$ and $G(x, y, z)$ is the minimum value

$$F \cup G = \{\max\left(F\left(\overrightarrow{x}\right), G\left(\overrightarrow{x}\right)\right) \mid \overrightarrow{x} \in \mathbb{R}^n\} \tag{6.13}$$

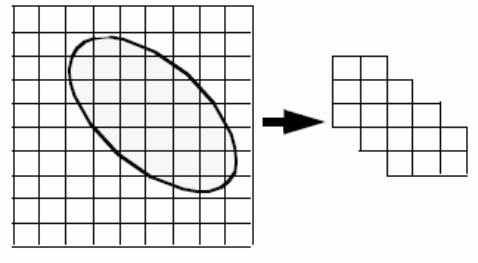
The intersection between two implicit functions is given by

$$F \cap G = \{\min\left(F\left(\overrightarrow{x}\right), G\left(\overrightarrow{x}\right)\right) \mid \overrightarrow{x} \in \mathbb{R}^n\} \tag{6.14}$$

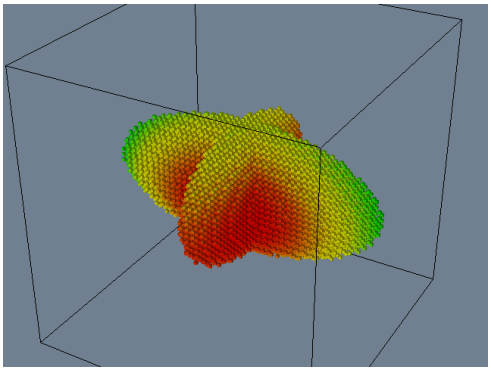
The difference of two implicit functions is given by

$$F - G = \{\min\left(F\left(\overrightarrow{x}\right), -G\left(\overrightarrow{x}\right)\right) \mid \overrightarrow{x} \in \mathbb{R}^n\} \tag{6.15}$$

Figure 6.23c shows a combination of simple implicit functions to create an ice-cream cone. The cone is created by clipping the (infinite) cone function with two planes. The ice cream is constructed by performing a difference operation on a larger sphere with a smaller offset sphere to create the "bite." The resulting surface was extracted using surface contouring with isosurface value 0.0.



(a) Selecting data with implicit function



(b) Isosurface of a sphere. (ExtractData.cxx or ExtractData.py)

Figure 6.24: Implicit functions used to select data: (a) 2D cells lying in ellipse are selected; (b) Two ellipsoids combined using the union operation used to select voxels from a volume. Voxels shrunk 50 percent.

6.5.4 Selecting Data

We can take advantage of the properties of implicit functions to select and cut data. In particular we will use the region separation property to select data. (We defer the discussion on cutting to "Cutting" on page 109.)

Selecting or extracting data with an implicit function means choosing cells and points (and associated attribute data) that lie within a particular region of the function. To determine whether a point *xyz* lies within a region, we simply evaluate the point and examine the sign of the result. A cell lies in a region if all its points lie in the region.

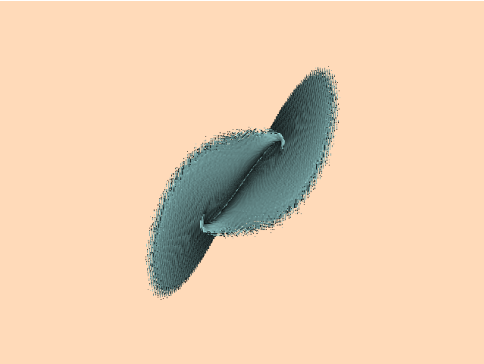
Figure 6.24a shows a 2D implicit function, here an ellipse, used to select the data (i.e., points, cells, and data attributes) contained within it. Boolean combinations also can be used to create complex selection regions as illustrated in Figure 6.24b. Here, two ellipses are used in combination

to select voxels within a volume dataset. Note that extracting data often changes the structure of the dataset. In Figure 6.24 the input type is a image data dataset, while the output type is an unstructured grid dataset.

6.5.5 Visualizing Mathematical Descriptions

Some functions, often discrete or probabilistic in nature, cannot be cast into the form of Equation 6.11. However, by applying some creative thinking we can often generate scalar values that can be visualized. An interesting example of this is the so-called *strange attractor*.

Strange attractors arise in the study of nonlinear dynamics and chaotic systems. In these systems, the usual types of dynamic motion — equilibrium, periodic motion, or quasiperiodic motion — are not present. Instead, the system exhibits chaotic motion. The resulting behavior of the system can change radically as a result of small perturbations in its initial conditions.



(a) (b)

Figure 6.25: Visualizing a Lorenz strange attractor by integrating the Lorenz equations in a volume. The number of visits in each voxel is recorded as a scalar function. The surface is extracted via marching cubes using a visit value of 50. The number of integration steps is 10 million, in a volume of dimensions 200^3. The surface roughness is caused by the discrete nature of the evaluation function.(Lorenz.cxx or Lorenz.py)

A classical strange attractor was developed by Lorenz in 1963 [17]. Lorenz developed a simple model for thermally induced fluid convection in the atmosphere. Convection causes rings of rotating fluid and can be developed from the general Navier Stokes partial differential equations for fluid flow. The Lorenz equations can be expressed in nondimensional form as

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - z) \\ \frac{dy}{dt} &= \rho x - y - xz \\ \frac{dz}{dt} &= xy - \beta z \end{aligned} \tag{6.16}$$

where x is proportional to the fluid velocity in the fluid ring, y and zz measure the fluid temperature in the plane of the ring, the parameters σ and ρ are related to the Prandtl number and Raleigh number, respectively, and β is a geometric factor.

Certainly these equations are not in the implicit form of Equation 6.11, so how do we visualize them? Our solution is to treat the variables x, y, and z as the coordinates of a three-dimensional space, and integrate Equation 6.16 to generate the system "trajectory", that is, the state of the system through time. The integration is carried out within a volume and scalars are created by counting the number of times each voxel is visited. By integrating long enough, we can create a volume representing the "surface" of the strange attractor, Figure 6.25. The surface of the strange attractor is extracted by using marching cubes and a scalar value specifying the number of visits in a voxel.

6.5.6 Implicit Modelling

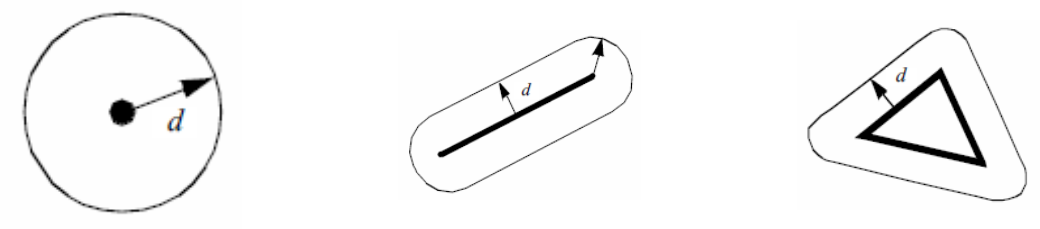


Figure 6.26: Distance functions to a point, line and triangle.

In the previous section we saw how implicit functions, or boolean combinations of implicit functions, could be used to model geometric objects. The basic approach is to evaluate these functions on a regular array of points, or volume, and then to generate scalar values at each point in the volume. Then either volume rendering (see "Volume Rendering" on page 132), or isosurface generation in combination with surface rendering, is used to display the model.

(a) Original (b) Union (c) Intersection (d) Difference

Figure 6.27: Distance functions to a point, line and triangle.

An extension of this approach, called implicit modeling, is similar to modeling with implicit functions. The difference lies in the fact that scalars are generated using a distance function instead of the usual implicit function. The distance function is computed as a Euclidean distance to a set of generating primitives such as points, lines, or polygons. For example, Figure 6.26 shows the distance functions to a point, line, and triangle. Because distance functions are wellbehaved monotonic functions, we can define a series of offset surfaces by specifying different isosurface values, where the value is the distance to the generating primitive. The isosurfaces form approximations to the true offset surfaces, but using high volume resolution we can achieve satisfactory results.

Used alone the generating primitives are limited in their ability to model complex geometry. By using boolean combinations of the primitives, however, complex geometry can be easily modeled. The boolean operations union, intersection, and difference (Equation 6.13, Equation 6.14, and Equation 6.15, respectively) are illustrated in Figure 6.27. Figure 6.28 shows the application

of implicit modeling to "thicken" the line segments in the text symbol "HELLO". The isosurface is generated on a volume $110 \times 40 \times 20$ at a distance offset of 0.25 units. The generating primitives were combined using the boolean union operator. Although Euclidean distance is always a nonnegative value, it is possible to use a signed distance function for objects that have an outside and an inside. A negative distance is the negated distance of a point inside the object to the surface of the object. Using a signed distance function allows us to create offset surfaces that are contained within the actual surface.

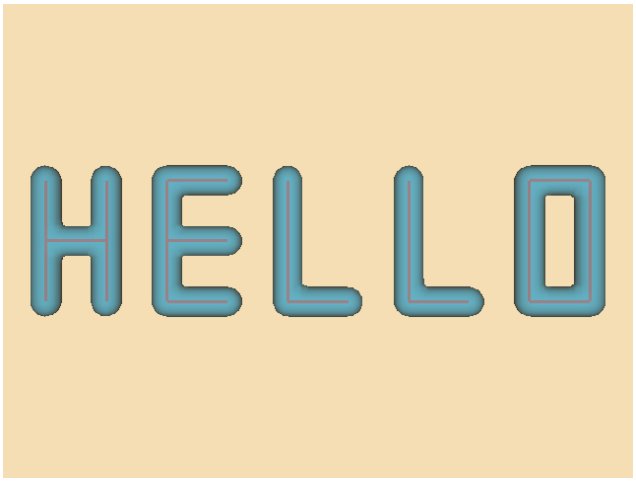


Figure 6.28: Implicit modelling used to thicken a stroked font. Original lines can be seen within the translucent implicit surface.(Hello.cxx or Hello.py)

Another interesting feature of implicit modeling is that when isosurfaces are generated, more than one connected surface can result. These situations occur when the generating primitives form concave features. Figure 6.29 illustrates this situation. If desired, multiple surfaces can be separated by using the connectivity algorithm described in “Connectivity” on page 135.

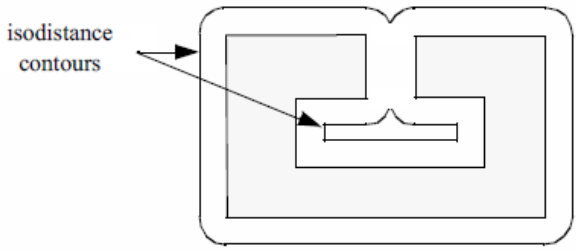


Figure 6.29: Concave features can result in multiple contour lines/surfaces.

6.5.7 Glyphs

Glyphs, sometimes referred to as icons, are a versatile technique to visualize data of every type. A glyph is an “object” that is affected by its input data. This object may be geometry, a dataset, or a graphical image. The glyph may orient, scale, translate, deform, or somehow alter the appearance of the object in response to data. We have already seen a simple form of glyph: hedgehogs are lines that are oriented, translated and scaled according to the position and vector value of a point. A variation of this is to use oriented cones or arrows. (See “Hedgehogs and Oriented Glyphs” on page 92 for more information.)

More elaborate glyphs are possible. In one creative visualization technique Chernoff [6] tied data values to an iconic representation of the human face. Eyebrows, nose, mouth, and other features were modified according to financial data values. This interesting technique built on the human capability to recognize facial expression. By tying appropriate data values to facial characteristics, rapid identification of important data points is possible.

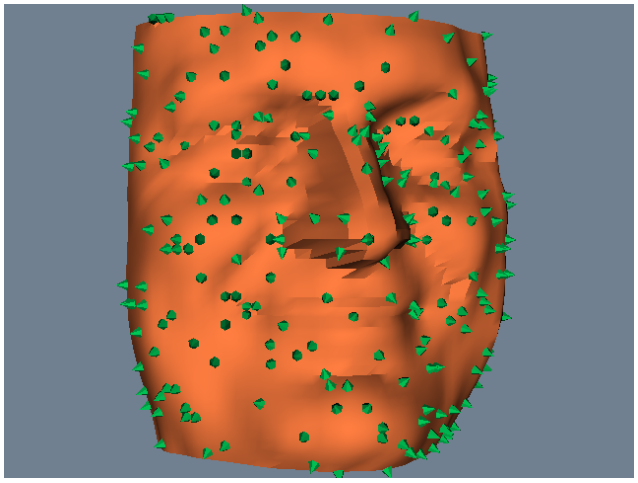


Figure 6.30: Glyphs indicate surface normals on model of human face. Glyph positions are randomly selected. (SpikeFran.cxx or SpikeFran.py)

In a sense, glyphs represent the fundamental result of the visualization process. Moreover, all the visualization techniques we present can be treated as concrete representations of an abstract glyph class. For example, while hedgehogs are an obvious manifestation of a vector glyph, isosurfaces can be considered a topologically twodimensional glyph for scalar data. Delmarcelle and Hesselink [8] have developed a unified framework for flow visualization based on types of glyphs. They classify glyphs according to one of three categories.

- *Elementary icons** represent their data across the extent of their spatial domain. For example, an oriented arrow can be used to represent surface normal.
- *Local icons* represent elementary information plus a local distribution of the values around the spatial domain. A surface normal vector colored by local curvature is one example of a local icon, since local data beyond the elementary information is encoded.
- *Global icons* show the structure of the complete dataset. An isosurface is an example of a global icon.

This classification scheme can be extended to other visualization techniques such as vector and tensor data, or even to nonvisual forms such as sound or tactile feedback. We have found this classification scheme to be helpful when designing visualizations or creating visualization techniques. Often it gives insight into ways of representing data that can be overlooked.

Figure 6.30 is an example of glyphing. Small 3D cones are oriented on a surface to indicate the direction of the surface normal. A similar approach could be used to show other surface properties such as curvature or anatomical keypoints.

6.5.8 Cutting

Often we want to cut through a dataset with a surface and then display the interpolated data values on the surface. We refer to this technique as *data cutting* or simply *cutting*. The data cutting

operation requires two pieces of information: a definition for the surface and a dataset to cut. We will assume that the cutting surface is defined by an implicit function. A typical application of cutting is to slice through a dataset with a plane, and color map the scalar data and/or warp the plane according to vector value.

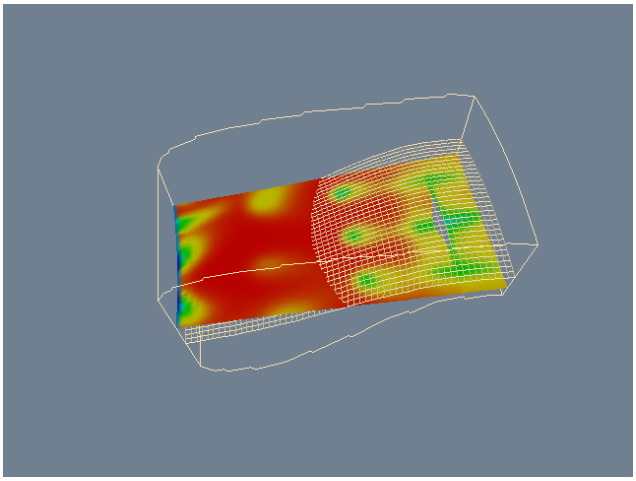
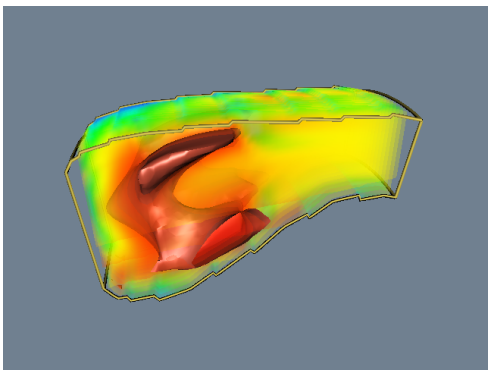


Figure 6.31: Cut through structured grid with plane. The cut plane is shown solid shaded. A computational plane of constant value is shown in wireframe for comparison. The colors correspond to flow density. Cutting surfaces are not necessarily planes: implicit functions such as spheres, cylinders, and quadrics can also be used. (CutStructuredGrid.cxx or CutStructuredGrid.py)

A property of implicit functions is to convert a position into a scalar value (see "Implicit Functions" on page 103). We can use this property in combination with a contouring algorithm (e.g., marching cubes) to generate cut surfaces. The basic idea is to generate scalars for each point of each cell of a dataset (using the implicit cut function), and then contour the surface value $F(x, y, z) = 0$.

The cutting algorithm proceeds as follows. For each cell, function values are generated by evaluating $F(x, y, z)$ for each cell point. If all the points evaluate positive or negative, then the surface does not cut the cell. However, if the points evaluate positive and negative, then the surface passes through the cell. We can use the cell contouring operation to generate the isosurface $F(x, y, z) = 0$. Data attribute values can then be computed by interpolating along cut edges.

(a)



(b)

Figure 6.32: 100 cut planes with opacity of 0.05. Rendered back–to–front to simulate volume rendering.(PseudoVolumeRendering.cxx or PseudoVolumeRendering.py)

Figure 6.31 illustrates a plane cut through a structured grid dataset. The plane passes through the center of the dataset with normal $(-0.287, 0, 0.9579)$. For comparison purposes a portion of the

grid geometry is also shown. The grid geometry is the grid surface $k = 9$ (shown in wireframe). A benefit of cut surfaces is that we can view data on (nearly) arbitrary surfaces. Thus, the structure of the dataset does not constrain how we view the data.

We can easily make multiple planar cuts through a structured grid dataset by specifying multiple isovalues for the cutting algorithm. Figure 6.32 shows 100 cut planes generated perpendicular to the camera's view plane normal. Rendering the planes from back to front with an opacity of 0.05 produces a simulation of volume rendering (see "Volume Rendering" on page 132).

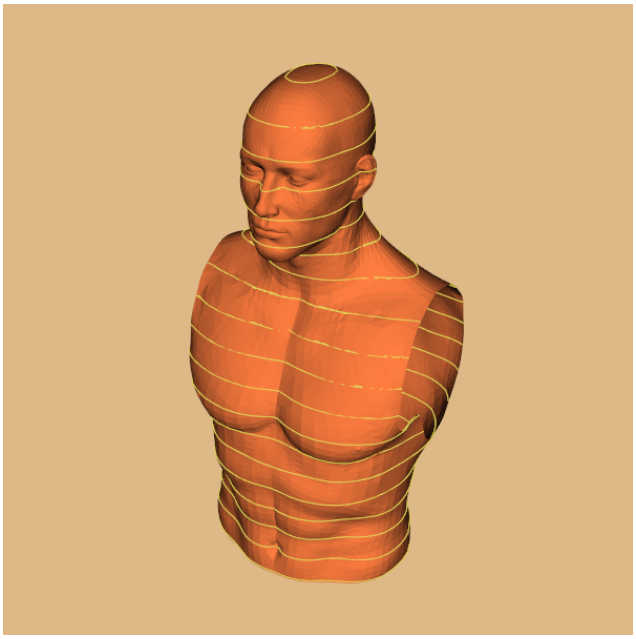


Figure 6.33: Cutting a surface model of the skin with a series of planes produces contour lines. Lines are wrapped with tubes for clarity. (CutWithScalars.cxx or CutWithScalars.py)

This example illustrates that cutting the volumetric data in a structured grid dataset produced polygonal cells. Similarly, cutting polygonal data produces lines. Using a single plane equation, we can extract "contour lines" from a surface model defined with polygons. Figure 6.33 shows contours extracted from a surface model of the skin. At each vertex in the surface model we evaluate the equation of the plane and $F(x.y.z) = c$ and store the valuec of the function as a scalar value. Cutting the data with 46 isovalues from 1.5 to 136.5 produces contour lines that are 3 units apart.

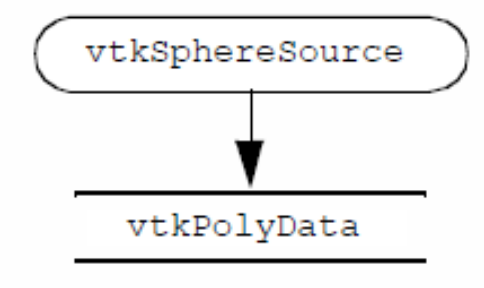
6.6 Putting It All Together

Algorithms are implemented in the *Visualization Toolkit* as process objects. These objects may be either sources, filters, or mappers (see "The Visualization Pipeline" on page 75). In this section we will describe how these objects are implemented.

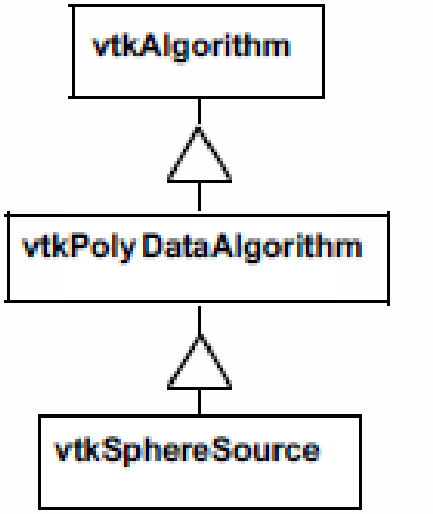
6.6.1 Process Object Design

Source Design

Source objects have no visualization data for input and one or more outputs, Figure 6.34. To create a source object, inheritance is used to specify the type of dataset that the process object creates for output. vtkSphereSource. This class inherits from vtkPolyDataAlgorithm, indicating that it creates polygonal data on output.



(a) Functional model



(b) Object model

Figure 6.34: Source object design. Example shown is a source object that creates a polygonal representation of a sphere.

Filter Design

Filter objects have one or more inputs and one or more outputs as shown in Figure 6.34. (You may also refer to "Pipeline Design and Implementation" on page 75.) To create Figure 6.35 a filter object, inheritance is used to specify the type of input and output data objects. illustrates this for the concrete source object vtkContourFilter (which implements marching cubes and other contouring techniques). It is worth examining this object diagram in detail since it is the basis for the architecture of the visualization pipeline.

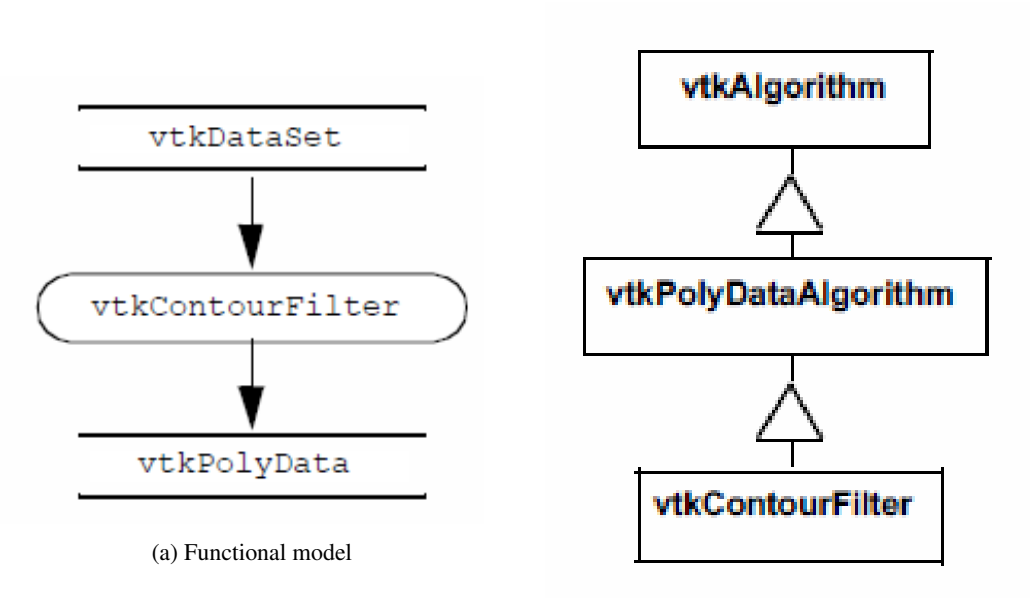
The superclasses of vtkContourFilter are vtkAlgorithm and vtkPolyDataAlgorithm. The class vtkPolyDataAlgorithm specifies the type of data vtkContourFilter produces on output (i.e., a vtkPolyData). Because this filter should take any subclass of vtkDataSet as input, it must override its superclasses implementation of the FillInputPortInformation() method to specify this. Note that inheritance from vtkPolyDataAlgorithm is optional — this functionality could be implemented directly in vtkContourFilter. This optional superclass is simply a convenience object to make class derivation a little easier.

What is left for vtkContourFilter to implement is its RequestData() method (as well as constructor, print method, and any other methods special to this class). Thus the primary difference between classes with equivalent inheritance hierarchies is the implementation of the RequestData() method.

Subclasses of vtkAlgorithm enforce filter input and output type by use of the FillInputPortInformation() and FillOutputPortInformation() methods. By default, its subclass vtkDataSetAlgorithm accepts input type vtkDataSet (or subclasses) and produces a vtkDataSet on output. (The type of the output is determined by the type of the input.) Since vtkDataSet is a base class for all data types, subclasses of vtkDataSetAlgorithm will accept any type as input. Specialized filters are derived from other classes. For example, filters that accept polygonal data might be derived from vtkPolyDataAlgorithm, and filters that accept unstructured grid datasets might be derived from

vtkUnstructuredGridAlgorithm.

We encourage you to examine the source code carefully for a few filter and source objects. The architecture is simple enough that you can grasp it quickly.



(a) Functional model

(b) Object model

Figure 6.35: Filter object design. The example shown is for an object that receives a general dataset as input and creates polygonal data on output.

Mapper Design

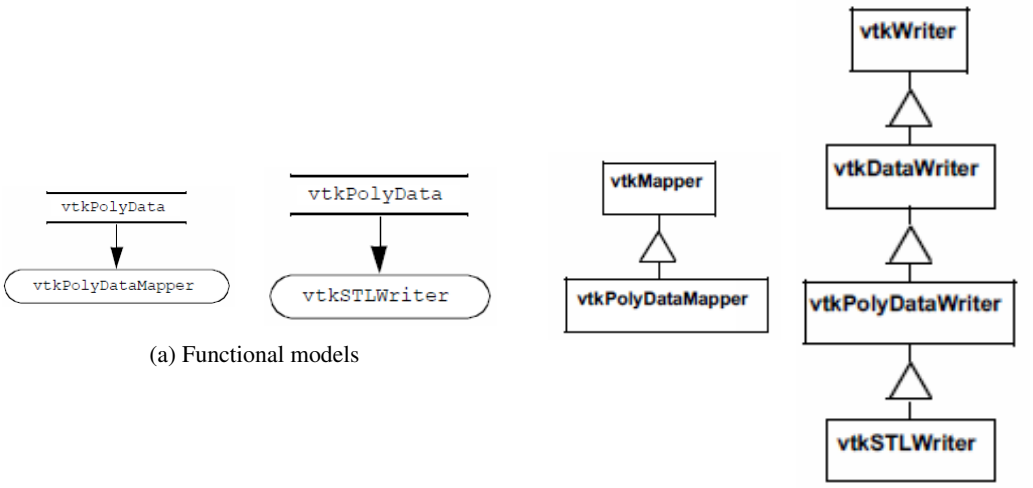
Mapper objects have one or more inputs and no visualization data output, Figure 6.36. Two different types of mappers are available in the *Visualization Toolkit*: graphics mappers and writers. Graphics mappers interface geometric structure and data attributes to the graphics library; writers write datasets to disk or other I/O devices.

Since mappers take datasets as input, type enforcement is required. Each mapper implements this functionality directly. For example, both classes vtkPolyDataMapper and vtkSTLWriter implement a SetInput() method to enforce the input to be of type vtkPolyData. Other mappers and writers enforce input type as appropriate.

Although writers and mappers do not create visualization data, they both have methods similar to the RequestData() method of the sources and filters. Each subclass of vtkMapper must implement the Render() method. This method is exchanged by the graphics system actors and its associated mappers during the rendering process. The effect of the method is to map its input dataset to the appropriate rendering library/system. Subclasses of the class vtkWriter must implement the WriteData() method. This method causes the writer to write its input dataset to disk (or other I/O device).

Color Maps

Color maps are created in the Visualization Toolkit using instances of the class vtkLookupTable. This class allows you to create a lookup table using HSVA (e.g., hue, saturation, value, and



(a) Functional models

(b) Object models

Figure 6.36: Mapper object design. Graphics mapper shown (e.g., vtkPolyDataMapper) maps polygonal data through graphics library primitives. Writer shown (e.g., vtkSTLWriter) writes polygonal data to stereo lithography format.

alpha opacity) specification. Although we discussed the HSV color system in Chapter 3: Computer Graphics Primer, we havent yet defined alpha opacity. We shall do so in Chapter 7: Advanced Computer Graphics, but until then consider the alpha value to be the opacity of an object. Alpha values of one indicate that the object is opaque, while alpha values of zero indicate that the object is transparent.

The procedure for generating lookup table entries is to define pairs of values for HSVA. These pairs define a linear ramp for hue, saturation, value, and opacity. When the Build() method is invoked, these linear ramps are used to generate a table with the number of table entries requested. Alternatively, vtkLookupTable also enables you to load colors directly into the table. Thus, you build custom tables that cannot be simply expressed as linear ramps of HSVA values. To demonstrate this procedure, we specify a starting and ending value for each of the components of HSVA, then we will create a rainbow lookup table from blue to red by using the following C++ code.

Listing 6.1: Create a rainbow lookup table.

```
vtkLookupTable *lut = vtkLookupTable::New();
  lut->SetHueRange(0.6667, 0.0);
  lut->SetSaturationRange(1.0, 1.0);
  lut->SetValueRange(1.0, 1.0);
  lut->SetAlphaRange(1.0, 1.0);
  lut->SetNumberOfColors(256);
  lut->Build();
```

Since the default values for SaturationRange, ValueRange, AlphaRange, and the number of lookup table colors are (1, 1), (1, 1), (1, 1), and 256, respectively, we can simplify this process to the following

Listing 6.2: Create a rainbow lookup table (simplified).

```
vtkLookupTable *lut = vtkLookupTable::New();
  lut->SetHueRange(0.6667, 0.0);
  lut->Build();
```

(The default values for HueRange are (0.0, 0.6667) — a red to blue color table.)

To build a black and white lookup table of 256 entries we use

Listing 6.3: Create a black and white lookup table.

```
vtkLookupTable *lut = vtkLookupTable::New();
  lut->SetHueRange(0.0, 0.0);
  lut->SetSaturationRange(0.0, 0.0);
  lut->SetValueRange(0.0, 1.0)
```

In some cases you may want to specify colors directly. You can do this by specifying the number of colors, building the table, and then inserting new colors. When you insert colors, the RGBA color description system is used. For example, to create a lookup table of the three colors red, green, and blue, use the following C++ code.

Listing 6.4: Directly specifying colors.

```
vtkLookupTable *lut = vtkLookupTable::New();
  lut->SetNumberOfColors(3);
  lut->Build();
  lut->SetTableValue(0, 1.0, 0.0, 0.0, 1.0);
  lut->SetTableValue(0, 0.0, 1.0, 0.0, 1.0);
  lut->SetTableValue(0, 0.0, 0.0, 1.0, 1.0);
```

Lookup tables in the *Visualization Toolkit* are associated with the graphics mappers. Mappers will automatically create a red to blue lookup table if no table is specified, but if you want to create your own, use the mapper->SetLookupTable(lut) operation where mapper is an instance of vtkMapper or its subclasses.

A few final notes on using lookup tables.

- Mappers use their lookup table to map scalar values to colors. If no scalars are present, the mappers and their lookup tables do not control the color of the object. Instead the vtkProperty object associated with the vtkActor class does. Use vtkProperty's method `actor->GetProperty()->SetColor(r,g,b)` where r, g, and b are floating point values specifying color.

- If you want to prevent scalars from coloring your object, use vtkMapper's method `mapper->ScalarVisibilityOff()` to turn off color mapping. Then the actor's color will control the color of the object.

- The scalar range (i.e., the range into which the colors are mapped) is specified with the mapper. Use the method `mapper->SetScalarRange(min, max)`.

You can also derive your own lookup table types. Look at vtkLogLookupTable for an example. This particular lookup table inherits from vtkLookupTable. It performs logarithmic mapping of scalar value to table entry, a useful capability when scalar values span many orders of magnitude.

Implicit Functions

As we have seen, implicit functions can be used for visualizing functions, creating geometry, and cutting or selecting datasets. VTK includes several implicit functions including a single plane (vtkPlane), multiple convex planes (vtkPlanes), spheres (vtkSphere), cones (vtkCone), cylinders (vtkCylinder), and the general quadric (vtkQuadric). The class vtkImplicitBoolean allows you to create boolean combinations of these implicit function primitives. Other implicit functions can be added to VTK by deriving from the abstract base class vtkImplicitFunction.

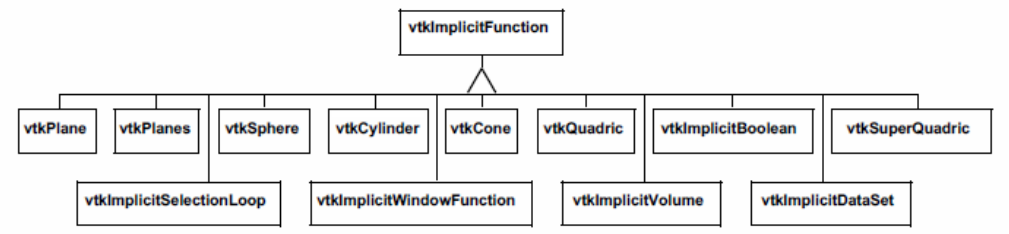


Figure 6.37: Inheritance hierarchy of vtkImplicitFunction and subclasses.

The existing inheritance hierarchy for implicit functions is shown in Figure 6.37. Subclasses of vtkImplicitFunction must implement the two methods Evaluate() and Gradient(x). The method Evaluate() returns the value of the function at point (x, y, z), while the method Gradient() returns the gradient vector to the function at point (x, y, z).

Contouring

Scalar contouring is implemented in the Visualization Toolkit with vtkContourFilter. This filter object accepts as input any dataset type. Thus, vtkContourFilter treats every cell type and each cell type must provide a method for contouring itself.

Contouring in VTK is implemented using variations of the marching cubes algorithm presented earlier. That is, a contour case table is associated with each cell type, so each cell will generate contouring primitives as appropriate. For example, the tetrahedron cell type implements "marching tetrahedron" and creates triangle primitives, while the triangle cell type implements "marching triangles" and generates lines segments.

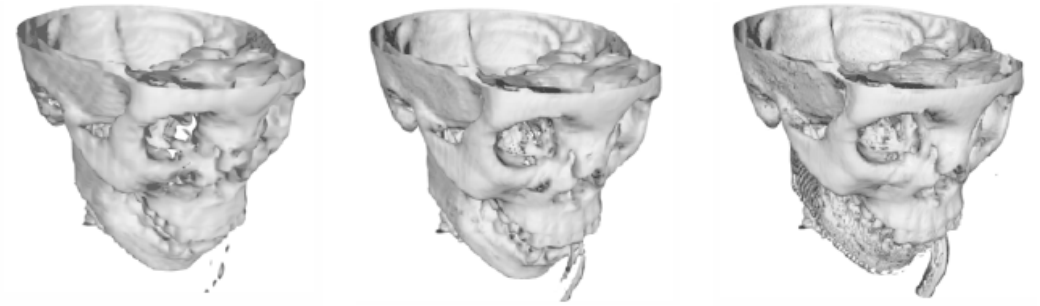
The implication of this arrangement is that vtkContourFilter will generate point, line, and surface contouring primitives depending on the combination of input cell types. Thus vtkContourFilter is completely general. We have created another contour filter, vtkMarchingCubes, that is specific to the dataset type image data (in particular, 3D volumes). These two filters allow us to compare (at least for this one algorithm) the cost of generality.

Recall from "Generality Versus Efficiency" on page 82 the issues regarding the trade-offs between general and specific algorithms. Figure 6.38 shows a comparison of CPU times for a volume dataset at $64 \times 64 \times 93$, $128 \times 128 \times 93$ and $256 \times 256 \times 93$. The volume is a CT dataset of a human head. Three cases were run. In the first case the vtkMarchingCubes object was used. The output of this filter is triangles plus point normals. In the second case, vtkContourFilter was run. The output of this filter is just triangles. In the last case, vtkContourFilter was combined with vtkPolyDataNormals (to generate point normals). The output of the combined filters is also triangles plus point normals.

The execution times are normalized to the smallest dataset using the vtkMarchingCubes object. The results are clear: The specific object outperforms the general object by a factor of 1.4 to 7, depending on data size and whether normals are computed. The larger differences occur on

the smaller datasets. This is because the ratio of voxel cells containing the isosurface to the total number of voxels is larger for smaller datasets. (Generally the total number of voxels increases as the resolution cubed, while the voxels containing the isosurface increase as the resolution squared.) As a result, more voxels are processed in the smaller datasets relative to the total number of voxels than in the larger datasets. When the datasets become larger, more voxels are empty and are not processed.

Although these results do not represent all implementations or the behavior of other algorithms, they do point to the cost of generality. Of course, there is a cost to specialization as well. This cost is typically in programmer time, since the programmer must rewrite code to adapt to new circumstances and data. Like all trade-offs, resolution of this issue requires knowledge of the application.

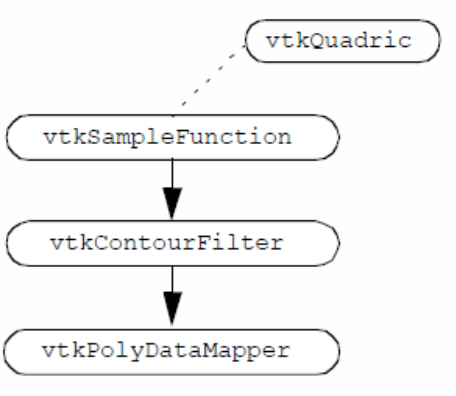


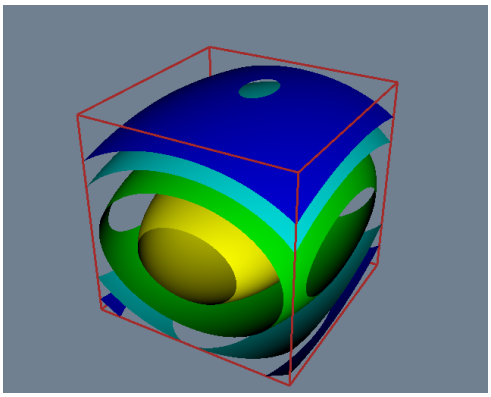
(a) Quarter resolution (b) Half resolution (c) Full resolution

Resolution	Specific (w/ normals)	General (no normals)	Factor Factor	General (w normals)	Factor
$64 \times 64 \times 93$	1.000	2.889	2.889	7.131	7.131
$128 \times 128 \times 93$	5.058	11.810	2.330	23.360	4.600
$256 \times 256 \times 93$	37.169	51.160	1.390	87.230	2.350

Figure 6.38: The cost of generality. Isosurface generation of three volumes of different sizes are compared. The results show normalized execution times for two different implementations of the marching-cubes isosurface algorithm. The specialized filter is *vtkMarchingCubes*. The general algorithms are first *vtkContourFilter* and then in combination with *vtkPolyDataNormals*.

An example use of vtkContourFilter is shown in Figure 6.39. This example is taken from Figure 4.1, which is a visualization of a quadric function. The class vtkSampleFunction samples the implicit quadric function using the vtkQuadric class. Although vtkQuadric does not participate in the pipeline in terms of data flow, it is used to define and evaluate the quadric function. It is possible to generate one or more isolines/isosurfaces simultaneously using vtkContourFilter. As Figure 6.39 shows, we use the GenerateValues() method to specify a scalar range, and the number of contours within this range (including the initial and final scalar values). vtkContourFilter generates duplicate vertices, so we can use vtkCleanPolyData to remove them. To improve the rendered appearance of the isosurface, we use vtkPolyDataNormals to create surface normals. (We describe normal generation in Chapter 9: Advanced Algorithms.)





(ContourQuadric.cxx or ContourQuadric.py)

```
1     // Define implicit function
2     vtkQuadric *quadric = vtkQuadric::New();
3       quadric->SetCoefficients(.5,1,.2,0,.1,0,0,.2,0,0);
4     vtkSampleFunction *sample = vtkSampleFunction::New();
5       sample->SetSampleDimensions(50,50,50);
6       sample->SetImplicitFunction(quadric);
7     vtkContourFilter *contour = vtkContourFilter::New();
8       contour->SetInputConnection(sample->GetOutputPort());
9       contour->GenerateValues(5,0,1.2);
10    vtkPolyDataMapper *contourMapper = vtkPolyDataMapper::New();
11      contourMapper->SetInputConnection( contour->GetOutputPort());
12      contourMapper->SetScalarRange(0,1.2);
13    vtkActor *contourActor = vtkActor::New();
14      contourActor->SetMapper(contourMapper);
15    // Create outline
16    vtkOutlineFilter *outline = vtkOutlineFilter::New();
17      outline->SetInputConnection(sample->GetOutputPort());
18    vtkPolyDataMapper *outlineMapper = vtkPolyDataMapper::New();
19      outlineMapper->SetInputConnection(outline->GetOutputPort());
20    vtkActor *outlineActor = vtkActor::New();
21      outlineActor->SetMapper(outlineMapper);
22      outlineActor->GetProperty()->SetColor(0,0,0);
```

Figure 6.39: Contouring quadric function. Pipeline topology, C++ code, and resulting image are shown.

6.6.2 Cutting

The vtkCutter class performs cutting of all VTK cell types. The SetValue() and GenerateValues() methods permit the user to specify which multiple scalar values to use for the cutting. vtkCutter requires an implicit function that will be evaluated at each point in the dataset. Then each cell is cut using the cell's Contour method. Any point attributes are interpolated to the resulting

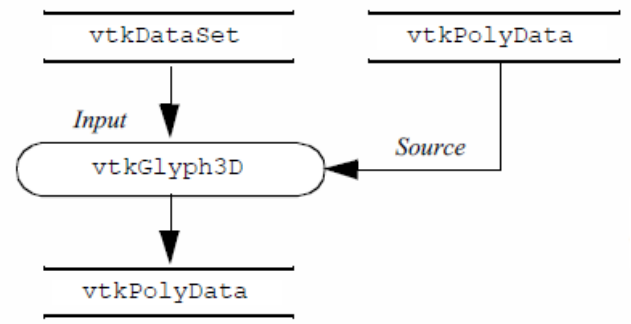


Figure 6.40: Data flow into and out of the *vtkGlyph3D* class.

cut vertices. The sorting order for the generated polygonal data can be controlled with the SortBy method. The default sorting order, SortByValue(), processes cells in the inner loop for each contour value. SortByCell() processes the cutting value in the inner loop and produces polygonal data that is suitable for backtofront rendering (see Figure 6.32). (The sorting order is useful when rendering with opacity as discussed in Chapter 7: Advanced Computer Graphics.) Notice the similarity of this filter to the vtkContourFilter. Both of these objects contour datasets with multiple isovalues. vtkCutter uses an implicit function to calculate scalar values while vtkContourFilter uses the scalar data associated with the dataset's point data.

6.6.3 Glyphs

The vtkGlyph3D class provides a simple, yet powerful glyph capability in the *Visualization Toolkit*. vtkGlyph3D is an example of an object that takes multiple inputs (Figure 6.40). One input, specified with the SetInputConnection() method, defines a set of points and possible attribute data at those points. The second input, specified with the SetSourceConnection() method, defines a geometry to be copied to every point in the input dataset. The source is of type vtkPolyData. Hence, any filter, sequence of filters creating polygonal data, or a polygonal dataset may be used to describe the glyph's geometry.

The behavior of an instance of vtkGlyph3D depends on the nature of the input data and the value of its instance variables. Generally, the input Source geometry will be copied to each point of the Input dataset. The geometry will be aligned along the input vector data and scaled according to the magnitude of the vector or the scalar value. In some cases, the point normal is used rather than the vector. Also, scaling can be turned on or off.

We saw how to use vtkGlyph3D in the example given in Figure 4.2. Cones were used as the glyph and were located at each point on the sphere, oriented along the sphere's surface normal.

6.6.4 Streamlines

Streamlines and particle motion require numerical integration to guide a point through the vector field. Vector visualization algorithms that we will see in later chapters also require numerical integration. As a result, we designed an object hierarchy that isolates the numerical integration process into a single base class. The base class is vtkStreamer and it is responsible for generating a particle path through a vector field of specified length (expressed as elapsed time). Each derived class of vtkStreamer takes advantage of this capability to move through the vector field but implements its own particular representational technique to depict particle motion. Streamlines (vtkStreamLine) draw connected lines while particle motion is shown by combining the output of vtkStreamPoints with the vtkGlyph3D object. Using vtkGlyph3D we can place spheres or oriented

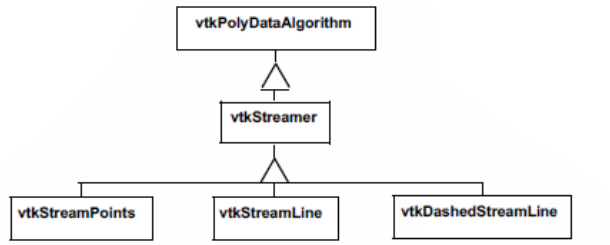


Figure 6.41: Inheritance hierarchy for vtkStreamer and subclasses.

objects such as cones or arrows at points on the particle path created by vtkStreamPoints. The inheritance hierarchy for vtkStreamer and subclasses is shown in Figure 4.2.

The integration method in vtkStreamer is implemented as a virtual function. Thus it can be overloaded as necessary. Possible reasons for overloading include implementing an integration technique of higher or lower accuracy, or creating a technique specialized to a particular dataset type. For example, the search process in a volume is much faster than it is for other dataset types, therefore, highly efficient vector integration techniques can be constructed.

The vector integration technique in VTK will accommodate any cell type. Thus, integration through cells of any topological dimension is possible. If the cells are of topological dimension 2 or less, the integration process constrains particle motion to the surface (2D) or line (1D). The particle may only leave a cell by passing through the cell boundary, and traveling to a neighboring cell, or exiting the dataset.

6.6.5 Abstract Filters

Attribute transformations create or modify data attributes without changing the topology or geometry of a dataset. Hence filters that implement attribute transformation (e.g., vtkElevationFilter) can accept any dataset type as input, and may generate any dataset type on output. Unfortunately, because filters must specialize the particular type of data they output, at first glance it appears that filters that create general dataset types on output are not feasible. This is because the type vtkDataSet is an abstract type and must be specialized to allow instantiation.

Fortunately, there is a solution to this dilemma. The solution is to use the "virtual constructor" NewInstance(). Although C++ does not allow virtual constructors, we can simulate it by creating a special virtual function that constructs a copy of the object that it is invoked on. For example, if this function is applied to a dataset instance of type vtkPolyData, the result will be a copy of that instance (Figure 6.42). (Note that we use reference counting to make copies and avoid duplicating memory.) The virtual constructor function NewInstance() is implemented in a number of VTK classes including datasets and cells.

Using the virtual constructor we can construct filters that output abstract data types like vtkDataSet. We simply apply NewInstance() to the input of the filter. This will then return a pointer to a concrete object that is the output of the filter. The result is a general filter object that can accept any dataset type for input and creates the general vtkDataSet type as output. In VTK, this functionality has been implemented in the abstract class vtkDataSetAlgorithm.

There are other filters that implement variations of this delegation technique. The class vtkPointSetAlgorithm is similar to vtkDataSetAlgorithm. This class takes as input any dataset whose geometry is explicitly defined via an instance of vtkPoints (or subclass), and generates on output an object of the same type (i.e., vtkPointSet). The class vtkMergeFilter combines dataset structure and point attributes from one or more input datasets. For example, you can read multiple files

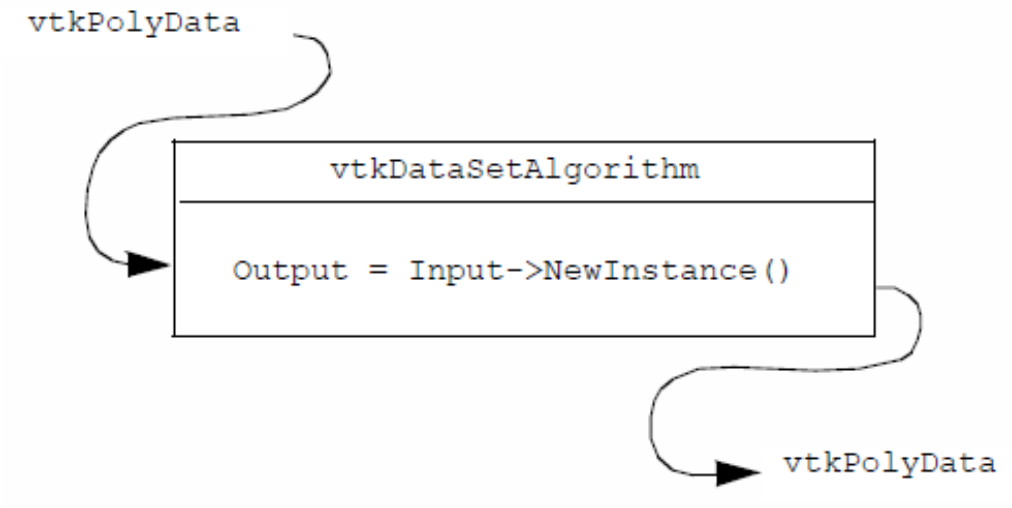


Figure 6.42: Depiction of data flow for abstract filter output. The output object type is the same as the input type.

and combine the geometry/topology from one file with different scalars, vectors, and normals from other files.

One difficulty using abstract filter types is that the output type may not match with the input type of a downstream filter. For example, the output of vtkElevationFilter is specified as vtkDataSet even though the input may be of type vtkPolyData, and we know from the previous discussion that the actual output type will be vtkPolyData. This difficulty is removed by using the filter vtkCastToConcrete, which allows you to runtime cast to the appropriate output type. In this case we would use the GetPolyDataOutput() from vtkCastToConcrete. After checking the validity of the cast, this method returns a dataset cast to vtkPolyData. Of course, this process requires that the input to vtkCastToConcrete be set before the output is requested.

6.6.6 Visualizing Blood Flow

In this example we'll combine a few different techniques to visualize blood flow in the human carotid arteries. Our data contains both vectors that represent the velocity of blood and scalars that are proportional to the magnitude of the velocity (i.e., speed).

We can provide context for the visualization by creating an isosurface of speed. This isosurface shows regions of fastest blood flow, and is similar to, but not the same as, the actual surface of the arteries. However, it provides us with a visual cue to the structure of the arteries. The first vector visualization technique we'll use is to generate vector glyphs (Unfortunately, we cannot just create glyphs at each point because of the number of points (over 167,000 points). To do so would result in a confusing mess, and the interactive speed would be poor. Instead, we'll use two filters to select a subset of the available points. These filters are vtkThresholdPoints and vtkMaskPoints.

vtkThresholdPoints allows us to extract points that satisfy a certain threshold criterion. In our example, we choose points whose speed is greater than a specified value. This eliminates a large number of points, since most points lie outside the arteries and have a small speed value.

The filter vtkMaskPoints allows us to select a subset of the available points. We specify the subset with the OnRatio instance variable. This instance variable indicates that every OnRatio point is to be selected. Thus, if the OnRatio is equal to one, all points will be selected, and if the OnRatio

is equal to ten, every tenth point will be selected. This selection can be either uniform or random. Random point selection is set using the RandomModeOn() and RandomModeOff() methods.

After selecting a subset of the original points, we can use the vtkGlyph3D filter in the usual way. A cone's orientation indicates blood flow direction, and its size and color correspond to the velocity magnitude. Figure 6.43 shows the pipeline, sample code, and a resulting image from this visualization. Note that we've implemented the example using the interpreted language Tcl. See Chapter 11: Visualization on the Web if you want more information about Tcl.

In the next part of this example we'll generate streamtubes of blood velocity. Again we use an isosurface of speed to provide us with context. The starting positions for the streamtubes were determined by experimenting with the data. Because of the way the data was measured and the resolution of the velocity field, many streamers travel outside the artery. This is because the boundary layer of the blood flow is not captured due to limitations in data resolution. Consequently, as the blood flows around curves, there is a component of the velocity field that directs the streamtube outside the artery. As a result it is hard to find starting positions for the streamtubes that yield interesting results. We use the source object vtkPointSource in combination with vtkThresholdPoints to work around this problem. vtkPointSource generates random points centered around a sphere of a specified radius. We need only find an approximate position for the starting points of the streamtubes and then generate a cloud of random seed points. vtkThresholdPoints is used to cull points that may be generated outside the regions of high flow velocity.

Figure 6.44 shows the pipeline, sample Tcl code, and a resulting image from the visualization. Notice that the isosurface is shown in wireframe. This provides context, yet allows us to see the streamtubes within the isosurface.

6.7 Chapter Summary

Visualization algorithms transform data from one form to another. These transformations can change or create new structure and/or attributes of a dataset. Structural transformations change either the topology or geometry of a dataset. Attribute transformations change dataset attributes such as scalars, vectors, normals, or texture coordinates.

Algorithms are classified according to the type of data they operate on. Scalar, vector, and tensor algorithms operate on scalar, vector, and tensor data, respectively. Modelling algorithms operate on dataset geometry or topology, texture coordinates, or normals. Modelling algorithms also may include complex techniques that may represent combinations of different data types.

Algorithms can be designed and implemented for general types of data or specialized for a specific type. General algorithms are typically less efficient than their specialized counterparts. Conversely, general algorithms are more flexible and do not require rewriting as new dataset types are introduced.

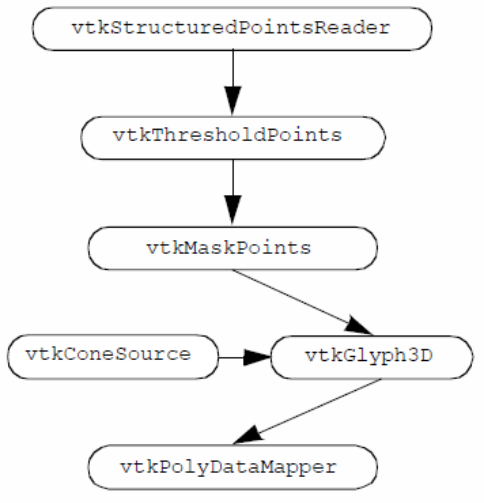
Important scalar algorithms include color mapping and contouring. Color maps are used to map scalar values to color values. Contouring algorithms create isosurfaces or isolines to indicate areas of constant scalar value.

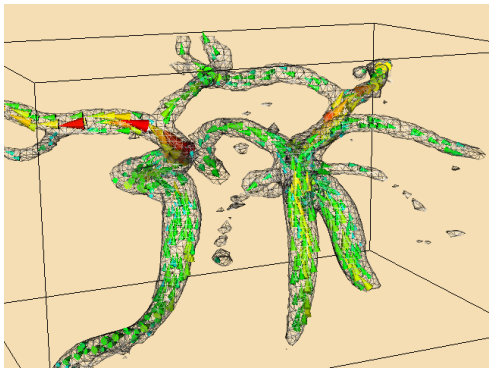
Glyphs such as hedgehogs are useful for visualizing vector data. These techniques are limited by the number of glyphs that can be displayed at one time. Particle traces or streamlines are another important algorithm for vector field visualization. Collections of particle traces can convey something of the structure of a vector field.

Real, symmetric tensors 3×3 can be characterized by their eigenvalues and eigenvectors.

Tensors can be visualized using tensor ellipsoids or oriented axes.

Implicit functions and sampling techniques can be used to make geometry, cut data, and visualize complex mathematical descriptions. Glyphs are objects whose appearance is associated with a particular data value. Glyphs are flexible and can be created to visualize a variety of data.

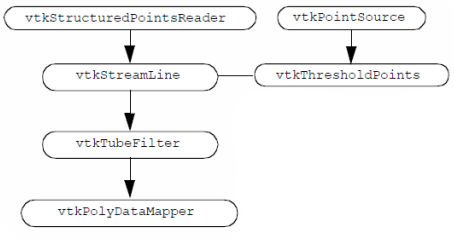


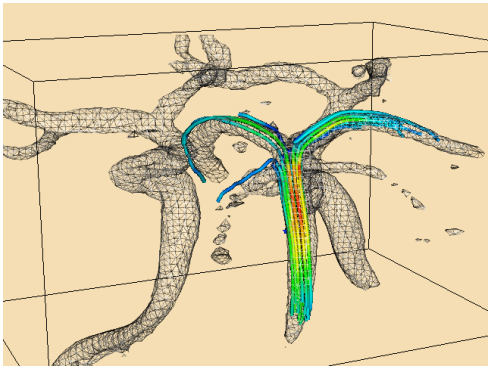


(CarotidFlowGlyphs.cxx or CarotidFlowGlyphs.py)

```
1   vtkStructuredPointsReader reader
2     reader SetFileName "$env(VTK_TEXTBOOK_DATA)/carotid.vtk"
3   vtkThresholdPoints threshold
4     threshold SetInputConnection [reader GetOutputPort]
5     threshold ThresholdByUpper 200
6   vtkMaskPoints mask
7     mask SetInputConnection [Threshold GetOutputPort]
8     mask SetOnRatio 10
9   vtkConeSource cone
10    cone SetResolution 3
11    cone SetHeight 1
12    cone SetRadius 0.25
13  vtkGlyph3D cones
14    cones SetInputConnection [mask GetOutputPort]
15    cones SetSourceConnection [cone GetOutputPort]
16    cones SetScaleFactor 0.5
17    cones SetScaleModeToScaleByVector
18  vtkLookupTable lut
19    lut SetHueRange.667 0.0
20    lut Build
21  vtkPolyDataMapper vecMapper
22    vecMapper SetInputConnection [cones GetOutputPort]
23    vecMapper SetScalarRange 2 10
24    vecMapaper SetLookupTable lut
```

Figure 6.43: Visualizing blood flow in human carotid arteries. Cone glyphs indicate flow direction and magnitude.





(CarotidFlow.cxx or CarotidFlow.py)

```
vtkStructuredPointsReader reader
  reader SetFileName "$env(VTK_TEXTBOOK_DATA)/carotid.vtk"
vtkPointSource source
  source SetNumberOfPoints 25
  source SetCenter 133.1 116.3 5.0
  source SetRadius 2.0
vtkThresholdPoints threshold
  threshold SetInputConnection [reader GetOutputPort]
  threshold ThresholdByUpper 275
vtkStreamTracer streamers
  streamers SetInputConnection [reader GetOutputPort]
  streamers SetSourceConnection [source GetOutputPort]
  streamers SetMaximumPropagationUnitToTimeUnit
  streamers SetMaximumPropagation 100.0
  streamers SetInitialIntegrationStepUnitToCellLengthUnit
  streamers SetInitialIntegrationStep 0.2
  streamers SetTerminalSpeed .1
vtkTubeFilter tubes
  tubes SetInputConnection [streamers GetOutputPort]
  tubes SetRadius 0.3
  tubes SetNumberOfSides 6
  tubes SetVaryRadiusToVaryRadiusOff
vtkPolyDataMapper streamerMapper
  streamerMapper SetInputConnection [tubes GetOutputPort]
  streamerMapper SetScalarRange 2 10
```

Figure 6.44: Visualizing blood flow in the human carotid arteries. Streamtubes of flow vectors.

6.8 Bibliographic Notes

Color mapping is a widely studied topic in imaging, computer graphics, visualization, and human factors. References [9] [26] [21] provide samples of the available literature. You also may

want to learn about the physiological and psychological effects of color on perception. The text by Wyszecki and Stiles [28] serves as an introductory reference.

Contouring is a widely studied technique in visualization because of its importance and popularity. Early techniques were developed for 2D data [27]. Three-dimensional techniques were developed initially as contour connecting methods [12] — that is, given a series of 2D contours on evenly spaced planes, connect the contours to create a closed surface. Since the introduction of marching cubes [16], many other techniques have been implemented. (A few of these include [19] [5] and [10]). A particularly interesting reference is given by Livnat et al. [29]. They show a contouring method with the addition of a preprocessing step that generates isocontours in near optimal time.

Although we barely touched the topic, the study of chaos and chaotic vibrations is a delightfully interesting topic. Besides the original paper by Lorenz [17], the book by Moon [18] is a good place to start.

Two- and three-dimensional vector plots have been used by computer analysts for many years [1]. Streamlines and streamribbons also have been applied to the visualization of complex flows [25]. Good general references on vector visualization techniques are given in [14] and [20].

Tensor visualization techniques are relatively few in number. Most techniques are glyph oriented [13] [15]. We will see a few more techniques in Chapter 9.

Blinn [2], Bloomental [4] [3] and Wyvill [11] have been important contributors to implicit modeling. Implicit modeling is currently popular in computer graphics for modeling "soft" or "blobby" objects. These techniques are simple, powerful, and are becoming widely used for advanced computer graphics modeling.

Bibliography

[1] A.J Fuller Baden and M.L.X dos Santos. “Computer Generated Display of 3D Vector Fields”. In: *Computer Aided Design* 12.2 (1980), pp. 61–66.

[2] J. F. Blinn. “A Generalization of Algebraic Surface Drawing”. In: *ACM Transactions on Graphics* 1.3 (July 1982), pp. 235–256.

[3] J. Bloomentha, ed. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, Inc., San Francisco, CA., 1997.

[4] J. Bloomenthal. “Polygonization of Implicit Surfaces”. In: *Computer Aided Geometric Desig* 5.4 (Nov. 1982), pp. 341–355.

[5] R. Scateni C. Montani and R. Scopigno. “A Modified LookUp Table for Implicit Disambiguation of Marching Cubes”. In: *Visual Computer* 10 (1994), pp. 353–355.

[6] H. Chernoff. “Using Faces to Represent Points in K-Dimensional Space Graphically”. In: *J. American Statistical Association* 68 (1973), pp. 361–368.

[7] S. D. Conte and C. de Boor. *Elementary Numerical Analysis*. McGrawHill Book Company, 1972.

[8] T. Delmarcelle and L. Hesselink. *A Unified Framework for Flow Visualization*. Ed. by R. S. Gallagher. CRC Press, Boca Raton, FL, 1995.

[9] H. J. Durrett, ed. *Color and the Computer*. Academic Press, Boston, MA, 1987.

[10] M. J. Durst. “Additional Reference to Marching Cubes”. In: *Computer Graphics* 22.2 (1988), pp. 72–73.

[11] G. Wyvill, C. McPheeters, B. Wyvill. “Data Structure for Soft Objects”. In: *Visual Computer* 2.4 (1986), pp. 227–234.

[12] H. Fuchs, Z. M. Kedem, and S. P. Uselton. “Optimal Surface Reconstruction from Planar Contours”. In: *Communications of the ACM* 20.10 (1977), pp. 693–702.

[13] R. B. Haber and D. A. McNabb. “Visualization Idioms: A Conceptual Model to Scientific Visualization Systems”. In: *Visualization in Scientific Computing*. Ed. by L. J. Rosenblum G. M. Nielson B. Shriver. IEEE Computer Society Press, 1990, pp. 61–73.

[14] J. Helman and L. Hesselink. “Representation and Display of Vector Field Topology in Fluid Flow Data Sets”. In: *Visualization in Scientific Computing*. Ed. by L. J. Rosenblum G. M. Nielson B. Shriver. IEEE Computer Society Press, 1990, pp. 61–73.

[15] W. C. de Leeuw and J. J. van Wijk. “A Probe for Local Flow Field Visualization”. In: *Proceedings of Visualization ’93*. IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 39–45.

[16] W. E. Lorensen and H. E. Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: *Computer Graphics* 21.3 (July 1987), pp. 163–169.

[17] E. N. Lorenz. “Deterministic NonPeriodic Flow”. In: *Journal of Atmospheric Science* 20 (1963), pp. 130–141.

[18] F. C. Moon. *Chaotic Vibrations*. WileyInterscience, New York, NY, 1987.

[19] G. M. Nielson and B. Hamann. “he Asymptotic Decider: Resolving the Ambiguity in Marching Cube”. In: *Proceedings of Visualization ’91*. IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 83–91.

[20] R. Richter, J. B. Vos, A. Bottaro, and S. Gavrilakis. “Scientific Visualization and Graphics Simulation”. In: ed. by D. Thalmann. John Wiley and Sons, 1990. Chap. Visualization of Flow Simulations, pp. 161–171.

[21] P. Rheingans. “Color, Change, and Control for Quantitative Data Display”. In: *Proceedings of Visualization ’92*. IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 252–259.

[22] A. S. Saada. *Elasticity Theory and Applications*. Pergamon Press, Inc., New York, NY, 1974.

[23] S. P. Timoshenko and J. N. Goodier. *Theory of Elasticity, 3d Edition*. McGrawHill Book Company, New York, NY, 1970.

[24] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1999.

[25] G. Volpe. “Technical Report AIAA-89-0140”. In: *27th Aerospace Sciences Meeting*. 1989.

[26] C. Ware. “Color Sequences for Univariate Maps: Theory, Experiments and Principles”. In: *IEEE Computer Graphics and Applications* 8.5 (1988), pp. 41–49.

[27] D. F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data*. Pergamon Press, 1992.

[28] G. Wyszecki and W. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. John Wiley and Sons, 1982.

[29] C. R. Johnson Y. Livnat H. W. Shen. “A Near Optimal Isosurface Extraction Algorithm for Structured and Unstructured Grids”. In: *IEEE Transactions on Visualization and Computer Graphics* 2.1 (Mar. 1996).

6.9 Exercises

1. Sketch contour cases for marching triangles. How many cases are there?

2. Sketch contour cases for marching tetrahedron. How many cases are there?

3. A common visualization technique is to animate isosurface value. The procedure is to smoothly vary isosurface value over a specified range.

   (a) Create an animation sequence for the quadric example (Figure 4.1a)

   (b) Create an animation sequence for the head sequence (Figure 6.11b)

4. Marching Cubes visits each cell during algorithm execution. Many of these cells do not contain the isosurface. Describe a technique to improve the performance of isosurface extraction by eliminating visits to cells not containing isosurface. (*Hint*: use a preprocessing step to analyze data. Assume that many isosurfaces will be extracted and that the preprocessing step will not count against execution time.)

5. Scanline rasterization proceeds along horizontal spans in graphics hardware (see “Rasterization” on page 44. Interpolation of color occurs along horizontal spans as well.

   (a) Show how the orientation of a polygon affects interpolated color.

(b) Discuss potential problems caused by orientation dependent viewing of visualizations.

6. Write a program to simulate beam vibration. Use the code associated with Figure 6.14a as your starting point.

7. Using the filters vtkStreamLine, vtkMaskPoints, and vtkGlyph3D, create a visualization consisting of oriented glyphs along a streamline.

8. Visualize the following functions.

   (a) Scalar $S(x, y, z) = sin(xy)$ for x, y, between 0 and π.

   (b) The effective stress field (a scalar field) from **Figure621**.

   (c) The vector field described in the combustor data (i.e., combq.bin and combxyz.bin).

9. Tensor ellipsoids are based on an ellipsoidal glyph. Describe two other glyphs that you might use.

10. Write a source object to generate a polygonal representation of a torus.

11. Design a glyph to convey airplane heading, speed, and altitude, and proximity (i.e., distance) to other planes.

12. Morphing is a process to smoothly blend images (2D) or geometry (3D) between two known images or geometry. Using an implicit modeling approach, how would you morph a torus into a cube?

13. Describe a technique to visualize vector information by animating a color map. (emphHint: By choosing a map carefully, you can give the illusion of motion across a surface.)

14. Isoline contours of different values are typically shown together in one image.

   (a) Describe the advantages and disadvantages of displaying isosurfaces simultaneously.

   (b) What two graphics properties might you adjust to improve the display of multiple isosurefaces?

15. Describe a parallel algorithm for marching cubes. Use a parallel architecture of your choice.

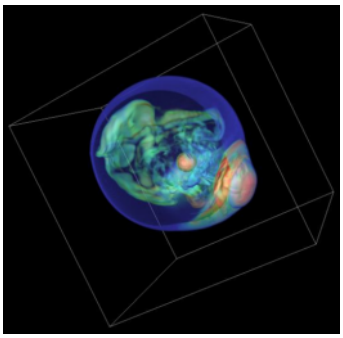
16. Decomposition can greatly increase the speed of an operation.

   (a) Prove that 3D Gaussian smoothing can be decomposed into three 1D operations.

   (b) Give the complexity of the decomposed filter and the same filter implemented as a 3D convolution.

   (c) Under what conditions can constant smoothing be decomposed into 1D operations.

7.0
Advanced Computer Graphics



Volume rendering of a supernova dataset.

Chapter 3 introduced fundamental concepts of computer graphics. A major topic in that chapter was how to represent and render geometry using surface primitives such as points, lines, and polygons. In this chapter our primary focus is on volume graphics. Compared to surface graphics, volume graphics has a greater expressive range in its ability to render inhomogeneous materials, and is a dominant technique for visualizing 3D image (volume) datasets.

We begin the chapter by describing two techniques that are important to both surface and volume graphics. These are simulating object transparency using simple blending functions, and using texture maps to add realism without excessive computational cost. We also describe various problems and challenges inherent to these techniques. We then follow with a focused discussion on volume graphics, including both object-order and image-order techniques, illumination models, approaches to mixing surface and volume graphics, and methods to improve performance. Finally, the chapter concludes with an assortment of important techniques for creating more realistic visualizations. These techniques include stereo viewing, antialiasing, and advanced camera techniques such as motion blur, focal blur, and camera motion.

7.1 Transparency and Alpha Values

Up to this point in the text we have focused on rendering opaque objects that is, we have assumed that objects reflect, scatter, or absorb light at their surface, and no light is transmitted through to their interior. Although rendering opaque objects is certainly useful, there are many applications that can benefit from the ability to render objects that transmit light. One important application of transparency is volume rendering, which we will explore in greater detail later in the chapter.

7.2 Texture Mapping

Texture mapping is a technique to add detail to an image without requiring modelling detail. Texture mapping can be thought of as pasting a picture to the surface of an object. The use of texture mapping requires two pieces of information: a *texture map* and *texture coordinates*. The texture map is the picture we paste, and the texture coordinates specify the location where the picture is pasted. More generally, texture mapping is a table lookup for color, intensity, and/or transparency that is applied to an object as it is rendered. Textures maps and coordinates are most often two-dimensional, but three-dimensional texture maps and coordinates are supported by most new graphics hardware.

7.3 Volume Rendering

Until now we have concentrated on the visualization of data through the use of geometric primitives such as points, lines, and polygons. For many applications such as architectural walk-throughs or terrain visualization, this is obviously the most efficient and effective representation for the data. In contrast, some applications require us to visualize data that is inherently volumetric (which we refer to as 3D image or volume datasets). For example, in biomedical imaging we may need to visualize data obtained from an MR or CT scanner, a confocal microscope, or an ultrasound study. Weather analysis and other simulations also produce large quantities of volumetric data in three or more dimensions that require effective visualization techniques. As a result of the popularity and usefulness of volume data over the last several decades, a broad class of rendering techniques known as volume rendering has emerged. The purpose of volume rendering is to effectively convey information within volumetric data.

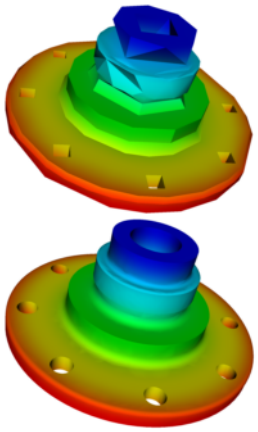
7.4 3D Widgets and User Interaction

Chapter 3 provided an introduction to interaction techniques for graphics (see “Introducing vtkRenderWindowInteractor” on page 61). In the context of visualization, interaction is an essential featureof systems that provide methods for data exploration and query. The classes vtkRenderWindowInteractor and vtkInteractorStyle are core constructs used in VTK to capturewindowing-system specific events in the render window, translate them into VTK events, and then take action as appropriate to that event invocation. In Chapter 3 we saw how these classes could be used to manipulate the camera and actors to interactively produce a desired view. This functionality, however, is relatively limited in its ability to interact with data. For example, users often wish tointeractively control the positioning of streamline starting points, control the orientation of a clipping plane, or transform an actor. While using interpreted languages (see “Interpreted Code” on page 61) can go a long way to provide this interaction, in some situations the ability to see what you are doing when placing objects is essential. Therefore, it is apparent that a variety of user interaction techniques is required by the visualization system if it is to successfully support real-world applications.

7.5 Exercises

8.0

Advanced Data Representation



Adaptive tessellation of higher-order cells.

This chapter examines advanced topics in data representation. Topics include topological and geometric relationships and computational methods for cells and datasets.

8.1 Coordinate Systems

We will examine three different coordinate systems: the global, dataset, and structured coordinate systems. Figure81 shows the relationship between the global and dataset coordinate systems, and depicts the structured coordinate system.

8.1.1 Global Coordinate System

The global coordinate system is a Cartesian, three-dimensional space. Each point is expressed as a triplet of values (x, y, z) along the x, y, and z axes. This is the same system that was described in Chapter 3 (see 3.7). The global coordinate system is always used to specify dataset geometry (i.e., the point coordinates),and data attributes such as normals and vectors. We will use the word "position" to indicate that we are using global coordinates.

8.2 Searching

Searching is an operation to find the cell containing a specified point p, or to locate cells or points in a region surrounding p. Algorithms requiring this operation include streamline generation, where we need to find the starting location within a cell; probing, where the data values at a point are interpolated from the containing cell; or collision detection, where cells in a certain region must be evaluated for intersection. Sometimes (e.g., image datasets), searching is a simple operation because of the regularity of data. However, in less structured data, the searching operation is more complex.

To find the cell containing p, we can use the following naive search procedure. Traverse all cells in the dataset, finding the one (if any) that contains p. To determine whether a cell contains a point, the cell interpolation functions are evaluated for the parametric coordinates (r, s, t). If these coordinates lie within the cell, then p lies in the cell. The basic assumption here is that cells do not overlap, so that at most a single cell contains the given point p. To determine cells or points lying in the region surrounding p, we can traverse cells or points to see whether they lie within the region around p. For example, we can choose to define the region as a sphere centered at p. Then, if a point or the points composing a cell lie in the sphere, the point or cell is considered to be in the region surrounding p.

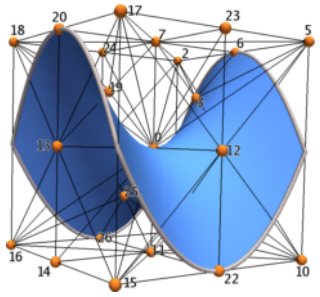
8.3 Putting It All Together

In this section we will finish our earlier description of an implementation for unstructured data. We also define a high-level, abstract interface for cells and datasets. This interface allows us to implement the general (i.e., dataset specific) algorithms in the *Visualization Toolkit*. We also describe implementations for color scalars, searching and picking, and conclude with a series of examples to demonstrate some of these concepts.

8.3.1 Picking

The Visualization Toolkit provides a variety of classes to perform actor (or vtkProp), point, cell, and world point picking (Figure838).

9.0
Advanced Algorithms



An isocontour of a tri-quadratic Lagrange-interpolant. Image courtesy D. Thompson and P. Pébay Sandia National Labs.

We return again to visualization algorithms. This chapter describes algorithms that are either more complex to implement, or less widely used for 3D visualization applications. We retain the classification of algorithms as either scalar, vector, tensor, or modelling algorithms.

9.1 Scalar Algorithms

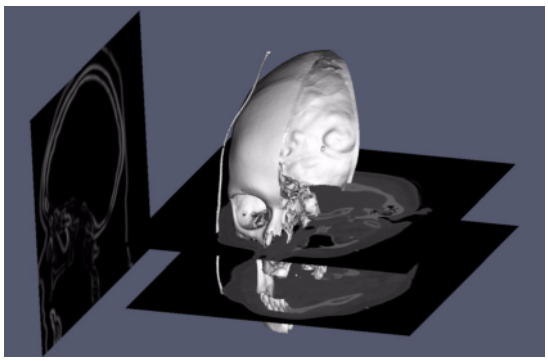
As we have seen, scalar algorithms often involve mapping scalar values through a lookup table, or creating contour lines or surfaces. In this section, we examine another contouring algorithm, dividing cubes , which generates contour surfaces using dense point clouds. We also describe carpet plots. Carpet plots are not true 3D visualization techniques, but are widely used to visualize many types of scalar data. Finally, clipping is another important algorithm related to contouring, where cells are cut into pieces as a function of scalar value.

9.1.1 Dividing Cubes

9.2 Putting It All Together

9.2.1 Connectivity

10.0
Image Processing



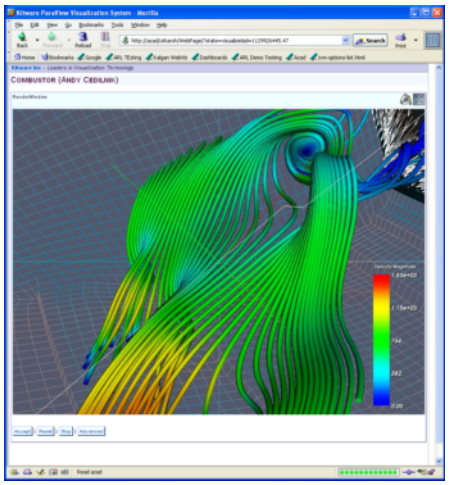
Gradient magnitude of a slice (vertical) of the Visible Woman CT dataset shown with an isosurface and two slices (horizontal) of the dataset.

IN this chapter we describe the image processing components of the Visualization Toolkit. The focus is on key representational ideas, pipeline issues such as data streaming, and useful algorithms for improving the appearance and effectiveness of image data visualizations.

10.1 Introduction

Image processing has been a mainstay of computing since the advent of the digital computer. Early efforts focused on improving image content for human interpretation.

11.0
Visualization on the Web



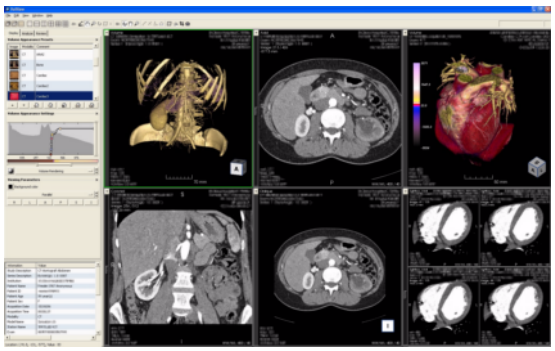
Streamline visualization with ParaView Enterprise Edition.

The early 1990s established the widespread use and accessibility of the World Wide Web. Once a network used primarily by researchers and universities, the Web has become something that is used by people throughout the world. The effects of this transformation have been significant, ranging from personal home pages with static images and text, to professional Web pages embedding animation and virtual reality. This chapter discusses some of those changes and describes how the World Wide Web can be used to make visualization more accessible, interactive, and powerful. Topics covered include the advantages and disadvantages of client-side versus server-side visualization, VRML, and Java3D, interwoven with demonstration examples.

11.1 Motivation

Before describing in detail how to perform visualization over the Web, it is important to understand what we expect to gain. Clearly people have been visualizing data prior to the invention of the Web, but what the Web adds is the ability for people throughout the world to share information quickly and efficiently. Like all successful communication systems, the Web enables people to interact and share information more efficiently compared to other methods. In many ways the Web shares the characteristic of computer visualization in its ability to communicate large amounts of data. For that reason, computer graphics and visualization are now vital parts of the Web, and are becoming widespread in their application.

12.0
Applications



Streamline visualization with ParaView Enterprise Edition.

We have described the design and implementation of an extensive toolkit of visualization techniques. In this chapter we examine several case studies to show how to use these tools to gain insight into important application areas. These areas are medical imaging, financial visualization, modeling, computational fluid dynamics, finite element analysis, and algorithm visualization. For each case, we briefly describe the problem domain and what information we expect to obtain through visualization. Then we craft an approach to show the results. Many times we will extend the functionality of the Visualization Toolkit with application specific tools. Finally, we present a sample program and show resulting images. The visualization design process we go through is similar in each case. First, we read or generate application-specific data and transform it into one of the data representation types in the Visualization Toolkit. Often this first step is the most difficult one because we have to write custom computer code, and decide what form of visualization data to use. In the next step, we choose visualizations for the relevant data within the application. Sometimes this means choosing or creating models corresponding to the physical structure. Examples include spheres for atoms, polygonal surfaces to model physical objects, or computational surfaces to model flow boundaries. Other times we generate more abstract models, such as isosurfaces or glyphs, corresponding to important application data. In the last step we combine the physical components with the abstract components to create a visualization that aids the user in understanding the data.

12.1 3D Medical Imaging

Radiology is a medical discipline that deals with images of human anatomy. These images come from a variety of medical imaging devices, including X-ray, X-ray Computed Tomography

(CT), Magnetic Resonance Imaging (MRI), and ultrasound. Each imaging technique, called an imaging modality, has particular diagnostic strengths.

List of Figures

List of Tables

List of Equations

Glossary

3D Widget An interaction paradigm enabling manipulation of scene objects (e.g., lights, camera, actors, and so on). The 3D widget typically provides a visible representation that can be intuitively and interactively manipulated.. 53

Abstract Class A class that provides methods and data members for the express purpose of deriving subclasses. Such objects are used to define a common interface and attributes for their subclasses..

Abstraction A mental process that extracts the essential form or unifying properties of a concept..

Alpha A specification of opacity (or transparency). An alpha value of one indicates that the object is opaque. An alpha value of zero indicates that the object is completely transparent..

Ambient Lighting The background lighting of unlit surfaces..

Animation A sequence of images displayed in rapid succession. The images may vary due to changes in geometry, color, lighting, camera position, or other graphics parameters. Animations are used to display the variation of one or more variables..

Antialiasing The process of reducing aliasing artifacts. These artifacts typically result from undersampling the data. A common use of antialiasing is to draw straight lines that don't have the jagged edges found in many systems without antialiasing..

API An acronym for application programmer's interface.. 3

Attribute A named member of a class that captures some characteristic of the class. Attributes have a name, a data type, and a data value. This is the same as a data member or instance variable..

Azimuth A rotation of a camera about the vertical (or view up) axis..

Base Class A superclass in C++..

Binocular Parallax The effect of viewing the same object with two slightly different viewpoints to develop depth information..

Boolean Texture A texture map consisting of distinct regions used to "cut" or accentuate features of data. For example, a texture map may consist of regions of zero opacity. When such a texture is mapped onto the surface of an object, portions of its interior becomes visible. Generally used in conjunction with a quadric (or other implicit function) to generate texture coordinates..

C++ A compiled programming language with roots in the C programming language. C++ is an extension of C that incorporates objectoriented principles..

Cell The atoms of visualization datasets. Cells define a topology (e.g., polygon, triangle) in terms of a list of point coordinates..

Cell Attributes Dataset attributes associated with a cell. See also *point attributes*..

Class An object that defines the characteristics of a subset of objects. Typically, it defines methods and data members. All objects instantiated from a class share that class's methods and data members..

Clipping Plane A plane that restricts the rendering or processing of data. Front and back clipping planes are commonly used to restrict the rendering of primitives to those lying between the two planes..

Color Mapping A scalar visualization technique that maps scalar values into color. Generally used to display the variation of data on a surface or through a volume..

Compiled System A compiled system requires that a program be compiled (or translated into a lowerlevel language) before it is executed. Contrast with *interpreted systems*..

Composite Cell A cell consisting of one or more primary cells..

Concrete Class A class that can be instantiated. Typically, abstract classes are not instantiated but concrete classes are..

Connectivity A technique to extract connected cells. Cells are connected when they share common features such as points, edges, or faces..

Constructor A class method that is invoked when an instance of that class is created. Typically the constructor sets any default values and allocates any memory that the instance needs. See also *destructor*..

Contouring 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 iso-lines. Contour surfaces are called isovalue surfaces or isosurfaces..

Critical Points Locations in a vector field where the local vector magnitude goes to zero and the direction becomes undefined..

CT (Computed Tomography) A data acquisition technique based on X-rays. Data is acquired in a 3D volume as a series of slice planes (i.e., a stack of n^2 points)..

Cutting A visualization technique to slice through or cut data. The cutting surface is typically described with an implicit function, and data attributes are mapped onto the cut surface. See also *boolean texture*..

Data Extraction The process of selecting a portion of data based on characteristics of the data. These characteristics may be based on geometric or topological constraints or constraints on data attribute values..

Data Flow Diagram A diagram that shows the information flow and operations on that information as it moves throughout a program or process..

Data Member A named member of a class that captures some characteristic of the class. Data members have a name, a data type, and a data value. This is the same as an attribute or instance variable..

Data Object An object that is an abstraction of data. For example, a patient's file in a hospital could be a data object. Typical visualization objects include structured grids and volumes. See also *process object*..

Data Visualization The process of transforming data into sensory stimuli, usually visual images. Data visualization is a general term, encompassing data from engineering and science, as well as information from business, finance, sociology, geography, information management, and other fields. Data visualization also includes elements of data analysis, such as statistical analysis. Contrast with *scientific visualization* and *information visualization*..

Dataset The general term used to describe visualization data. Datasets consist of structure (geometry and topology) and dataset attributes (scalars, vectors, tensors, etc.)..

Dataset Attributes The information associated with the structure of a dataset. This can be scalars, vectors, tensors, normals, and texture coordinates, or arbitrary data arrays that may be contained in the field..

Decimation A type of polygon reduction technique that deletes points in a polygonal mesh that satisfies a co-planar or co-linear condition and replaces the resulting hole with a new triangulation..

Delaunay Triangulation A triangulation that satisfies the Delaunay circumsphere criterion. This criterion states that a circumsphere of each simplex in the triangulation contains only the points defining the simplex..

Delegation The process of assigning an object to handle the execution of another object's methods. Sometimes it is said that one object forwards certain methods to another object for execution..

Demand-driven A method of visualization pipeline update where the update occurs only when data is requested and occurs only in the portion of the network required to generate the data..

Derived Class A class that is more specific or complete than its superclass. The derived class, which is also known as the subclass, inherits all the members of its superclass. Usually a derived class adds new functionality or fills in what was defined by its superclass. See also *subclass*..

Destructor A class method that is invoked when an instance of that class is deleted. Typically the destructor frees memory that the instance was using. See also *constructor*..

Device Mapper A mapper that interfaces data to a graphics library or subsystem..

Diffuse Lighting Reflected light from a matte surface. Diffuse lighting is a function of the relative angle between the incoming light and surface normal of the object..

Displacement Plots A vector visualization technique that shows the displacement of the surface of an object. The method generates scalar values by computing the dot product between the surface normal and vector displacement of the surface. The scalars are visualized using color mapping..

Display Coordinate System A coordinate system that is the result of mapping the view coordinate system onto the display hardware..

Divergence In numerical computation: the tendency of computation to move away from the solution. In fluid flow: the rapid motion of fluid particles away from one another..

Dividing Cubes A contour algorithm that represents isosurfaces as a dense cloud of points..

Dolly A camera operation that moves the camera position towards (from the camera focal point. *dolly in*) or away (*dolly out*)..

Double Buffering A display technique that is used to display animations more smoothly. It consists of using two buffers in the rendering process. While one buffer is being displayed, the next frame in the animation is being drawn on the other buffer. Once the drawing is complete the two buffers are swapped and the new image is displayed..

Dynamic Memory Model A data flow network that does not retain intermediate results as it executes. Each time the network executes, it must recompute any data required as input to another process object. A dynamic memory model reduces system memory requirements but places greater demands on computational requirements..

Dynamic Model A description of a system concerned with synchronizing events and objects..

Effective Stress A mathematical combination of the normal and shear stress components that provide a measure of the stress at a point. Effective stress is a scalar value, while stress is represented with a tensor value. See *stress*..

Eigenfields Vector fields defined by the eigenvectors of a tensor..

Eigenvalue A characteristic value of a matrix. Eigenvalues often correspond to physical phenomena, such as frequency of vibration or magnitude of principal components of stress..

Eigenvector A vector associated with each eigenvalue. The eigenvector spans the space of the matrix. Eigenvectors are orthogonal to one another. Eigenvectors often correspond to physical phenomena such as mode shapes of vibration..

Elevation A rotation of a camera about the horizontal axis..

Entity Something within a system that has identity. Chairs, airplanes, and cameras are things that correspond to physical entities in the real world. A database and isosurface algorithm are examples of nonphysical entities..

Event-driven A method of visualization pipeline update where updates occur when an event affects the pipeline, e.g., when an object instance variable is set or modified. See also *demand-driven*..

Execution The process of updating a visualization network..

Explicit Execution Controlling network updates by performing explicit dependency analysis..

Exporter An object that saves a VTK scene definition to a file or other program. (A scene consists of lights, cameras, actors, geometry, properties, texture, and other pertinent data.) See also *importer*..

Fan-in The flow of multiple pieces of data into a single filter..

Fan-out The flow of data from a filter's output to other objects..

Feature Angle The angle between surface normal vectors, e.g., the angle between the normal vectors on two adjacent polygons..

Filter A process object that takes at least one input and generates at least one output..

Finite Difference Method A numerical technique for the solution of partial differential equations (PDEs). Finite difference methods replace the PDEs with truncated Taylor series approximations. This results in a system of equations that is solved on a computer. Typical applications include fluid flow, combustion, and heat transfer..

Finite Element Method (FEM) A numerical technique for the solution of partial differential equations. FEM is based on discretizing a domain into elements (and nodes) and constructing basis (or interpolation) functions across the elements. From these functions a system of linear equations is generated and solved on the computer. Typical applications include stress, heat transfer, and vibration analysis..

Flat Shading A shading technique where the lighting equation for a geometric primitive is calculated once, and then used to fill in the entire area of the primitive. This is also known as faceted shading. See also *gouraud shading* and *phong shading*..

Functional Model The description of a system based on what it does..

Generalization The abstraction of a subset of classes to a common superclass. Generalization extracts the common members or methods from a group of classes to create a common superclass. See also *specialization* and *inheritance*..

Geometry Used generally to mean the characteristic position, shape, and topology of an object. Used specifically (in tandem with topology) to mean the position and shape of an object..

Glyph A general visualization technique used to represent data using a meaningful shape or pictorial representation. Each glyph is generally a function of its input data and may change size, orientation, and shape; or modify graphics properties in response to changes in input..

Gouraud Shading A shading technique that applies the lighting equations for a geometric primitive at each vertex. The resulting colors are then interpolated over the areas between the vertices. See also *flat shading* and *Phong shading*..

Hedgehog A vector visualization technique that represents vector direction and magnitude with oriented lines..

Height Field A set of altitude or height samples in a rectangular grid. Height fields are typically used to represent terrain..

Hexahedron A type of primary 3D cell. The hexahedron looks like a "brick." It has six faces, 12 edges, and eight vertices. The faces of the hexahedron are not necessarily planar..

Homogeneous Coordinates An alternate coordinate representation that provides more flexibility than traditional Cartesian coordinates. This includes perspective transformation and combined translation, scaling, and rotation..

Hyperstreamline A tensor visualization technique. Hyperstreamlines are created by treating the eigenvectors as three separate vectors. The maximum eigenvalue/eigenvector is used as a vector field in which particle integration is performed (like streamlines). The other two vectors control the cross-sectional shape of an ellipse that is swept along the integration path. See also *streampolygon*..

Image Data A dataset whose structure is both geometrically and topologically regular. Both geometry and topology are implicit. A 3D image dataset is known as a volume. A 2D image dataset is known as a pixmap..

Image-Order Techniques Rendering techniques that determine for each pixel in the image plane which data samples contribute to it. Image-order techniques are implemented using ray casting. Contrast with *object-order techniques*..

Implicit Execution Controlling network updates by distributing network dependency throughout the visualization process objects. Each process object requests that its input be updated before it executes. This results in a recursive update/execution process throughout the network..

Implicit Function A mathematical function of the form, where $Fxyz() = c$, c is a constant often $= 0$..

Implicit Modelling A modelling technique that represents geometry as a scalar field. Usually the scalar is a distance function or implicit function distributed through a volume..

Importer An object that interfaces to external data or programs to define a complete scene in VTK. (The scene consists of lights, cameras, actors, geometry, properties, texture, and other pertinent data.) See also *exporter*..

Information Visualization The process of transforming information into sensory stimuli, usually visual images. Information visualization is used to describe the process of visualizing data without structure, such as information on the World Wide Web; or abstract data structures, like computer file systems or documents. Contrast with *scientific visualization* and *data visualization*..

Inheritance A process where the attributes and methods of a superclass are bestowed upon all subclasses derived from that superclass. It is said that the subclasses inherit their superclasses' methods and attributes..

Instance An object that is defined by a class and used by a program or application. There may be many instances of a specific class..

Instance Variable A named member of a class that captures a characteristic of the class. Instance variables have a name, a data type, and a data value. The phrase, instance variable, is often abbreviated as ivar. This is the same as an attribute or data member..

Intensity The light energy transferred per unit time across a unit plane perpendicular to the light rays..

Interpolation Functions Functions continuous in value and derivatives used to interpolate data from known points and function values. Cells use interpolation functions to compute data values interior to or on the boundary of the cell..

Interpreted System An interpreted system can execute programs without going through a separate compilation stage. Interpreted systems often allow the user to interact and modify the program as it is running. Contrast with *compiled systems*..

Irregular Data Data in which the relationship of one data item to the other data items in the dataset is arbitrary. Irregular data is also known as unstructured data..

Iso-parametric A form of interpolation in which interpolation for data values is the same as for the local geometry. Compare with *sub-parametric* and *super-parametric*..

Isosurface A surface representing a constant valued scalar function. See *contouring*..

Isovalue The scalar value used to generate an isosurface..

Jacobian A matrix that relates one coordinate system to another..

Line A cell defined by two points..

Manifold Topology A domain is manifold at a point p in a topological space of dimension n if the neighborhood around p is homeomorphic to an n-dimensional sphere. Homeomorphic means that the mapping is one to one without tearing (i.e., like mapping a rubber sheet from a square to a disk). We generally refer to an object's topology as manifold if every point in the object is manifold. Contrast with nonmanifold topology..

Mapper A process object that terminates the visualization network. It maps input data into graphics libraries (or other devices) or writes data to disk (or a communication device)..

Marching Cubes A contouring algorithm to create surfaces of constant scalar value in 3D. Marching cubes is described for volume datasets, but has been extended to datasets consisting of other cell types..

Member Function A member function is a function or transformation that can be applied to an object. It is the functional equivalent to a data member. Member functions define the behavior of an object. Methods, operations, and member functions are essentially the same..

Method A function or transformation that can be applied to an object. Methods define the behavior of an object. Methods, operations, and member functions are essentially the same..

Modal Lines Lines on the surface of a vibrating object that separate regions of positive and negative displacement..

Mode Shape The motion of an object vibrating at a natural frequency. See also *eigenvalues* and *eigenvectors*..

Model Coordinate System The coordinate system that a model or geometric entity is defined in. There may be many different model coordinate systems defined for one scene..

Morph A progressive transformation of one object into another. Generally used to transform images (2D morphing) and in some cases geometry (3D morphing)..

Motion Blur An artifact of the shutter speed of a camera. Since the camera's shutter stays open for a finite amount of time, changes in the scene that occur during that time can result in blurring of the resulting image..

MRI (Magnetic Resonance Imaging) A data acquisition technique based on measuring variation in magnetic field in response to radio-wave pulses. The data is acquired in a 3D region as a series of slice planes (i.e., a stack of n^2 points)..

Multidimensional Visualization Visualizing data of four or more variables. Generally requires a mapping of many dimensions into three or fewer dimensions so that standard visualization techniques can be applied..

Multiple Input Process objects that accept more than one input..

Multiple Output Process objects that generate more than one output..

Nonmanifold Topology Topology that is not manifold. Examples include polygonal meshes, where an edge is used by more than two polygons, or polygons connected to each other at their vertices (i.e., do not share an edge). Contrast with *manifold topology*..

Normal A unit vector that indicates perpendicular direction to a surface. Normals are a common type of data attribute..

Object An abstraction that models the state and behavior of entities in a system. Instances and classes are both objects..

Object Factory An object used to construct or instantiate other objects. In VTK, object factories are implemented using the class method New()..

Object Model The description of a system in terms of the components that make up the system, including the relationship of the components one to another..

Object-Order Techniques Rendering techniques that project object data (e.g., polygons or voxels) onto the image plane. Example techniques include ordered compositing and splatting..

Object-Oriented A software development technique that uses objects to represent the state and behavior of entities in a system..

Octree Decomposition A technique to decompose a cubical region of three-dimensional space into smaller cubes. The cubes, or octants, are related in tree fashion. The root octant is the cubical region. Each octant may have eight children created by dividing the parent in half in the x, y, and z directions..

OMT *Object Modelling Technique*. An object-oriented design technique that models software systems with object, dynamic, and functional diagrams..

Operation A function or transformation that can be applied to an object. Operations define the behavior of an object. Methods and member functions implement operations..

Overloading Having multiple methods with the same name. Some methods are overloaded because there are different versions of the same method. These differences are based on argument types, while the underlying algorithm remains the same. Contrast with *polymorphic*..

Painter's Algorithm An object-order rendering technique that sorts rendering primitives from back to front and then draws them..

Parallel Projection A mapping of world coordinates into view coordinates that preserves all parallel lines. In a parallel projection an object will appear the same size regardless of how far away it is from the viewer. This is equivalent to having a center of projection that is infinitely far away. Contrast with *perspective projection*..

Parametric Coordinates A coordinate system natural to the geometry of a geometric object. For example, a line may be described by the single coordinate s even though the line may lie in three or higher dimensions..

Particle Trace The trajectory that particles trace over time in fluid flow. Particle traces are everywhere tangent to the velocity field. Unlike streamlines, particle lines are time-dependent..

Pathline The trajectory that a particle follows in fluid flow..

Perspective Projection A mapping of world coordinates into view coordinates that roughly approximates a camera lens. Specifically, the center of projection must be a finite distance from the view plane. As a result closer, objects will appear larger than distant objects. Contrast with *parallel projection*..

Phong Shading A shading technique that applies the lighting equations for a geometric primitive at each pixel. See also *flat shading* and *Gouraud shading*..

Pitch A rotation of a camera's position about the horizontal axis, centered at its viewpoint. See also *yaw* and *roll*. Contrast with *elevation*..

Pixel Short for picture element. Constant valued elements in an image. In VTK, a two-dimensional cell defined by an ordered list of four points..

Point A geometric specification of position in 3D space..

Point Attributes Data attributes associates with the points of a dataset..

Polygon A cell consisting of three or more co-planar points defining a polygon. The polygon can be concave but without embedded loops..

Polygon Reduction A family of techniques to reduce the size of large polygonal meshes. The goal is to reduce the number of polygons, while preserving a "good" approximation to the original geometry. In most techniques topology is preserved as well..

Polygonal Data A dataset type consisting of arbitrary combinations of vertices, polyvertices, lines, polylines, polygons, and triangle strips. Polygonal data is an intermediate data form that can be easily rendered by graphics libraries, and yet can represent many types of visualization data..

Polymorphic Having many forms. Some methods are polymorphic because the same method in different classes may implement a different algorithm. The semantics of the method are typically the same, even though the implementation may differ. Contrast with *overloading*..

Prandtl number A dimensionless number, named after the German physicist Ludwig Prandtl, defined as the ratio of momentum diffusivity to thermal diffusivity..

Probing Also known as sampling or resampling. A data selection technique that selects data at a set of points..

Process Object A visualization object that is an abstraction of a process or algorithm. For example, the isosurfacing algorithm marching cubes is implemented as a process object. See also *data object*..

Progressive Mesh A representation of a triangle mesh that enables incremental refinement and derefinement. The data representation is compact and is useful for transmission of 3D triangle meshes across a network. See also *polygon reduction*..

Properties A general term used to describe the rendered properties of an actor. This includes lighting terms such as ambient, diffuse, and specular coefficients; color and opacity; shading techniques such as flat and Gouraud; and the actor's geometric representation (wireframe, points, or surface)..

Pyramid A type of primary 3D cell. The pyramid has a quadrilateral base connected to a single apex point. It has five faces, eight edges, and five vertices. The base face of the pyramid is not necessarily planar..

Quadratic Edge A type of primary 1D cell with a quadratic interpolation function. The quadratic edge is defined by three points: two end points and a mid-edge node..

Quadratic Hexahedron A type of primary 3D cell with quadratic interpolation functions. The quadratic edge is defined by twenty points: eight corner points and twelve mid-edge nodes..

Quadratic Quadrilateral A type of primary 2D cell with quadratic interpolation functions. The quadratic quadrilateral is defined by eight points: four corner points and four mid-edge nodes..

Quadratic Tetrahedron A type of primary 3D cell with quadratic interpolation functions. The quadratic tetrahedron is defined by ten points: four corner points and six mid-edge nodes..

Quadratic Triangle A type of primary 2D cell with quadratic interpolation functions. The quadratic triangle is defined by six points: three corner points and three mid-edge nodes..

Quadric A function of the form $f(x,y,z) = a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9$ The quadric equation can represent many useful 3D objects such as spheres, ellipsoids, cylinders, and cones..

Quadrilateral (Quad) A type of primary 2D cell. The quadrilateral is four sided with four vertices. The quadrilateral must be convex..

Rayleigh number A dimensionless number associated with buoyancy–driven flow, also known as free convection or natural convection. When the Rayleigh number is below a critical value for that fluid, heat transfer is primarily in the form of conduction; when it exceeds the critical value, heat transfer is primarily in the form of convection..

Reader A source object that reads a file or files and produces a data object..

Reference Counting A memory management technique used to reduce memory requirements. Portions of memory (in this case objects) may be referenced by more than one other object. The referenced object keeps a count of references to it. If the count returns to zero, the object deletes itself, returning memory back to the system. This technique avoids making copies of memory..

Region of Interest A portion of a dataset that the user is interested in visualizing. Sometimes abbreviated ROI..

Regular Data Data in which one data item is related (either geometrically or topologically) to other data items. Also referred to as structured data..

Rendering The process of converting object geometry (i.e., geometric primitives), object properties, and a specification of lights and camera into an image. The primitives may take many forms including surface primitives (points, lines, polygons, splines), implicit functions, or volumes..

Resonant Frequency A frequency at which an object vibrates..

Roll A rotation of a camera about its direction of projection. See also *azimuth*, *elevation*, *pitch*, and *yaw*..

Sampling Selective acquisition or sampling of data, usually at a regular interval. See also *probing*..

Scalar A single value or function value. May also be used to represent a field of such values..

Scalar Generation Creating scalar values from other data such as vectors or tensors. One example is computing vector norm..

Scalar Range The minimum and maximum scalar values of a scalar field..

Scene A complete representation of the components required to generate an image or animation including lights, cameras, actors. properties, transformations, geometry, texture, and other pertinent information..

Scene Graph A hierarchical, acyclic, directed tree representation of a scene. The graph order (depth first) controls when objects are processed by the graphics system..

Scientific Visualization The process of transforming data into sensory stimuli, usually visual images. Generally used to denote the application of visualization to the sciences and engineering. Contrast with *data visualization* and *information visualization*..

Searching The process of locating data. Usually the search is based on spatial criteria such as position or being inside a cell..

Segmentation Identification and demarcation of tissue types. Segmentation is generally applied to CT and MRI data to associate soft tissue with a particular body organ or anatomical structure..

Simplex The convex combination of n independent vectors in n-space forms an n-dimensional simplex. Points, lines, triangles, and tetrahedra are examples of simplices in 0D, 1D, 2D, and 3D..

Source A process object that produces at least one output. Contrast with *filter*..

Specialization The creation of subclasses that are more refined or specialized than their superclass. See also *generalization* and *inheritance*..

Specular Lighting Reflected lighting from a shiny surface. Specular lighting is a function of the relative angle between the incoming light, the surface normal of the object, and the view angle of the observer..

Splatting A method to distribute data values across a region. The distribution functions are often based on Gaussian functions..

State Diagram A diagram that relates states and events. Used to describe behavior in a software system..

Static Memory Model A data flow network that retains intermediate results as it executes. A static memory model minimizes computational requirements, but places greater demands on memory requirements..

Strain A nondimensional quantity expressed as the ratio of the displacement of an object to its length (normal strain), or angular displacement (shear strain). Strain is a tensor quantity. See also *stress*..

Streakline The set of particles that have previously passed through a particular point..

Streamline Curves that are everywhere tangent to the velocity field. A streamline satisfies the integral curve $\frac{d\vec{x}}{ds} = \vec{v}(x, t')$ at sonme time t'..

Streampolygon A vector and tensor visualization technique that represents flow with tubes that have polygonal cross sections. The method is based on integrating through the vector field and then sweeping a regular polygon along the streamline. The radius, number of sides, shape, and rotation of the polygon are allowed to change in response to data values. See also *hyperstreamline*..

Streamribbon A vector visualization technique that represents vectors with ribbons that are everywhere tangent to the vector field..

Streamsurface . A surface that is everywhere tangent to a vector field. Can be approximated by generating a series of streamlines along a curve and connecting the lines with a surface..

Streamwise Vorticity A measure of the rotation of flow around a streamline..

Stress A measure of force per unit area. Normal stress is stress normal to a given surface, and is either compressive (a negative value) or tensile (a positive value). Shear stress acts tangentially to a given surface. Stress is related to strain through the linear proportionality constants (theE modulus of elasticity), (Poisson's ratio), and (modulusG of elasticity in shear). Stress is a tensor quantity. See also *strain*..

Structured Data Data in which one data item is related (either geometrically or topologically) to other data items. Also referred to as regular data..

Structured Grid A dataset whose structure is topologically regular but whose geometry is irregular. Geometry is explicit and topology is implicit. Typically, structured grids consist of hexahedral cells..

Structured Points *Preferred term is* Image Data - A dataset whose structure is both geometrically and topologically regular. Both geometry and topology are implicit. A 3D structured point dataset is known as a volume. A 2D structured point dataset is known as a pixmap..

Sub-parametric A form of interpolation in which interpolation for data values is of higher order than that for the local geometry. Compare with *iso-parametric* and *super-parametric*..

Subclass A class that is more specific or complete than its superclass. The subclass, which is also known as the derived class, inherits all the members of its superclass. Usually a subclass will add some new functionality or fill in what was defined by its superclass. See also *derived class*..

Subsampling Sampling data at a resolution at less than final display resolution..

Super-parametric A form of interpolation in which interpolation for data values is of lower order than that for the local geometry. Compare with *iso-parametric* and *sub-parametric*..

Superclass A class from which other classes are derived. See also *base class*..

Surface Rendering Rendering techniques based on geometric surface primitives such as points, lines, polygons, and splines. Contrast with *volume rendering*..

Tensor A mathematical generalization of vectors and matrices. A tensor of rank k can be considered a k-dimensional table. Tensor visualization algorithms treat 3 x 3 real symmetric matrix tensors (rank 2 tensors)..

Tensor Ellipsoid A type of glyph used to visualize tensors. The major, medium, and minor eigenvalues of a tensor33Âť define an ellipsoid. The eigenvalues are used to scale along the axes..

Tetrahedron A 3D primary cell that is a simplex with four triangular faces, six edges, and four vertices..

Texture Animation Rapid application of texture maps to visualize data. A useful example maps a 1D texture map of varying intensity along a set of lines to simulate particle flow..

Texture Coordinate Specification of position within texture map. Texture coordinates are used to map data from Cartesian system into 2D or 3D texture map..

Texture Map A specification of object properties in a canonical region. These properties are most often intensity, color, and alpha, or combinations of these. The region is typically a structured array of data in a pixmap (2D) or in a volume (3D)..

Texture Mapping A rendering technique to add detail to objects without requiring extensive geometric modelling. One common example is to paste a picture on the surface of an object..

Texture Thresholding Using texture mapping to display selected data. Often makes use of alpha opacity to conceal regions of minimal interest..

Thresholding A data selection technique that selects data that lies within a range of data. Typically scalar thresholding selects data whose scalar values meet a scalar criterion..

Topological Dimension The dimension or number of parametric coordinates required to address the domain of an object. For example, a line in 3D space is of topological dimension one because the line can be parametrized with a single parameter..

Topology A subset of the information about the structure of a dataset. Topology is a set of properties invariant under certain geometric transformation such as scaling, rotation, and translation..

Transformation Matrix A $4x4matrix$ of values used to control the position, orientation, and scale of objects..

Triangle A primary 2D cell. The triangle is a simplex with three edges and three vertices..

Triangle Strip A composite 2D cell consisting of triangles. The triangle strip is an efficient representation scheme for triangles where points $n + 2$ can represent n triangles..

Triangular Irregular Network (TIN) An unstructured triangulation consisting of triangles. Often used to represent terrain data..

Triangulation A set of nonintersecting simplices sharing common vertices, edges, and/or faces..

Type Checking The process of enforcing compatibility between objects..

Type Converter . A type of filter used to convert from one dataset type to another..

Uniform Grid A synonym for image data..

Unstructured Data Data in which one data item is unrelated (either geometrically or topologically) to other data items. Also referred to as irregular data..

Unstructured Grid A general dataset form consisting of arbitrary combinations of cells and points. Both the geometry and topology are explicitly defined..

Unstructured Points A dataset consisting of vertex cells that are positioned irregularly in space, with no implicit or explicit topology..

Vector A specification of direction and magnitude. Vectors can be used to describe fluid velocity, structural displacement, or object motion..

Vector Field Topology Vector fields are characterized by regions flow diverges, converges, and/or rotates. The relationship of these regions one to another is the topology of the flow..

Vertex A primary 0D cell. Is sometimes used synonymously with point or node..

View Coordinate System The projection of the world coordinate system into the camera's viewing frustrum..

View Frustrum The viewing region of a camera defined by six planes: the front and back clipping planes, and the four sides of a pyramid defined by the camera position, focal point, and view angle (or image viewport if viewing in parallel projection)..

Visual Programming A programming model that enables the construction and manipulation of visualization applications. A typical implementation is the construction of a visualization pipeline by connecting execution modules into a network..

Visualization The process of converting data to images (or other sensory stimuli). Alternatively, the end result of the visualization process..

Visualization Network A series of process objects and data objects joined together into a dataflow network..

Volume A regular array of points in 3D space. Volumes are often defined as a series of 2D images arranged along the z-axis..

Volume Rendering The process of directly viewing volume data without converting the data to intermediate surface primitives. Contrast with *surface rendering*..

Vorticity A measure of the rotation of fluid flow..

Voxel Short for volume element. In VTK, a primary three-dimensional cell with six faces. Each face is perpendicular to one of the coordinate axes..

Warping A scalar and vector visualization technique that distorts an object to magnify the effects of data value. Warping may be used on vector data to display displacement or velocity, or on scalar data to show relative scalar values..

Wedge A type of primary 3D cell. The wedge has two triangular faces connected with three quadrilateral faces. It has five faces, nine edges, and six vertices. The quadrilateral faces of the wedge are not necessarily planar..

World Coordinate System A three-dimensional Cartesian coordinate system in which the main elements of a rendering scene are positioned..

Writer A type of mapper object that writes data to disk or other I/O device..

Yaw A rotation of a camera's position about the vertical axis, centered at its viewpoint. See also *pitch* and *roll*. Contrast with *azimuth*..

Z-Buffer Memory that contains the depth (along the view plane normal) of a corresponding element in a frame buffer..

Z-Buffering A technique for performing hidden line (point, surface) removal by keeping track of the current depth, or z value for each pixel. These values are stored in the z-buffer..