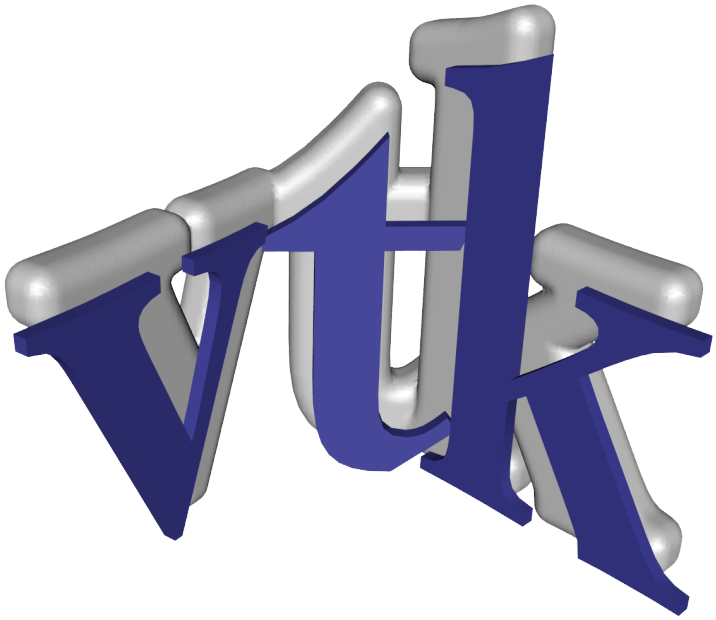
The Visualization Toolkit
An Object-Oriented Approach To 3D Graphics
Fourth Edition



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 . . . 5
Acknowledgments . . . 7

1 Introduction 9
1.1 What Is Visualization? . . . 9
1.1.1 Terminology . . . 9
1.1.2 Examples of Visualization . . . 10
1.2 Why Visualize? . . . 11
1.3 Imaging, Computer Graphics, and Visualization . . . 12
1.4 Origins of Data Visualization . . . 13
1.5 Purpose of This Book . . . 14
1.6 What This Book Is Not . . . 15
1.7 Intended Audience . . . 15
1.8 How to Use This Book . . . 15
1.9 Software Considerations and Example Code . . . 16
1.10 Chapter-by-Chapter Overview . . . 17
1.11 Legal Considerations . . . 19
1.12 Bibliographic Notes . . . 19

2 Object-Oriented Design 23
2.1 Introduction . . . 23
2.2 Bibliographic Notes . . . 23

3 Computer Graphics Primer 27
3.1 Introduction . . . 27
3.2 A Physical Description of Rendering . . . 27
3.2.1 Image-Order and Object-Order Methods . . . 28
3.2.2 Surface versus Volume Rendering . . . 29
3.2.3 Visualization Not Graphics . . . 30
3.3 Color . . . 30
3.4 Lights . . . 32
3.5 Surface Properties . . . 33
3.6 Cameras . . . 36
3.7 Coordinate Systems . . . 38
3.8 Coordinate Transformation . . . 40
3.9 Actor Geometry . . . 43
3.9.1 Modelling . . . 43
3.9.2 Actor Location and Orientation . . . 43
3.10 Graphics Hardware . . . 44
3.10.1 Raster Devices . . . 44

3.10.2 Interfacing to the Hardware . . . 46
3.10.3 Rasterization . . . 48
3.10.4 Z-Buffer . . . 50
3.11 Putting It All Together . . . 52
3.11.1 The Graphics Model . . . 52
3.11.2 Achieving Device Independence . . . 54
3.11.3 Examples . . . 56
3.12 Chapter Summary . . . 71
3.13 Bibliographic Notes . . . 71
3.14 Exercises . . . 73

4 The Visualization Pipeline 77
4.1 Overview . . . 77

5 Basic Data Representation 79
5.1 Introduction . . . 79
5.1.1 Characterizing Visualization Data . . . 79

6 Fundamental Algorithms 81
6.1 Introduction . . . 81

7 Advanced Computer Graphics 83
7.1 Transparency and Alpha Values . . . 83
7.2 Texture Mapping . . . 83
7.3 Volume Rendering . . . 84
7.4 3D Widgets and User Interaction . . . 84

8 Advanced Data Representation 85
8.1 Coordinate Systems . . . 85
8.1.1 Global Coordinate System . . . 85
8.2 Putting It All Together . . . 85
8.2.1 Picking . . . 85

9 Advanced Algorithms 87
9.1 Scalar Algorithms . . . 87
9.1.1 Dividing Cubes . . . 87

10 Image Processing 89
10.1 Introduction . . . 89

11 Visualization on the Web 91
11.1 Motivation . . . 91

12 Applications 93
12.1 3D Medical Imaging . . . 93

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

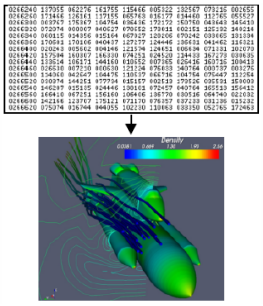


Figure 1.1: Visualization transforms numbers to images.

Visualization — "2: the act or process of interpreting in visual terms or of putting into visual form, Webster's 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.

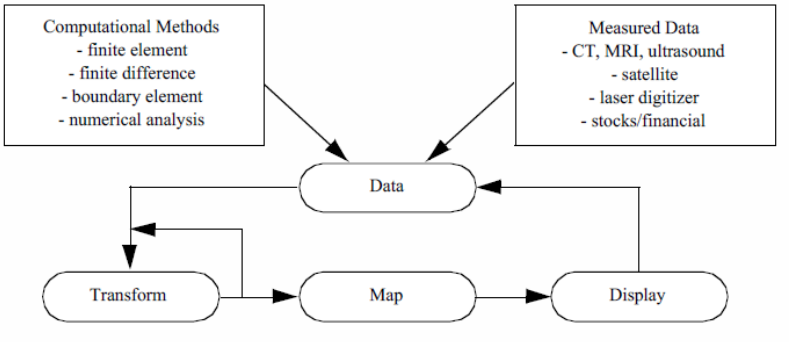


Figure 1.2: The visualization process. Data from various sources is repeatedly transformed to extract, derive, and enhance information. The resulting data is mapped to a graphics system for display.

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

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 1.2 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/Graph-

ics/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: Application

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. Mac-intosh 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 User's 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 API's, 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 User's 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

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

Today's 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 system's 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 series—of which [9] and [19] are quite popular—are 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] D. Robson A. Goldberg. 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] S. Orlow K. Gorlen 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 Programmer's 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/.

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 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

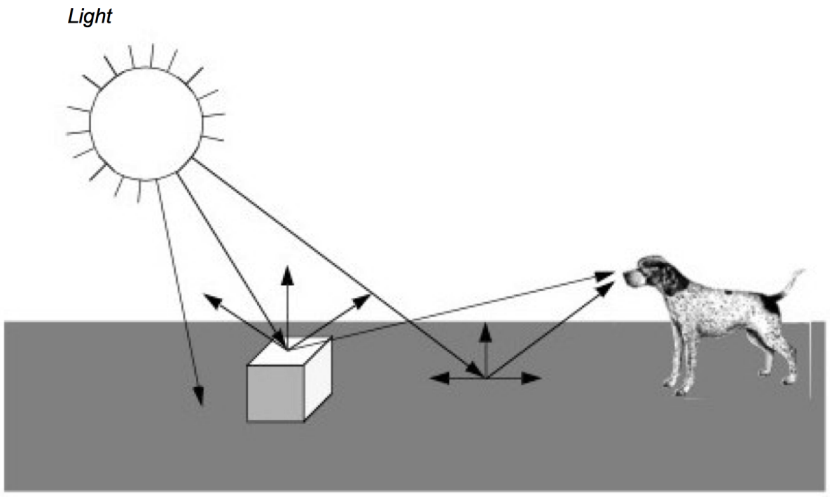


Figure 3.1: Physical generation of an image.

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 71 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.

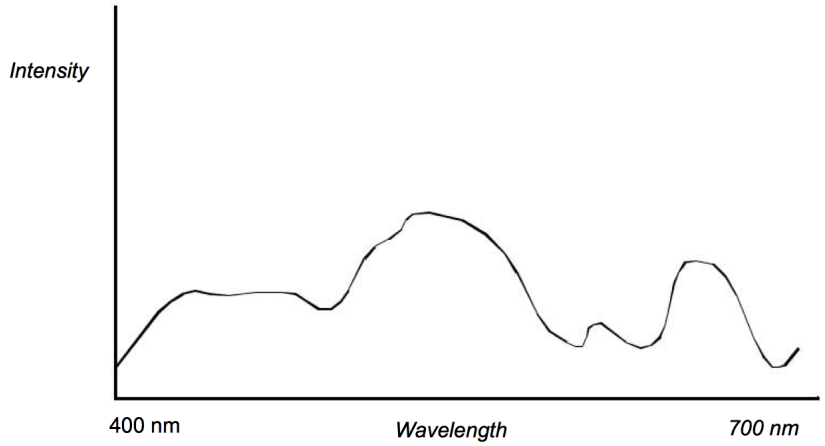


Figure 3.2: Wavelength versus Intensity plot.

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.

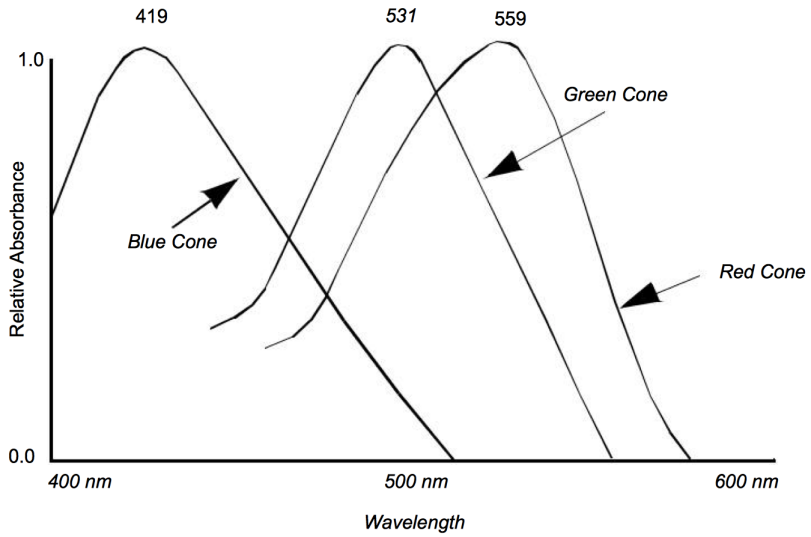


Figure 3.3: Relative absorbance of light by the three types of cones in the human retina.

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

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

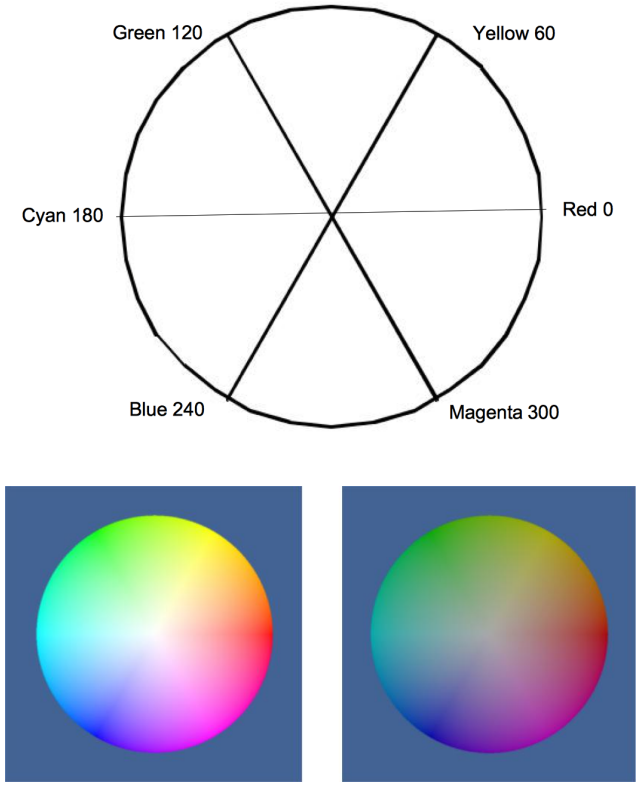


Figure 3.4: On the top, circular representation of hue. The other two images on the bottom are slices through the HSV color space. The first slice has a value of 1.0, the other has a value of 0.5.

in a fluorescent light). The point source lighting model assumes that the light is emitted in all directions from a single point 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

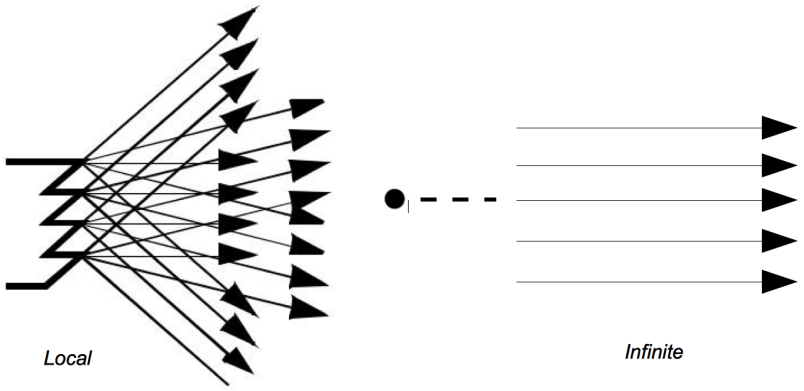


Figure 3.5: Local light source with a finite volume versus an infinite point light source.

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 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

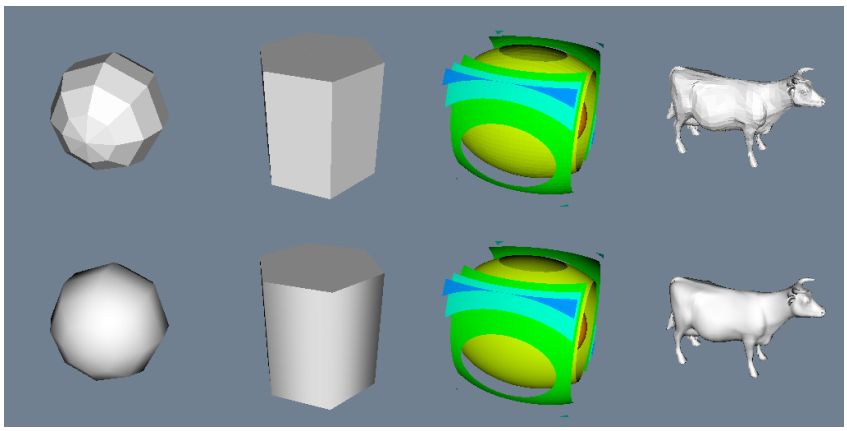


Figure 3.6: Flat and Gouraud shading. Different shading methods can dramatically improve the look of an object represented with polygons. On the top, flat shading uses a constant surface normal across each polygon. On the bottom, Gouraud shading interpolates normals from polygon vertices to give a smoother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

(which varies between the top and bottom rows) controls the intensity of the specular lighting. The specular power, O_sp, 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

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

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

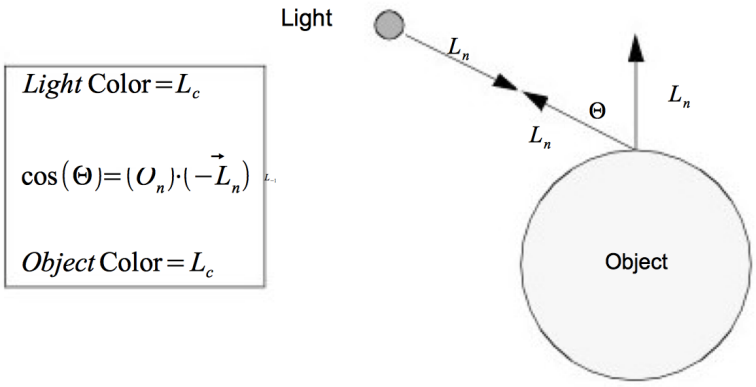


Figure 3.7: Diffuse lighting

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

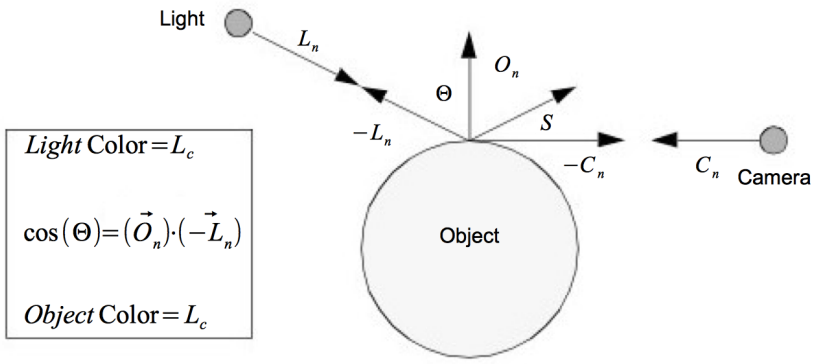


Figure 3.8: Specular lighting.

ambient, diffuse and specular lighting into one equation.

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

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.

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.

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.

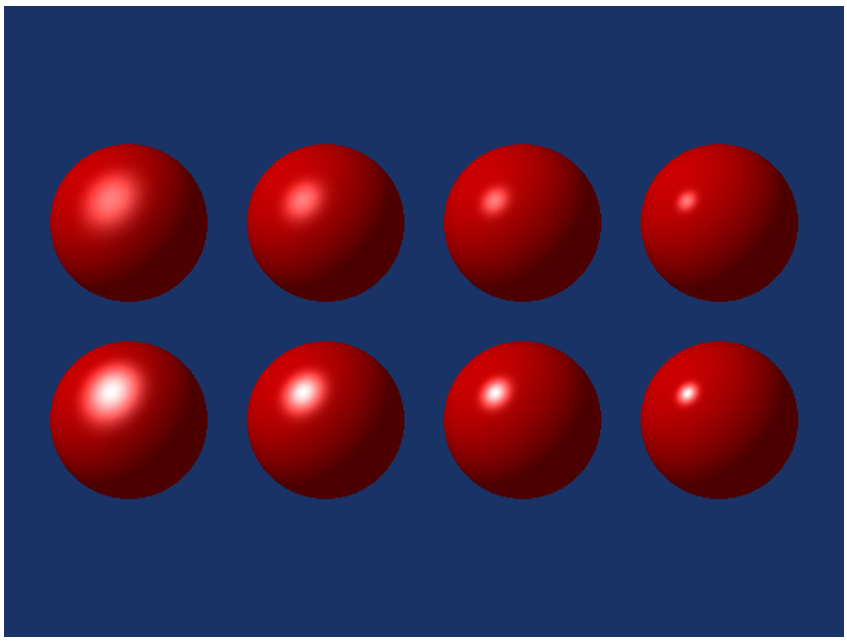


Figure 3.9: Effects of specular coefficients. Specular coefficients control the apparent "shininess" of objects. The top row has a specular intensity value of 0.5; the bottom row 1.0. Along the horizontal direction the specular power changes. The values (from left to right) are 5, 10, 20, and 40.

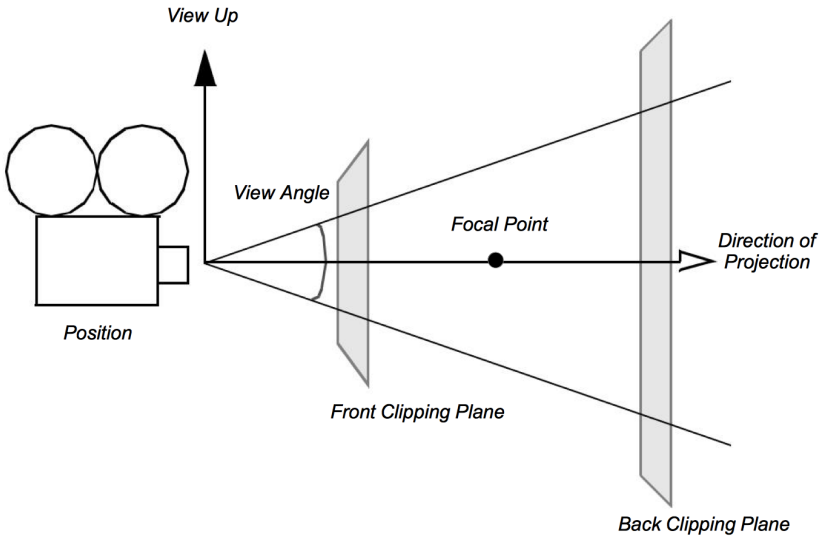


Figure 3.10: Camera attributes.

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 camera's clipping range. The location of the planes are measured from the camera's 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 camera's 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 camera's 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 camera's position constant and instead modify the focal point. Changing the yaw rotates the focal point about the view up centered at the camera's 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 camera's position. Dollying in and out moves the camera's 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 camera's view angle, so that more or less of the scene falls within the view frustum.

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 camera's position and other properties determine which rays of light get captured and projected. More specifically, only rays of light that intersect the camera's position, and are within its viewing frustum, will affect the resulting 2D image.

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,

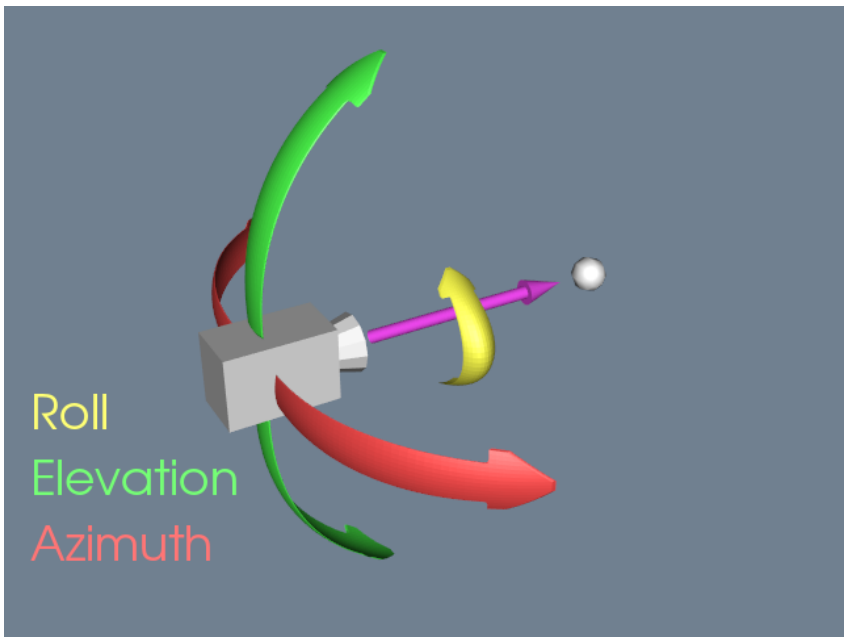


Figure 3.11: Camera movements around the focal point.

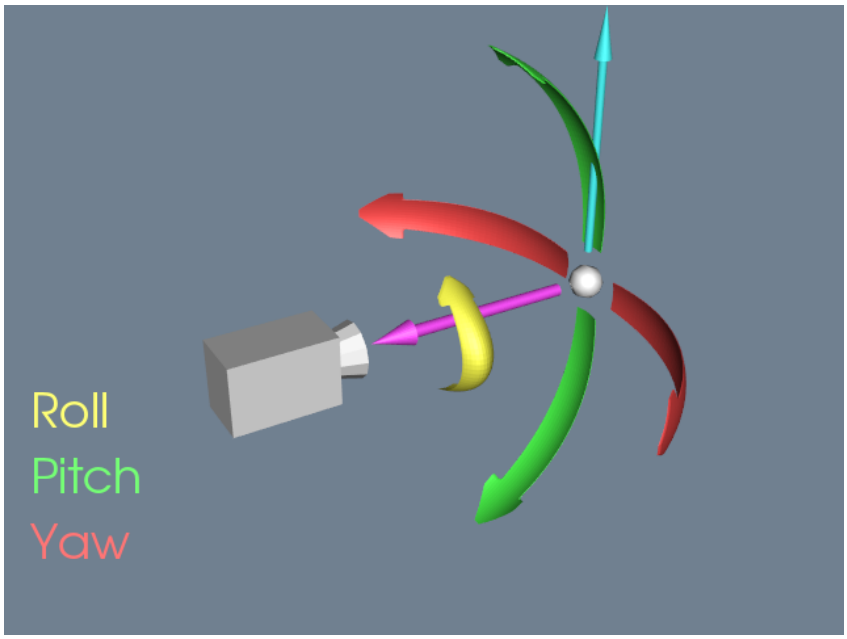


Figure 3.12: Camera movements around the camera position.

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 actor's responsibilities is to convert from the model's 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 camera's 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 window's 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.

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 50.

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:

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

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

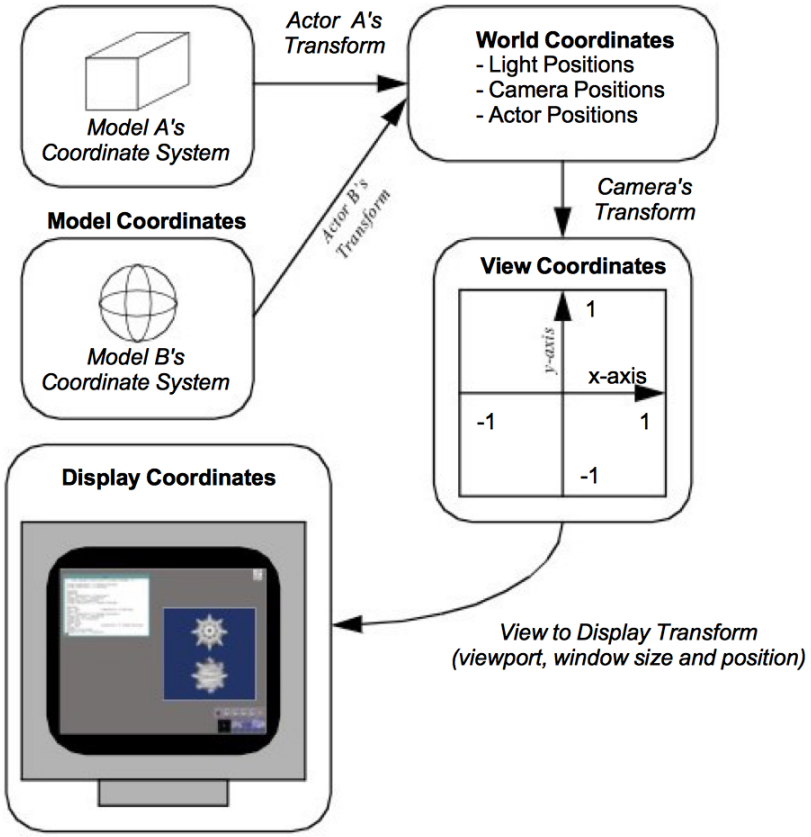


Figure 3.13: Modelling, world, view and display coordinate system.

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

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 object's 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).

3.9.2 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 actor's 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.

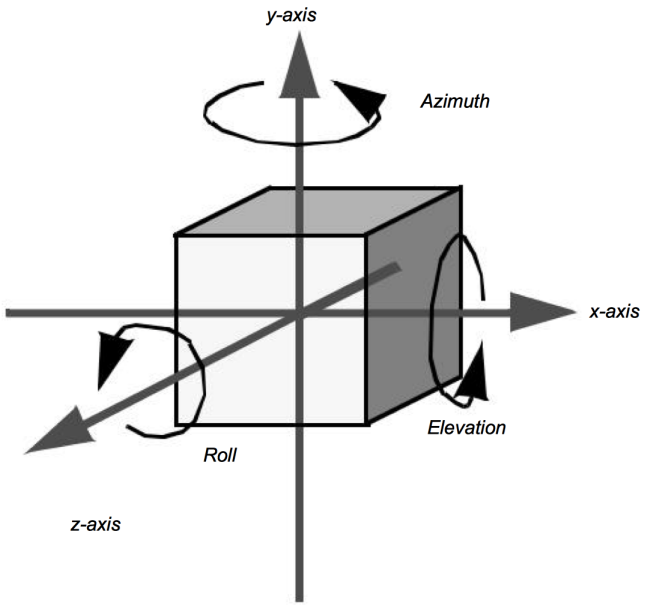


Figure 3.14: Actor coordinate system.

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.

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.

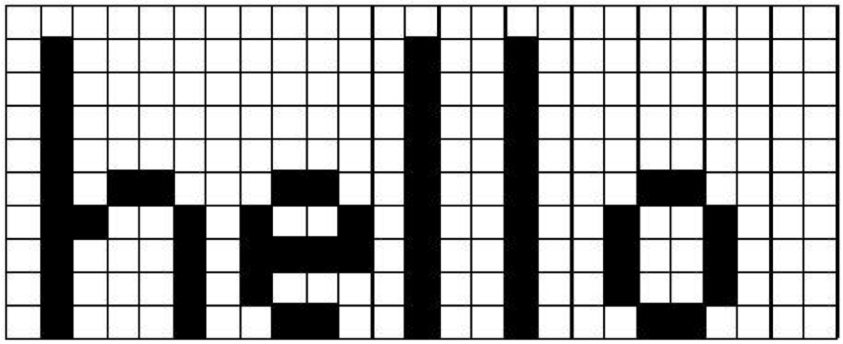


Figure 3.15: A pixel array for the word ”hello”.

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.

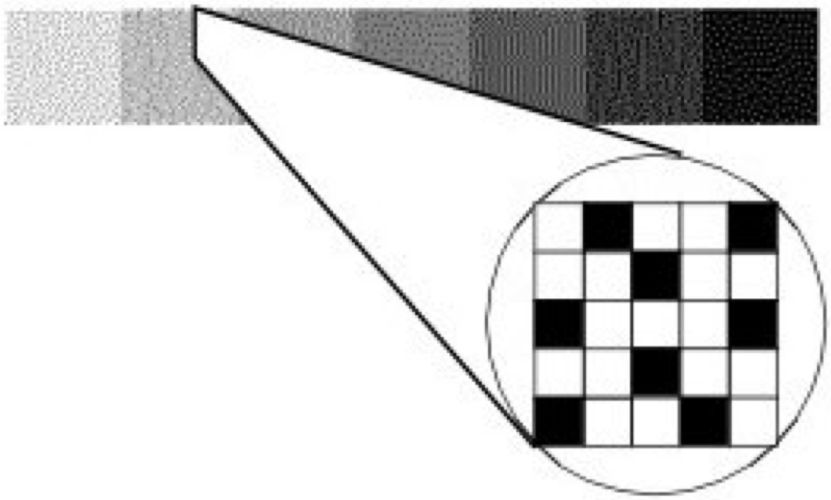


Figure 3.16: Black and white dithering.

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 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.

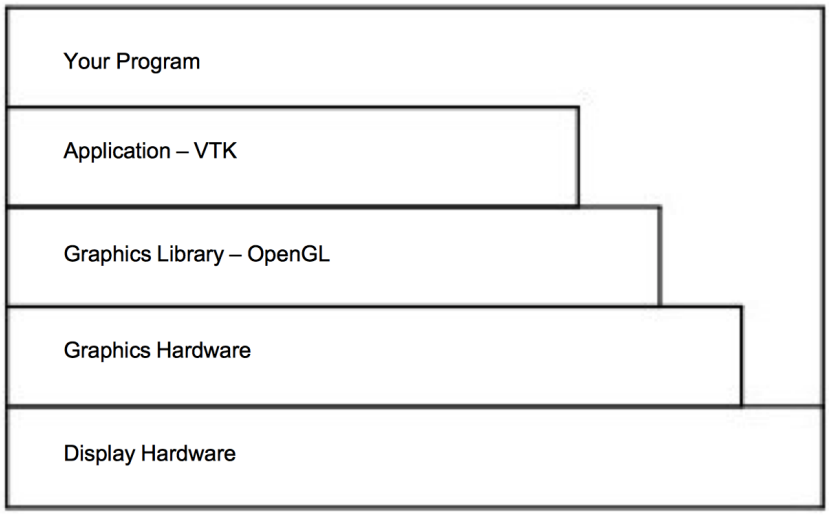


Figure 3.17: Typical graphics interface hierarchy.

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 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.

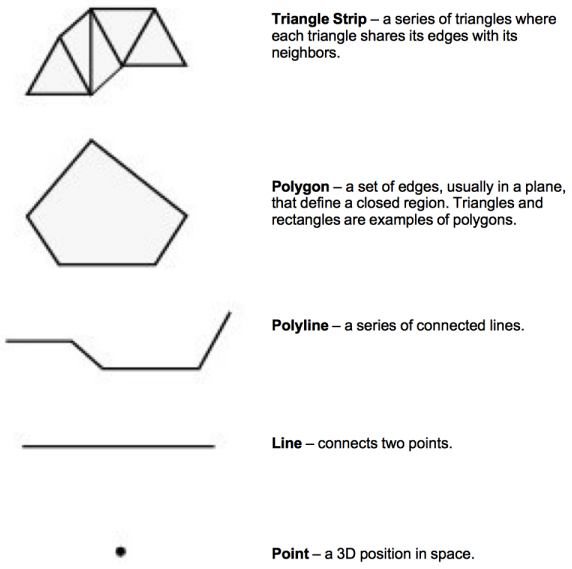


Figure 3.18: Graphics primitives.

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 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.

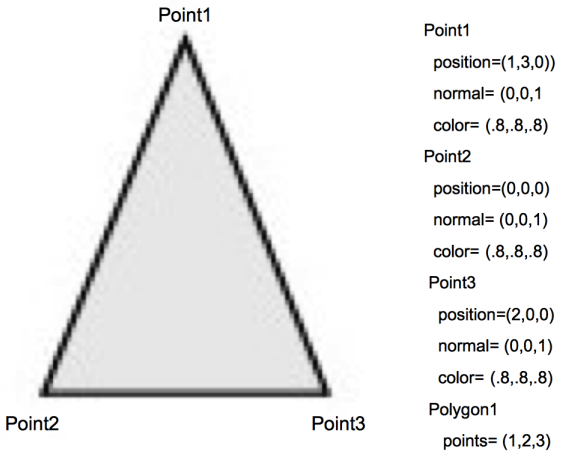


Figure 3.19: An example polygon.

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 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.

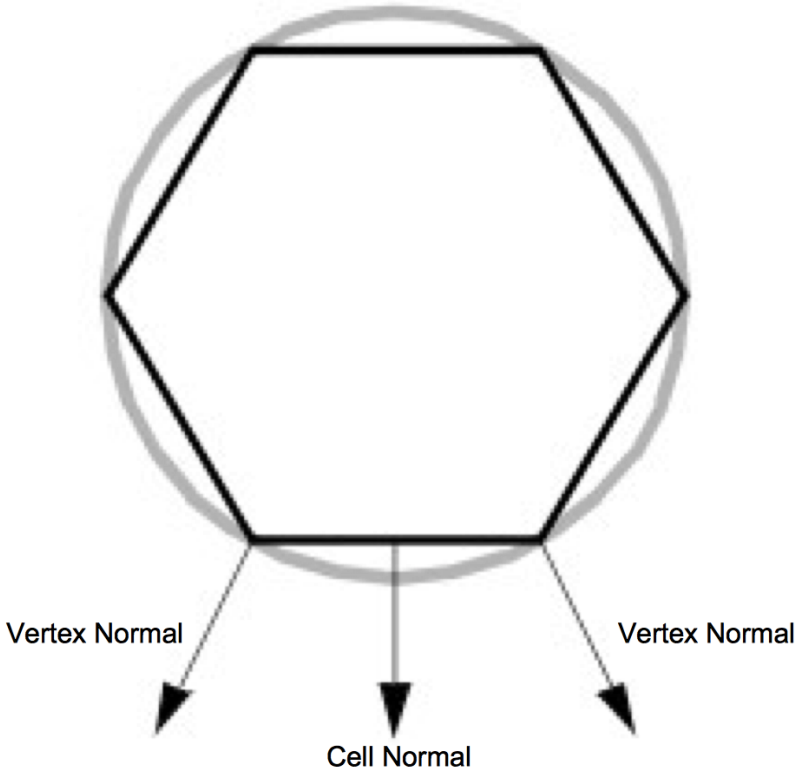


Figure 3.20: Vertex and polygon normals.

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 poly-gon 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

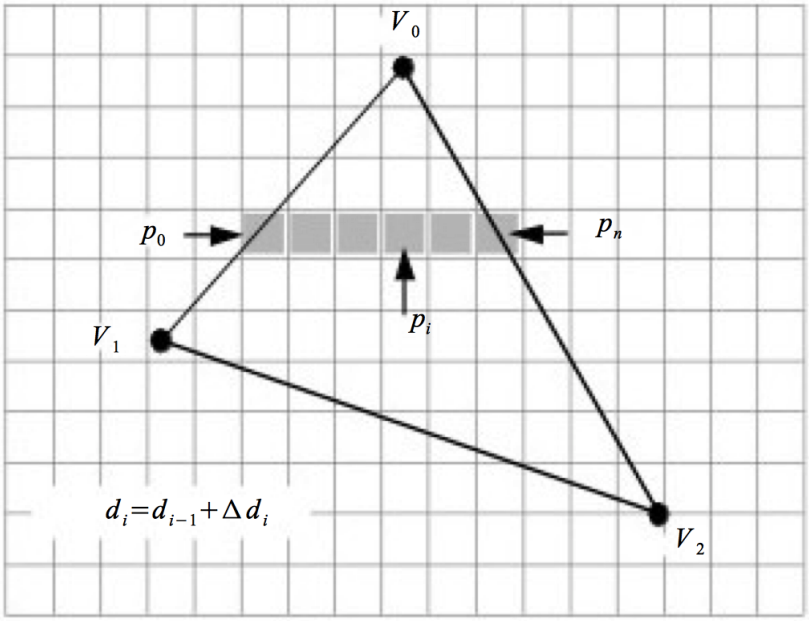


Figure 3.21: Rasterizing a convex polygon. Pixels are processed in horizontal spans (or scan-lines) in the image plane. Data values d_i at point p_i are interpolated along the edges and then along the scan-line using delta data values. Typical data values are RGB components of color.

flat and Gouraud shading are commonly used methods. The complexity of Phong shading has prevented it from being widely supported in hardware.

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 camera's view vector) and then render them in that order. This is called the painter's algorithm or painter's 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

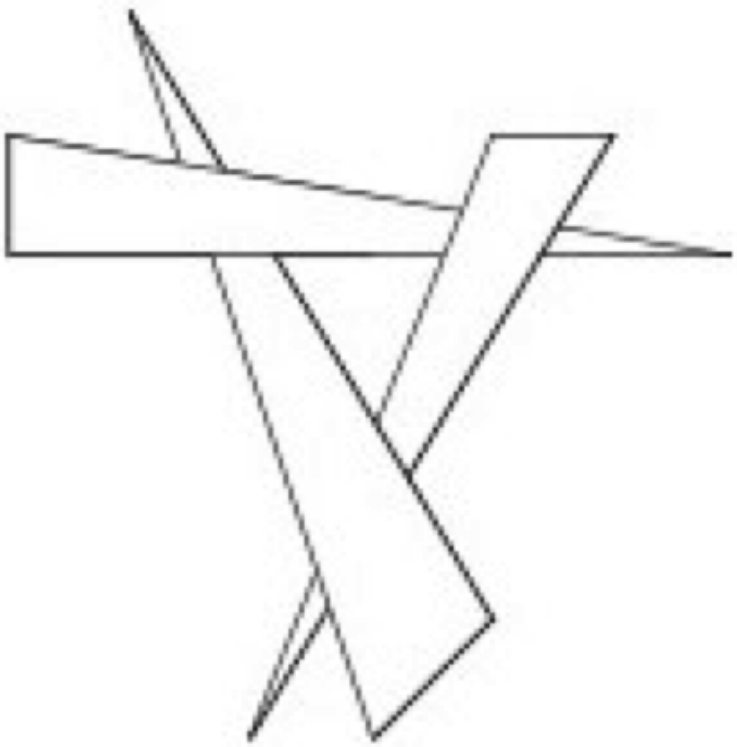


Figure 3.22: Problem with Painter's algorithm.

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 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

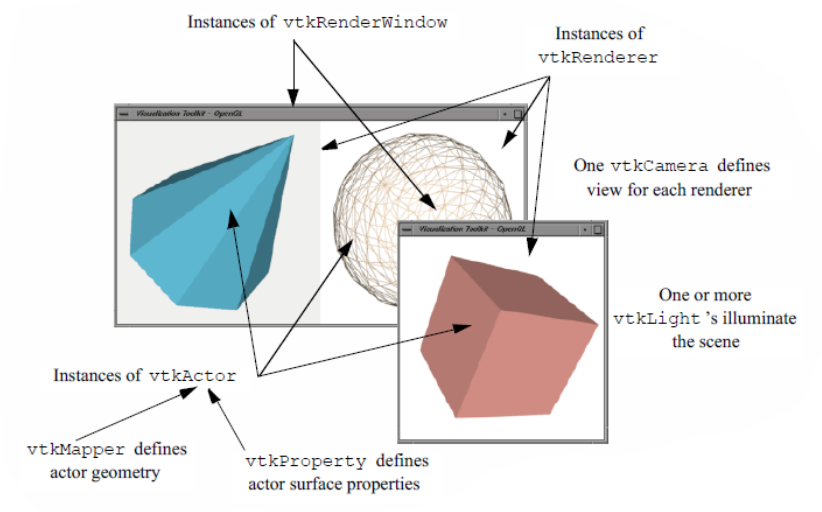


Figure 3.23: Illustrative diagram of graphics objects (Model.cxx) and (Model.py).

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.

- 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, 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 double 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.

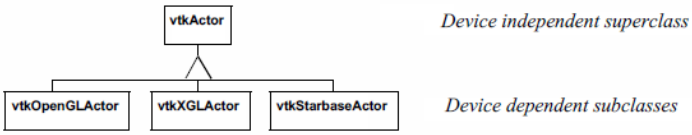
Finally, vtkMapper (and its subclasses) defines object geometry and, optionally, vertex colors. In addition, vtkMapper refers to a table of colors (i.e., vtkLookupTable) that are used to color the geometry. (We discuss mapping of data to colors in "Color Mapping" on page163.) We will examine the mapping 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.

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.



(a) Inheritance of device classes. (Note: in VTK 4.2 the Starbase and XGL graphics libraries are no longer supported.)

```
vtkActor *vtkActor::New()
{
  vtkObject* ret = vtkGraphicsFactory::CreateInstance("vtkActor");
  return (vtkActor*)ret;
}
```

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

```
  if (!strcmp("OpenGL",rl) || !strcmp("Win32OpenGL",rl) ||
 !strcmp("CarbonOpenGL",rl) || !strcmp("CocoaOpenGL",rl))
     {
     if(strcmp(vtkclassname, "vtkActor") == 0)
       {
       return vtkOpenGLActor::New();
       }
```

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

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

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 User's 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. 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
```

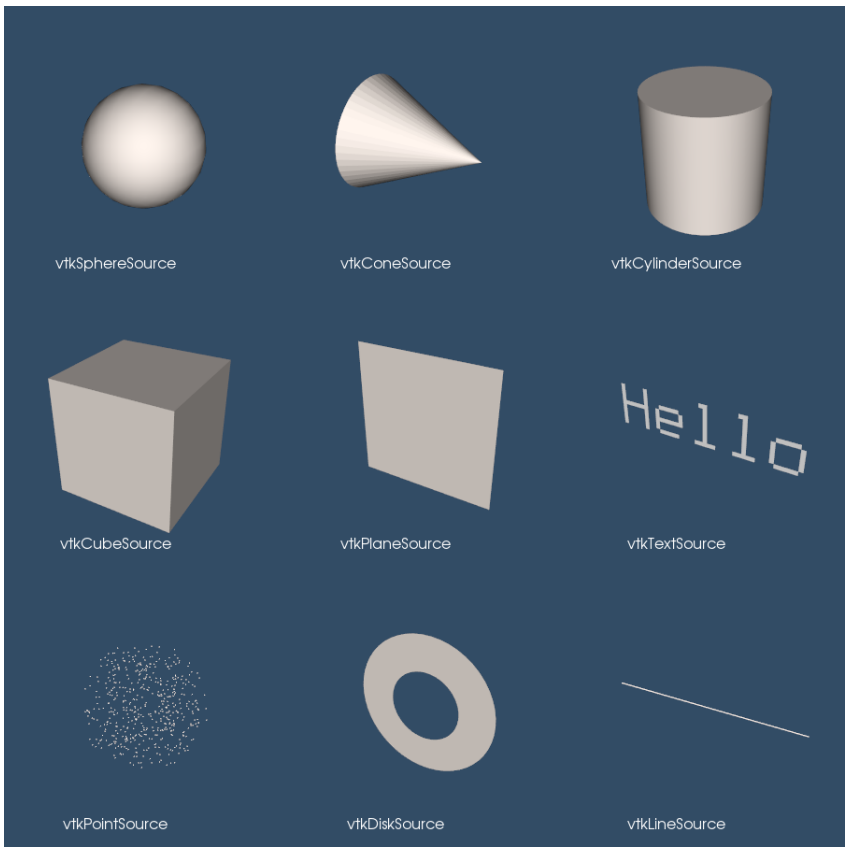


Figure 3.25: Examples of source objects that procedurally generate polygonal models. These nine images represent just some of the capability of VTK. From upper left in reading order: sphere, cone, cylinder, cube, plane, text, random point cloud, disk (with or without hole), and line source. Other polygonal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

```
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() ); vtkActor
27   *coneActor = vtkActor::New(); coneActor->SetMapper( coneMapper );
28
29   vtkRenderer *ren1= vtkRenderer::New();
30   ren1->AddActor( coneActor );
31   ren1->SetBackground( 0.1, 0.2, 0.4 );
32
33   vtkRenderWindow *renWin = vtkRenderWindow::New();
34   renWin->AddRenderer( ren1 ); renWin->SetSize( 300, 300 );
35
36   vtkMyCallback *mo1 = vtkMyCallback::New();
37   ren1->AddObserver(vtkCommand::StartEvent,mo1); mo1->Delete();
38
39   int i;
40   for (i = 0; i < 360; ++i)
41   {
42   //   render the image
43     renWin->Render();
44   // rotate the active camera by one degree
45     ren1->GetActiveCamera()->Azimuth( 1 );
46   }
47
48   cone->Delete();
49   coneMapper->Delete();
50   coneActor->Delete();
51   ren1->Delete();
52   renWin->Delete();
53   return 0;
54 }
```

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 63) 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 84).

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.)

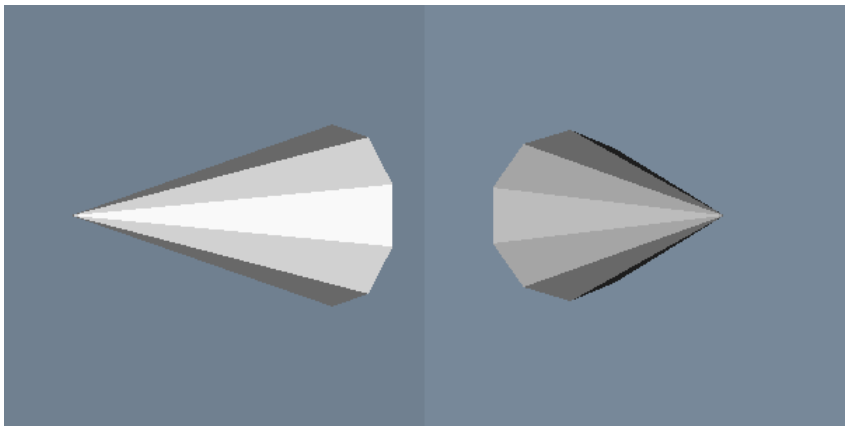


Figure 3.26: Two frames of output from Cone3.cxx. See Cone3.cxx or Cone3.py

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 ); renWin->AddRenderer( ren2 );
13 renWin->SetSize( 600, 300 );
14
15 ren1->GetActiveCamera()->Azimuth(90);
16
17 int i;
18 for (i = 0; i < 360; ++i)
19   {
20   // render the image renWin->Render();
21   // rotate the active camera by one degree
22     ren1->GetActiveCamera()->Azimuth( 1 );
23     ren2->GetActiveCamera()->Azimuth( 1 );
24   }
```

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.

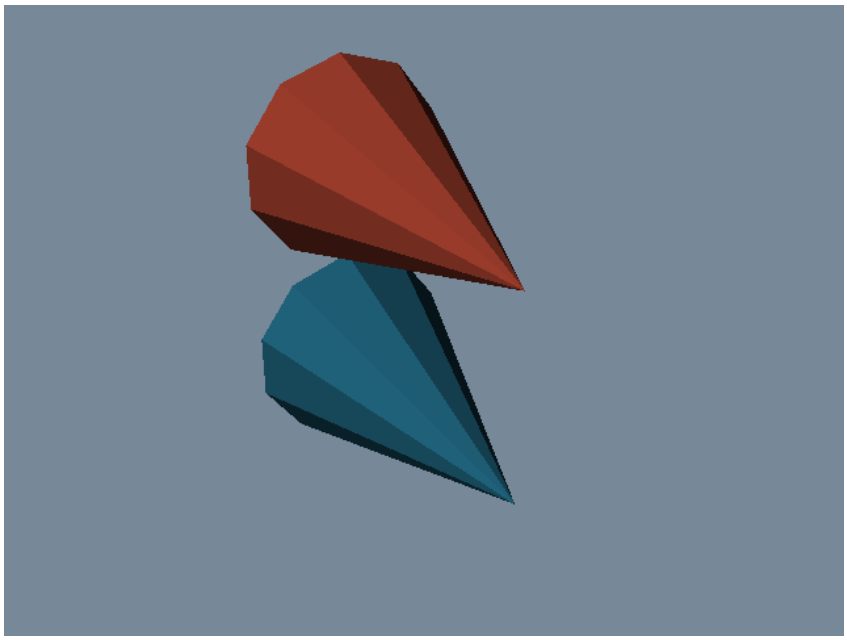


Figure 3.27: Modifying properties and transformation matrix. See Cone4.cxx or Cone4.py

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

```
1 vtkActor *coneActor = vtkActor::New();
2 coneActor->SetMapper(coneMapper);
3 coneActor->GetProperty()->SetColor(0.2, 0.63, 0.79);
4 coneActor->GetProperty()->SetDiffuse(0.7);
5 coneActor->GetProperty()->SetSpecular(0.4);
```

```
6  coneActor->GetProperty()->SetSpecularPower(20);
7
8  vtkProperty *property = vtkProperty::New();
9  property->SetColor(1.0, 0.3882, 0.2784);
10 property->SetDiffuse(0.7);
11 property->SetSpecular(0.4);
12 property->SetSpecularPower(20);
13
14 vtkActor *coneActor2 = vtkActor::New();
15 coneActor2->SetMapper(coneMapper);
16 coneActor2->GetProperty()->SetColor(0.2, 0.63, 0.79);
17 coneActor2->SetProperty(property); coneActor2->SetPosition(0, 2, 0);
18
19 vtkRenderer *ren1= vtkRenderer::New();
20 ren1->AddActor( coneActor );
21 ren1->AddActor( coneActor2 );
22 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 57). 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

```
1  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
2  iren->SetRenderWindow(renWin);
3
4  vtkInteractorStyleTrackballCamera *style =
5   vtkInteractorStyleTrackballCamera::New();
6
```

```
7 iren->SetInteractorStyle(style);
8 iren->Initialize();
9 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

```
1  package require vtk
2  package require vtkinteraction
3  vtkConeSource cone
4  cone SetHeight 3.0
5  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
```

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

```
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 .
```

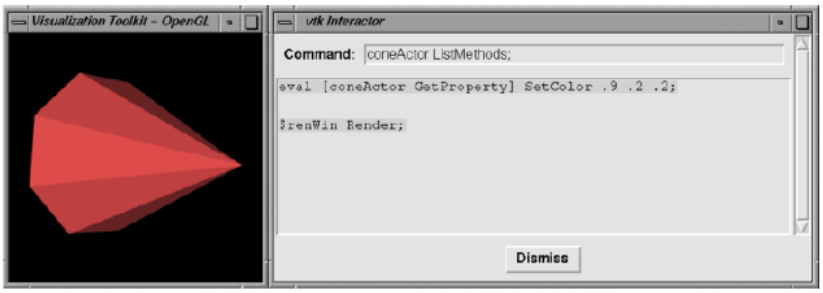


Figure 3.29: Using Tcl and Tk to build an interpreted application (Cone5.tcl).

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 **pageref????) 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.

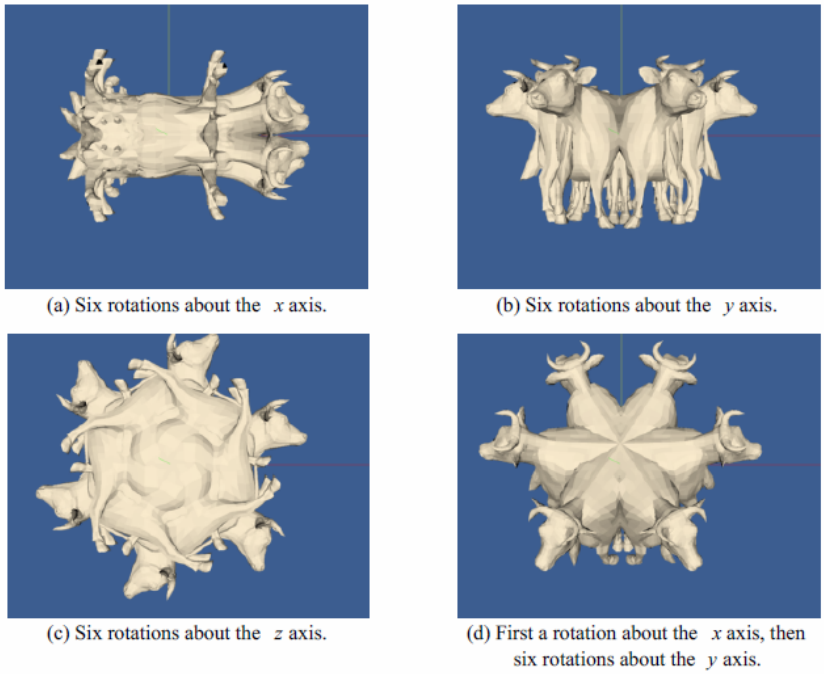
4. Undo the translation of step 1 and move the actor to its final location.

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 window's 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.

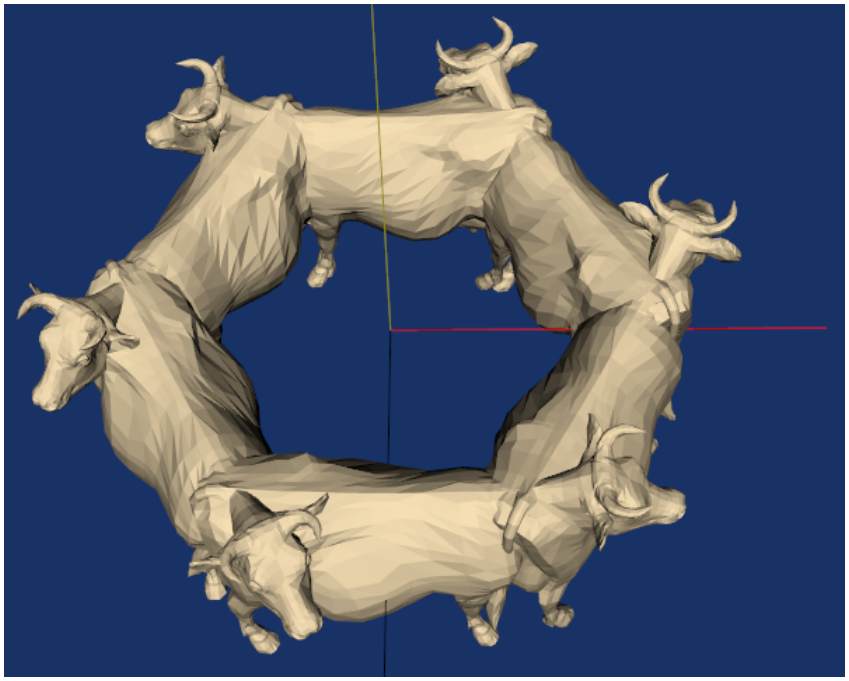
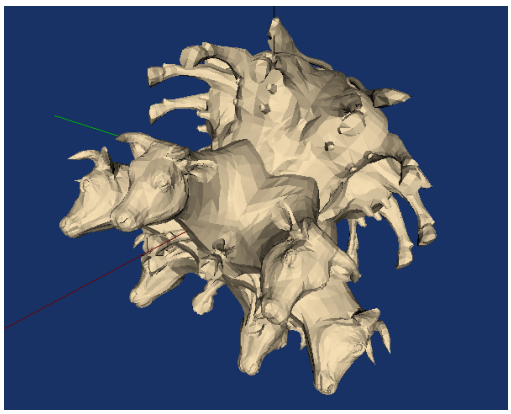


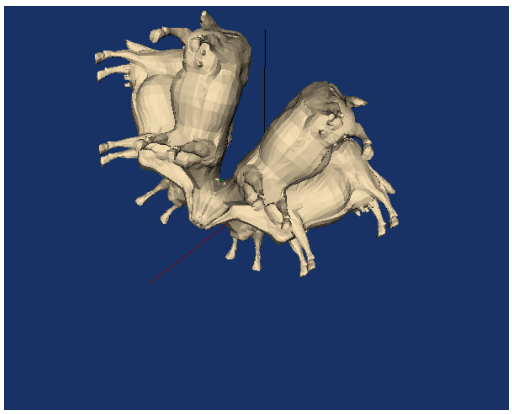
Figure 3.31: Modifying properties and transformation matrix. See WalkCow.cxx or WalkCow.py

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 65). Here is an example of how to create a simple assembly hierarchy (from



(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.

assembly.tcl).

Listing 3.6: Part of assembly.tcl

```
vtkSphereSource sphere
vtkPolyDataMapper sphereMapper
sphereMapper SetInputConnection [sphere GetOutputPort] vtkActor
sphereActor
sphereActor SetMapper sphereMapper
sphereActor SetOrigin 2 1 3
sphereActor RotateY 6
sphereActor SetPosition 2.25 0 0
[sphereActor GetProperty] SetColor 1 0 1

vtkCubeSource cube
vtkPolyDataMapper cubeMapper
cubeMapper SetInputConnection [cube GetOutputPort] vtkActor
cubeActor
cubeActor SetMapper cubeMapper
cubeActor SetPosition 0.0 .25 0
[cubeActor GetProperty] SetColor 0 0 1
vtkConeSource cone
vtkPolyDataMapper coneMapper
coneMapper SetInputConnection [cone GetOutputPort] vtkActor
coneActor
coneActor SetMapper coneMapper
coneActor SetPosition 0 0 .25
[coneActor GetProperty] SetColor 0 1 0
vtkCylinderSource cylinder
vtkPolyDataMapper cylinderMapper
cylinderMapper SetInputConnection [cylinder GetOutputPort]
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 }
```

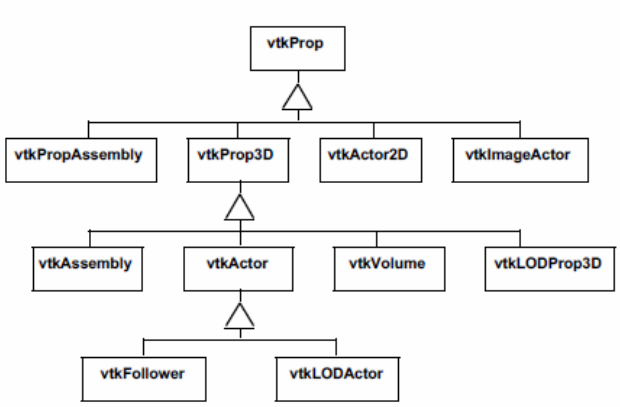


Figure 3.33: The vtkProp hierarchy. Props that can be transformed in 3D space are a subclass of vtkProp3D. Images can be drawn effectively with vtkImageActor. Overlay text and graphics use vtkActor2D. Hierarchical groups of vtkProps are gathered into a vtkPropAssembly. Volume rendering uses vtkVolume. 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 85.

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 84).

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.

[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 camera's 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 56 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

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.

5.0

Basic Data Representation

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

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.

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 62). 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 62) 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.

8.0

Advanced Data Representation

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. Figure8–1 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 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.

8.2.1 Picking

The Visualization Toolkit provides a variety of classes to perform actor (or vtkProp), point, cell, and world point picking (Figure8–38).

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

10.0

Image Processing

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

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

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

1.1 Visualization transforms numbers to images. 9
1.2 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. 13

3.1 Physical generation of an image. 28
3.2 Wavelength versus Intensity plot. 31
3.3 Relative absorbance of light by the three types of cones in the human retina. 32
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. 33
3.5 Local light source with a finite volume versus an infinite point light source. 34
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. 35
3.7 Diffuse lighting 35
3.8 Specular lighting. 36
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. 37
3.10 Camera attributes. 37
3.11 Camera movements around the focal point. 39
3.12 Camera movements around the camera position. 39
3.13 Modelling, world, view and display coordinate system. 41
3.14 Actor coordinate system. 44
3.15 A pixel array for the word "hello". 45
3.16 Black and white dithering. 45
3.17 Typical graphics interface hierarchy. 46
3.18 Graphics primitives. 47
3.19 An example polygon. 48
3.20 Vertex and polygon normals. 49
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. 50
3.22 Problem with Painter's algorithm. 51
3.23 Illustrative diagram of graphics objects (Model.cxx) and (Model.py). 52

3.24 Achieving device independence using (a) inheritance and object factories (b) and (c). 55
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 58
3.26 Two frames of output from Cone3.cxx. See Cone3.cxx or Cone3.py 60
3.27 Modifying properties and transformation matrix. See Cone4.cxx or Cone4.py 61
3.28 In VTK the C++ library is automatically wrapped with the interpreted languages Tcl, Python, and Java. 63
3.29 Using Tcl and Tk to build an interpreted application (Cone5.tcl). 64
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. 66
3.31 Modifying properties and transformation matrix. See WalkCow.cxx or WalkCow.py 68
3.32 The cow rotating about a vector passing through her nose. 69
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. 70

List of Tables

3.1 Common colors in RGB and HSV space 32

List of Equations

3.1 Ambient Lighting 34
3.2 Diffuse Lighting Contribution 34

3.3 Specular Lighting . . . 35
3.4 Ambient, Diffuse and Specular Lighting . . . 36
3.5 Cartesian to Homogenous Conversion . . . 40
3.6 Translation . . . 41
3.8 Homogenous to Cartesian . . . 42
3.9 Scaling . . . 42
3.10 $X-$axis rotation . . . 42
3.11 $Y-$axis rotation . . . 42
3.12 $Z-$axis rotation . . . 42
3.13 Transform from $x-y-z$ to $x'-y'-z'$ . . . 42
3.14 vtkActor transforms . . . 65

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

API An acronym for application programmer's interface.. 3