Computer GRAPHICS

Samit Bhattacharya

Assistant Professor
Department of Computer Science Engineering
IIT Guwahati

OXFORD
UNIVERSITY PRESS

Dedicated to
My family

Preface

The term *computer graphics* roughly refers to the field of study that deals with the display mechanism (the hardware and software) of a computer. In the early days, for most of us, a computer meant what we got to *see* on the monitor. Then came the *laptops*, in which the display and the CPU tower (along with the peripheral keyboard unit) were combined into a single and compact unit for easy portability. The cathode ray tube (CRT) displays were replaced by the liquid crystal display (LCD) technology. However, the idea of computer was still restricted to the display screen for a majority of the users.

For the younger generation, the personal computer (PC) is no longer a 'computer'. It is replaced by a plethora of devices, although the laptops have managed to retain their charm and appeal due to portability. These devices come in various shapes and sizes with varying degrees of functionality. The most popular of these is the ubiquitous *smartphone*. Although much smaller in size compared to a PC, smartphones are similar to very powerful PCs of the yesteryears; with powerful multicore processing units, high-resolution displays, and large memory. Then we have the tablets (or tabs), which are slightly larger in size (although still much smaller than a PC), and the *fablets*, having features of both a phone and a tab. Such devices also include wearable computers such as the smart watch or the Google glass. Even the televisions nowadays have many computing elements that has led to the concept of *smart TVs*. This is made possible with a rapid change in technology, including display technology. Instead of the CRT, we now have devices that use LCD, plasma panel, light-emitting diode (LED), organic light-emitting diode (OLED), thin-film transistor (TFT), and so on, for the design of display units.

However, regardless of the current state-of-the-art technology in computing is, the idea of a 'computer' is shaped primarily by what we get to 'see' on the display unit of a computing system. Since perception matters the most in the popularity of any system, it is important for us to know the components of a computing system that give rise to this perception—the display hardware and the associated software and algorithms. Therefore, it is very important to learn the various aspects of computer graphics to understand the driving force behind the massive change in consumer electronics that is sweeping the world at present.

ABOUT THE BOOK

Computer Graphics is a textbook aimed at the undergraduate students of computer science engineering, information technology, and computer applications. It seeks to provide a thorough understanding of the core concepts of computer graphics in a semester-level course. The contents of this book are designed for a one-semester course on the subject keeping in mind the difficulty faced by a first-time learner of the subject. The book aims to help students in self-learning through illustrative diagrams, examples, and practice exercises.

The chapters are organized following the three-dimensional (3D) graphics pipeline stages. In this method, students learn graphics starting from 3D concepts, making it easier to master

the concepts. This is an accepted way of approaching the subject although it is in contrast with the common methodology followed in the existing literature, in which the students learn the two-dimensional (2D) concepts first and then are introduced to the 3D concepts.

KEY FEATURES

- Follows the 3D graphics pipeline-based discussion
- Provides a separate chapter on the applications of computer graphics
- Supports concepts through numerous illustrative diagrams and images
- Provides a large number of worked-out examples, algorithms, and pseudocodes at appropriate places
- Contains chapter-end summary and key terms that enable quick recapitulation

ONLINE RESOURCES

The following resources are available to support the faculty using this book:

- Solutions Manual
- PowerPoint Presentations

ORGANIZATION OF THE BOOK

The chapters in this book are organized following the 3D graphics pipeline stages. In the first chapter, the 3D graphics pipeline is explained. Subsequent chapters deal with the stages of the pipeline, namely the object representation techniques, modeling/geometric transformations, illumination and lighting model, 3D viewing, clipping, hidden surface removal, and scan conversion. The last few chapters of the book cover advanced topics such as graphics hardware, multimedia, hypermedia, and computer animation, which are intended to widen the knowledge base of the reader. The contents of these chapters are briefly described here.

Chapter 1 introduces the field to the reader. It starts with the general idea of computer graphics and the application areas. This is followed by a brief description on the historical development of the field. The core issues in computer graphics are presented next. In order to illustrate the core issues and challenges, the basic graphics hardware is presented along with the detailed description of the cathode ray tube (CRT) technology. The pipeline stages are introduced next, followed by a brief description of each stage.

Chapter 2 deals with the first stage of the graphics pipeline, namely the object representation techniques. The widely used representation techniques are reviewed at the beginning of the chapter, with illustrations. This is followed by a brief description of the mesh representation, quadric surface representation, and blobby object representation, all part of the boundary representation techniques. Splines, another boundary representation technique, are discussed in detail in the subsequent part of the chapter. The discussion includes spline types and sub-types, spline surface generation, and implementation of splines. The space partition-based representations that include the concept of voxels, the octree, the quadtree, the BSP method, and constructive solid geometry (CSG) are described next. Advanced representation

techniques such as fractals, particle systems, skeleton model, and the scene graphs are introduced in this chapter. The chapter ends with a comparative discussion among the various methods of representations.

Chapter 3 describes the second stage of the graphics pipeline—modeling transformation. The core idea is introduced at the beginning. This is followed by a description of the basic 2D modeling transformations, namely translation, rotation, scaling, and shearing. The idea of matrix representation of transformations and the homogeneous coordinate system are introduced next. The composition of transformation as matrix multiplication is described subsequently, followed by the description of 3D transformations. Illustrative examples are used throughout the chapter to explain the transformations.

Chapter 4 presents the third and important stage of the graphics pipeline—the coloring of points on the screen. It starts with the basic idea behind the idea of color computation, followed by the detailed description of a simple lighting model to *compute* color at screen pixels. The components of the model, including the calculation of ambient light, diffuse light, and specular reflection, along with the procedure to compute the attenuation affects (both radial and angular) are explained in the model description. The three shading models—flat shading, Gouraud shading, and Phong shading—are discussed next. The mapping of the computed color to the device-supported intensity levels including the related concepts of Gamma correction, halftoning, and dithering are described in detail in the chapter. Numerical examples are presented throughout the chapter to illustrate the computations.

Chapter 5 presents topics that pertain to the idea of color, its representation, and realistic modeling. The chapter starts with a description of the physiology of our visual processing system and the idea of color perception. This is followed by the various models used to represent colors in a computer that include the RGB, the XYZ, the CMY, and the HSV models. The chapter ends with a description of the texture synthesis techniques that make object surfaces appear realistic. All three texture synthesis techniques—projected texture, texture mapping, and solid texture-are covered along with illustrative examples.

Chapter 6 deals with the fourth stage of the graphics pipeline—the viewing transformation. Viewing transformation is done through a series of steps and transformations, known as the viewing pipeline. The stages of the viewing pipeline is introduced at the beginning of the chapter. This is followed by the description of setting up the viewing coordinate system and the view coordinate transformation, in that order. The projection transformation, that is, the transformation from 3D viewing coordinate system to 2D view plane, is described next. The concepts of the view volumes and the canonical view volumes are also introduced with illustration. The chapter also covers the window-to viewport-transformation, the last stage of viewing pipeline.

Before projection to view plane, unwanted portions of a 3D scene are removed or clipped-out. The process is called clipping, which is discussed in ***Chapter 7***. The chapter first discusses the clipping in two dimensions. Four 2D clipping algorithms are described with illustrative examples—the Cohen–Sutherland algorithm, the Liang–Barsky algorithm, the Sutherland–Hodgeman algorithm, and the Weiler–Atherton algorithm—covering line and polygon clipping in 2D. Extensions of these algorithms for clipping in three dimensions are presented next, with illustrative examples.

Chapter 8 presents another pre-processing step in the graphics pipeline—hidden surface removal. This chapter deals with the procedures used to remove portions of a scene that are hidden from the viewer with respect to the viewing position. The chapter begins with the basic principles, known as the *coherence* properties. The simplest approach, namely the back face elimination method is described next, followed by the more sophisticated approaches that include the Z-buffer method, the A-buffer algorithm, the depth-sorting method or the painter's algorithm, the Warnock's algorithm, and the octree method. All these algorithms are presented with illustrative examples for easy understanding.

The final stage of the graphics pipeline, namely the rendering or scan conversion, is described in ***Chapter 9***. In this chapter, basic line scan conversion algorithms are introduced. These include the DDA algorithm and the Bresenham's algorithm. The line scan conversion is followed by the circle scan conversion and the midpoint algorithm. The fill area scan conversion algorithms are presented next including the seed fill, the flood fill, and the scan line polygon fill algorithm. The basic idea behind the rendering of characters is dealt with separately in the chapter. An important consideration in rendering is anti-aliasing. The core concepts of anti-aliasing and the major approaches are outlined and discussed in this chapter. The approaches include the area filtering and super sampling techniques.

No discussion on graphics stages shall be complete without a description of the fundamentals of graphics hardware and software, which is the subject matter of ***Chapter 10***. The chapter starts with an introduction to a generic computer graphics systems along with its components. The various input and output devices that are the inseparable parts of an interactive graphics system are described next. These include the current state-of-the-art developments in the display technology; the hardcopy output systems such as the printer and plotters; and the input devices including the mouse, keyboard, joystick, trackball, touch input, and data gloves. The basic idea of a graphics processing unit (GPU) and GPU programming are also covered in this chapter. The chapter includes a brief description of graphics software along with an introduction to the OpenGL, an open source library for graphics programming.

The last two chapters (Chapters 11 and 12) contain topics that are applications of the computer graphics fundamentals. In ***Chapter 11***, computer animation techniques are introduced. It starts with a description of the principles that drive computer animation. The principles are followed by brief descriptions of the major animation techniques, namely keyframing, motion capture, physically based methods, and procedural animation techniques. The concept of multimedia is introduced in ***Chapter 12***. The chapter contains a discussion on the related concept of hypermedia, the methods for multimedia content authoring, and the components of multimedia including the fundamentals of digital audio and video. An important component of multimedia is the data compression techniques. The chapter introduces the core concepts and various techniques for video and audio data compression including the JPEG, the H.261, and the MPEG standards.

The book has three appendices to supplement the core materials discussed in the chapter.

Appendix A contains the basic mathematical background required to understand the topics covered in this book.

Appendix B contains an introductory discussion on the fractals, an advanced object representation technique (briefly mentioned in Chapter 2).

The fundamentals of the ray tracing method are covered in ***Appendix C***. The method is useful for photorealistic image synthesis and employs advanced rendering techniques.

ACKNOWLEDGEMENTS

Writing a book, any book, be it text or non-text, is always difficult. It is even more so for a first-time writer like me. I never realized before this book, the amount of patience and perseverance involved in book writing. The book probably would not have seen the light of the day without the constant and active support from various people, including my family and the staff of Oxford University Press, India.

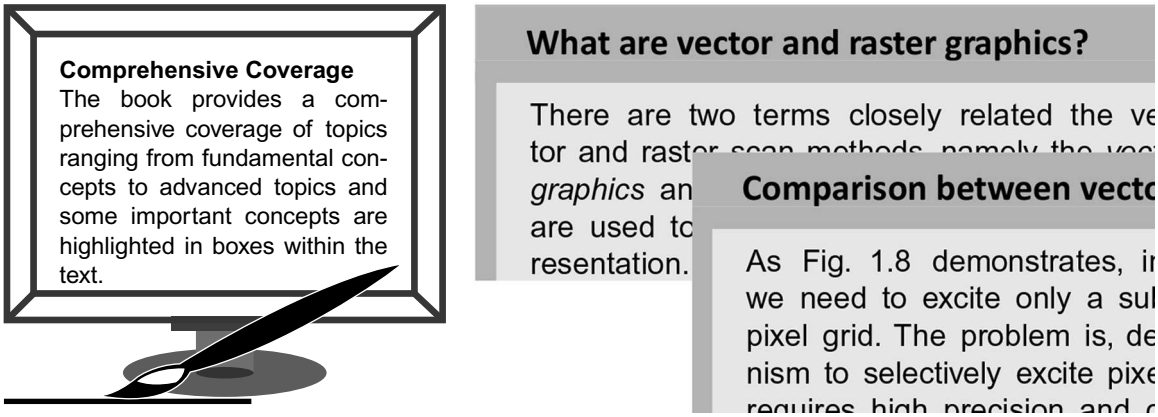
I express my gratitude to my wife, Patralekha Bhattacharya for her patience, encouragement, and active support during the writing of the book. My son, Jitamitra Bhattacharya also helped a lot in his own way through his little naughty activities, which helped make the writing enjoyable. I am also indebted to my parents, Mr Ashwini Bhattacharya and Mrs Gita Bhattacharya, my brother Mr Amit Bhattacharya, my parents-in-law Mr Golok Mukherjee and Mrs Shyamali Mukherjee, and brother-in-law Mr Anik Mukherjee for their constant support and encouragement. Without their help, this book wouldn't have been possible.

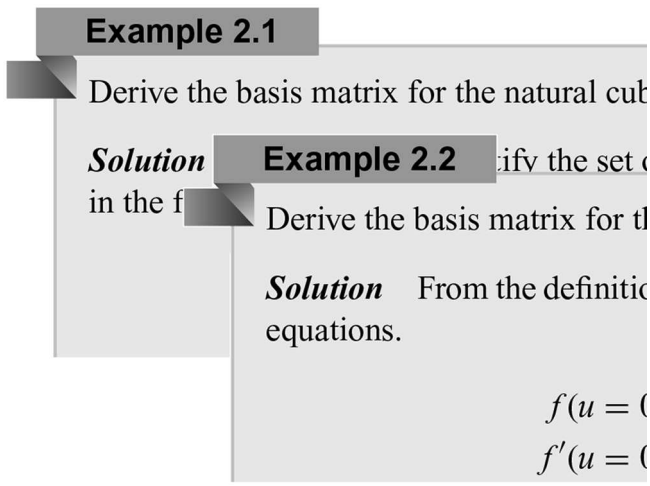
During the process of writing this book, I felt like giving up many times owing to the other academic and professional commitments. However, I could not do so due to the persistent and active support of the editorial team at the Oxford University Press India. I would also like to thank all the editors at the press, involved with this book, for their constant help and guidance during the manuscript preparation.

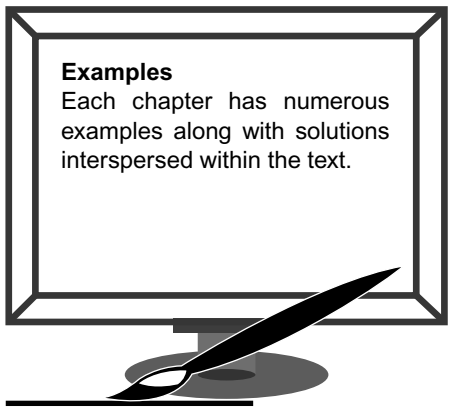
Finally, I thank all my friends, colleagues, students, and well-wishers who directly or indirectly helped me in making this book a reality.

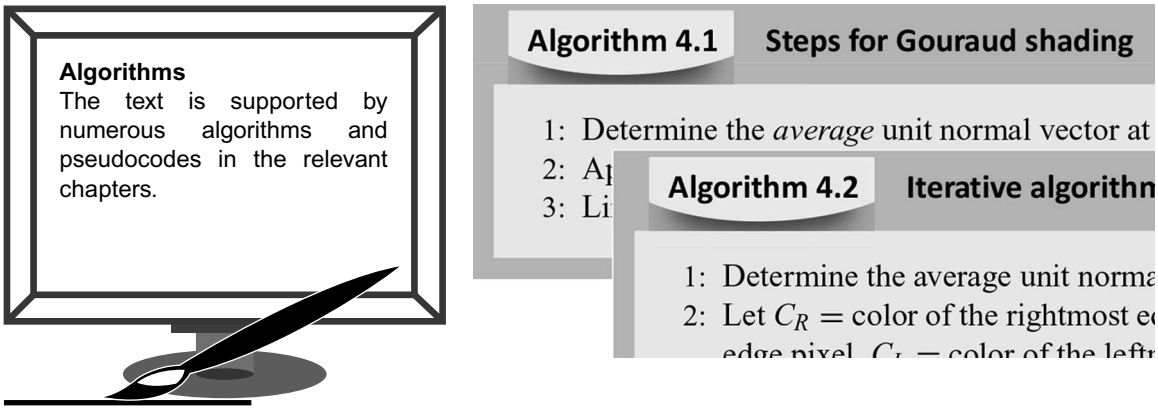
Samit Bhattacharya

Features of


Comprehensive Coverage
The book provides a comprehensive coverage of topics ranging from fundamental concepts to advanced topics and some important concepts are highlighted in boxes within the text.
What are vector and raster graphics?
There are two terms closely related the ve
tor and raster scan methods, namely the vec
graphics an
are used to
resentation.
Comparison between vecto
As Fig. 1.8 demonstrates, i
we need to excite only a sul
pixel grid. The problem is, de
nism to selectively excite pixe
requires high precision and


Example 2.1
Derive the basis matrix for the natural cub
Solution
Example 2.2
ify the set
in the f
Derive the basis matrix for th
Solution From the definitio
equations.
f(u =
f′(u =


Examples
Each chapter has numerous examples along with solutions interspersed within the text.


Algorithms
The text is supported by numerous algorithms and pseudocodes in the relevant chapters.
Algorithm 4.1 Steps for Gouraud shading
1: Determine the average unit normal vector at
2: Ap
3: Li
Algorithm 4.2 Iterative algorithm
1: Determine the average unit norma
2: Let C_R = color of the rightmost e
edge pixel, C_L = color of the left

the Book

SUMMARY

In this chapter, we have discussed the fundam
brief discussion on the physiology of vision. V
are primarily responsible for our perception c
of color. We also learnt that the existence of
to the generation of a color, makes it possible
the exact natural process.

Summary
The key concepts discussed in the chapter are highlighted at the end of each chapter to enable quick revision.

Key Terms
All chapters provide a list of key terms along with their definitions for a quick recapitulation of important terms discussed.

KEY TERMS

Composite transformation – composition (by matrix multiplic
transformations to obtain a new transformation
Differential scaling – when the amounts of change to an obj
(X, Y, and Z) are not the same
Homogeneous coordinate system – an abstract mathema
n-dimensional point with a $(n + 1)$ vector
Local/Object coordinates – the Cartesian coordinate referenc
ual objects
Modeling transformation – transforming objects from local
through some transformation operation

EXERCISES

5.1 Explain the process by which we sense colors.
5.2 It is not possible to exactly mimic the lighting process that computer graphics work?
5.3 What is the basis for developing models with three primar
5.4 Discuss the limitation of the RGB model that is overcome
5.5 Briefly discuss the relationship between the RGB and the model useful?
5.6 Explain the significance of using the HSV model instead c
5.7 Mention the three broad texture synthesis techniques.

Review Questions
Concept-review questions and numerical problems are provided at the end of every chapter to test the readers' understanding of the concepts learnt.

Companion Online Resources for Instructors

Visit www.oupinheonline.com to access both teaching and learning solutions online.
The following resources are available to support the faculty using this book

- Solutions manual
- PowerPoint slides

Steps to Register

Step 1: Getting Started

- Go to www.oupinheonline.com

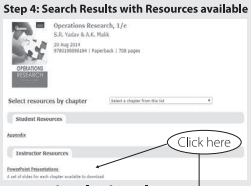
Step 2: Browse quickly by:

- BASIC SEARCH
  - → Author
  - → Title
  - → ISBN
- ADVANCED SEARCH
  - ▷ Subjects
  - ▷ Recent titles

Step 3: Select Title

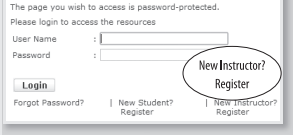
Select title for which you are looking for resources.

Step 4: Search Results with Resources available



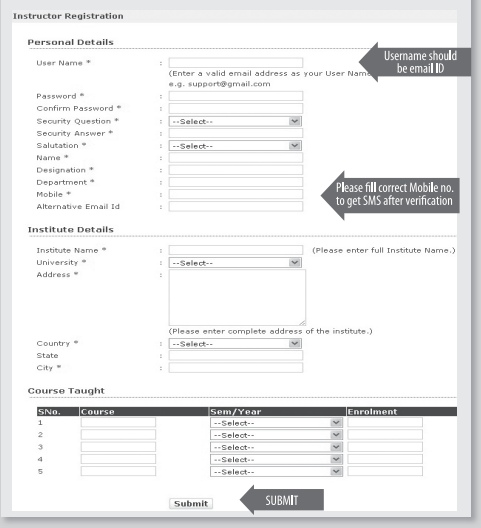
Step 5: To download Results

Step 6: Login to download



Step 7: Registration Form

Please fill correct details and *marked fields are mandatory



Step 8: Message after completing the registration form

Thank you for registering with us. We shall revert to you within 48 hours after verifying the details provided by you. Once validated please login using your username and the password and access the resources.

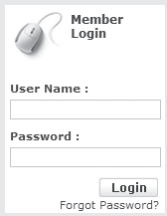
Step 9: Verification

You will receive a confirmation on your mobile & email ID.

For any further query please write to us at HEMarketing.in@oup.com with your mobile number

Step 10: Visit us again after validation

- Go to www.oupinheonline.com
- Login from Member Login



Step 11: My Subscriptions

You can view Subscriptions in your account

- Click on the title
- Select Chapter or "Select All"
- Click on "Download All"
- Click on "I Accept"
- A zip file will be downloaded on your system. You may use this along with the textbook.

Brief Contents

Detailed Contents

CHAPTER

1 Overview of Computer Graphics

Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the field of computer graphics and its application areas
- Trace the historical development of the field
- Understand the various components of a computer graphics system
- Have a basic understanding of the display hardware in terms of the cathode ray tube display technology
- Identify the stages of the image synthesis process

INTRODUCTION

With a computer, we can and usually do lot of things. We create documents and presentations. For example, consider the screenshot in Fig. 1.1 that was taken during the preparation of this book with MS Word^{TM}.

Notice the components present in the image. The primary components, of course, are the (alphanumeric) characters. These characters were entered using a keyboard. While in a document, the alphanumeric characters are the most important, there are other equally important components that are part of any word processing software. In this figure, these components are the *menu options* and the editing tool *icons* on top. Some of these options are shown as text while the others are shown as images (icons). Thus, we see a mix of characters and images that constitute the *interface* of a typical word processing system.

Next, consider Fig. 1.2, which is an interface of a Computer-aided Design (CAD) tool. It shows the design of some machinery parts on a computer screen, along with some control buttons on the right-hand side. The part itself is constructed from individual components, with specified properties (dimension, etc.). An engineer can specify the properties of those individual components and try to assemble them *virtually* on the computer screen, to check if there is any problem in the specifications. This saves time, effort, and cost, as the engineer does not need to actually develop a physical prototype and perform the specification checks. This is the advantage of using a CAD tool.

Figure 1.3 shows two instances of *visualization*, another useful activity done with computers. Figure 1.3(a) shows the visualization of a DNA molecule. It shows that, with the

2 Computer Graphics

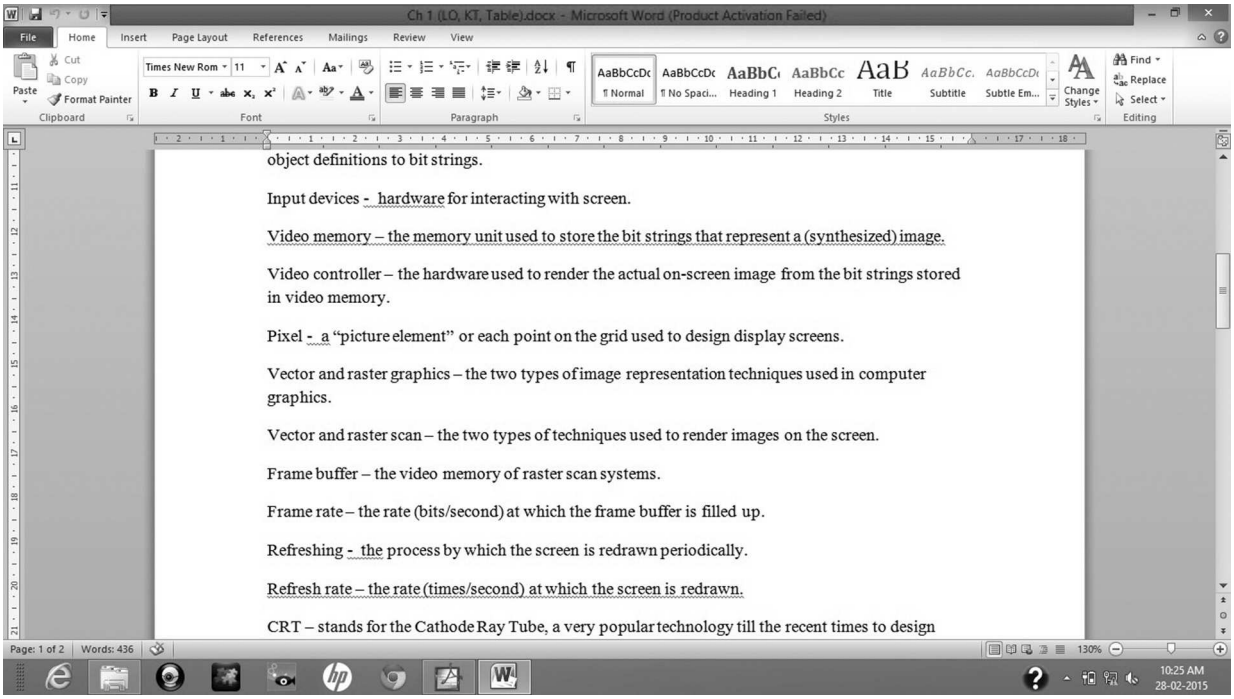


Fig. 1.1 Screen capture of a *page* during document preparation using MS Word

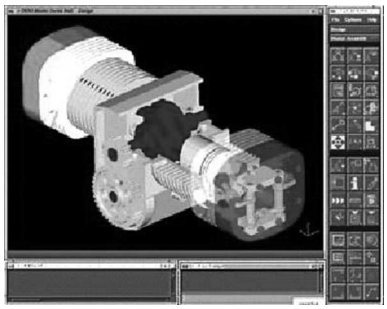
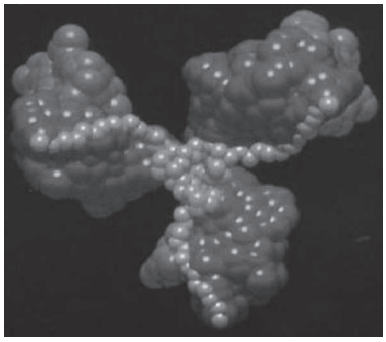


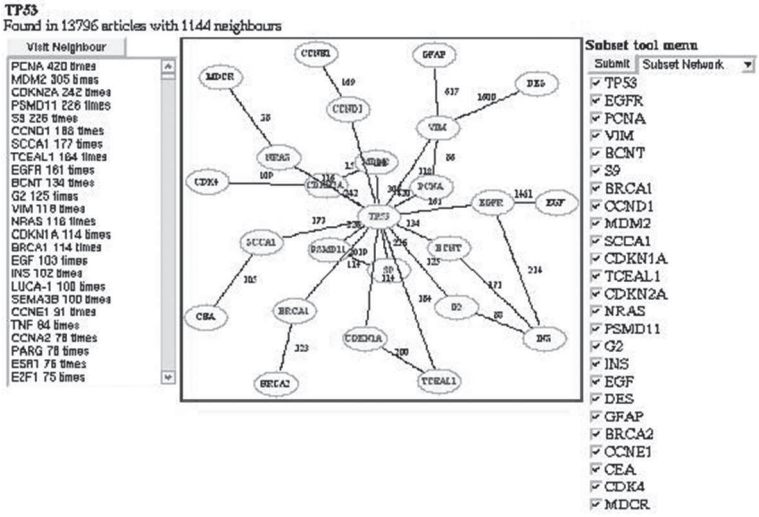
Fig. 1.2 A CAD system interface—the right-hand side contains buttons to perform various engineering tasks

aid of computers, we could *see* something that is not possible with the naked eye. Such a type of visualization is called *scientific visualization*, where we try to visualize things that occur in nature and that we cannot otherwise see. Figure 1.3(b), on the other hand, shows an instance of traffic in a computer network. Basically, it shows the status of the network at that instant, such as the active nodes, the active links, the data flow path, and so on. As you can see, this figure shows something, which is not *natural* (i.e., it shows information about some man-made entity). Visualization of this latter kind is known as *information visualization*.

Each of the aforementioned points is basically an example of the usage of *computer graphics*. The spectrum of such applications is very wide. In fact, it is difficult to list



(a)



(b)

Fig. 1.3 Two examples of visualization (a) Visualization of a DNA molecule (b) Network visualization

all applications as virtually everything that we see around us involving computers contains some applications of computer graphics. Apart from the examples we saw and the typical desktop/laptop/tablet/palmtop applications that we traditionally refer to as *computers*, computer graphics techniques are used in the mobile phones we use, information kiosks at popular spots such as airports, ATMs, large displays at open air music concerts, air traffic control panels, the latest movies to hit the halls, and so on. The applications are so diverse and widespread that it is no exaggeration to say that for a layman in this digital age, the term *computer graphics* has become synonymous with the *computer*.

In all these examples, we see instances of *images* displayed on a computer screen. These images are constructed with *objects*, which are basically geometric shapes (characters and icons) with colors assigned to them. When we write a document, we are dealing with letters, numbers, punctuations, and symbols. Each of these is an object, which is rendered on the screen with a different style and size. In case of drawing, we deal with basic shapes such

as circles, rectangles, curves, and so on. For animation videos or computer games, we are dealing with virtual characters, which may or may not be human-like. The images or parts thereof can be manipulated (interacted with) by a user with input devices such as mouse, keyboard, joystick, and so on.

The question is *how can a computer do all these stuff?* We know that computers understand only the binary language, that is, the language of 0s and 1s. Letters of an alphabet, numbers, symbols, or characters are definitely not strings of 0s or 1s, or are they? How we can represent such objects in a language understood by computers so that those can be processed by the computer. How can we map from the computer's language to something that we can perceive (with physical properties such as shape, size, color)? In other words, how we can create or represent, synthesize, and render imagery on a computer display? This is the fundamental question that is studied in the field of computer graphics.

This fundamental question can further be broken down into a set of four basic questions.

1. Imagery is constructed from its constituent parts. How to represent those parts?
2. How to synthesize the constituent parts to form a complete realistic imagery?
3. How to allow the users to manipulate the imagery constituents on-screen?
4. How to create the impression of motion?

Computer graphics seeks the answer to these questions. A couple of things are important here. First, the term *computer screen* here is used in a very broad sense and encompasses all sorts of displays including small displays on the mobile devices such as smart phones, tablets, etc., interactive white boards, interactive table tops, as well as large displays such as display walls. Obviously, these variations in displays indicate corresponding variations in the underlying computing platforms. The second issue is, computer graphics seeks *efficient* solutions to these questions. As the computing platforms vary, the term *efficiency* refers to ways that make or try to make optimal resource usage for a given platform. For example, displaying something on mobile phone screens requires techniques that are different from displaying something on a desktop. This is because of the differences in CPU speed, memory capacity, and power consumption issues in the two platforms.

Let us delve a little deeper. In the early era of computers, displays constituted a terminal unit capable of showing only characters. In subsequent developments, ability to show complex 2D images was introduced. However, with the advent of technology, the memory capacity and processor speeds of computing systems have increased greatly. Along with that, the display technology has also improved significantly. Consequently, our ability to display complex processes such as 3D animation in a realistic way has improved to a great extent. There are two aspects of a 3D animation. One is to synthesize frames, the other is to combine them and render in a way to generate the effects of motion. Both these are complex and resource-intensive tasks, which are the main areas of activities in the present-day computer graphics.

Thus, computer graphics can be described in brief as the process of rendering static images or animation (sequence of images) on computer screens in an efficient way. In the subsequent chapter, we shall look into the details of this process.

1.1 HISTORICAL DEVELOPMENT OF THE FIELD

The evolution of the field of computer graphics is intricately linked to the evolution of the computer itself. We cannot separately describe the history of computer graphics without mentioning the developments that took place in shaping up the concept and technology of the present-day computers. However, we shall restrict ourselves to a concise account of the development of the field, mentioning only the major milestones in the process.

The term *computer graphics* was first coined by William Fetter of Boeing Corp. in 1960. Sylvan Chasen of Lockheed Corp. in 1981 proposed phase-wise classification of the development of the field. He mentioned four distinct phases in the development process.

1. Conception to birth or the gestational period (1950–1963)
2. Childhood (1964–1970)
3. Adolescence (1970–1981)
4. Adulthood (1981–)

Conception to birth The gestational period clearly coincides with the early developmental phase of the computing technology itself. What we take for granted (interactive graphical user interphase—GUI) in any computing system today, was not present even in the imagination of people back then. An early system from this phase, which demonstrated the power of computer graphics, was the SAGE (Semi Automatic Ground Environment) air defense system of the US Air Force (see Fig. 1.4). The system was a product of the Whirlwind project (started in 1945 at the MIT, USA). In this project, the system received positional data related to an aircraft from a radar station. This radar data was shown as an aircraft on a CRT screen, superimposed on a geographical region drawn on the screen. An input device, called a *light gun* or *light pen*, was used by the operators to request identification information about the aircraft (see Fig. 1.4). When the light gun was pointed at the symbol for the plane on the screen, an event was sent to the Whirlwind, which then sent text about the plane's identification, speed, and direction to be displayed on the screen.

Although the SAGE (Whirlwind) system demonstrated traces of interactive graphics (with the use of light pens to provide input to the system), the true potential of interactive computer graphics caught people's attention after the development of the *Sketchpad* by Ivan Sutherland in 1963, as part of his doctoral dissertation at MIT. The Sketchpad used the light pen

Fig. 1.4 SAGE system with light pen
Source: https://design.osu.edu/carlson/history/lesson2.html

to create engineering drawings directly on the CRT screen (see Fig. 1.5). Precise drawings could be created, manipulated, duplicated, and stored. The Sketchpad was the first GUI long before the term was coined and pioneered several concepts of graphical computing, including memory structures to store objects, rubber-banding of lines, the ability to zoom in and out on the display, and the ability to make perfect lines, corners, and joints. This achievement made Sutherland to be acknowledged by many as the *grandfather* of interactive computer graphics.

In addition to the SAGE and the Sketchpad systems, the gestational period saw development of many other influential systems such as the first computer game (*Spaceware* developed by Steve Russell and team in 1961 on a PDP-I platform) and the first CAD system (DAC-1 by IBM, formally demonstrated in 1964 though the work started in 1959, see Fig. 1.6).

Adolescence In 1971, Intel released the first commercial microprocessor (the 4004). This brought in a paradigm shift in the way computers are made, having profound impact on the advent of the field of computer graphics. In addition, the adolescence period (1970–1981) saw both the developments of important techniques for realistic and 3D graphics as well as several applications of the nascent field, particularly in the field of entertainment

Fig. 1.5 The use of the sketchpad software with a light pen to create precise drawings
Source: https://design.osu.edu/carlson/history/lesson3.html

Fig. 1.6 The first CAD system by IBM (called DAC-1)—DAC stands for *Design Augmented by Computer*

and movie making, which helped to popularize the field. Notable developments during this period include the works on lighting model, texture and bump mapping, and ray tracing. This period also saw the making of movies such as the Westworld (1973, first movie to make use of computer graphics) and Star Wars (1977). The worldwide success of Star Wars demonstrated the potential of computer graphics.

Adulthood The field entered its adulthood period (1981 onwards) standing on the platform created by these pioneering works and early successes. The year saw the release of the IBM PC by IBM, which helped computers to proliferate among the masses. In order to cater to this new and emerging market, the importance of computer graphics was felt more intensely. The focus was shifted from *graphics for expert* to *graphics for laymen*. This shift in focus accelerated works for developing new interfaces and interaction techniques and eventually gave rise to a new field of study: the human-computer interaction or HCI in short.

The development of software and hardware related to computer graphics has become a self-sustaining cycle now. As more and more user-friendly systems emerge, they create more and more interest among people. This in turn brings in new enthusiasm and investments on innovative systems. The cycle is certainly helped by the huge advancements in processor technology (from CPU to GPU), storage (from MB to TB), and display (CRT to touch screen and situated walls) technology. The technological advancements have brought in a paradigm shift in the field. It is now possible to develop algorithms to generate photo-realistic 3D graphics in real time. Consequently the appeal and application of computer graphics have increased manifold. The presence of all these factors implies that the field is growing rapidly and will continue to grow in the foreseeable future. Table 1.1 highlights the major developments in computer graphics.

Table 1.1 Major developments in computer graphics

Phase	Period	Major developments
Gestational period	1950–1963	The SAGE system
		The Sketchpad system
		Spaceware—the first computer game
		First CAD system
Childhood	1964–1970	Mainly consolidation of the earlier ideas
Adolescence	1971–1980	First commercial microprocessor by Intel Corp.
		Development of techniques for 3D realistic graphics (lighting models, bump mapping, ray tracing)
		Application of computer graphics to movie making (Westworld, Star Wars)
Adulthood	1981–Present	Release of the first personal computer (IBM PC) in 1981

1.2 MAJOR ISSUES AND CONCERNS IN COMPUTER GRAPHICS

In the formative stages of the field, primary concern was generation of 2D scenes. Although still in use, 2D graphics is no longer the thrust area. Its place has been taken over by 3D graphics and animations. Consequently, there are three primary concerns in computer graphics today.

Modeling Creating and representing the geometry of objects in the 3D world

Rendering Creating and displaying a 2D image of the 3D objects

Animation Describing how the image changes over time

Modeling not only deals with modeling of solid geometric objects but also modeling of phenomena such as smoke, rain, and fire. Rendering deals with displaying the modeled objects/scenes on the screen. The issues here are many and includes color and illumination (simulating optics), determination of visible surfaces (with respect to a viewer position), introducing texture patterns on object surfaces (to mimic realism), transforming from 3D description to 2D image and so on. Imparting motion on the objects to simulate movements is dealt with in animation. Modeling of motion and interaction between objects are the key concerns here.

Hardware-related issues While these are primarily issues that are addressed in graphics software, there are many hardware-related issues such as the following:

1. The quality and cost of the display technology, with the trade-off between the two being an important consideration
2. The selection of the appropriate interaction devices (what type of input devices should we have to make the interaction *intuitive*)
3. Design of specialized graphic devices to speed up the rendering process

In fact, many graphic algorithms are nowadays executed in hardware for better performance. How to design such hardware at an affordable cost is the primary concern here.

In subsequent chapters of this book, we shall explore how the issues are addressed. However, we shall restrict ourselves primarily to the discussion of the issues addressed in software. Nevertheless, in the next section, we shall briefly introduce the basic architecture of a graphic system, which will aid in the understanding of further discussions.

1.3 PRELIMINARIES: BASICS OF GRAPHICS SYSTEM

In computer graphics, what do we do? We synthesize a 2D image that can be displayed on a screen. Figure 1.7 shows the schematic of a generic system architecture that is followed by modern-day graphics system. In the figure, the task of image generation is relegated to a separate system called the *display controller*, which takes its input from the CPU (host computer in the figure) as well as external *input devices* (mouse, keyboard, etc.). The generated image is stored in digital form in a *video memory*, which is a (dedicated) part of the memory hierarchy of the system. The stored image is taken as input by the *video controller*, which

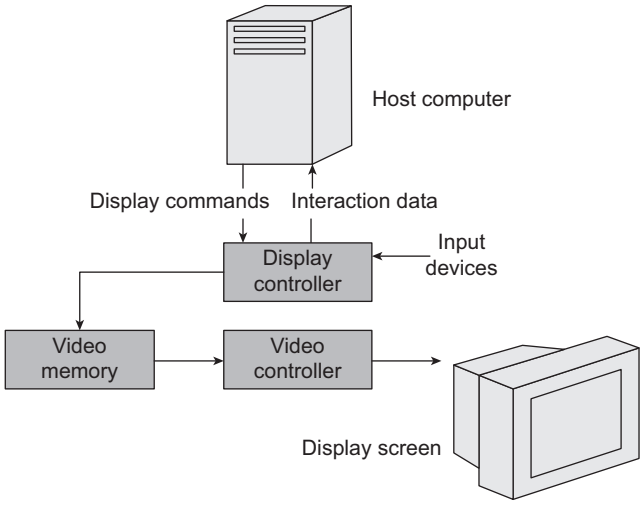


Fig. 1.7 Generic architecture of a graphics system

converts the digital image to analog voltages that drive electro-mechanical arrangements, which ultimately render the image on the screen.

Let us delve a little deeper to understand the working of the graphics system. The process of generating an image for rendering is a multi-stage process, involving lots of computation (we shall discuss these computations in subsequent parts of the book). If all these computations are to be carried out by the CPU, then the CPU may get very less time for doing other computational tasks. As a result, the system cannot do much except graphics. In order to avoid such situations and increase system efficiency, the task of rendering is usually carried out by a dedicated component of the system (the

What is a *frame buffer*? Why it is needed?

The video memory depicted in Fig. 1.7 for a raster scan system is more generally known as the *frame buffer*. The buffer contains one location corresponding to each pixel location. Thus the size is equal to the resolution of the screen. As we mentioned, the display processor in the display controller performs the computations and the video controller performs the actual rendering. The display processor works at the speed of the CPU (nanosecond scale). However, the video controller is typically an electro-mechanical arrangement, which is much slower (millisecond scale). Consequently, there is a mismatch between the speeds of these two components. If the output of the display controller is fed directly as the input of the video controller, the resulting image can get distorted as the next input may come before the video controller finishes processing the current input. To synchronize their operation, the frame buffer is used. In fact, if we use a single frame buffer, the synchronization problem may occur. Consequently, at least two frame buffers are used in practice (called *double buffering*), called *primary* and *secondary* buffers. The video controller takes input from the primary buffer. During this process, the display controller fills up the secondary buffer. Once the secondary buffer is filled up, it is now designated as primary and the primary becomes secondary (role reversal) and the process repeats.

graphics card in our computers) having its own processing unit (called GPU or graphics processing unit). The CPU, when encountering the task of displaying something, simply assigns the task to this separate graphics unit, which is termed as the display controller in Fig. 1.7.

Thus the display controller generates the image to be displayed on the screen. The generated image is in digital format (strings of 0s and 1s). The place where it is stored is the *video memory*, which in modern systems is part of the separate graphics unit (the *VRAM* in the graphic card). The *display screen*, however, contains picture elements or *pixels* (such as phosphor dots and gas-filled cells). The pixels are arranged in the form of a grid. When these pixels are excited by electrical means, they emit lights with specific intensities, which give us the sensation of the colored image on the screen. The mechanism for exciting pixels is the responsibility of the *video controller*, which takes as input the digital image stored in the memory and activates suitable electro-mechanical mechanism such that the pixels can emit light.

Graphic Devices

Graphic devices can be divided into two broad groups, based on the method used for rendering (or excitation of pixels): (a) vector scan devices and (b) raster scan devices.

Vector scan devices In vector scan (also known as *random-scan stroke-writing*, or *calligraphic*), an image is viewed as composed of continuous geometric primitives such as lines and curves. Clearly, this is what most of us intuitively think about images. From the system's perspective, the image is rendered by rendering these primitives. In other words, a vector scan device excites only those pixels of the grid that are part of these primitives.

Raster scan devices In contrast, in raster scan devices, the image is viewed as represented by the whole pixel grid. In order to render a raster image, it is therefore necessary to consider all the pixels. This is achieved by considering the pixels in sequence (typically left to right, top to bottom). In other words, the video controller starts with the top-left pixel. It checks if the pixel needs to be excited. Accordingly, it excites the pixel or leaves it unchanged. It then moves to the next pixel on the right and repeats the steps. It continues till it reaches the last pixel in the row. Afterward, the controller considers the first pixel in the next (below the current) row and repeats the steps. This continues till the right-bottom pixel of the grid. The process of such sequential consideration of the pixel

What are vector and raster graphics?

There are two terms closely related the vector and raster scan methods, namely the *vector graphics* and the *raster graphics*. These terms are used to indicate the nature of image representation. When an image is represented in terms of continuous geometric primitives such as lines and curves, we call it vector graphics. On the other hand, images represented in terms of a pixel grid are known as raster graphics. Note that they indicate only the image representation and not the underlying hardware rendering process, unlike the vector and random scan. Thus, we can use a raster scan method to render a vector graphics.

Comparison between vector and raster graphics

As Fig. 1.8 demonstrates, in vector graphics we need to excite only a subset of the whole pixel grid. The problem is, designing a mechanism to selectively excite pixels on a pixel grid requires high precision and complex hardware. As opposed to this, the scanning-based rendering in raster devices requires simpler hardware to implement. Consequently, the vector devices are costlier than raster devices. Also, due to selective exciting, vector graphics is good for rendering wireframe (i.e., outline) images. For complex scenes, flicker is visible as the rendering mechanism has to make lots of random movements. Raster systems do not have these problems. For these reasons, the displays that we use are mostly raster systems.

grid is known as *scanning*. Each row in the pixel grid in a scanning system is called a *scan line*.

This difference between the two rendering methods is illustrated in Fig. 1.8.

Refreshing An important related concept is *refreshing*. The lights emitted from the pixel elements, after excitation, starts decaying over time. As a result, the scene looks faded on the screen. Also, since the pixels in a scene get excited at different points of time, they do not fade in sync. Consequently, the scene looks distorted. In order to avoid such undesirable effects, what is done is to keep on exciting the pixels periodically. Such periodic excitation of the same pixels is known as *refreshing*. The number of times a scene is refreshed per second is called the *refresh rate*, expressed in Hz (Hertz, the frequency unit). Typical refresh rate required to give a user the perception of static image is at least 60 Hz.

So far, we have discussed the broad concepts of the display system, namely the video controller, the pixel grid, and the raster and vector displays. The discussion was very generic and applies to any graphic system. Let us understand these generic concepts in terms of an actual system, namely the cathode ray tube (CRT) displays. Although CRTs are no longer in wide use, discussion on CRT serves pedagogical purpose as it enables us to discuss all the relevant concepts.

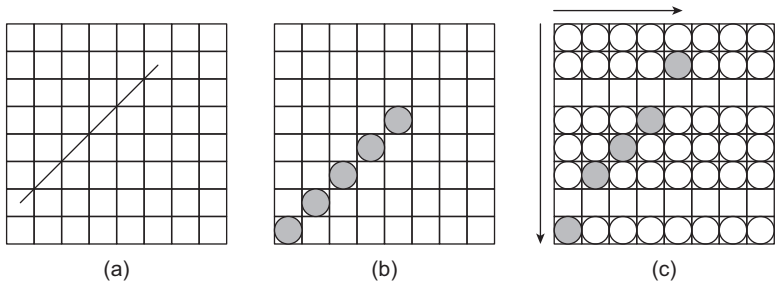


Fig. 1.8 Difference between vector and raster scan devices (a) Image to be rendered on the pixel grid (b) Vector scan method—only those pixels through which the line passes are excited (black circles) (c) Raster scan method—all pixels are considered during scanning—white circles show the pixels not excited; black circles show excited pixels; arrows show scanning direction

Differentiate between refresh rate and frame rate. Does higher frame rate ensure better image quality?

The frame rate of a computer is how often a video processing device can calculate the intensity values for the next frame to be displayed, that is, the rate at which the frame buffer is filled with new intensity values. Refresh rate refers to how often the display device can actually render the image.

No, frame rate is not a measure of how good the display is. Too high a frame rate is not beneficial as any frames sent for rendering above the display's refresh rate are not rendered and simply lost. Thus it is important to have a display capable of high refresh rates in order to synchronize the two.

1.3.1 CRT Displays

The CRT technology, invented in 1885 ruled the computer displays till recently. We are all familiar with it. A typical CRT display is shown in Fig. 1.9, with a schematic of the working of the technology. As the figure shows, CRT displays contain a long funnel-shaped vacuum *tube* (that gives them the bulky shape). At one end of the tube (the back side) is a component known as the *electron gun*, which is a heated metal cathode. The heating is achieved by passing current through a filament inside the cathode. When heated, the metallic cathode surface generates negatively charged electrons. Through the use of controlled deflection mechanism (the first and second anodes in the figure with high positive voltages), these electrons are focused into a narrow *electron beam* (the *cathode ray*). The front side of the tube (the screen) is coated with phosphor dots (in a grid form). The electron beams are *guided* towards specific points (phosphor dots) on the grid with the use of another *deflection* mechanism. In the deflection mechanism, the electron beam is deflected horizontally or vertically using electrical or magnetic fields created by the vertical or horizontal deflection plates shown in the figure. Once the beam strikes a phosphor dot, the dot emits photons (light) with intensity proportional to that of the beam. The light emission by phosphor dots gives us the impression of image on the screen.

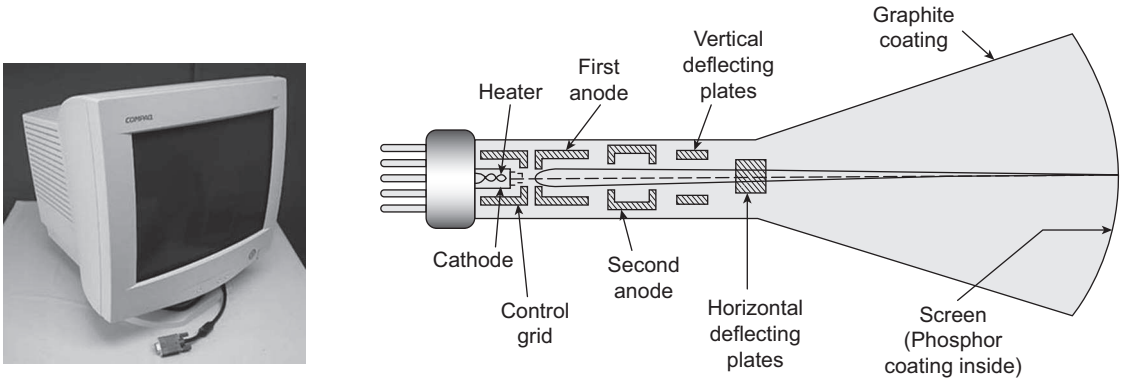


Fig. 1.9 (a) Typical CRT (b) Schematic representation of inner working of CRT

In summary, therefore, in a CRT, the electron gun generates cathode rays, which hits the phosphor dots to emit light, eventually giving us the sensation of image. Now, let us go back to the terms we introduced in the previous section and try to understand those concepts in light of the CRT displays. The video controller of Fig. 1.7 is the system responsible for generating requisite voltage/fields to generate cathode rays of appropriate intensity and guide the rays to hit specific phosphor dots (pixels) on the screen. In vector displays, the video controller guides the electron gun to only the pixels of interest. In case of raster systems, the electron gun actually moves in a raster sequence (left to right, top to bottom).

What we have discussed so far assumes that each pixel is a single phosphor dot. By varying the intensity of the cathode ray, we can generate light of different intensities. This gives us images having different shades of gray at the most. In the case of color displays, the arrangement is a little different. We now have three electron guns instead of one. Similarly, each pixel position is composed of three phosphor dots, each corresponding to the red (R),

How is the resolution of a CRT screen determined?

As mentioned before, the phosphor dots (pixels) are arranged in the form of a grid on the screen. This grid is typically referred to as *resolution* of the screen and expressed as C $\times$ R (C = number of columns, R = number of rows). How do we determine the resolution? This is determined on the basis of the properties of the pixels. The intensity of the light emitted by a phosphor dot, after it is hit by the cathode ray, follows a certain distribution around the point of impact (typically assumed to be the center of the pixel). The intensity gradually falls off as we move away from the center. This change in intensity follows a Gaussian distribution. Now, when two pixels are placed side by side, the lights emitted by each of them will be distinguishable only if the overlap of their intensity distributions is outside of 60% of peak intensity distribution of each. Otherwise, we will not be able to distinguish between the spots clearly. The resolution is determined based on this principle. The idea is illustrated in the following figure.

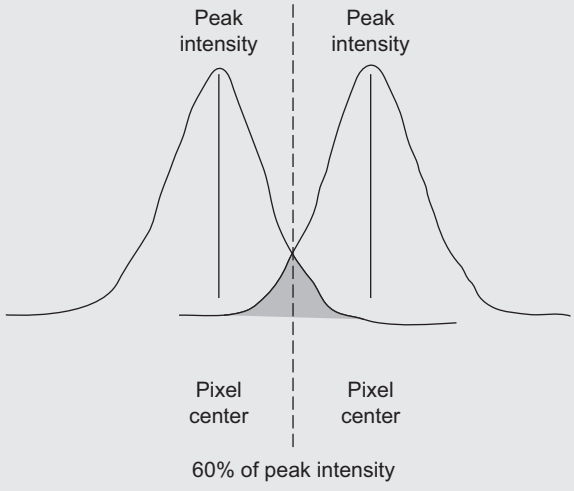


Illustration of emitted intensity distribution around pixels in a CRT device

green (G), and blue (B) colors. As you know, these three are called the *primary* colors. Any color can be generated by mixing these three in appropriate quantities. The same process is simulated here. Each electron gun corresponds to one of the R, G, and B dots. By controlling electron gun intensities separately, we can generate different shades of R, G, and B for each pixel, resulting in a new color. Figure 1.10 shows the schematic of the process.

There are two ways computer graphics with color displays are implemented. In the first method, the individual color information for each of the R, G, and B components of a pixel are stored directly in the corresponding location of the frame buffer. This method is called *direct coding*. Although the video controller gets the necessary information from the frame buffer directly to drive the electron guns, the method requires a large frame buffer to store all possible color values (that means all possible combinations of the RGB values. This set is also known as *color gamut*). An alternative scheme, used primarily during the early periods of computers when the memory was not so cheap, makes use of a *color look-up table* (CLT). In this scheme, a separate look-up table (a portion of memory) is used. Each entry (row) of the table contains a specific RGB combination. There are N such combinations (entries) in the table. The frame buffer location in this scheme does not contain the color itself. Instead, a pointer to the appropriate entry in the table that contains the required color is stored. The scheme is illustrated in Fig. 1.11.

Note that the scheme is based on a premise: we only require a small fraction of the whole color gamut in practice and we, as designers, know those colors. If this assumption is not valid (i.e., we want to generate high quality images with large number of colors), then the CLT method will not be of much use.

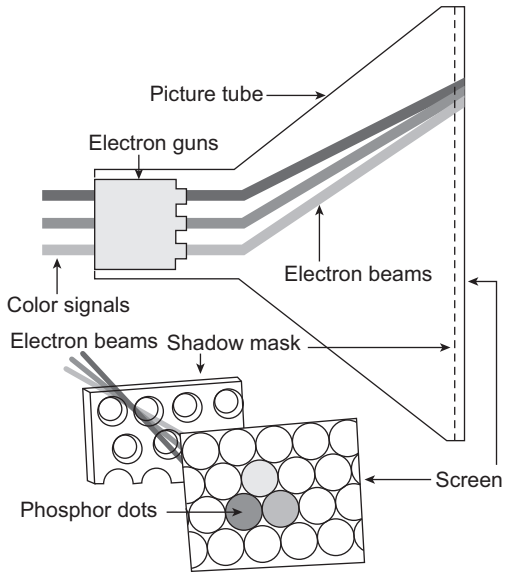


Fig. 1.10 Schematic of a color CRT. The three beams generated by the three electron guns are passed through a *shadow mask*—a metallic plate with microscopic holes that direct the beams to the three phosphor dots of a pixel.

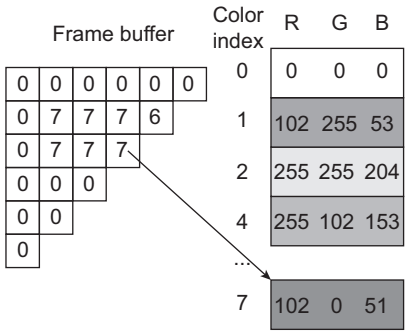


Fig. 1.11 Schematic of color look-up scheme

1.4 GRAPHICS PIPELINE: STAGES OF RENDERING PROCESS

In the preceding discussion, we were talking about the color values stored in the frame buffer. How are these values obtained? As we mentioned, the display processor *computes* these values in stages. These stages together are known as the *graphics pipeline*.

Object representation At the very beginning, we need to *define* the objects that will be part of the image that we see on the display. There are several object representation techniques available to cater to the task of efficient creation and manipulation of images (scenes).

Modeling/Transformation The objects are defined in their own (local) coordinate system. We need to put them together to construct the image, which is having its own coordinate system (known as the *world coordinate*). This process, of putting individual objects into a scene, is known as the *modeling/geometric transformations*, which is the second stage of the pipeline.

Lighting Once the (3D) scene is constructed, the objects need to be assigned colors, which is our third stage. Color is a psychological phenomena linked to the way light behaves (i.e., the laws of optics). Thus, in order to assign color to our scene, we need to implement methods that mimic the optical laws.

Highlight the advantage of color look-up scheme over direct coding with an example.

Suppose each of the R, G, and B is represented with 8 bits. That means we can have between 0 and 255 different shades for each of these primary colors. Consequently, the total number of colors (the color gamut) that we can generate is $256 \times 256 \times 256 = 16$ M.

Direct coding: The size of each frame buffer location is $8 \times 3 = 24$ bits. Thus, the size of the frame buffer for a system with resolution 100×100 will be $24 \times 100 \times 100 = 234$ Kb.

Color look-up scheme: Assume that out of 16 M possible colors, we know that in our image we shall use only 256 colors (combinations of R, G and B). We shall keep these 256 colors in the look-up table. Thus the size of the table is 256 with each row having 24 bits color value. Each of the table location requires 8 bits to access. Therefore, to access any table location, we need to have 8 bits in each frame buffer location. What will be our storage requirement for the 100×100 screen? It is the frame buffer size + the table size. In other words, the size of the frame buffer will be, $(8 \times 100 \times 100) + (256 \times 24) = 84$ Kb, much less than the 234 Kb required for direct coding.

Viewing pipeline/transformation After the aforementioned stages, we have a 3D scene with appropriate colors. The next task is to map it to 2D *device coordinate*. This is analogous of taking a snapshot of a scene with a camera. Mathematically, the snapshot taking involves several intermediate operations. First, we set-up a camera (also called *view*) coordinate system. Then the world coordinate scene is transferred to the view coordinate system (known as the *viewing transformation*). From there, we transfer the scene to the 2D view plane (the *projection transformation*).

For projection, we need to define a region in the viewing coordinate space (called *view volume*). Objects inside the volume are projected. The objects that lie outside are removed from the consideration for projection. The process of removing objects outside view volume is called *clipping*. Along with clipping, another task is performed at this stage. When we project, we consider a viewer position. With respect to that position, some objects will be fully visible, some partially visible while some will be altogether invisible, although all of the objects are within the same volume. In order to capture this viewing effect, the process of *hidden surface removal* (also known as the *visible surface detection*) is carried out. Once both clipping and hidden surface removal are performed, the scene is projected on the view plane.

From the view plane, the 2D projected scene is transferred to a region on the device coordinate system (called *viewport*). The process is known as the *window-to-viewport transformation*.

These series of transformations (viewing, projection, and viewport) along with the tasks of clipping and hidden surface removal is sometimes referred to as the *viewing pipeline* and constitute the fourth stage of the graphics pipeline.

Scan conversion The device coordinate is a continuous space. However, the display, as we have seen before, contains a pixel grid, which is a discrete space. Therefore, we need to transfer the viewport on the (continuous) device coordinate to the (discrete) screen coordinate system. This process is called *scan conversion* (also called *rasterization*). An added concern here is how to minimize distortions (called *aliasing effect*) that result from the transformation from continuous to discrete spaces. Anti-aliasing techniques are used during the scan conversion stage, to minimize such distortions. Scan conversion with anti-aliasing together forms the last and final stage of the 3D graphics pipeline.

The stages of the pipeline are shown in Fig. 1.12. We shall discuss each of these stages in subsequent chapters of the book. However, the sequence of the stages mentioned here is purely theoretical. In practice, the sequence may not be followed strictly. For example, the

How does object space differ from image space?

The pipeline we discussed before is object space, as it starts from the object definition culminating in image generation. All intermediate operations are done on the objects. On the other hand, there are methods where the operations start from pixels (images) and moves backward (to object definitions). Such methods are said to work in the image space. An example is the ray-tracing method. However, in this book, we shall concentrate on the object space methods only.

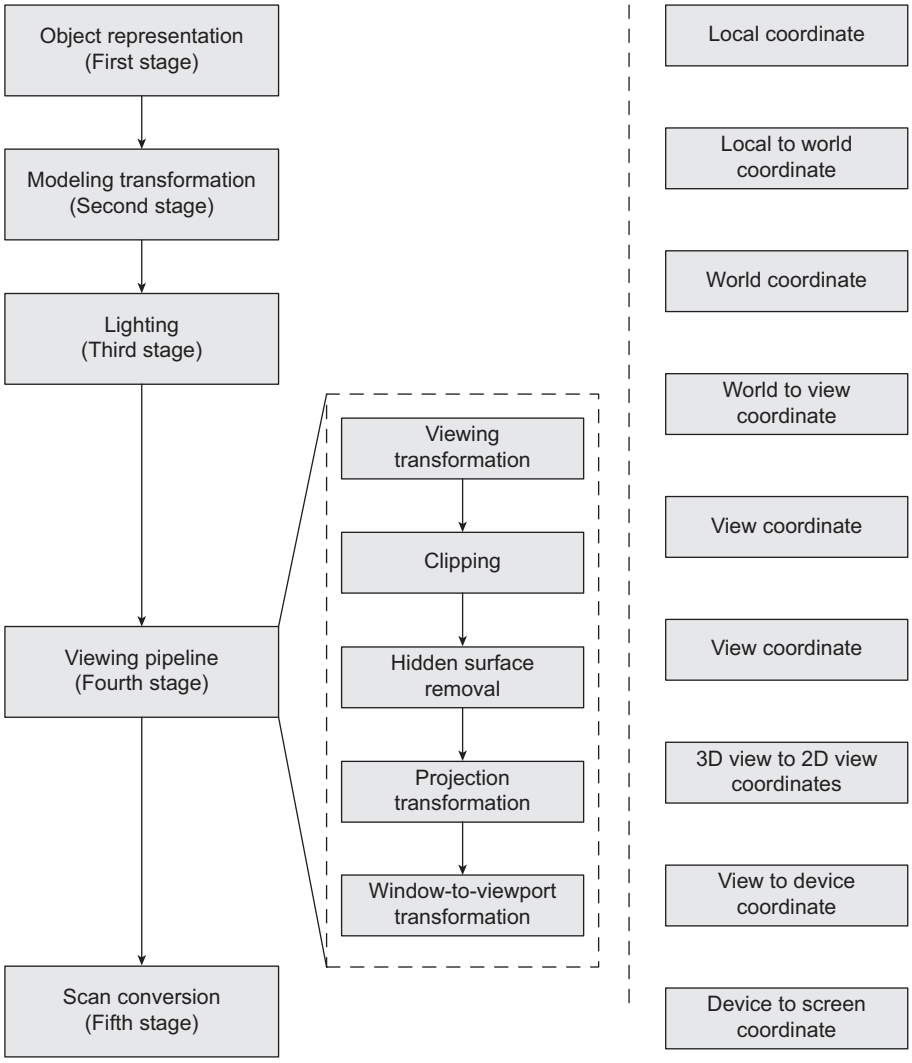


Fig. 1.12 Stages of 3D graphics pipeline—Boxes in the second column show the substages of the fourth stage; the third column shows the coordinate systems in which each stage operates

assigning of colors (third stage) may be performed after projection to reduce computation. Similarly, the hidden surface removal may be performed after projection.

1.5 ROLE OF GRAPHIC LIBRARIES

In the preceding discussions, we outlined the theoretical background of computer graphics. However, a programmer need not always have to implement the stages of the pipeline, in order to make the system work. Instead, the programmer can use the application programming interfaces (APIs) provided by the graphics libraries to perform the pipeline stages.

Examples of graphic libraries include OpenGL (stands for *Open source Graphics Library*) and DirectX (by Microsoft).

These APIs are essentially predefined sets of functions, which, when invoked with the appropriate arguments, perform the specific tasks. Thus, these functions eliminate the need for the programmer to know every detail of the underlying system (the processor, memory, and OS) to build a graphics application. For example, the function 'xyz' in OpenGL assigns the color abc to a 3D point. Note that the color assignment does not require the programmer to know details such as how color is defined in the system, how such information is stored (which portion of the memory) and accessed, how the operating system manages the call, which processor (CPU/GPU) handles the task, and so on. Graphics applications such as painting systems, CAD tools, video games, or animations are developed using these functions.

SUMMARY

In this chapter, we have touched upon the background required to understand topics discussed in later chapters. We have briefly seen some of the applications and discussed the history of the field, along with the current issues. We also got introduced to the generic architecture of a graphics system and brief overview of important concepts such as display controller, video controller, frame buffer, pixels, vector and raster devices, CRT displays, and color coding methods. We shall make use of this information in the rest of the book. The other important concept we learnt is the graphics pipeline. The rest of the book shall cover the stages of the pipeline. In the following chapter, we introduce the first stage: the object representation techniques.

BIBLIOGRAPHIC NOTE

There are many online sources that give good introductory idea on computer graphics. The website https://design.osu.edu/carlson/history/lessons.html includes in-depth discussion, with illustrative images, of the historical evolution and application areas of computer graphics. The textbooks on computer graphics by Hearn and Baker [2004] and Foley et al. [1995] also contain comprehensive introduction to the field. There is a rich literature on application of computer graphics techniques to various domains as diverse as aircraft design Bouquet [1978], energy exploration Gardner and Nelson [1983], scientific data visualization Hearn and Baker [1991] and visualization of music Mitroo et al. [1979]. A good source to learn about the past works, current trends, and research direction in the field are the issues of the well-known journals and conference proceedings in the field. Links to various bibliographics resources can be found in http://www.cs.rit.edu/~ncs/graphics.html. Also see the *Bibliographic Note* section of Chapter 11 for more references on graphics hardware.

KEY TERMS

Color look-up scheme – color management scheme used in computer graphics in which the color information is stored in a separate table.
CRT – stands for the cathode ray tube, a very popular technology till the recent times to design display screens
Direct coding – color management scheme used in computer graphics in which the color information is stored in the frame buffer itself

Display controller – generic name given to the component of a graphics system that converts abstract object definitions to bit strings
Electron gun – hardware to excite pixels on a CRT screen
Frame buffer – video memory of raster scan systems
Frame rate – rate (bits/second) at which the frame buffer is filled up
Graphics pipeline – set of stages in sequence that are used to synthesize an image from object definitions
Input device – hardware for interacting with screen
Pixel – a picture element or each point on the grid used to design display screens
Refresh rate – rate (times/second) at which the screen is redrawn
Refreshing – process by which the screen is redrawn periodically
SAGE system – the first computer graphics system
Sketchpad – the first interactive computer graphics system
Vector and raster graphics – the two types of image representation techniques used in computer graphics
Vector and raster scan – the two types of techniques used to render images on the screen
Video controller – hardware used to render the actual on-screen image from the bit strings stored in video memory
Video memory – memory unit used to store the bit strings that represent a (synthesized) image
Visualization – the techniques to visualize real or abstract objects or events

EXERCISES

1.1 At the beginning of this chapter, we learnt a few areas of application of computer graphics. As we mentioned, such applications are numerous. Find out at least 10 more applications of computer graphics (excluding those mentioned at the beginning).

1.2 Suppose you are trying to develop a computer game for your iPhone. Make a list of all the issues that are involved (Hint: combine the discussions of Sections 1.2 and 1.4).

1.3 In Figure 1.7, the generic architecture of a typical graphics system is shown. As we know, almost all the electronic devices we see around us contain some amount of graphics. Therefore, they can be dubbed as *graphics systems*. In light of the understanding of the generic architecture, identify the corresponding components (i.e., the display, the two controllers, video memory, I/O mechanism) for the following types of devices (you can select any one commercial product belonging to each of the categories and try to find out the name of the underlying technologies).

(a) Digital watch
(b) Smartphone
(c) Tablet
(d) ATM
(e) HD TV

1.4 Explain double buffering. Why do we need it?

1.5 While trying to display a scene on the screen, you observe too much flickering. What may be the cause for this? How can the flickering be overcome?

1.6 In some early raster displays, refreshing was performed at 30 Hz, which was half that of the minimum refresh rate required to avoid flicker. This was done due to technological limitations as well as to reduce system cost. In order to avoid flickers in such systems, the scan lines were divided into odd set $(1, 3, 5 \cdots)$ and even set $(2, 4, 6 \cdots)$. In each cycle, only one set of lines was scanned. For example, first scan odd set, then even set, then odd set, and so on. This method is called *interlacing*. List with proper explanation all the factors that determined if the method would work.

1.7 Assume you are playing an online multiplayer game on a system having a 100 × 100 color display with 24 bits for each color. A double buffering technique is used for rendering. Your system is receving 5 MBps of data from the server over the network. Assuming no data loss, will you experience flicker?

1.8 What is the main assumption behind the working of the color look-up table? Suppose you have a graphic system with a display resolution of 64 × 64. The color look-up table has 16 entries. Calculate the percentage saving of space due to the use of the table. Assume colors are represented by 24 bits.

1.9 Discuss why vector graphics is good for drawing wire frames but not filled objects. Suppose that you are a gaming enthusiast. Will you prefer a vector-based system or a raster-based system?

1.10 In a raster scan system, the scanning process starts (from top-left pixel) with the application of a vertical sync pulse (Vp). The time between the excitation of the last (right-most) pixel in the current scan line and that of the first (left-most) pixel of the next scan line is known as the horizontal retrace time (HT). Scanning for the next line starts with the application of a horizontal sync pulse (Hp). The time it takes to reset the scanning process for the next frame (i.e. the time gap between the excitation of the bottom-right pixel of the current frame and the top-left pixel of the next frame) is known as the vertical retrace time (VT). Calculate the desirable value of M for a CRT raster device having resolution $M \times 100$ with HT = 5 s, VT = 500 s and 1 s electron gun movement time between two pixels along a scan line.

1.11 Assume you have a raster device with screen resolution = 100 × 100. It is a color display with 9 bits/pixel. What should be the access rate (i.e., time required to access each bit) of the video memory to avoid flicker in this system?

1.12 Consider the two objects shown in Fig. 1.13(a).
We want to render a scene in which the bar (the right object in (a)) is inside the hole of the cube (left object in (a)), as shown in (b). Discuss the tasks performed in each pipeline stage for rendering the scene.

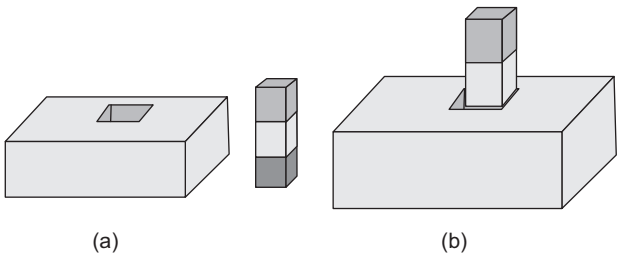


Fig. 1.13 The object and the scene of Exercise 1.12

CHAPTER 2

Object Representation Techniques

Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the four broad classes of object representation techniques used in computer graphics
- Have a basic understanding of the boundary representation techniques including quadric surfaces and blobby object representation
- Learn in detail about the spline representation techniques covering both interpolation and approximation splines
- Get a detailed overview of the space portioning methods-octree, BSP, and CSG
- Get an introductory idea on advanced representation techniques including skeletal model, scene graphs, particle systems, and fractals
- Get introduced to the issues in the section of appropriate representation technique for a given problem

INTRODUCTION

In a computer-generated scene, objects of different shapes and sizes are present. They can range from tiny snowflakes to complex characters. A snowflake, for example, has an elegant shape and should not be represented with a simple sphere. Similarly, an animated character needs to be depicted with its associated complexities so that it looks realistic. In order to generate scenes where all these disparate objects are present, we first need to have some way to represent them. Any representation will not work as we seek answers to the following two questions.

1. How can we represent all these different objects with their characteristic complexities so that those can be rendered realistically in a synthesized environment?
2. How can we have a representation that makes the process of rendering *efficient* (i.e., can we perform the operations of the different stages of the pipeline in ways that optimize space and time complexities)?

In order to answer these questions, a plethora of object representation techniques are used in computer graphics. In this chapter, we shall survey and discuss the pros and cons of those techniques.

2.1 CATEGORIZATION OF REPRESENTATION TECHNIQUES

A large number of techniques are in use in computer graphics to represent objects. We can categorize them along the following types.

Point-sample rendering In simulating a 3D scene, we can first capture raw data such as color, surface normal vector and depth information of different points on the scene. The capturing of information is done through the use of input sensors such as range scanner, range finder, or computer vision techniques. The information is captured in the form of points on the scene, with each point having the information as attributes. Those points can subsequently be processed to generate the scene. Thus, our representation here is a set of raw data points of the scene, each with some attribute values.

Boundary representation Some methods represent an object by representing the individual object surfaces. The surface can be polygonal or curved. An example is shown in Fig. 2.1, where the cube is represented as a collection of interlinked rectangles A–F.

Space partitioning In some methods, the 3D space occupied by the object is divided into several disjoint (or non-overlapping) regions. Any point inside the object lies in exactly one

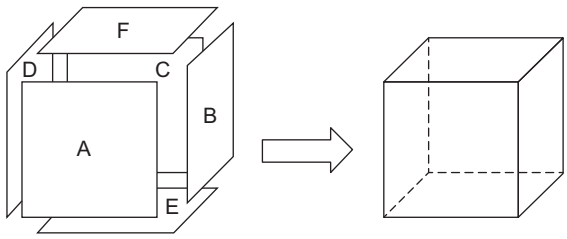


Fig. 2.1 An example of boundary representation of an object

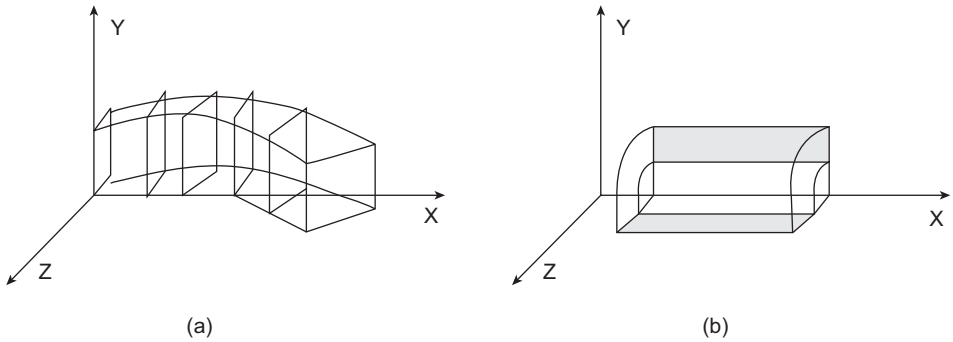


Fig. 2.2 Two types of sweep representations (a) Surface is represented as a square simultaneously translating and rotating along the X-axis (b) Object is represented as a closed polygon revolving around the X-axis

of the regions. The space-partitioning representations are often created in a hierarchical way in which a space is divided into several subregions, and then the same method is recursively applied to each of the subregions. A common way to depict the regions is to organize them in the form of a tree, called a space-partitioning tree.

Sweep representation Two sweep representation techniques used in computer graphics are the sweep surface and surface of revolution. In sweep surface, 3D surfaces are obtained by traversing an entity such as a point, line, polygon, or curve, along a path in space in a specified manner. When a 2D entity is rotated around an axis, we get a surface of revolution (a special case of sweep surfaces). These concepts are illustrated in Fig. 2.2.

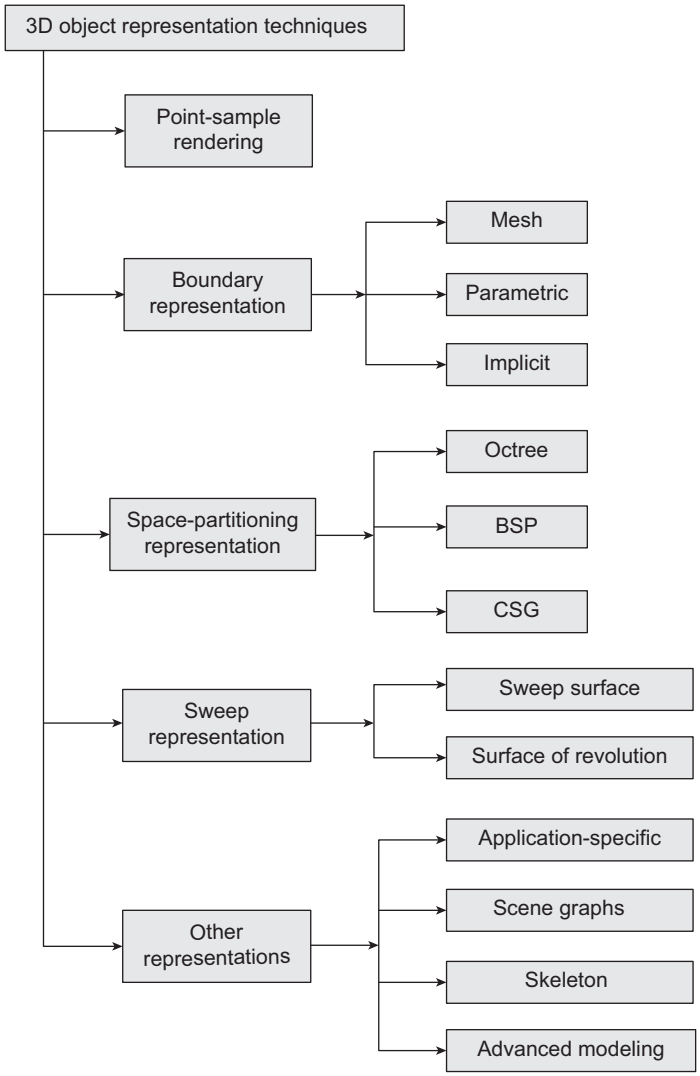


Fig. 2.3 Different object representation techniques used in 3D computer graphics

These broader categories may consist of several representation techniques. The three types of boundary representation techniques are the *mesh representation*, *parametric representation*, and *implicit representation*. Space-partitioning methods include techniques such as *octrees*, *BSP trees*, and *constructive solid geometry* (CSG). Apart from these broad classes, some other techniques are used either to represent application-specific objects, or to represent a set of objects each of which has been represented using either of the three categories or to represent complex photo-realistic objects. Fig. 2.3 summarizes these different techniques. In subsequent sections, we shall briefly discuss about each of these techniques.

2.2 BOUNDARY REPRESENTATION TECHNIQUES

There are several ways in which an object can be represented in terms of its bounding surfaces. The most basic technique is to use polygons to represent surfaces. Polygonal surfaces are represented using vertex/edge lists that store the information about all the vertices/edges on the surface and their relationships. For polyhedral objects (i.e., objects with polygonal surfaces), such lists are used, as shown in Fig. 2.4.

For objects having non-polygonal surfaces, the surfaces are approximated to polygonal (triangle or simple convex polygon) meshes and the meshes are represented using vertex/edge lists. The mesh representation is the most basic form of representation for any objects of a rendered scene. All other representations are converted to mesh representation at the end of the pipeline before the objects are rendered. Fig. 2.5 shows an example of an object with curved surfaces represented using a triangular mesh.

One important issue in mesh representation is the extent to which a surface is *tessellated* (i.e., how many polygons should be used to approximate surfaces). Clearly, the more the number of polygons, the better the approximation is. However, more (coarser) subdivisions also implies more storage and computation. Thus the subdivision rule is important to optimize space and time complexities of mesh representation and usually depends on the application and the resources available.

Although mesh representation is the most basic form of representing any object, it is not the most convenient from the designer's point of view. Instead of working with meshes,

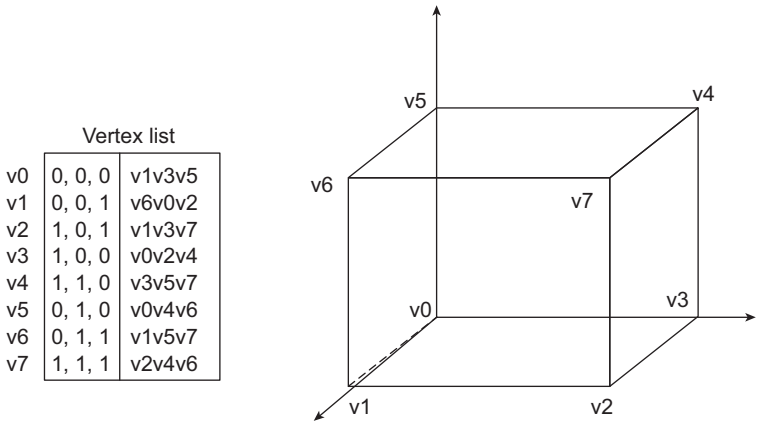


Fig. 2.4 Example of vertex list representation. In the list, each vertex is listed along with its coordinate and the other vertices with which it forms an edge.

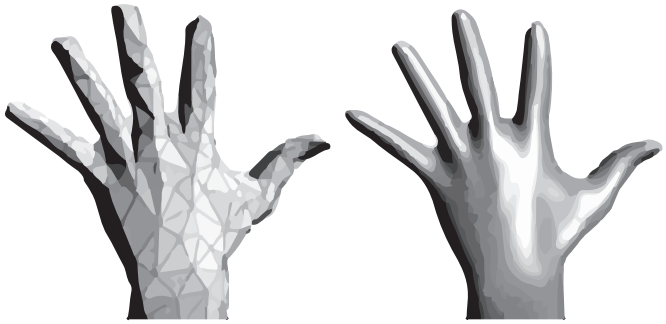


Fig. 2.5 Example of mesh representation (a) Triangular mesh (b) Rendered image

designers like to use representations that mimic the actual object rather than its approximation. Implicit and parametric surface representations are high-level representation techniques which are used to represent curved surfaces more accurately and conveniently. In implicit representation, the surfaces are defined in terms of the implicit functional form. In parametric representations, surfaces points are defined in Euclidean space in terms of some parameters. A frequently used class of objects in computer graphics, which are represented using implicit or parametric form, are the *quadric surfaces*.

2.2.1 Quadric Surfaces

Objects, which (or the surface of which) are described with second-degree equations (i.e., quadratic equations), are generally termed as quadric surfaces. Spheres are the most commonly used among all quadric surfaces. In implicit form, we can represent a spherical surface with radius r and centered at origin as the set of points in the Cartesian coordinate system that satisfy the following equation (Eq. 2.1).

$$x^2 + y^2 + z^2 = r^2 \tag{2.1}$$

The same surface can also be described in parametric form. For that, we require the latitude and longitude angles θ and ϕ (see Fig. 2.6). In terms of these angles, the parametric representation of the sphere is:

$$\begin{aligned} x &= r\cos\phi\cos\theta, -\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2} \\ y &= r\cos\phi\sin\theta, -\pi \leq \theta \leq \pi \\ z &= r\sin\phi \end{aligned}$$

Figure 2.7 shows another frequently used quadric surface: an ellipsoid. The implicit and parametric form of the ellipsoid are shown assuming that the ellipsoid is centered at the origin with radii r_x, r_y, and r_z along the respective (X, Y, and Z) axis.

Other examples of quadric surfaces are tori, paraboloids, and hyperboloids. Sometimes, additional parameters are introduced into the quadric surface equations to provide increased flexibility for adjusting object shapes. Usually, two additional parameters are incorporated in quadric surface representations. These surfaces are called *superquadrics* such as

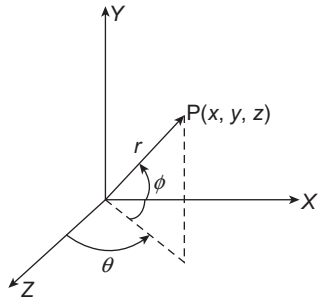


Fig. 2.6 Parametric coordinate (r, θ, ϕ) of a point on a spherical surface

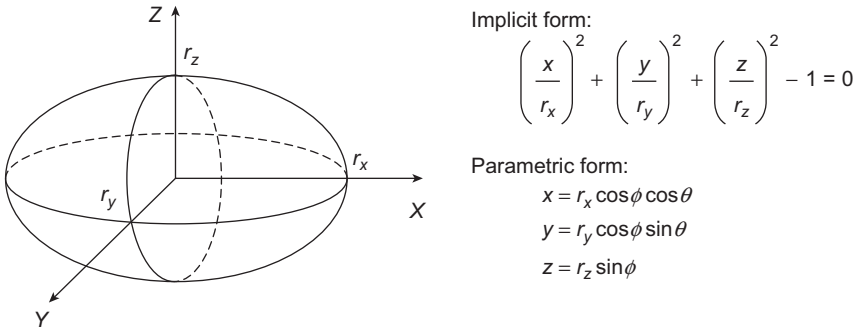


Fig. 2.7 An ellipsoid centered at origin represented in implicit form

superellipsoids. However, we will not go into the details of such representations. Interested readers may refer to the reading materials mentioned in the bibliographic notes at the end of the chapter.

2.2.2 Blobby Objects

There are some objects whose shapes show a certain degree of *fluidity*. That means, these objects change their shape during motion or when they come closer to other objects. Although the objects have curved surfaces, we can not use standard shapes to represent them. This is since the standard shapes fail to represent the surface fluidity in a realistic way. Such obejcts are generally referred to as *blobby objects*. Examples include molecular structures, liquid and water droplets, melting objects, and animal and human muscle shapes. There are several methods developed to represent blobby objects. Usually some *distribution function* is used over a region of space.

One method is to use a combination of Gaussian density functions (sometimes called Gaussian bumps). As we know, a Guassian density function, shown in Fig. 2.8, is characterized by two parameters: the *height* and the *standard deviation*. When we combine many such functions by varying the two parameters (plus some additional ones), we can represent a blobby object. A surface function is then represented as,

$$f(x, y, z) = \sum_i b_i e^{-a_i r_i^2} - T = 0 \tag{2.2}$$

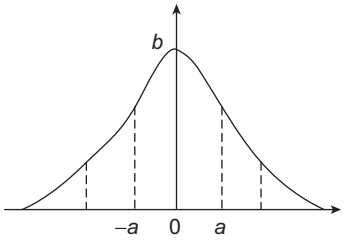


Fig. 2.8 A Gaussian density function or Gaussian bump, centered at the origin

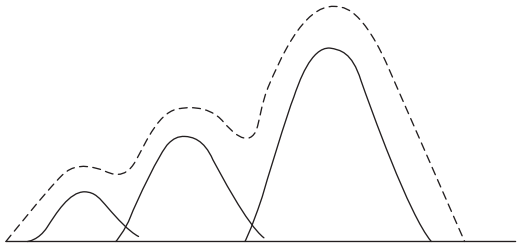


Fig. 2.9 A blobby object generated with three Gaussian bumps—the dotted lines represent the surface, whereas the solid lines show individual bumps

subject to the condition $r_i^2 = x_i^2 + y_i^2 + z_i^2$. By varying a_k and b_k, we can generate the desired amount of *blobbiness* in the individual components of the surface. If we set b_k negative, we can generate *dents* instead of *bumps* in the object. The parameter T is some specified threshold. A blobby surface with three Gaussian density functions is shown in Fig. 2.9.

The *metaball* model is another method to represent blobby objects. In this, a quadratic density function is used (instead of the Guassian bumps in Eq. 2.2), which is of the form

$$f(r) = \begin{cases} b(1 - \frac{3r^2}{d^2}), & \text{if } 0 < r \leq \frac{d}{3} \\ \frac{3}{2}b(1 - \frac{r}{d})^2, & \text{if } \frac{d}{3} < r \leq d \\ 0, & \text{if } r > d \end{cases} \tag{2.3}$$

In Eq. 2.3, b is called the scaling factor, r is the radius of the blobby object from its center, and d is the maximum radius of the object. In other words, d is bound on the spread of the object around its center. However, it is very difficult or even impossible to represent any arbitrary surface in either implicit or parametric form. A special type of parametric representation, known as spline representation or simply splines, can be used to represent any shapes. Owing to its widespread use in computer graphics, we discuss the concept of splines in more details in the following section.

2.3 SPLINE REPRESENTATIONS

In order to generate complex shapes, we need to represent curves. Before we proceed further, we should note that representation here refers to the parametric representation.

For a curve, we represent it (or its Cartesian coordinates) using a single parameter u[1] as in Eq. 2.4.

$$x = f_1(u) \tag{2.4a}$$

$$y = f_2(u) \tag{2.4b}$$

A useful analogy is to think of u as denoting time. In other words, we are drawing the curve on a 2D Cartesian space over a period of time. At any instant of time, we place a Cartesian point. Thus, any value of u denotes a specific instant of time, at which point we can determine the corresponding coordinate values using Eq. 2.4. In subsequent discussions, we shall assume this parametric representation of curve.

The question is, how can we represent a curve easily and efficiently? We can have the curve represented as a set of small line segments. Obviously the line segments have to be very small to make the curve look smooth. Although this method is easy, it may not be efficient as we may have to provide a large number of points depending on the nature of the curve. An alternative is to work out the curve equation. This certainly can provide an easy and efficient solution. The only problem is, we may not be able to find out the equation itself!

Let us look at the problem from a user's perspective. As a user, we want to generate a curve (of any shape). However, we are not interested to provide as input to the system a (very) large number of points (to represent it as a collection of small line segments) or a precise equation of the curve (which we may not know). Instead, we may provide a limited set of points that *defines* the curve (i.e., the curve passes *through* or *nearby* those points). Such points are known as *control points*. We expect the system to draw the curve by *interpolating* those control points that we provide.

The process of interpolation essentially refers to fitting a curve that passes through or nearby the control points. Mathematicians generally agree that polynomial interpolation (i.e., trying to fit a polynomial curve) is simple, efficient, and easy to manipulate among all types of interpolation. Depending on the number of control points, we may decide on the degree of the interpolating polynomial. For example, if two control points are given, a linear interpolation is advisable, whereas for three control points, quadratic polynomials serve best, and so on. In general, for $n + 1$ control points, we may try to fit a polynomial of degree n, as shown in Fig. 2.10.

In order to determine the polynomial from the set of control points, we need to solve the system of $n + 1$ equations shown on the right-hand side of Fig. 2.10. For a large n, the process of interpolation clearly becomes cumbersome as we need to solve a large number of equations (note that we need to find two separate parametric equations for x and y, for a 2D curve). In addition, there may be another problem when we are dealing with a large n. If the user wants to change the shape slightly (by changing one or few of the control points), the entire curve may have to be recalculated. This is known as the problem of *local*

[1]When the parameter can have any arbitrary range of values, it is usually denoted by t. However, if the parameter can take values only within the range [0,1], it is usually denoted by u. we will also follow this convention.

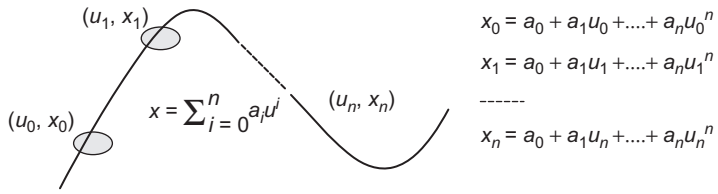


Fig. 2.10 An interpolated polynomial through a set of *n* input points

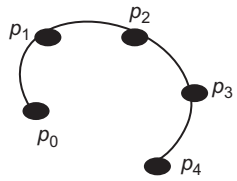


Fig. 2.11 Example of a curve constructed by joining two polynomial pieces

controllability, that is, we are not able to change the curve locally (around the set of points that we want to change), rather we have to change the whole curve (a new curve fitting process). Thus, every time we want to make small changes to our interpolated curve, we have to recompute the curve anew (by solving the set of equations). Certainly, this is not an efficient solution.

In order to overcome the limitation of interpolation (i.e., fitting a single curve to the set of input points), we can use a different approach. We may partition the control points into subsets with fewer points (usually = 3). Next, we may fit lower degree polynomials to each of these subsets. These individual polynomials of lower degree (or polynomial pieces), when joined together, give the overall curve.

Figure 2.11 illustrates the idea of joining polynomial pieces to represent the overall curve. In the figure, we specified five control points (p_0[2] to p_4). The five points are divided into two subsets $p_0\ p_1\ p_2$ and $p_2\ p_3\ p_4$. Each subset is fitted with a quadratic (degree 2) polynomial. Together, they represent the whole curve.

The idea of fitting a set of control points with several polynomials (of lower degree) than a single higher degree polynomial is the spline representation. The curve that we get is called a spline curve or simply spline. In graphics, it is common to use splines that are made up of third degree ($n = 3$) or cubic polynomials. In subsequent discussions, we shall concentrate on such splines only.

2.3.1 Continuity Conditions

Since we are joining several polynomials in order to create a spline representation, it is important to ensure that they join smoothly so that the resulting curve looks smooth. The objective can be achieved if we can make the splines conform to the *continuity conditions*. Continuity conditions are of two types, namely, (a) parametric continuity and (b) geometric continuity. The *n*th order parametric continuity, usually denoted by C^n, states that the

[2]Note that each p_i is basically a vector (x_i, y_i) representing a 2D point.

adjoining curves meet and the first to nth order parametric derivatives of the adjoining curve functions are equal at their common boundary. Thus, C^0 indicates that the adjoining curves meet, C^1 indicates that the first order parametric derivatives of the adjoining curves at the common boundary are equal, C^2 indicates that both the first and the second order derivatives are equal, and so on. The parametric continuity conditions are sufficient, but not necessary to ensure geometric smoothness of the spline. For that, conformation to the geometric continuity conditions are required. The zero-order geometric continuity, denoted as G^0 is similar to C^0. G^1 or the first-order geometric continuity indicates that the tangent directions at the common boundary of adjoining curves are equal, but their magnitudes can be different. The second-order geometric continuity (G^2) states that both the tangent directions and curvatures at the common boundary of the adjoining curves are equal.

2.3.2 Types of Splines

While conforming to the parametric or geometric continuity conditions can ensure smooth curves, we also need to keep in mind the fact that the overall design should support local controllability. Otherwise, the advantages of having a spline representation instead of a single interpolated polynomial may be lost. In order to address these issues, different types of splines are used in computer graphics. Such splines are broadly of the following two types (illustrated in Fig. 2.12).

Interpolating splines We try to fit a spline that passes through all the control points. The three commonly used interpolating splines are the natural cubic splines, the Hermite cubic splines, and the Cardinal cubic splines.

Approximating splines In such splines, the curve passes closer to the control points. The control points define the boundary (known as the *convex hull*), which in turn determines the overall shape of the curve. The two common approximating splines are the cubic Bezier curves and the cubic B-splines.

Let's first discuss how we can represent a spline mathematically. Then, we shall delve deeper into the two types of splines.

There are two ways commonly used to represent splines: basis matrix and the basis/blending functions. The two are equivalent to each other. We can convert the basis

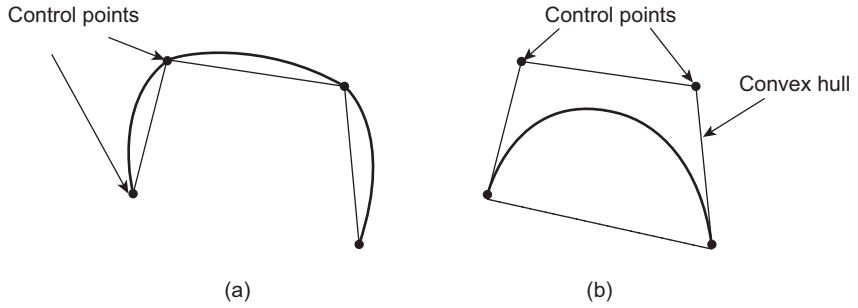


Fig. 2.12 Two types of splines (a) Interpolating spline (b) Approximating spline

matrix representation to the basis/blending functions representation and vice versa. We shall try to understand these representations in terms of a simple example.

2.3.3 Representation by Basis Matrix

Consider a polynomial of degree one, whose parametric form is shown in Eq. 2.5.

$$f(u) = a_0 + u.a_1 \tag{2.5}$$

In Eq. 2.5, we should keep in mind that a_i is a vector $(a_{ix}\ a_{iy})$ for x and y coordinates and the function $f(u)$ should have the corresponding form $[f_x(u)$ or $f_y(u)]$. For convenience, we shall work with this compact form rather than the form for individual coordinates. We can represent Eq. 2.5 in matrix form as $f(u) = U.A$ where U and A are the parameter and coefficient matrices, respectively as follows.

$$U = \begin{bmatrix} 1 & u \end{bmatrix},\ A = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

In order to determine the function of Eq. 2.5, we need to have at least two control points (i.e., $n+1$ control points to obtain a polynomial of degree n). Let us denote those two control points as p_0 and p_1. We shall make use of these points to parameterize the polynomial, which means that we shall assume certain parameter values for these control points. For example, we may assume that the two control points denote the values of the function at the boundary values of the parameter (i.e., at $u = 0$ and $u = 1$). Then we can set up the system of equations as shown in Eq. 2.6.

$$p_0 = f(u = 0) = a_0 + 0.a_1 \tag{2.6a}$$
$$p_1 = f(u = 1) = a_0 + 1.a_1 \tag{2.6b}$$

In matrix notation, we can express Eq. 2.6 as $P = C.A$ (dot product of two matrices) where the matrices P, C, and A are defined as follows.

$$P = \begin{bmatrix} p_0 \\ p_1 \end{bmatrix},\ C = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},\ A = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$$

Note how the matrix C is constructed. We took the coefficients of the a_i (from a_1 to a_n in that order) terms in each equation of the system of equations to form the corresponding row (first equation for first row, etc.). Since we have obtained C by imposing certain parameterization conditions (constraints) on the control points, C is called the *constraint* matrix.

Since $P = C.A$, we can say that $A = C^{-1}.P$. The inverse of the constraint matrix (C^{-1}) is called the *basis* matrix B. Thus, we can express the polynomial $f(u)$ as in Eq. 2.7.

$$f(u) = U.A = U.C^{-1}.P = U.B.P \tag{2.7}$$

Derivation of Eq. 2.7 demonstrates that the basis matrix for an interpolating polynomial that satisfies the parameterization conditions (constraints) is fixed. Hence, it can be used to

uniquely characterize the polynomial. Since a spline is made up of polynomial pieces, if each of these pieces are made from the same type of polynomial (i.e., degree and constraints are the same), then the overall spline can be uniquely characterized by the basis matrix of the polynomial pieces. That is the basis matrix representation of splines.

2.3.4 Representation by Blending Functions

If we expand the right-hand side of Eq. 2.7, we get a weighted sum of polynomials with the control points being the weights. For our specific example (of a linear polynomial), the expansion will result in Eq. 2.8.

$$f(u) = U.B.P = \begin{bmatrix} 1 & u \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = (1-u)p_0 + up_1 \tag{2.8}$$

The individual polynomials in the weighted sum, such as the terms $1-u$ and u in Eq. 2.8, are called the basis/blending functions. As in the case of basis matrix, the blending functions for a given polynomial are also fixed and hence can be used to characterize the polynomial. Hence, a spline made up of several pieces of the same polynomial type, can also be represented in terms of the blending functions of the constituent polynomials. Thus we can say that for our linear polynomial piece that satisfies the constraints, that the two control points represent the values of the function at the boundary values of the parameter u, is characterized by the set of blending functions $1-u, u$. The compact representation of a curve in terms of blending functions is shown in Eq. 2.9, where $n+1$ is the number of control points.

$$f(u) = \sum_{i=0}^{n} p_i b_i(u) \tag{2.9}$$

2.3.5 Interpolating Splines: Natural, Hermite, and Cardinal Cubics

One of the first splines used in computer graphics is the natural cubic splines. As the name suggests, the spline is made up of pieces of third degree polynomials. Each of these pieces is defined by four control points (p_0, p_1, p_2, and p_3). The first and the last points denote the two boundary points (i.e., for $u = 0$ and $u = 1$) of the polynomial piece while the other two are the first and second derivatives at $u = 0$.

The splines support C^2 continuity. In other words, p_3 of the ith piece should be the same as the p_0 of the adjoining $(i+1)$th piece. Also, the first and second derivatives of the ith piece at $u = 0$ should be the same as that of p_1 and p_2 of the $(i+1)$th piece.

A problem with the natural cubics is that it does not have *local controllability* (i.e., local changes require re-computing the whole curve). The Hermite cubics are another type of interpolating splines that support local controllability. However, these have only C^1 continuity (implying that they give less smooth curves than natural cubics). Each cubic piece of the Hermite splines is defined by four control points p_0, p_1, p_2, and p_3. The control points p_0 and p_2 are the values of the cubic at the parameter boundaries ($u = 0$ and $u = 1$), p_1 is the first derivative of the cubic at $u = 0$ and p_2 is the first derivative at $u = 1$.

Example 2.1

Derive the basis matrix for the natural cubic splines.

Solution First, we shall identify the set of constraints for the spline from its definition, as shown in the following set of equations.

$$\begin{aligned} p_0 &= f(u=0) = a_0 + 0.a_1 + 0^2.a_2 + 0^3.a_3 \\ p_1 &= f'(u=0) = 0 + 1.a_1 + 2.0^1.a_2 + 3.0^2.a_3 \\ p_2 &= f'(u=1) = 0 + 1.a_1 + 2.1^1.a_2 + 3.1^2.a_3 \\ p_3 &= f(u=1) = a_0 + 1.a_1 + 1^2.a_2 + 1^3.a_3 \end{aligned}$$

From these equations, we can set up the constraint matrix C as we have done in the previous section.

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The inverse of C is our basis matrix B. Hence, for natural cubics, the basis matrix is as below.

$$B_{natural} = C^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ -1 & -1 & -0.5 & 1 \end{bmatrix}$$

Since the natural cubic spline is made up of cubic polynomials that follow the same constraints, the whole spline can be represented in terms of the basis matrix, as we discussed in the previous section.

We can derive the basis matrix for the Hermite cubics in a similar way as that of the natural cubics. This process is left as an exercise. The basis matrix for Hermite cubics is given as follows:

$$B_{Hermite} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix}$$

At the expense of smoothness, we can have local controllabity with the Hermite cubic splines. However, a problem with these splines is that, we have to specify the first order derivatives as control points for the boundary values of each piece. This puts extra burden on the user. With the cardinal cubic spline, this problem can be overcome. Similar to the Hermite cubics, the cardinal cubics are also made up of polynomial pieces, each of which are defined by four control points. Among them, p_1 and p_2 are the boundary values of each

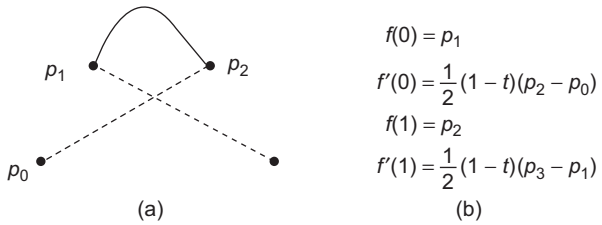


Fig. 2.13 (a) Illustration of the cardinal cubic spline pieces (b) The system of equations derived from the definition

piece (at $u = 0$ and $u = 1$). p_0 and p_3 are used to obtain the first order derivatives at the boundaries of each piece, as shown in Fig. 2.13.

The term t in Fig. 2.13(b), used to obtain the derivatives is called the *tension* parameter. It determines the shape of the curve. If $t = 0$, the corresponding spline is called the Catmull-Rom/Overhauser spline. Note that the cardinal cubics conform to the C^1 continuity.

The basis matrix, which we can obtain using a procedure similar to the one we described for the natural cubics, is given as follows, where $S = \frac{1-t}{2}$.

$$B_{cardinal} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -S & 0 & S & 0 \\ 2S & S-2 & 3-2S & -S \\ -S & 2-S & S-2 & S \end{bmatrix}$$

2.3.6 Approximating Splines: Cubic Bezier Curves and B-splines

Bezier curves are among the widely used representation techniques in computer graphics. Its name is derived from the name of the French engineer Pierre Bezier, who first used it to design Renault car bodies. Since a Bezier cubic is an approximating spline, the polynomial pieces do not pass through all the four control points (p_0, p_1, p_2, and p_3). Instead, each of these pieces originates at the first control point and end at the last control point (i.e., the first and the last control points are on the curve). The other two control points serve to determine the convex hull within which the curve lies. Moreover, using the two control points, we can define the first order derivatives of the polynomial piece at the boundary values of the parameter ($u = 0$ and $u = 1$), as shown in Eq. 2.10.

$$f'(u = 0) = 3(p_1 - p_0) \tag{2.10a}$$
$$f'(u = 1) = 3(p_3 - p_2) \tag{2.10b}$$

The Bezier cubics can also be described in terms of the equivalent blending function representation, as we have seen before. The general form of the blending function based representation of a Bezier curve with $n + 1$ control points is shown in Eq. 2.11.

$$P(u) = \sum_{k=0} p_k BEZ_{k,n}(u), \quad 0 \leq u \leq 1 \tag{2.11}$$

Example 2.2

Derive the basis matrix for the cubic Bezier curves.

Solution From the definition of the Bezier curves, we can establish the following set of constraint equations.

$$\begin{aligned} f(u=0) &= p_0 = a_0 + 0.a_1 + 0^2.a_2 + 0^3.a_3 \\ f'(u=0) &= 3(p_1 - p_0) = 0 + 1.a_1 + 2.0^1.a_2 + 3.0^2.a_3 \\ f(u=1) &= p_3 = a_0 + 1.a_1 + 1^2.a_2 + 1^3.a_3 \\ f'(u=1) &= 3(p_3 - p_2) = 0 + 1.a_1 + 2.1^1.a_2 + 3.1^2.a_3 \end{aligned}$$

We can rearrange the terms in these equations to get the following set of constraints.

$$\begin{aligned} p_0 &= a_0 \\ p_1 &= a_0 + \frac{1}{3}a_1 \\ p_2 &= a_0 + \frac{2}{3}a_1 + \frac{1}{3}a_2 \\ p_3 &= a_0 + a_1 + a_2 + a_3 \end{aligned}$$

Therefore, the constraint matrix is:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 \\ 0 & \frac{2}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The inverse of C is our basis matrix B. Hence, for natural cubics, the basis matrix is as follows.

$$B_{bezier} = C^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ 1 & 3 & -3 & 1 \end{bmatrix}$$

Since the natural cubic spline is made up of cubic polynomials that follow the same constraints, the whole spline can be represented in terms of the basis matrix, as we discussed in the previous section.

However, unlike the previous cases, the blending Bezier functions $BEZ_{k,n}(u)$ are of special nature. They are known as the Bernstein polynomial and are represented as in Eq. 2.12.

$$BEZ_{k,n}(u) = C(n,k)u^k(1-u)^{n-k} \tag{2.12a}$$

$$C(n,k) = \frac{n!}{k!(n-k)!} \tag{2.12b}$$

From Eq. 2.12, we can derive the blending functions for a cubic Bezier (i.e., $n = 3$) as shown in Eq. 2.13.

$$BEZ_{0,3}(u) = (1-u)^3 \tag{2.13a}$$
$$BEZ_{1,3}(u) = 3u(1-u)^2 \tag{2.13b}$$
$$BEZ_{2,3}(u) = 3u^2(1-u) \tag{2.13c}$$
$$BEZ_{3,3}(u) = u^3 \tag{2.13d}$$

Although the Bezier curves are a nice and elegant way to approximate any given shape, they do not support local controllability. The B-spline, on the other hand, can support C^2 continuity (i.e., very smooth curve) while at the same time has local controllability. The key to understand this concept is to recollect the notion of representing a polynomial in terms of other polynomials, as we have seen in the blending function representation of polynomial pieces (see Eq. 2.9 and associated discussion). The most general form of such representation is shown in Eq. 2.14.

$$f(t) = \sum_{i=1}^{n} p_i b_i(t) \tag{2.14}$$

Equation 2.14 states that the function f can be represented as a linear combination of blending functions b_i's with the control points p_i's serving as the coefficients. Note the notational changes from Eq. 2.9. Here, we are using t to denote the parameter instead of u (that means, the parameter range need not be [0,1]). Let each b_i be a polynomial. Clearly, we can think of f as a spline which is made up of polynomial pieces (b_i's). We can represent each b_i as a combination of other functions, like the overall function f. Then, conceptually each b_i is also a spline (as it is made up of polynomial pieces). We call such splines as B(asis)-spline. Thus, what we get is a spline curve made up of B-splines. That is the idea of B-spline (where the name actually indicates the constituent splines). Let us illustrate this in terms of an example.

Suppose we are given 4 control points. We want to fit a curve through these points. We have decided to use a linear B-spline (i.e., B-spline pieces made up of linear polynomials). Each B-spline will have 2 linear pieces as shown in Eq. 2.15 (we shall see later how to construct them).

$$b_i(t) = \begin{cases} t-i & i \le t < i+1 \\ 2-t+i & i+1 \le t < i+2 \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

Then, our curve function will be of the form shown in Eq. 2.16.

$$f(t) = p_1 b_1(t) + p_2 b_2(t) + p_3 b_3(t) + p_4 b_4(t) \tag{2.16}$$

Note that each B-spline is defined between a certain sub-interval of the parameter. Moreover, within this sub-interval, the pieces (constituents of the B-spline) have their own range within which they are defined. Such points in the parameter range, where a piece starts (or

ends) are called *knot points* or simply *knots*. For the *i*th B-spline, the knots are i, $i+1$, and $i+2$. Thus, for 4 control points, i takes the values 1 to 4. Hence, the knots are [1, 2, 3, 4, 5, 6]. This vector of knots, in increasing order of the parameter value, is known as the *knot vector*.

From this example, we can see several key characteristics of B-splines. Each B-spline is made up of k $(= d + 1)$ pieces, where d is the degree of the polynomials. The parameter ranges between 1 to $n + k$ with $n + k$ knots. In our example $n = 4, k = 2$, the range is between 1 to 6, with 6 knots. The *i*th B-spline is defined within the knots i and $i + k$. Thus, the 2nd $(i = 2)$ B-spline in our example is defined between 2 to 4. Each B-spline has $k + 1$ knots (in the example, each B-spline has 3 knots). As we can see, k is very crucial in determining B-spline characteristics. Hence, it is often called the B-spline parameter.

2.3.7 Types of B-Splines

Three types of B-splines are used in graphics. When the knots are uniformly spaced in the knot vector, we have *uniform B-spline* as in our example. If the spacing between consecutive knots in the knot vector is not same, we get *non-uniform B-splines*. A third type is called NURBS which stands for *non-uniform rational B-spline*. A NURBS is basically a ratio of two quantities as shown in Eq. 2.17, where h_i's are *scalar* weights and b_i's are non-uniform B-splines. Note that the same B-splines are used in both numerator and denominator.

$$f(t) = \frac{\sum_{i=1}^{n} h_i p_i b_{i,k}(t)}{\sum_{i=1}^{n} h_i b_{i,k}(t)} \tag{2.17}$$

So far, we have discussed the basic idea of B-splines and its characteristics. How can we determine the B-spline piecewise function for a given k and T (the knot vector)? We can use the Cox-De Boor recurrence relation, shown in Eq. 2.18 for the purpose.

$$b_{i,1,T}(t) = \begin{cases} 1 & T_i \le t < T_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{2.18a}$$

$$b_{i,k,T}(t) = \frac{t - T_i}{T_{i+k-1} - T_i} b_{i,k-1}(t) + \frac{T_{i+k} - t}{T_{i+k} - T_{i+1}} b_{i+1,k-1}(t) \tag{2.18b}$$

How can we use Eq. 2.18 to derive the B-splines? Let us illustrate with an example.

We can in fact derive the basis matrix for the uniform linear B-spline from the blending function representation. The trick is to transform the piecewise functions to parametric functions where the parameter range is [0, 1], as follows.

$$b_{i,2}(t) = \begin{cases} u & u = t - i, \quad i \le t < i + 1 \\ -u + 1 & u = t - (i + 1), \quad i + 1 \le t < i + 2 \end{cases} \tag{2.19}$$

Next, construct the basis matrix using the following rule. For a B-spline with parameter k, construct a $k \times k$ basis matrix. Let the basis matrix be $B = [C_1 C_2 \cdots C_n]$, where C_i denotes the *i*th column vector. Then, C_i contains the coefficients of the *i*th piecewise function from

Example 2.3

Illustrate the use of the Cox-De Boor recurrence relation.

Solution Let us try to derive the piecewise functions for the uniform linear B-spline (Eq. 2.14). Note that we have the knot vector $T = [1, 2, 3, \Omega]$, since we are dealing with a uniform spline. Therefore, $T_i = i$ and also $k = 2$. Then,

$$\begin{aligned} b_{i,2}(t) &= \frac{t-i}{(i+1)-i} b_{i,1}(t) + \frac{(i+2)-t}{(i+2)-(i+1)} b_{i+1,1}(t) \\ &= (t-i)b_{i,1}(t) + (i+2-t)b_{i+1,1}(t) \end{aligned}$$

Each B-spline on the right-hand side of this expression is non-zero only for a particular parameter range. For instance, $b_{i,1}(t)$ is non-zero only between i and $i+1$. So, if $i \leq t < i+1$, only the first of the B-splines in the expression is non-zero. Hence, we can rewrite the expression as follows.

$$C = \begin{cases} t-i & i \leq t < i+1 \\ 2-t+i & i+1 \leq t < i+2 \\ 0 & \text{otherwise} \end{cases}$$

This is the form we have seen in Eq. 2.15. Thus, using Eq. 2.18, we have derived the piecewise functions of the uniform linear B-spline.

the *bottom* in our listing of piecewise functions. Thus, C_1 is the coefficient of the last piece, C_2 is the coefficient of the 2nd last piece, and so on. In each column, the coefficients are arranged as $C_{i,0}$ = coefficient of the highest degree term, $C_{i,1}$ = coefficient of the next highest degree term and so on, where $C_{i,j}$ = the jth element of the ith column. Hence, for our example, we can set up the basis matrix as,

$$B_{linear\_bspline} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

2.3.8 Displaying Spline Curves

In the preceding sections, we have discussed the general idea of splines. Let us come back to the basic problem: how to fit a spline curve to a given set of control points? The simplest way to do this is to determine (interpolate) new control points using the spline equation. Then, we shall join the control points using line segments. The blending function representation can be used for the purpose.

As an illustration, suppose we are given two control points p_0 (1, 2) and p_1 (5, 6). We plan to use a linear interpolating spline to interpolate these points. We can create 3 new control points by evaluating Eq. 2.9 (the blending function representation) for 3 different values of the parameter u as shown in Fig. 2.14 (u takes the values 0.25, 0.5, and 0.75). Thus we now have five points. We can join these points using line segments to fit the curve. Instead of 3, we could have computed any number of new points. Instead of linear splines, we could have used any spline in the same way.

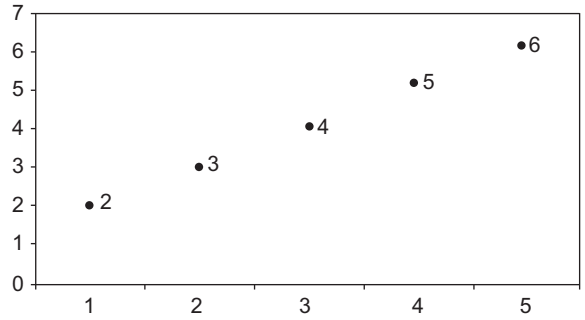


Fig. 2.14 The process of displaying a spline. In the process, new control points are computed using the blending function representation for different values of the parameter. Then, the points are joined with line segments.

Note that in the simple scheme, we have to evaluate the blending functions again and again. When we want to display cubic splines at a high rate, such computations may slow the rendering process. In fact, there are other methods that can reduce computations. We shall discuss one such method in the following algorithm, known as the De Casteljau algorithm or the subdivision method, which is used to render Bezier curves.

2.3.9 De Casteljau Algorithm

As we mentioned before, the subdivision method is useful for displaying Bezier curves. However, we can use this method to draw any spline, with some added computations. Suppose we are trying to fit a curve $f(u)$ to a given set of four control points $P_1 = \{p_1, p_2, p_3, p_4\}$. We have decided to use a cubic spline for the purpose. If the basis matrix of the spline is B_1 and the parameter matrix is u, then we can represent $f(u)$ as $f(u) = U.B_1.P_1$ (see Eq. 2.7 and associated discussion). Now, the same curve $f(u)$ can be obtained using another cubic spline with basis matrix B_2, albeit with a new set of control points $P_2 = \{p'_1, p'_2, p'_3, p'_4\}$. Thus, we can say, $UB_1P_1 = UB_2P_2$. From this equation, we can decide the control point set P_2 as in Eq. 2.20.

$$P_2 = B_2^{-1}B_1P_1 \tag{2.20}$$

Using this method, we can convert any spline curve to its equivalent Bezier representation by finding the new set of control points. For example, if we are given four control points (P_1) and we have used a uniform cubic B-spline (B_1) to fit the curve, we can compute the new set of control points (P_2) using the Bezier cubic basis matrix (B_2) and Eq. 2.20. On the set P_2, we can apply the subdivision method to compute the points on the curve.

2.3.10 Spline Surfaces

So far we have discussed the process of fitting a spline curve to a set of control points. How can we construct a surface? This can be done by extending the idea of spline curves. In order to define a surface, we need to define to a grid of control points. Using this grid, we can construct individual spline curves, which together shall form the surface, as shown in Fig. 2.15.

Example 2.4

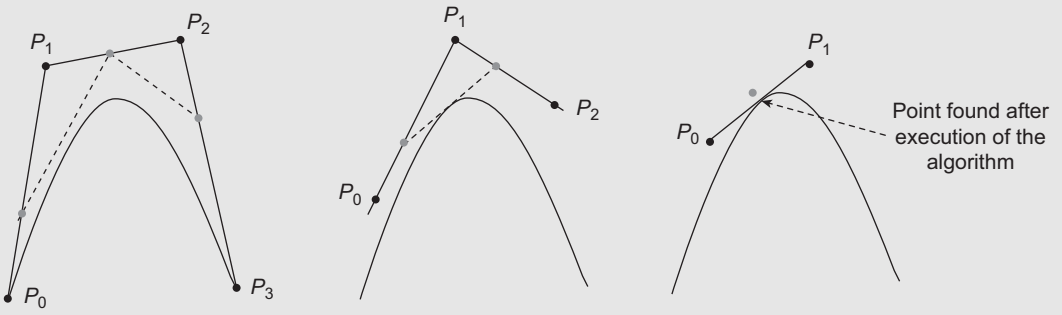
De Casteljau algorithm (Sub-division method)

Solution The basic idea of the method is as follows:
Input: n control points p_1 to p_n

Step 1: Join consecutive pairs p_i and p_j with line segments.
Step 2: Continue till a single point is obtained.

Step 2.1: Divide each line segment in the ratio d:1−d to get $n-1$ new points where d is any real number > 0.
Step 2.2: Join these new points with line segments.
Step 2.3: Go to Step 1.

The following figure shows an example of the method for $n = 4$ and $d = \frac{1}{3}$. In the figure, the greyed circles show the points found at each stage of the algorithm. The marked circle in the rightmost figure is the output point of the algorithm.



Note that in each run of the method, we get one new point. By executing the method m times for m values of d, we can compute m new points. When these points are joined with line segments, we get the Bezier curve.

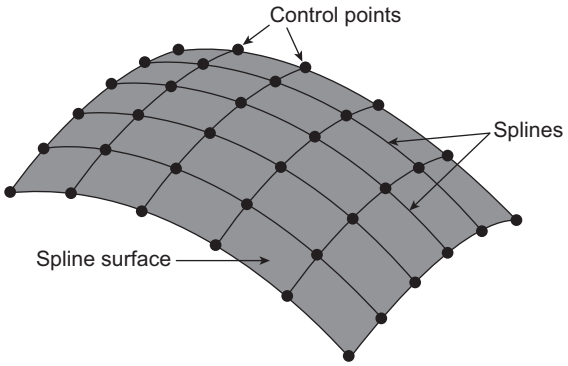


Fig. 2.15 A spline surface made from individual splines

2.4 SPACE-PARTITIONING REPRESENTATION

Space partitioning methods, as we mentioned before, address the problem of object representation through the use of the space (volume) enclosed by the boundaries. The simplest of all such representations is the voxel. A voxel, simply put, is a 3D equivalent of pixel. Like pixel grid, we can create a voxel grid to represent a 3D space. The voxels in the grid are uniform-sized cubes/parallelepipeds. Each voxel carries information such as intensity, temperature, and so on, such that they uniquely describe a 3D scene.

We can create such a grid using an Octree method. In this method, a 3D region is provided as input. A recursive procedure is invoked on this input space, to divide it into eight subregions and the recursion continues till we arrive at a pre-set uniform size of the subregions (say a unit cube). The recursive procedure creates a tree, each node of which contains eight children, hence the name octree. The leaf nodes of the tree represent the 3D space. The process is illustrated in Fig. 2.16. Figure 2.17(a) shows the voxel grid created using the octree in Fig. 2.17(b). By uniquely associating properties such as color, density, temperature, and so on, with each voxel, we can represent different objects in a 3D scene.

Binary space partitioning (BSP) is another way of representing space. Similar to the octree method, this method is also implemented as a recursive procedure. However, unlike the octree method, here in each step of the recursion, we divide the space into two. The division is performed by using planes, which may or may not be parallel to either of the XY, YZ, or ZX planes (i.e., planes formed by the principle axes). The result of the recursive procedure of the BSP method is the *BSP tree*, as illustrated in Fig. 2.17, in which the leaf nodes

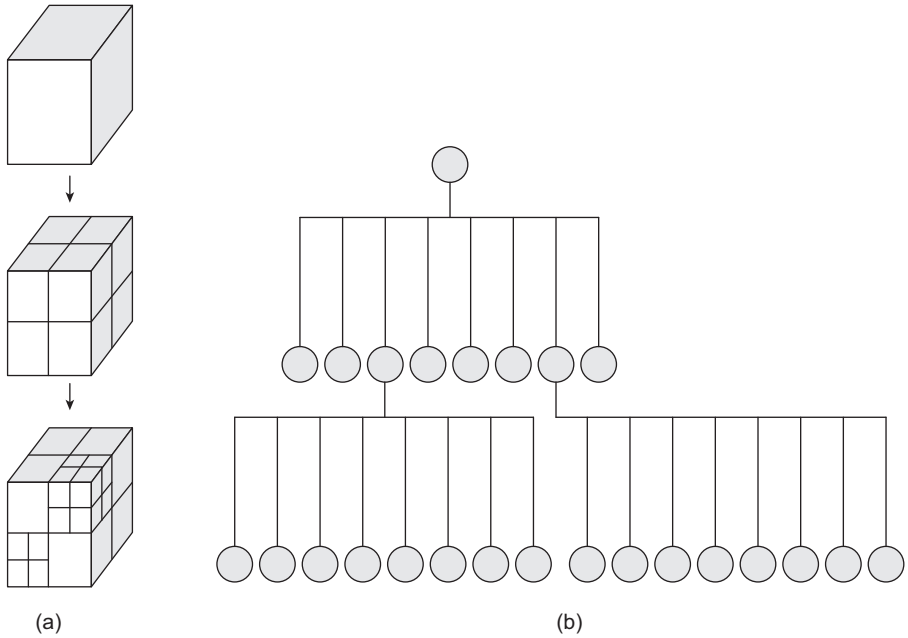


Fig. 2.16 A 3D space represented as voxels (a) Voxel grid (b) Octree method

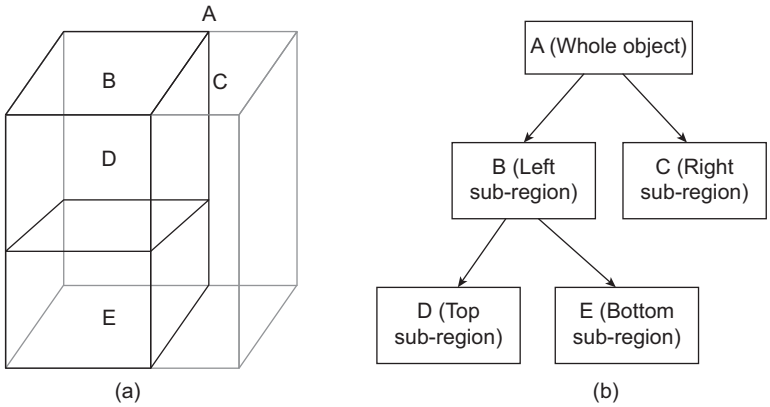


Fig. 2.17 (a) 3D space (b) Represented as BSP tree

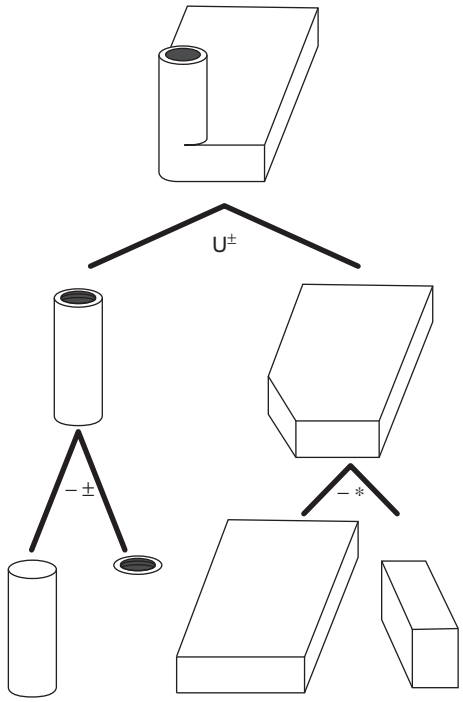


Fig. 2.18 Illustration of the CSG method—The leaf nodes are the primitive shapes

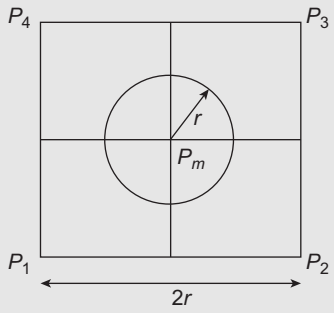
represent the 3D space. Note that in this example, the division is done using two orthogonal planes aligned to the XZ and YZ planes. However, this is not a strict requirement and planes of any orientation can be used for partitioning.

A problem with voxel-based representation is that the memory required to store the voxels of a voxel-grid is large. This is so since we are dividing the space to uniform-sized voxels irrespective of the space properties. For example, if a certain region in the space has the same properties everywhere, we will divide it into voxels nonetheless, although each of these voxels will have the same attributes (same color, temperature, density, etc.). We can save storage

Example 2.5

Construction of a BSP tree

Consider the following figure. We want to write a function createBSP() to represent the circle using a BSP tree.



Solution For constructing the tree, we shall take as argument the surrounding region R (the square in the figure) represented by the four corner points P_1, P_2, P_3, and P_4. We partition the region into four subregions in each call of the function. The partitioning is done using lines parallel to the axes. The pseudocode of the function createBSP(region) is shown here, assuming that the circle has radius r.

```
void createBSP(R)
  Add R as an intermediate node in the tree.
  Create four new regions R1–R4 by joining the midpoints of the sides of the square.
  for(int i = 0; i ≤ 4; i++)
   if size of Ri = unit square (assuming a pixel is represented as a unit square)
     if distance between the centers of R and Ri ≤ r
       Add Ri as a leaf node to the BSP tree and mark it as 'inside'
     else Add Ri as a leaf node to the BSP tree and mark it as 'outside'
   else
     Add Ri as an intermediate node in the tree
     createBSP(Ri)
```

space if, instead of using many voxels to represent a homogeneous region (i.e., a region in space with similar attributes), we use a single unit. In other words, we want to divide the space into non-uniform partitions.

One way to do that is to modify the octree method such that each internal node of the tree contains 0 or 8 children. In this modified method, we use the recursive procedure as before. In each step of the recursion, we divide the input space into eight regions based on the properties of the space. If the attributes of the whole 3D space are same everywhere, we do not divide it. The BSP method can also be adopted in a similar way. Thus, in the modified

BSP method, we shall either divide a region into two subregions or not (i.e., each node of the BSP tree shall have 0 or 2 children).

While octree or BSP methods are based on division of space, the constructive solid geometry (CSG) method relies on joining of spaces to represent objects. In this method, a set of primitive shapes are used. A set of Boolean operators (union, intersection, difference, etc.) are defined over the set of primitive shapes. The operators can be applied hierarchically over a number of shapes to represent complex objects, as illustrated in Fig. 2.18.

2.5 OTHER REPRESENTATIONS

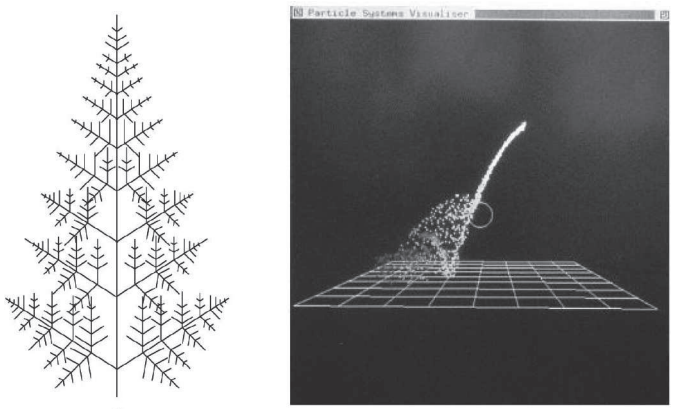
The techniques we have discussed so far are useful to approximate objects of different shapes and sizes. However, for complex shapes, we can use more advanced techniques.

Fractal Representation

Consider the shape of Fig. 2.19(a). While we can use any of the earlier techniques to represent the shape, more realistic rendering can be done if we use *fractal representation*. This is so due to the special properties of the shape: namely infinite details at every point and certain self-similarity between object parts and overall features of the object. Other examples of such shapes include mountains, clouds, corals, and snowflakes, for which fractal representation is suitable. Appendix B contains a more detailed discussion on the idea of fractals and its use in computer graphics.

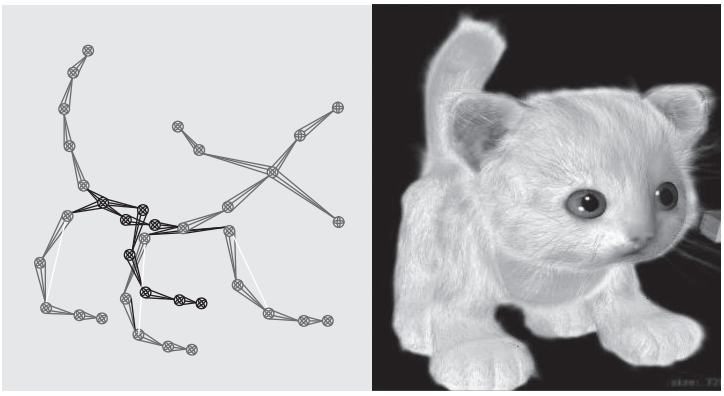
Particle System Representation

The fractal method is a procedural representation technique in which an object is represented as a procedure (i.e., a set of rules). *Particle systems* provide another representation technique, as illustrated in Fig. 2.19(b). In this technique, objects are represented as a set of



(a) (b)

Fig. 2.19 Advanced modeling techniques (a) Fern leaf, an example of fractal representation, a procedural method (b) Waterfall can be modeled using particle system



(a) (b)

Fig. 2.20 (a) A skeleton model (b) Subsequent rendering of animate figure (kitten)

particles, which move over time with some specified properties. The method is suitable to model phenomena such as waterfall, smoke, hair, cloth, and the like.

Skeletal Model Representation

A widely used representation technique in computer animation is the *skeleton* model. Contrary to what the name suggests, the technique can be used for representing vertebrate (human and non-human), invertebrate as well as inanimate objects. In this model, an object is represented as a hierarchy of 'bones' that forms the 'inner layer'. To these bones is attached the 'skin' (outer layer). When some operation is applied on some node in the bone hierarchy, rules are provided to decide the change in the overall bone structure as well as the skin. An example of the use of the skeleton model is shown in Fig. 2.20.

Scene Graph Representation

While the techniques discussed so far are primarily used to represent individual objects, the *scene graph* is used to represent the whole scene itself. A scene graph is basically a graph-like data structure, with individual objects serving as the nodes. The objects in the scene can be represented using any technique we have discussed so far. The scene graph structure can be used to specify the spatial and temporal properties of those objects in relation to the overall scene as well as to each other.

2.6 ISSUES IN MODEL SELECTION

It should be apparent from the preceding discussions that there is a rich variety of techniques used to represent objects/scenes in computer graphics. An obvious question is, which technique to use and when? The answer depends on many factors. While we always want to represent things in the most realistic way, use of advanced techniques such as particle systems is not always possible. This is so since each technique comes with a cost. The cost may be computational or storage requirements or both. Depending on the resources available, we have to make the decision. For example, although it is desirable to represent coastlines (of a sea in a scene) with fractals, if we are considering a mobile platform with limited

storage, limited capacity processor, and limited battery life, we may have to use some simpler methods than fractals. Of course, we lose something (the realism) but gain something (a working approximation).

Sometimes, we also consider the *ease of manipulation*. This is particularly important in animations where scenes change rapidly. If it requires a lot of time to manipulate (e.g., rotation, translation, scaling, clipping, projection, hidden surface removal, etc.) objects represented in a particular way, the quality of animation will be reduced. Hence, we should look for other simpler types in such cases.

Ease of acquiring data (e.g., how many data items we need to supply/provide for the representation) can be another important concern. For example, if we are using vertex list representation to represent a curved surface approximated using a mesh, we may need to provide a large number of vertices. Instead, using fewer control points, we can approximate the same surface with splines.

Thus, there will always be some trade-offs to manage. We have to take the decision (about choosing a particular representation technique) depending on the resources at hand (the computing platform), the nature of interaction (are we comfortable with supplying large amount of data or we don't want to compromise on that), and the effect we are satisfied with (are we looking for perfect photo-realism or will a good approximation be acceptable).

SUMMARY

What we have learnt in this chapter? The first thing is, in order to model various objects and phenomena in a synthesized scene, we can resort to a rich variety of techniques. While point-sample rendering and sweep representations have their use, more widely used are the boundary representation techniques and the space partitioning methods. The boundary representation techniques include mesh representation, implicit modeling, and parametric representations.

The spline representation, which is a type of parametric representation technique, is widely used in computer graphics to represent complex and irregular shapes. Both interpolating (natural cubics, Hermite cubics, and Cardinal cubics) and approximating (Bezier and B-splines) are used, each with their own advantages and disadvantages.

In space partitioning methods, uniform (voxels) and non-uniform space partitioning using the BSP trees or octrees are used. CSG is another space partitioning method for object representation, which works by applying Boolean operators on a set of primitive shapes.

Although the aforementioned methods can be used to approximate anything, we can make use of advanced techniques such as fractal geometry methods or particle systems to represent complex shapes more realistically. Moreover, scene graphs provide a mechanism to represent whole scenes in a systematic manner. On the other hand, the skeleton models are good to manipulate and hence are primarily used in computer animations where the scene changes rapidly.

Finally, we have briefly discussed the factors involved in addressing the trade-offs in choosing a representation scheme. The factors include the available resources (computational and storage), the desired accuracy and realism (what we get and if that is sufficient for the purpose), as well as the ease of manipulation (how easily we can manipulate the representations to perform various tasks such as transformations, clipping etc.).

Object representation is the first and the most basic stage of a graphics pipeline. On the objects thus defined, we perform several operations before these are rendered, as we have discussed in Chapter 1 (the pipeline). In the next chapter, we discuss the geometric modeling,

which is the next stage in our pipeline. This stage can be construed as the process of assembling different objects defined in their own co-ordinate systems (using any of the representation techniques discussed in this chapter) into a whole scene.

BIBLIOGRAPHIC NOTE

A large number of object representation techniques are used in computer graphics. We have discused a few of those at an introductory level. More techniques at such introductory level can be found in Hearn and Baker [2004] and Foley et al. [1995]. There are many good sources for in-depth understanding of the techniques. In fact, some of the broad techniques are full fields of study in itself with a large body of literature. A useful source of information on parametric curves and surfaces is Rogers and Adams [1990]. Detailed discussion on splines and their application to computer graphics can be found in Mortenson [1985] and Bartels et al. [1987]. A standard reference for a more mathematical discussion on splines is De Boor [2001]. Algorithms for quadtree and octree applications are given in Yamaguchi et al. [1984] and Brunet and Navazo [1990]. BSP-tree methods are presented in Gordon and Chen [1991]. Solid-modeling methods are discussed in Requicha and Rossignac [1992]. Mandelbrot [1977] is a good source on fractals. Reeves [1983] contains information on particle systems. Physically-based modeling methods are presented in Barzel [1992].

KEY TERMS

Approximating spline – the spline curve that passes close to the control points
Basis matrix – the inverse of a constraint matrix
Basis/blending functions – Another way of representing splines
Bazier curves – a type of approximating splines
Bernstein polynomials – the set of blending functions for a Bezier curve
Binary space partitioning (BSP) method – a space partition method that uses BSP data structure for object representation
Blobby objects – objects that are fluidic in nature such as a water droplet
Boundary representation – representing an object in terms of its boundary
Cardinal cubic splines – a type of interpolating splines
Constraint matrix – a matrix representation of constraints on the control points
Constructive solid geometry (CSG) – a space partitioning method that uses a set of basic objects and Boolean operators to represent arbitrary objects
Continuity condition – ensures that there is continuity at the joining points in joining more than one polynomial pieces in a spline
Control points – a set of points that define a spline
Cox-De Boor recurrence relation – recurrence relation to derive the basis functions for B-splines
Cubic B-splines – a type of approximating splines
De Casteljau algorithm – a sub-division method to display Bezier curves
Fractal method – a procedural technique for object representation
Geometric continuity – a set of conditions that ensures geometric smoothness of spline curves
Hermite cubic spline – a type of interpolating splines
Implicit representation – object representation techniques that use some parameters in the Eucledian space
Interpolating spline – the spline curve that passes through all its control points
Interpolation – the process of fitting a curve to a given set of points

Knot points/knots – points that mark the beginning or end of constituent polynomials of a B-spline
Knot vector – a vector representing all the knots
Local controllability – the ability to change a small portion of a curve shape by changing only a subset of the control points around that portion
Mesh representation – representation of an object in terms of a polygonal mesh
Metaball model – a type of blobby object representation technique
Natural cubic spline – a type of interpolating splines
Non-uniform B-splines – when the knot spacings are not the same
NURBS – a special type of non-uniform B-splines
Octree method – a space-partition method that uses octree data structure to represent objects
Parametric continuity – a set of conditions that ensures continuity of spline curves
Parametric representation – object representation techniques that use implicit functions
Particle system – a procedural technique for object representation
Point sample – a set of raw data points
Point-sample rendering – representing objects in terms of a set of raw data points
Quadric surfaces – surfaces that can be represented with quadratic equations
Scene graphs – hierarchical scene representation technique
Skeleton model – a popular technique used to represent objects in computer animation
Space partitioning – the technique to represent an object in terms of the space it occupies
Space-partitioning tree – a tree data structure used to represent the space occupied by an object
Spline – a mathematical abstraction, taken from the field of ship building, used to represent curves as a set of polynomial pieces joined together
Surface of revolution – One type of sweep representation technique
Sweep representation – a technique to represent an object in terms of a basic shape and its motion path
Sweep surface – one type of sweep representation technique
Uniform B-spline – when the knot spacings are same
Vertex/Edge list – special data structure used in mesh representation

EXERCISES

2.1 Find out at least two examples for each of the object representation techniques shown in Fig. 2.3, excluding those mentioned in this chapter.

2.2 Suppose you are designing a painting system, where the user can paint shapes in an interactive way using either of the following methods.

(a) Draw the shape by selecting from a library of pre-defined shapes.
(b) Draw the shape using a free-form curve shape (i.e., a curve that can be made to take any shape).

What representation method you should use to design the system? Discuss.

2.3 Consider the object shown in Fig. 2.2(b). Write an algorithm (in pseudocode) to represent the object.

2.4 Explain the concept of splines. Why are they useful?

2.5 What is the difference between interpolating and approximating splines? Discuss their merits and demerits.

2.6 Consider the control points (1,1), (6,3), (9,5), and (12,2). Perform the following.

(a) Determine three new control points using natural cubic, hermite cubic, and cardinal cubic splines.
(b) Determine three new control points using the subdivision method for cubic Bezier curves.

(c) Determine three new control points for fitting a uniform cubic B-spline using the subdivision method.

2.7 Determine the basis matrix for the uniform quadratic B-spline using the Cox-DeBoor relation.

2.8 Write an algorithm (in pseudocode) to represent a sphere using voxels.

2.9 Consider the BSP tree construction algorithm in Section 2.4. In the algorithm, we divide the space till the level of unit squares. Adapt this method such that the division occurs only when a region contains the object. Assuming $r = 4$, calculate the saving in space if the adapted method is used. Does it saves time also?

2.10 Consider Fig. 2.21. It shows an object made from two parallelepipeds including one with a circular hole in its middle. Determine the primitive shapes and operations required to construct this object using CSG method. Show the CSG tree.

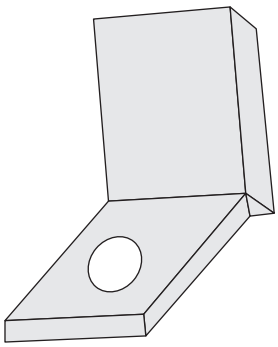


Fig. 2.21 Composite object of Exercise 2.10

2.11 What are the factors you should consider to decide about a particular representation method? Discuss.

2.12 Although we discussed many representation techniques, we also mentioned that all are first converted to mesh representation before rendering. Why do you think this is so?

CHAPTER 3

Modeling Transformations

Learning Objectives

After going through this chapter, the students will be able to

- Get an idea of modeling transformations
- Learn about the four basic modeling transformations—translation, rotation, shearing, and scaling—in both two and three dimensions
- Understand the homogeneous coordinate system used for representing modeling transformations
- Have a basic understanding of the matrix representations of modeling transformations
- Learn and derive composite transformations from the basic transformations through matrix multiplications, both in two and three dimensions

INTRODUCTION

We have come across different methods and techniques to represent objects in Chapter 2. These techniques, however, allow us to represent objects *individually*, in what is called *local/object* coordinates. In order to compose a scene, the objects need to be assembled together in the so called *scene/world* coordinate system. That means, at the time of defining the objects, the shape, size, and position of the object is not important. However, when individual objects are assembled in a scene, these factors become very important. Consequently, we have to perform operations to *transform* objects (from its *local* coordinate to the *scene/world* coordinate). The stage of the graphics pipeline in which this transformation takes place is known as the modeling transformation. Figure 3.1 illustrates the idea.

Thus, modeling transformation effectively implies applying some *operations* on the object definition (in local coordinate) to transform them as a component of the world coordinate scene. There are several such operations possible. However, all these operations can be derived from the following basic operations (the letter in the parenthesis beside each name shows the common notation for the transformations).

Translation (T) Translates the object from one position to another position

Rotation (R) Rotates the object by some angle in either the clockwise or anticlockwise direction around an axis

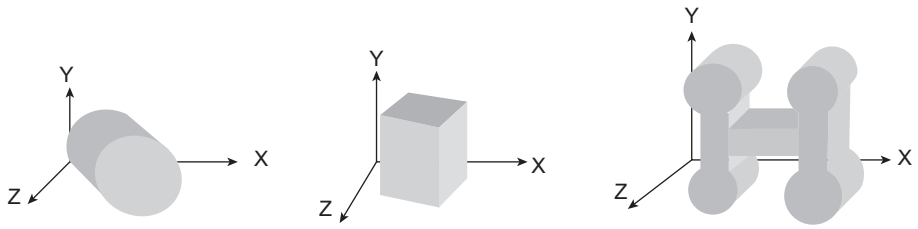


Fig. 3.1 In (a), two objects are defined in their *local* coordinates. In (b), these objects are assembled (in *world scene* coordinate) to compose a complex scene.
Note: In a scene, objects are used multiple times, at different places, and in different sizes. The operations required to transform objects from their local coordinate to world coordinate are collectively called modeling transformations.

Scaling (S) Reduces/Increases the size of the object

Shearing (Sh) Changes shape of the object (although, strictly speaking, this is not a basic transformation as it can be derived from a composition of rotation and scaling, we shall treat it as basic in this book)

As you can see, these operations change the geometric properties (shape, size, and location) of the objects. Hence, these are also called *geometric* transformations.

We shall adopt a different approach in describing the transformations. Although our main focus is 3D graphics pipeline, we start with the description of 2D transformations and then show how these are applied in 3D. This is done for simplicity and we shall take this approach in the later chapters also. In this chapter, we shall learn about the four basic modeling transformations, their representation, and the composition of these basic transformations to derive new transformations.

3.1 BASIC TRANSFORMATIONS

Among the four basic transformations, translation is the simplest. In order to translate a point $p(x, y)$ to a new location $p'(x', y)$, we displace the point by an amount t_x and t_y along the X and Y directions, respectively, as shown in Fig. 3.2. If the displacement is along the positive X-axis, it is positive displacement, otherwise the displacement is negative. The same is true for vertical displacements.

It is clear from the figure that the new point can be obtained by simply adding the displacements to the corresponding coordinate values. Thus, we have the relationships shown in Eq. 3.1 between the old and new coordinates of the point for translation.

$$x' = x + t_x \tag{3.1a}$$

$$y' = y + t_y \tag{3.1b}$$

Unlike translation where linear displacement takes place, rotation involves angular displacement. In other words, the point moves from one position to another on a circular track about some axis. For simplicity, let us assume that we want to rotate a point around the Z-axis counterclockwise by an angle ϕ. The scenario is shown in Fig. 3.3.

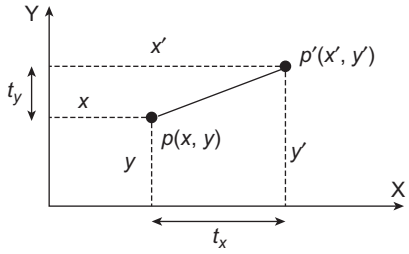


Fig. 3.2 Illustration of the translation operation

As the figure shows, the new coordinates can be expressed in terms of the old coordinates as in Eq. 3.2, where r is the radius of the circular trajectory.

$$x = r\cos\theta, y = rsin\theta \tag{3.2a}$$

$$x' = r\cos(\theta + \phi) = r\cos\theta\cos\phi - r\sin\theta\sin\phi = x\cos\phi - y\sin\phi \tag{3.2b}$$

$$y' = r\sin(\theta + \phi) = r\sin\theta\sin\phi + r\cos\theta\cos\phi = x\sin\phi + y\cos\phi \tag{3.2c}$$

By convention, counterclockwise angular movements are taken as positive, as in the aforementioned derivation, whereas clockwise movement is taken as negative. Thus, in case of clockwise movement, the displacement angle (ϕ) in Eq. 3.2 should be replaced with ($-\phi$).

In both the transformations (translation and rotation), we have shown the transformations applied on a point. For an object, we shall simply apply the operation on the points that make up the surfaces of the object (e.g., apply operation on all the points on the vertex list for a regular object or on the control points for an object defined with spline surfaces, etc.). Note that, in this way we can change the orientation of an object by applying rotation on the surface points.

With translation and rotation, we can change the position and orientation of objects. Scaling allows us to change (increase or decrease) the object size. Mathematically, scaling a point is defined as multiplying its coordinates by some scalers, called the *scaling factors*. Thus, given a point $p(x, y)$, we can scale it by a factor s_x along X-axis (direction) and s_y along Y-axis (direction) to get the new point $p'(x', y')$, as shown in Eq. 3.3.

$$x' = s_x x \tag{3.3a}$$

$$y' = s_y y \tag{3.3b}$$

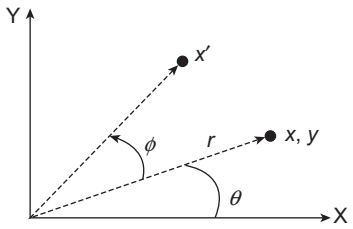


Fig. 3.3 The counterclockwise rotation operation about Z-axis by an angle ϕ

When the scaling factor is same along both the X and Y directions, the scaling is called *uniform*. Otherwise, it is *differential* scaling. Thus, in order to scale up/down any object, we need to apply Eq. 3.3 on its surface points, with scaling factor greater/less than one, as illustrated in Fig. 3.4. Note in the figure that the application of scaling *repositions* the object also (see the change in position of vertices in the figure).

We have so far seen transformations that can change the position and size of an object. With shearing transformation, we can change the shape of an object. The general form of the transformation to determine the new point (x', y') from the current point (x, y) is shown in Eq. 3.4, where sh_x and sh_y are the *shearing factors* along the X and Y directions, respectively.

$$x' = x + sh_x y \tag{3.4a}$$

$$y' = y + sh_y x \tag{3.4b}$$

Similar to the scaling operation, we can apply Eq. 3.4 to all the surface points of the object to shear it, as illustrated in Fig. 3.5. Note in the figure that shearing may also reposition the object like scaling (see the change in position of vertices in the figure).

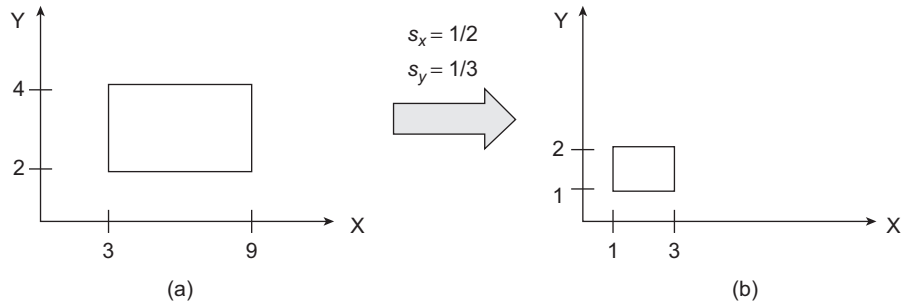


Fig. 3.4 Illustration of the scaling operation. Figure 3.4(a) is shrunk to the shape shown in Fig. 3.4(b). Note that the new vertices are obtained by applying Eq. 3.3 on the vertices of the object.

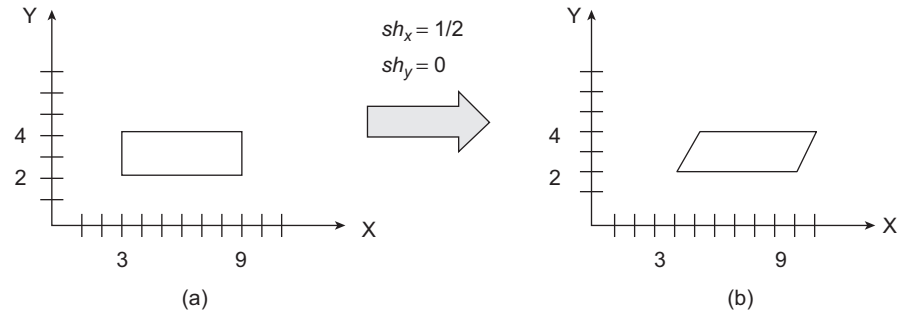


Fig. 3.5 The shearing of the object along horizontal direction (characterized by a positive shear factor along X-direction and a shear factor of 0 along the Y-direction). Figure 3.5(a) is distorted to the shape shown in Fig. 3.5(b). The new vertices are obtained by applying Eq. 3.4 on the vertices of the object.

3.2 MATRIX REPRESENTATION AND HOMOGENEOUS COORDINATE SYSTEM

The equation form of the transformations in Table 3.1 is not very useful in developing graphics packages/libraries in a modular way. Instead, an alternative equivalent representation is used, in which each of these transformations is represented as a matrix. For example, scaling can be represented in matrix form as $S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$. Given a point $P(x, y)$, we can represent it as a (column) vector $P = \begin{bmatrix} x \\ y \end{bmatrix}$. Then, the new vertices of a scaled object can be obtained by multiplying the matrices together (i.e., $P' = S.P$). A similar approach can be adopted for rotation and shearing also.

However, if we choose to adopt 2×2 matrices for representing transformations, problem occurs for translation. We cannot represent translation with a 2×2 matrix. In order to address this issue, we take recourse to one mathematical trick known as the *homogeneous coordinate system*. A homogeneous coordinate system is an abstract representation technique in which we represent a 2D point $P(x, y)$ with a 3-element vector $P_h(x_h, y_h, h)$, with the relationship $x = \frac{x_h}{h}, y = \frac{y_h}{h}$. The term h is the homogenous factor and can take any non-zero value.

Table 3.1 Four basic types of geometric transformations along with their characteristics

Transformations [Notation]	General form	Remarks
Translation $[T(t_x, t_y)]$	$x' = x + t_x$ $y' = y + t_y$	
Rotation $[R(\phi)]$	$x' = x \cos\phi - y \sin\phi$ $y' = x \sin\phi + y \cos\phi$	• ϕ = angle of rotation • Clockwise rotation → negative angle (replace ϕ with $-\phi$) • Anticlockwise rotation → positive angle
Scaling $[S(s_x, s_y)]$	$x' = s_x x$ $y' = s_y y$	• s_x, s_y: Scaling factors along the X and Y axes, respectively, can take any real value (=1 if no scaling) • Scaling factor > 1, size increases • Scaling factor < 1, size decreases • Along with size, position may change
Shear $[Sh = (sh_x, sh_y)]$	$x' = x + sh_x y$ $y' = y + sh_y x$	• sh_x, sh_y: Shearing factors along the X and Y axes, respectively, can take any real value (= 0 if no scaling). • Along with shape, position may change.

Homogeneous coordinate system

A convenient way to represent transformations in matrix form. It is an abstract 3D representation of 2D points (in general, *n*-D point represented by $n+1$ vector).

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow P_h \begin{bmatrix} x_h \\ y_h \\ h \end{bmatrix}$$

The homogeneous factor h can take any non-zero value. To get back the original coordinates, perform the following operations: $x = \frac{x_h}{h}, y = \frac{y_h}{h}$.

The point $(x_h, y_h, 0)$ is assumed to be at infinity and the point $(0, 0, 0)$.

Any point of the form $(x_h, y_h, 0)$ is assumed to be at infinity and the point $(0, 0, 0)$ is not allowed in this coordinate system.

When we represent geometric transformations in homogeneous coordinate system, our earlier 2×2 matrices will transform to 3×3 matrices (in general, any $N \times N$ transformation matrix is converted to $N + 1 \times N + 1$ matrix). Also, for geometric transformations, we consider $h = 1$ (in later chapters, we shall see other transformations with $h \neq 1$). With these changes, the matrices for the four basic transformations are given in Table 3.2.

Table 3.2 Matrix representation (in homogeneous coordinates) of the four basic geometric transformations

Transformations	Homogeneous matrix form
Translation $[T(t_x, t_y)]$	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$
Rotation $[R(\phi)]$	$\begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Scaling $[S(s_x, s_y)]$	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Shear $[Sh(sh_x, sh_y)]$	$\begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

3.3 COMPOSITION OF TRANSFORMATIONS

Consider Fig. 3.6. The Figure 3.6(a) shows an object (the rectangle ABCD with length 2 units and height 1 unit) in its local coordinate. This object is used to define the chimney of the house in Fig. 3.6(b) (in world coordinate scene).

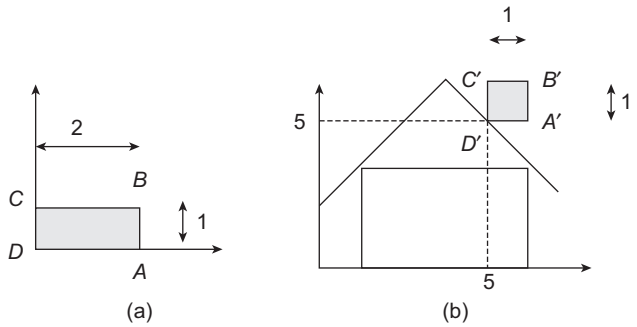


Fig. 3.6 Example of composite transformation. The object of the Fig. 3.6(a) is transformed to the object in Fig. 3.6(b), after application of a series of transformations

Clearly, the transformation of the object ABCD (in local coordinate) to A′B′C′D′ (in world coordinate) is not possible with a single basic transformation. In fact, we need two transformations: scaling and translation.

How we can calculate the new object vertices? We shall follow the procedure as before, namely multiply the current vertices with the transformation matrix. Only, here we have a transformation matrix that is the *composition* of two matrices, namely the scaling matrix and the translation matrix. The composite matrix is obtained by multiplying the two matrices *in sequence*, as shown in the following steps.

Step 1: Determine the basic matrices

Note that the object is halved in length while the height remains the same. Thus, the scaling matrix is, $S = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. The current vertex $D(0, 0)$ has now positioned at $D'(5, 5)$. Thus, there are 5 unit displacements along both horizontal and vertical directions. Therefore, the translation matrix is, $T = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix}$.

Step 2: Obtain the composite matrix

The composite matrix is obtained by multiplying the basic matrices in sequence. We follow the *right-to-left* rule in forming the multiplication sequence. The first transformation applied on the object is the rightmost in the sequence. The next transformation is placed on the left and we continue in this way till the last transformation. Thus, our composite matrix for the example is obtained as follows.

$$M = T.S = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix}$$

Step 3: Obtain new coordinate positions

Next, multiply the surface points with the composite matrix as before, to obtain the new surface points. In this case, we simply multiply the current vertices with the composite matrix to obtain the new vertices.

$$A' = MA = \begin{bmatrix} 0.5 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \\ 1 \end{bmatrix}$$

$$B' = MB = \begin{bmatrix} 0.5 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \\ 1 \end{bmatrix}$$

$$C' = MC = \begin{bmatrix} 0.5 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 1 \end{bmatrix}$$

$$D' = MD = \begin{bmatrix} 0.5 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \\ 1 \end{bmatrix}$$

Note that the results obtained are in homogeneous coordinates. In order to obtain the Cartesian coordinates, we divide the homogeneous coordinate values with the homogeneous factor, which is 1 for geometric transformations. Thus, the Cartesian coordinates of the final vertices are as follows:

$$\begin{aligned} A' &= (6/1, 5/1) = (6, 5), \\ B' &= (6/1, 6/1) = (6, 6), \\ C' &= (5/1, 6/1) = (5, 6), \text{ and} \\ D' &= (5/1, 5/1) = (5, 5) \end{aligned}$$

In composite transformations, we multiply basic matrices. We know that matrix multiplication is not commutative. Therefore, the sequence is very important. If we form the sequence wrongly, then we will not get the correct result. In the previous example, if we form the composite matrix M as $M = S.T$, then we will get the wrong vertices (do the calculations and check for yourself).

How have we decided in the previous example, the sequence, namely first scaling and then translation? Remember that in scaling, the position changes. Therefore, if we translate first to the final position and then scale, the vertex positions would have changed. In stead, if we first *scale with respect to the fixed point D (the origin)* and then translate the object (by applying the same displacement to all the vertices), then the problem of repositioning of the vertices will not occur. That is precisely what we did.

The example before is a special case where the fixed point was the origin itself. In general, the fixed point can be anywhere in the coordinate space. In such cases, we shall apply the aforementioned approach with slight modification.

Suppose we want to scale with respect to the fixed point $F(x, y)$. In order to determine the composite matrix, we assume the following sequence of steps.

1. The fixed point is translated to origin ($-$x and $-$y units of displacements in the horizontal and vertical directions, respectively).
2. Scaling is performed with respect to origin.
3. The fixed point is translated back to its original place.

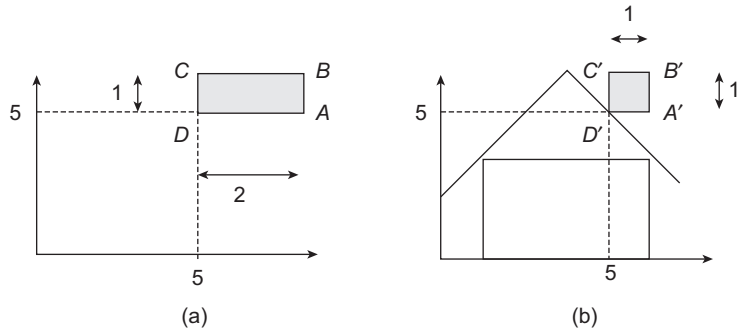


Fig. 3.7 Example of scaling with respect to an arbitrary fixed point (a) Object definition (b) Object position in world coordinate

Thus, the composite matrix $M = T(t_x = x, t_y = y).S(s_x, s_y).T(t_x = -x, t_y = -y)$. Figure 3.7 illustrates the concept. This is a modification of Fig. 3.6. In this, the object is now defined (Fig. 3.7(a)) with the vertices $A(7, 5)$, $B(7, 6)$, $C(5, 6)$, and $D(5, 5)$. Its world coordinate position is shown in Fig. 3.7(b). Note that the scaling is done keeping $D(5, 5)$ fixed. Hence, the composite matrix $M = T(t_x = 5, t_y = 5)S(s_x = 0.5, s_y = 1)T(t_x = -5, t_y = -5)$.

A similar situation arises in the case of rotation and shearing. In rotation, so far we assumed that the object is rotated around the Z-axis. In other words, we assumed rotation with respect to the origin through which the Z-axis passes. Similar to scaling, we can derive the rotation matrix with respect to any fixed point in the XY coordinate space through which the rotation axis (parallel to the Z-axis) passes. We first translate the fixed point to origin (aligning the axis of rotation with the Z-axis), rotate the object, and then translate the point back to its original place. Thus, the composite matrix is $M = T(t_x = x, t_y = y).R(\phi).T(t_x = -x, t_y = -y)$, where (x, y) is the fixed point coordinate. Composite matrix for shearing with respect to any arbitrary (other than origin) fixed point is derived in a similar way: $M = T(t_x = x, t_y = y).Sh(sh_x, sh_y).T(t_x = -x, t_y = -y)$.

What happens when more than one basic transformation is applied to an object with respect to any arbitrary fixed point? We apply the same process to obtain the composite transformation matrix. We first translate the fixed point to origin, perform the basic

Rotation, scaling, and shearing with respect to any arbitrary fixed point (other than origin)

The transformation matrix is obtained as a composition of basic transformations. All follow the same procedure.

1. Translate the fixed point to origin.
2. Perform the transformation (rotation/scaling/shearing).
3. Translate the fixed point back to its original place.

The transformation matrix at the fixed point (x, y) is

$$M = \begin{cases} T(t_x = x, t_y = y).R(\phi).T(t_x = -x, t_y = -y) & \text{Rotation} \\ T(t_x = x, t_y = y).S(s_x, s_y).T(t_x = -x, t_y = -y) & \text{Scaling} \\ T(t_x = x, t_y = y).Sh(sh_x, sh_y).T(t_x = -x, t_y = -y) & \text{Shearing} \end{cases}$$

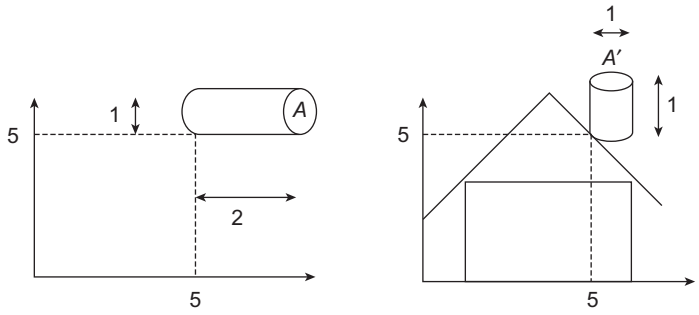


Fig. 3.8 Example of composite transformations with respect to an arbitrary fixed point (5,5). The object in Fig. 3.8(a) is transformed in Fig. 3.8(b). Note that two basic transformations are involved: scaling and rotation.

transformations in sequence, and then translate the fixed point back to its original place. An example is shown in Fig. 3.8.

Figure 3.8(a) shows the object (cylinder) with length 2 units and diameter 1 unit, defined in its own (local) coordinate. The cylinder is placed on the roof of the house in Fig. 3.8(b) (world coordinate), after scaling it horizontally by half and rotating it 90° anticlockwise with respect to the fixed point (5,5). How to compute the new (world coordinate) position of the object? We apply the approach outlined before.

Step 1: Obtain the composite matrix.

(a) Translate the fixed point (5,5) to origin.
(b) Scale by 1/2 in X-direction.
(c) Rotate anticlockwise by 90°.
(d) Translate the fixed point back to (5,5).

Composite matrix $M =$
$T(t_x = 5, t_y = 5)R(90^\circ)S(s_x = 0.5, s_y = 1)T(t - x = -5, t_y = -5)$

$$= \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & -5 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -10 & 10 \\ 0.5 & 0 & 2.5 \\ 0 & 0 & 1 \end{bmatrix}$$

Step 2: Multiply surface point (column vectors) with composite matrix to obtain new position (left as an exercise).

3.4 TRANSFORMATIONS IN 3D

Three-dimensional transformations are similar to 2D transformations, with some minor differences.

1. We now have 4×4 transformation matrices (in homogeneous coordinate system) instead of 3×3. However, the homogeneous factor remains the same ($h = 1$).
2. In 2D, all rotations are defined about Z-axis (or an axis parallel to it). However, in 3D, we have three basic rotations, with respect to each of the principle axes X, Y, and Z. Also,

the transformation matrix for rotation about any arbitrary axis (any axis of rotation other than the principle axes) is more complicated than in 2D.

3. The general form of the shearing matrix is more complicated than in 2D.

In shearing, we can now define distortion along one or two directions keeping one direction fixed. For example, we can shear along X and Y directions, keeping Z direction fixed. Therefore, the general form looks different from the one in 2D.

3.4.1 3D Shearing Transformation Matrix

In the general form, there are six shearing factors, each can take any real value or zero (if no shear along that particular direction). The factors sh_{xy} and sh_{xz} are used to shear along Y and Z directions, respectively, leaving the x coordinate value unchanged. Similarly, sh_{yx} and sh_{yz} are used to shear along X and Z directions, respectively, leaving the y coordinate value unchanged; The factors sh_{zx} and sh_{zy} are used to shear along X and Y directions, respectively, leaving the z coordinate value unchanged.

$$\text{Shearing}[sh_{xy}, sh_{xz}, sh_{yx}, sh_{yz}, sh_{zx}, sh_{zy}] = \begin{bmatrix} 1 & sh_{xy} & sh_{xz} & 0 \\ sh_{yx} & 1 & sh_{yz} & 0 \\ sh_{zx} & sh_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformation matrices for translation, rotation, and scaling in 3D are shown in Table 3.3, which are similar to their 2D counterparts, except that there are three rotation matrices in 3D.

The composite matrix for scaling and shearing with respect to any arbitrary fixed point is determined in a similar way as in 2D, namely translate the fixed point to origin, perform the operation and then translate the point back to its original place. Rotation about any arbitrary axis, however, is more complicated as discussed next.

Table 3.3 The matrix representation (in homogeneous coordinates) of the three basic geometric transformations in 3D

Transformations	Homogeneous matrix form
Translation $[T(t_x, t_y, t_z)]$	$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rotation about X-axis $[R_X(\phi)]$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

(Contd)

Table 3.3 (*Contd*)

Transformations	Homogeneous matrix form
Rotation about Y-axis $[R_Y(\phi)]$	$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Rotation about Z-axis $[R_Z(\phi)]$	$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Scaling $[S(s_x, s_y, s_z)]$	$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

3.4.2 3D Rotation About Any Arbitrary Axis

The schematic of 3D rotation about any arbitrary axis is shown in Fig. 3.9, where we want to rotate an object by an angle θ counterclockwise around an axis of rotation passing through the two points P1 and P2.

As shown in Fig. 3.9, the transformation matrix is a composition of five transformations.

1. Translation to origin ($T[-x, -y, -z]$: (x, y, z) is the coordinate of P2).
2. Alignment of the axis of rotation with the Z-axis (it can be X- or Y-axis also).
   (a) Rotation about X-axis, to put the axis of rotation on the XZ plane [$R_X(\alpha)$: α is the angle of rotation about X-axis].
   (b) Rotation about Y-axis, to align the axis of rotation with Z-axis [$R_Y(\beta)$: β is the angle of rotation about Y-axis].
3. Rotation of the object about Z-axis [$R_Z(\theta)$: *theta* is the angle of rotation of the object about the axis of rotation].
4. Reverse rotation about Y− and X-axes to bring the axis of rotation back to its original orientation.
5. Translation of the axis of rotation back to its original position.

Therefore, the composite transformation matrix is,

$$M = T^{-1}R_X^{-1}(\alpha)R_Y^{-1}(\beta)R_Z(\theta)R_Y(\beta)R_X(\alpha)T$$

Note that the inverse rotations essentially change the sign of the angle. For example, if α is defined counterclockwise, then the inverse rotation will be clockwise. In other words, $R_X^{-1}(\alpha) = R_X(-\alpha)$.

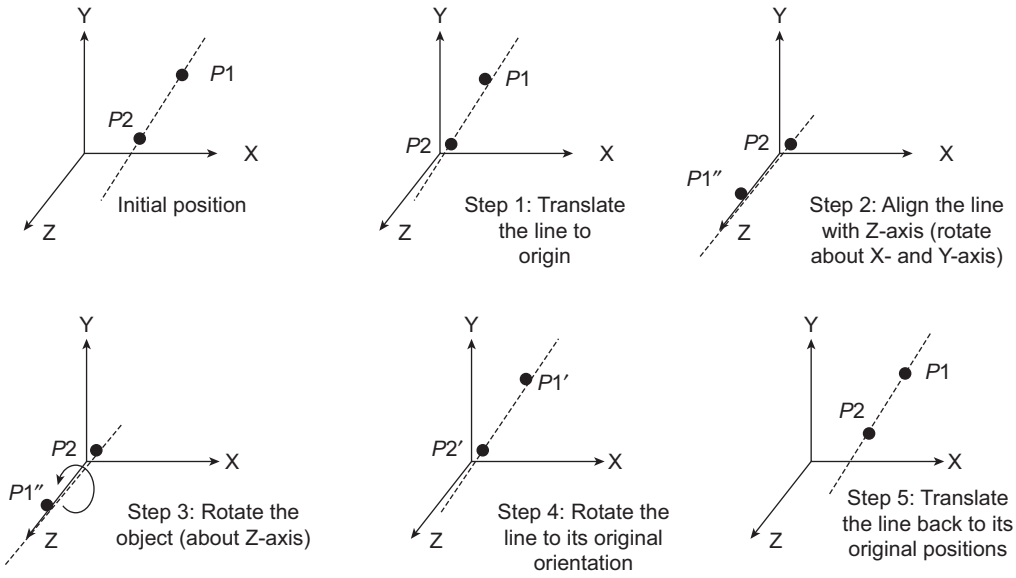


Fig. 3.9 Schematic of rotation about any arbitrary axis in 3D

Example 3.1

An object ABCD is defined in its own coordinate as $A(1, 1, 0)$, $B(3, 1, 0)$, $C(3, 3, 0)$, and $D(1, 3, 0)$. The object is required to construct a *partition wall* $A'B'C'D'$ in a world-coordinate scene (A' corresponds to A and so on). The new vertices are $A'(0, 0, 0)$, $B'(0, 4, 0)$, $C'(0, 4, 4)$, and $D'(0, 0, 4)$. Calculate the composite transformation matrix to perform the task.

Solution[1] The situation is depicted in Fig. 3.10.

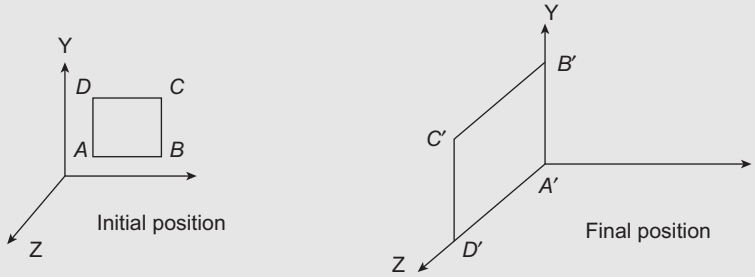


Fig. 3.10 The initial and final positions of the object

Initially the square is in the XY plane with each side equal to 2 units and center at (2, 2, 0). The final square is on the YZ plane with side equal to 4 units and the center at (0, 2, 2). The transformations required are as follows:

[1]It can be done in multiple ways. The solution presented here is one of those.

1. Translate center to origin → $T(-2, -2, 0)$.
2. Rotate by 90° anticlockwise around z-axis → $R_Z(90°)$.
3. Rotate by 90° anticlockwise around y-axis → $R_Y(90°)$.
4. Scale by 2 in Y and Z direction → $S(1, 2, 2)$.
5. Translate center to $(0, 2, 2)$ → $T(0, 2, 2)$.

The composite matrix $M = T(0, 2, 2)S(1, 2, 2)R_Y(90°)R_Z(90°)T(-2, -2, 0)$.

Example 3.2

Consider a circular track (assume negligible width, radius = 4 units, centered at origin). The track is placed on the XZ plane. A sphere (of unit radius) is initially resting on the track with center on the +Z axis, with a black spot on it at (0, 0, 6). When pushed, the sphere rotates around its own axis (parallel to Y axis) at a speed of 1°/min as well as along the track (complete rotation around the track requires 6 hrs). Assume all rotations are anticlockwise. Suppose an observer is present at (3, 0, 7). Will the black spot be visible to the observer after the sphere rotates and slides down for an hour and half?

Solution The situation is illustrated in Fig. 3.11.

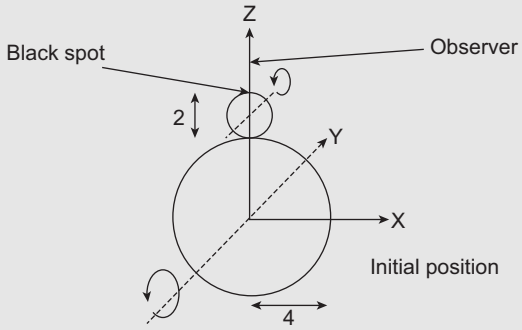


Fig. 3.11 Initial configuration

In the problem, we are required to determine the position of the black spot after one and half hours. We need to determine the transformation matrix M, which, when multiplied to the initial position of the black spot, gives its transformed location. Clearly, M is a composition of two rotations, one for rotation of the sphere around its own axis (R_{axis}) and the other for the rotation of the sphere around the circular track (R_{track}).

Since R_{axis} is performed around an axis parallel to Y-axis, we can formulate R_{axis} as a composition of translation (of the axis to the Y-axis), the actual rotation with respect to the Y axis and reverse translation (of the axis to its original place). Therefore, $R_{axis} = T(0, 0, 5).R_Y(\theta).T(0, 0, -5)$. Since the sphere can rotate around its axis at a speed of 1°/min, in one and half hours, it can rotate 90°. Therefore, $\theta = 90°$.

At the same time, the sphere is rotating along the circular track with a speed of $360°/6 = 60°$/hour. Therefore, in one and half hours, the sphere can move $90°$ along the track.
Therefore,

$$M = R_{track}(90°)R_{axis}(90°)$$
$$= R_Y(90°)T(0, 0, 5)R_Y(90°)T(0, 0, -5)$$
$$= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, the position of the point after one and half hours is

$$P' = MP = \begin{bmatrix} -1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ -1 \\ 1 \end{bmatrix}.$$

The transformed position of the black spot $(5, 0, -1)$ is clearly not visible from the observer's position.

SUMMARY

In this chapter, we have learnt to construct a scene from basic object definitions. The scene is constructed by applying transformations on the objects. In the process, we learnt the concepts of *local coordinate* (the coordinate system in which the object is defined) and *world coordinate* (the coordinate in which the objects are assembled to construct the scene).

There are four basic transformations, which are used either individually or in sequence to perform various transformations on an object. The four are translation, rotation, scaling, and shearing, which can change the position, shape, and size of an object.

While we can represent transformations in analytical form, it is more convenient to represent them as matrices for implementing graphics systems. In order to be able to represent all transformations in matrix form, we use the *homogeneous coordinate* system, which is essentially an abstraction and a mathematical trick. 2D transformations are represented as 3×3 matrices in homogeneous coordinate system.

When we compose two or more basic transformations, we follow the *right-to-left* rule, meaning that the first transformation is placed as the rightmost, the second transformation is placed on the left, and so on till the last transformation. Then we multiply the transformation

matrices together to obtain the composite transformation. However, while composing the basic transformations, it is very important to arrange them in proper sequence. Otherwise, the composite matrix will be wrong.

The transformations in 3D are performed in almost a similar manner as 2D transformations. The only notable differences are (a) the matrices are 4×4, (b) there are three basic rotation matrices for each of X, Y and Z axis (as opposed to one in 2D), (c) the way shearing matrix looks, and (d) the way rotation about any arbitrary axis takes place.

After the first two stages (object definition and geometric transformations), we now know how to construct a scene. The next task is to assign colors to it, so that it looks realistic. Color assignment is the next stage of the graphics pipeline, which we shall discuss in the next chapter.

BIBLIOGRAPHIC NOTE

The *Graphics Gems* series of books (Glassner [1990], Arvo [1991], Kirk [1992], Heckbert [1994] and Paeth [1995]) contains useful additional information on geometric transformation. Blinn and Newell [1978] contains discussion on homogeneous coordinates in computer graphics. More discussion on the use of homogeneous coordinates in computer graphics can be found in Blinn [1993].

KEY TERMS

Composite transformation – composition (by matrix multiplication) of two/more basic modeling transformations to obtain a new transformation
Differential scaling – when the amounts of change to an object size along the axial directions (X, Y, and Z) are not the same
Homogeneous coordinate system – an abstract mathematical technique to represent any n-dimensional point with a $(n + 1)$ vector
Local/Object coordinates – the Cartesian coordinate reference frame used to represent individual objects
Modeling transformation – transforming objects from local coordinates to world coordinates through some transformation operation
Rotation – the basic modeling transformation that changes the angular position of an object
Scaling factor – the amount of change of object size along a particular axial direction
Scaling – the basic modeling transformation that changes the size of an object
Scene/World coordinates – the Cartesian coordinate reference frame used to represent a scene comprising of multiple objects
Shearing factor – the amount of change in shape of an object along a particular axial direction
Shearing – the basic modeling transformation that changes the shape of an object
Translation – the basic modeling transformation that changes the position of an object
Uniform scaling – when the amounts of change to an object size along the axial directions (X, Y, and Z) are the same

EXERCISES

3.1 What is the primary objective of the modeling transformation stage? In which coordinate system(s) does it work?
3.2 Discuss how matrix representation helps in implementing modeling transformation in computer graphics.

3.3 Suppose you want to animate the movement of a pendulum, fixed at the point (0,5). Initially, the pendulum was on the Y-axis (along $-$Y direction) with its tip touching the origin. The movement is gradual with a rate of 10°/s. It first moves counterclockwise for 9 s, then returns (gradually) to its original position, then moves clockwise for 9 s, then again returns (gradually) to its original position and continues in this manner. Determine the transformation matrix for the pendulum in terms of t. Use the matrix to determine the position of the pendulum tip at $t = 15s$.

3.4 Derive the matrices for the following 2D transformations.

(a) Reflecting a point about origin
(b) Reflecting a point about the X-axis
(c) Reflecting a point about the Y-axis
(d) Reflecting a point about any arbitrary point

3.5 Explain homogeneous coordinate system. Why do we need it in modeling transformation?

3.6 Although we have treated the shearing transformation as basic, it can be considered as a composite transformation of rotation and scaling. Derive the shearing transformation matrix from rotation and scaling matrices.

3.7 Consider a line with end points $A(0, 0)$ and $B(1, 1)$. After applying some transformation on it, the new positions of the end points have become $A'(0, -1)$ and $B'(-1, 0)$. Identify the transformation matrix.

3.8 A triangle has its vertices at $A(1, 1)$, $B(3, 1)$, and $C(2, 2)$. Modeling transformations are applied on this triangle which resulted in new vertex positions $A'(-3, 1)$, $B'(3, 1)$, and $C'(2, 0)$. Obtain the transformation matrix.

3.9 A square plate has vertices at $A(1, 1)$, $B(-1, 1)$, $C(-1, 3)$, and $D(1, 3)$. It is translated by 5 units along the $+$X-direction and then rotated by 45° about $P(0, 2)$. Determine the final coordinates of the plate vertices.

3.10 Consider an object made up of a triangle ABC with $A(1, 2)$, $B(3, 2)$, and $C(2, 3)$, on top of a rectangle $DEBA$ with $D(1, 1)$ and $E(3, 1)$. Calculate the new position of the vertices after applying the following series of transformations on the object.

(a) Scaling by half along the X-direction, with respect to the point (2, 2)
(b) Rotation by 90° counterclockwise, with respect to the point (3, 1)

3.11 Consider Fig. 3.12. In this figure, the thin circular ring (with radius = 1 unit) is rolling down the inclined path with a speed of 90°/s. Assuming the ring rolls down along a straight

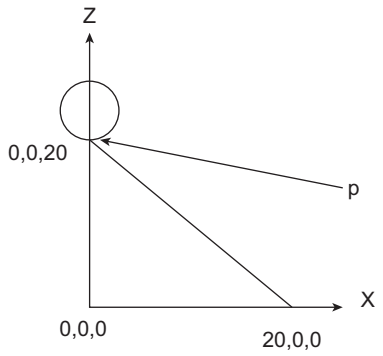


Fig. 3.12 Initial position of the ring

line without any deviation, determine the transformation matrix to obtain the coordinate of any surface point on the ring at time t. Use the matrix to determine the position of p at $t = 10$s.

3.12 Consider a sphere of diameter 5 units, initially (time $t = 0$) centered at the point (10,0,0). The sphere rotates around the Z-axis counterclockwise with a speed of 1°/min. An ant can move along the vertical great circular track (parallel to the XZ plane) on the sphere counterclockwise. It is initially ($t = 0$) located at the point (10,0,5) and can cover 1 unit distance along the track in 1 sec. (assume $\pi \approx 3$). Determine the composite matrix for the ant's movement and use it to compute the ant's position at $t = 10$ min.

CHAPTER 4

Illumination, Lighting Models, and Intensity Representation

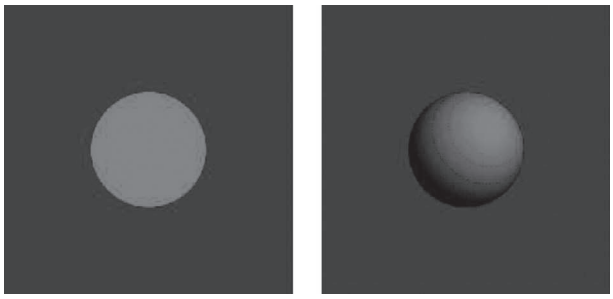
Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the fundamental activities performed in the illumination stage
- Learn about the concept of coloring of a point and the factors that influence our perception of color
- Understand the point light source and spotlight source
- Learn about the ambient, diffuse, and specular reflection types that contribute towards the perception of color
- Gain an understanding of the simple lighting model to compute color at a given point
- Learn about light attenuation, its effect on color perception, and the way it is computed
- Understand the flat, Gouraud, and Phong shading models
- Learn about the process of mapping the real color values to the device supported intensity levels represented as bit patterns
- Understand the concept of gamma correction
- Get an overview of the process of halftoning-used to represent multiple intensity levels in bi-level devices
- Learn about the dithering process and the Floyd–Steinberg error diffusion method for dithering

INTRODUCTION

In computer graphics, we are concerned about synthesizing an image. The image is nothing but a *snapshot* of a *scene* consisting of objects. So far, we have discussed the different methods and techniques to represent individual objects (Chapter 2) as well as putting the objects together to construct a 3D scene (Chapter 3). The next task is to assign colors *in an appropriate way* to the points in the scene. Without proper coloring, we will not be able to perceive *depth*, which is of utmost importance in creating the impression of three dimensions (see for example Fig. 4.1). The process of assigning appropriate colors to the points of a scene is conceptually the same as illuminating the scene with light, which is done with the help of lighting models. In this chapter, we shall discuss the process of assigning colors to a scene, which is the third stage of the 3D graphics pipeline.



(a) (b)

Fig. 4.1 In (a), the same color is applied to all the points on the ball. In (b), the ball is colored in a way such that we get the impression of 3D. The shading in (b) is what we are refering to as the *appropriate coloring* to give us an impression of 3D.

As we shall see, a lighting model *computes* colors of the surface points in terms of (continuous) intensity values (which are real numbers). We know that computers cannot work with continuous values, everything has be discretized in terms of 0s and 1s. Therefore, once the lighting model performs its task (of computing colors), we need to map the continuous values to some discrete representations that can be manipulated by the computer. In this chapter, we shall also discuss the mapping process.

4.1 BACKGROUND

When we talk about *color of a point*, we are basically referring to the following process: there is a (may be more than one also) source which emits light. The emitted light energy falls upon the point. Sometimes, the emitted light falls upon the point after getting reflected from the surrounding surfaces. The transport of the light energy from light source to the point is called *illumination*. The incident light then gets reflected from the point and reaches the eye of the viewer. The intensity of the light that reaches the viewer's eye is the *perceived color* or simply the *color* of the point. This process is illustrated in Fig. 4.2.

The process of computing the luminous intensity (i.e., outgoing light) at the point is known as *lighting*. In computer graphics, we are interested in simulating the lighting process. A related term is *shading* (also known as *surface rendering*), which refers to the process of assigning colors to pixels.

The luminous intensity or color of a point depends on both the properties of the light source and the surface on which the point lies. The surface properties include optical properties such as the reflectance and refractance as well as the geometric attributes such as the position and orientation with respect to the light source. In computer graphics, typically three types of light sources are considered for simulation of the lighting process: the *point light source*, the *directional light source* or *spotlight*, and the *ambient light*.

A point light source emits light equally in all directions from a single point (i.e., does not have any dimension). Such sources are characterized by their position and the intensity of

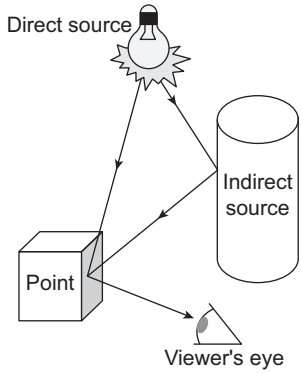


Fig. 4.2 The picture depicts the process behind the perception of color of a point. The light emitted from the light bulb (direct source) as well as coming after reflection from the cylinder (indirect source) is incident on the point. It then gets reflected from the point and comes to the viewer's eye, which gives the viewer the sensation of color at the point

the emitted light. Infinitely distant light sources such as sunlight can be modeled with point light source. In such cases, the light source is characterized by only the intensity of the emitted light. The directional light sources or spotlights are used to simulate the *beam of light* effect. A spotlight consists of a point light source, which emits light within an angular limit, as shown in Fig. 4.3. Points within this limit are illuminated while outside points are not. Thus, the spotlights are characterized by the position of the light source, the angular limit, and the intensity of the emitted light.

Even if an object is not illuminated directly by some light source (such as a point source or spotlight), we are still able to see it. This is a result of indirect illumination from emitters, which are the surrounding surfaces that reflect light. For example, the cylinder in Fig. 4.2 is one such emitter. It is usually very expensive to calculate (in real time) the contribution of those emitters to the color of a point. To avoid expensive calculations, a simplification

How is shading different from lighting?

Both the terms lighting and shading refer to the process of computing outgoing light (the color) at a point. However, in the context of computer graphics, the two terms represent two different ways of computing colors at a point. Lighting refers to the computation of colors at a point taking into account the properties of the light source and the surface on which the point lies. In other words, lighting refers to the simulation of the optical phenomenon. Usually, this is done through some model of lighting, which involves lots of computations as we shall see in subsequent discussions. In computer graphics, generating a realistic image involves computation of color at a large number of points of the scene. Consequently, applying lighting models for the purpose will be inappropriate to generate a realistic 3D image in real-time. In order to alleviate this problem, an alternative approach is used. In the alternative approach, the object surfaces are first mapped to pixels. Lighting model is used to compute colors for a selected small number of pixels (surface points). Those color values are then used to *interpolate* the colors of the remaining pixels that are part of the surfaces. The interpolation involves much less computation, thereby saving lots of time. This interpolation-based alternative process of pixel coloring is called shading.

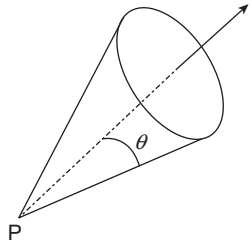


Fig. 4.3 The spotlight, characterized by the point light source P and the angular limit θ

is used in computer graphics, called the ambient light source. Ambient sources do not have any spatial or directional characteristics. They are assumed to illuminate all surfaces equally and are characterized by the ambient light intensity.

4.2 SIMPLE LIGHTING MODEL

In the preceding discussion, we saw that the color of a point is basically the intensity of the light reflected from that point reaching the viewer's eye. Therefore, we need to calculate the intensity of the reflected light to assign color to the point. This is done with the lighting models. In this section, we shall discuss a simple lighting model. In order to explain the model, we shall first assume monochromatic light and a single point light source. Later on we shall generalize the idea.

4.2.1 Simple Model for Monochromatic Single Point Light Source

Let us have a relook at Fig. 4.2. We can see that the light reflected from the point is a result of two incident lights: one coming from the light source (direct light) and the other coming from the cylinder (ambient light). Thus, the reflected light intensity is a *sum* of the intensities of the ambient light reflection and the direct light reflection. Reflection from a point can occur in either of the two ways: *diffuse reflection* and *specular reflection*. From a rough or grainy surface, incident light tends to reflect in all directions. This is known as diffuse reflection. Diffuse reflection can occur for both the direct light source and the ambient light. For a shiny/smooth surface, however, light gets reflected in a specific direction (or region). If a viewer is situated within that region (or along the direction), a bright spot is visible. This is known as specular reflection. These different types of reflections are shown in Fig. 4.4.

What Fig. 4.4 shows is that the intensity of the light reaching the eye of a viewer from a point can be modeled as the sum of three intensities.

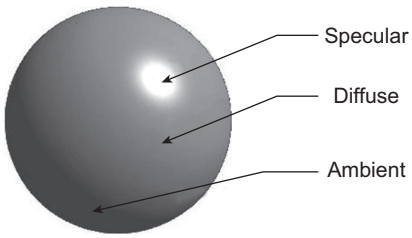


Fig. 4.4 The figure illustrates how the color of a point depends on the light source (direct/ambient) and reflection types (specular/diffuse)

1. Intensity of the ambient light after diffuse reflection from the point (I_{amb}).
2. Intensity of the light from the light source after diffuse reflection from the point (I_{diff}).
3. Intensity of the light from the light source after specular reflection from the point (I_{spec}).

In other words, the intensity (color) of a surface point (I_p) can be represented as shown in Eq. 4.1.

$$I_p = I_{amb} + I_{diff} + I_{spec} \tag{4.1}$$

The reflected light intensity is a fraction of the incident light intensity. This fraction is determined by a surface property, known as *reflection coefficient* or *reflectivity*. In order to control the lighting effect in our synthesized image, we shall define the following three reflection coefficients. For either of these coefficients, the value varies between 0.0 (representing dull surface with no reflection) and 1.0 (representing shiny surface that reflects almost all the incident light).

1. The diffuse reflection coefficient for ambient light k_a.
2. The diffuse reflection coefficient for direct light k_d.
3. The specular reflection coefficient for direct light k_s.

In order to model the contribution of the ambient light intensity, we shall assume that there is an ambient light in the scene with intensity I_a (that means, every surface is fully illuminated by the scene's ambient light I_a). Therefore, we can use Eq. 4.2 to compute the intensity contribution of the ambient light in the overall intensity of the reflected light.

$$I_{amb} = k_a.I_a \tag{4.2}$$

The contribution due to the diffuse reflection of the direct light is modeled by making an important assumption, namely that all the surfaces in the scene are *ideal diffuse reflectors* or *Lambertian reflectors*. Such surfaces follow the *Lambert's cosine law*, which states that *the energy reflected by a small portion of a surface from a light source in a given direction is proportional to the cosine of the angle between that direction and the surface normal*. The law implies that the amount of incident light from a light source on a Lambertian surface is proportional to the cosine of the angle between the surface normal and the direction of the incident light (the angle is called the *angle of incidence*, see Fig. 4.5). Assume that there is a direct light source emitting light with intensity I_s. The angle of incidence at the point is θ. Then, the amount of light incident on the point, according to the Lambert's law, is $I_s \cos\theta$. Then, the contribution to the overall intensty of a point by the diffuse reflection of the light coming from a direct light source can be computed using Eq. 4.3.

$$I_{diff} = k_d.I_s \cos\theta \tag{4.3}$$

Let $\hat{L}$ and $\hat{N}$ denote the unit direction vector to the light source from the point and the unit surface normal vector at the point, respectively. Then, $\cos\theta = \hat{N}.\hat{L}$. Hence, we can rewrite Eq. 4.3 as Eq. 4.4.

$$I_{diff} = \begin{cases} k_d I_s(\hat{N}.\hat{L}) & \text{if } \hat{N}.\hat{L} > 0 \\ 0 & \text{if } \hat{N}.\hat{L} \leq 0 \end{cases} \tag{4.4}$$

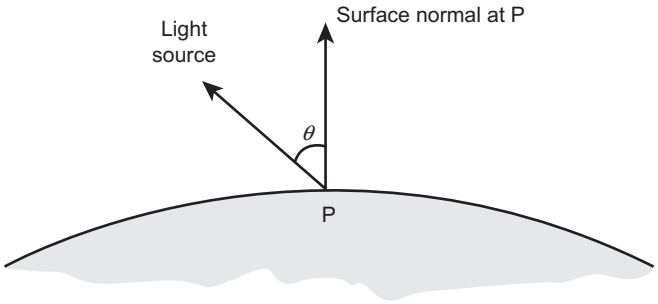


Fig. 4.5 The figure illustrates the idea of Lambertian surface. The amount of incident light at P from the light source is proportional to cos θ where θ is the angle of incidence

It may be noted that in modeling the ambient light contribution (Eq. 4.2), the position of the viewer and the light source are not considered. The light source position becomes important in modeling contribution due to diffuse reflection of the light from the light source (Eq. 4.4). The other component of the intensity of the reflected light at a point is the specular reflection intensity I_{spec}. Computation of I_{spec} involves consideration of the light source position as well as the viewer's position. Let us try to understand the I_{spec} computation using Fig. 4.6, which introduces the notions involved.

Phong Specular Reflection Model

In order to compute the specular intensity I_{spec}, we shall use the Phong specular reflection model, which is an empirically derived model proposed by Bui Tuong Phong in 1973. The model states that 'the intensity of specular reflection is proportional to the cosine of the angle (ϕ in Fig. 4.6) between the viewing vector ($\vec{V}$ in Fig. 4.6) and the specular reflection vector ($\vec{R}$ in Fig. 4.6), raised to a power'. In other words, I_{spec} is proportional to $\cos^{n_s}\phi$ ($0^\circ \leq \phi \leq 90^\circ$). The quantity n_s is the *specular reflection exponent*, which allows us to generate different types of reflection effects by setting appropriate values. Shiny surface effect can be generated by setting large values (> 100) to n_s, whereas n_s values closer to 1 generates the effect of reflection from rough surfaces.

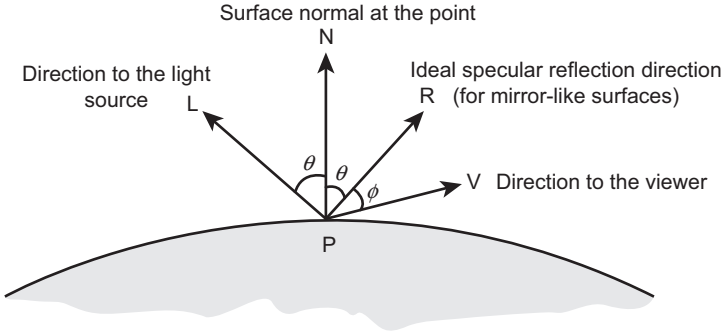
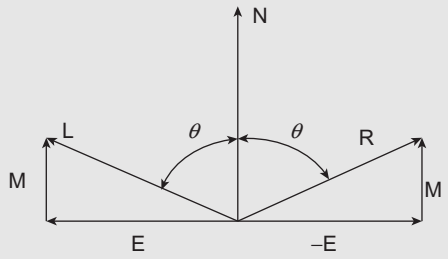


Fig. 4.6 Illustration of the idea of specular reflection and specular intensity (I_{spec}) computation. The angle θ between $\vec{N}$ and $\vec{R}$ in Fig. 4.6 is the *specular reflection angle*, which is equal to the angle of incidence (the angle between $\vec{L}$ and $\vec{N}$) according to Snell's law.

Representing $\hat{R}$ in terms of $\hat{N}$ and $\hat{L}$

The ideal specular reflection direction $\hat{R}$ in Eq. 4.5 is more conveniently represented in terms of $\hat{N}$ and $\hat{L}$. The idea is illustrated in the following figure.



In the figure, L and R are the light direction and ideal specular reflection direction vectors, respectively and N is the surface normal. E and M are two auxilary vectors. Note that $\vec{R} = \vec{M} - \vec{E}$. Since $\vec{L} = \vec{E} + \vec{M}$, we have $\vec{E} = \vec{L} - \vec{M}$. Therefore, $\vec{R} = \vec{M} - (\vec{L} - \vec{M}) = 2\vec{M} - \vec{L}$.

Note that $\vec{M}$ is simply the projection of $\vec{L}$ onto $\vec{N}$. Thus, we can derive that $\vec{M} = \frac{\vec{L}.\vec{N}}{|\vec{N}|^2}\vec{N}$. Since $\vec{N}$ is unit normal, $|\vec{N}|^2 = 1$. Thus, $\vec{M} = (\vec{L}.\vec{N})\vec{N}$. Hence, $\vec{R} = 2(\vec{L}.\vec{N}).\vec{N} - \vec{L}$.

Using the Phong model, we can compute the specular intensity contribution as $I_{spec} = k_d.I_s \cos^{n_s}\phi$. However, note that $\cos\phi = \hat{V}.\hat{R}$. Therefore, we can use Eq. 4.5 to compute I_{spec}.

$$I_{spec} = \begin{cases} k_s I_s(\hat{V}.\hat{R})^{n_s} & \text{if } \hat{V}.\hat{R} > 0 \\ 0 & \text{if } \hat{V}.\hat{R} \leq 0 \end{cases} \tag{4.5}$$

4.2.2 Intensity Attenuation

The computations we have discussed so far assume that the intensity of the light does not change as it moves from the source to the surface point. Now assume there are two surface points, one closer to the light source than the other. The intensity of the light received by either of these points will be the same. Hence, the color computed at these points using our simple lighting model will be the same. In other words, all surfaces are illuminated with equal intensities (assuming same optical characteristics), irrespective of their distance from the light source. Obviously, this is not desirable as it leads to *indistinguishable* overlapping of the surfaces when those are projected on screen.

In order to overcome this problem, we incorporate *intensity attenuation* in our model in the form of *attenuation factors*. Two such factors are used: *radial attenuation factor* (AF_{rad}) and *angular attenuation factor* (AF_{ang}). The radial factor accounts for the effect of the diminishing light intensity over distance. The inverse quadratic function shown in Eq. 4.6 is typically used to compute the radial attenuation factor. In Eq. 4.6, d is the distance between the light source and surface point; the coefficients a_0, a_1, and a_2 are varied to produce better realistic effects.

$$AF_{rad} = \frac{1}{a_0 + a_1 d + a_2 d^2} \tag{4.6}$$

The angular attenuation, on the other hand, is useful primarily to generate the spotlight effect. The factor helps us take into account the fact that the light intensity changes as the

angle between the spotlight cone axis and the surface point direction changes (i.e., the angle θ in Fig. 4.3). A commonly used function to compute the angular attenuation is shown in Eq, 4.7, where the angular attenuation exponent a_l is assigned some positive value.

$$AF_{ang} = \begin{cases} 0 & \text{if the surface point is outside the angular limit } \theta \\ \cos^{a_l}\phi & 0 \leq \phi \leq \theta \end{cases} \tag{4.7}$$

In order to take intensity attenuation into account, we adjust Eq. 4.1 as shown in Eq. 4.8. Note that by setting the factor value to 1.0, we can eliminate the effect of attenuation from our lighting calculation.

$$I_p = I_{amb} + AF_{rad}.AF_{ang}[I_{diff} + I_{spec}] \tag{4.8}$$

4.2.3 Simple Model for Colored Light

With monochromatic light source, we can generate different shades of gray. In order to generate color image, we have to consider *color light source*. In a color light source, the emitted light intensity has three components corresponding to the primary colors red, green, and blue. In other words, light source intensity is a three element vector: $I_s = (I_{sR}, I_{sG}, I_{sB})$. The reflection coefficients are also specified as vectors. Thus, we specify $k_a = (k_{aR}, k_{aG}, k_{aB})$, $k_d = (k_{dR}, k_{dG}, k_{dB})$ and $k_s = (k_{sR}, k_{sG}, k_{sB})$. Each component of the color is then calculated separately by applying Eq. 4.8 with the appropriate source intensity and the coefficient values.

4.2.4 Simple Model for Multiple Point Light Sources

With the model represented by Eq. 4.8, we can generate the lighting effects in the presence of a single light source. We may wish to generate the effect of illumination by multiple (point) light sources. In such cases, Eq. 4.8 takes a more general form as in Eq. 4.9, where diffuse and specular reflections due to each (ith) light source are computed separately and then added together.

$$I_p = I_{amb} + \sum_{i=1}^{n}[AF_{rad}AF_{ang}(I_{diff} + I_{spec})]_i \; n = \text{total number of light sources} \tag{4.9}$$

Note that for multiple colored light sources, Eq. 4.9 is applied for individual components (R, G, and B) of the source intensities.

Local vs _global_ lighting models

The simple lighting model we discussed is an example of *local lighting models*. In such models, emphasis is given on the direct impact of the light source in determining the color of a point. The ambient light sources are considered in a very simplified manner. In contrast, a *global lighting model* tries to mimic the illumination process more accurately by taking into account secondary effects such as the movement of light through transparent/translucent material (i.e., both reflection and refraction are considered) and the bouncing of the light from one object surface to another. Thus, global models are more complex and usually require more time and resources.

4.2.5 Transparent Surfaces

So far we assumed *opaque* surfaces; surfaces that do not allow light to pass through them. If any other object is present *behind* the surface, it is *not visible*. For such surfaces, it is sufficient to consider only reflection. Now consider an empty glass. If we put it in front of any other object, say an apple, we get to see the apple *through* the glass. Such surfaces, which allow objects behind them to be seen, are called *transparent* surfaces. Sometimes, some transparent surfaces such as frosted glass and some plastic material make the objects behind appear blurred. Those surfaces are known as *translucent* surfaces.

For an opaque object, we assumed that the color at any surface point is solely determined by the reflected lights from that point. This is not the case for transparent surfaces. In addition to the reflection, color at any transparent surface point is also determined by a second component: *transmitted light*. What happens is as follows: Some portion of the light incident at any surface point *passes through* the surface material. This phenomenon is known as *refraction*. This refracted light then gets *reflected* from the surface behind and again passes through the transparent surface towards the viewer. The idea is schematically illustrated in Fig. 4.7.

Clearly, we need to model the refraction path of light through the materials for realistic synthesis of transparent material. In general, refraction models are complex and computation intensive. These require considerations such as the *refractivity* (represented by the *index of refraction*) and the resultant shift in light direction. However, we can avoid all these complexities and use a simplistic approximate model to mimic the phenomenon, as we did in the case of reflection. The model is shown in Eq. 4.10.

$$I = (1 - k_l)I_{refl} + k_l I_{trans} \tag{4.10}$$

In this model, intensity at any point on a transparent surface is a weighted sum of the reflected and transmitted lights. In the equation, I_{refl} is the light reflected from the surface

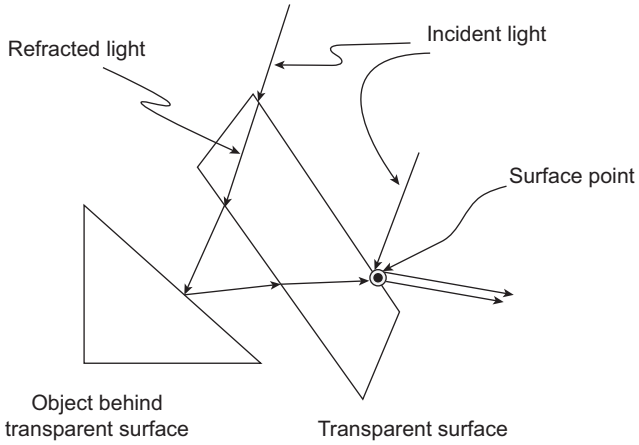


Fig. 4.7 The mechanism behind transparent surfaces. Color at any surface point is determined as a combination of the reflected and refracted lights.

point (can be calculated using our simple lighting model). I_{trans} is the intensity of the background light (i.e., reflected light intensity from the background object) transmitted through the surface. Again, we can calculate I_{trans} with the simple lighting model. k_l is called the *transparency index*, which is used to control the degree of the transparency effect. It is assigned a value between 0.0 and 1.0. The term $(1 - k_l)$ is called the *opacity factor*. For example, if $k_l = 0.7$, 70% of the background light is transmitted through the surface, indicating a highly transparent surface. On the other hand, transparency reduces (and opacity increases) as the value of k_l gets closer to 0.0.

4.3 SHADING MODELS

With the simple lighting model, we can compute the intensity of the reflected light (color) of a surface point in a 3D scene. As we saw, the computations involve lots of operations. Consequently, generating an image using computer graphics techniques is very expensive in terms of computing resources (processor, memory, etc.) and time. Most of the graphics applications nowadays require the screen image to change frequently (e.g., computer animation, games), which makes the process even more complicated as the computations have to be carried out repeatedly and within a short time span.

We can greatly reduce the number of computations in image generation by using the shading models. These models allow us to assign colors to the screen pixels without performing the lighting calculations for all the points to be rendered. Colors of only a few of the points are computed using the lighting model. Those colors are then used to interpolate colors at the other surface points mapped to the screen pixels. Note that there are two differences in the coloring of an image using lighting and shading models.

1. Application of the lighting model is very expensive as it involves large number of operations. Shading models, on the other hand, are interpolation-based approaches, which can be applied using efficient incremental procedures.
2. Lighting models are applied on the scene description (in 3D world coordinate system). In contrast, shading models work at the pixel-level, after the scene is mapped to the screen.

4.3.1 Flat Shading

Flat shading is the simplest of the shading models and involves the least number of computations. In flat shading, the color of any one point on a surface is computed using the lighting model. This color is then assigned to all other points on the surface that are mapped to screen pixels. Clearly, this may result in unrealistic images. Consequently, flat shading is not good to color any surface. In order to make the method work, the surface should satisfy the following properties.

1. The surface should be part of a polyhedral object and not a polygonal approximation of a curved surface. In other words, the surface should be defined as rather than approximated to a polygon.
2. All light sources should be sufficiently far from the surface. In other words, the vector dot product $\hat{N}.\hat{L}$ and the attenuations are constant over the surface.

3. Viewing position is sufficiently far from the surface, ensuring that the vector dot product $\hat{V}.\hat{R}$ is constant over the surface.

4.3.2 Gouraud Shading

Although flat shading is very fast due to minimum computation, it can lead to unrealistic images if the surfaces do not satisfy the properties we discussed in the previous section. With a little increase in computation, we can color surfaces more realistically using the Gouraud shading[1] without requiring the surfaces to satisfy the properties. In Gouraud shading, we assign colors to pixels using Algorithm 4.1.

Note that in the first step, we mentioned polygonal surfaces only. However, the surface may be polygonal by definition or due to the polygonal mesh approximation of a curved surface, unlike in the flat shading. Now, let us try to understand the steps of the algorithm. The first step implies that the lighting model is used to compute colors only at the vertices. Such vertices may be shared by more than one surface, as shown in Fig. 4.8. For each of these vertices, we calculate the *average unit normal vector*, which is the average of the unit normals of the surfaces that share the vertex. If the vertex is shared by n surfaces and the unit normal of the ith surface is denoted by $\hat{N}_i$, then the average unit normal $\hat{N}_{AVG}$ at the

Algorithm 4.1 **Steps for Gouraud shading**

1: Determine the *average* unit normal vector at each vertex of the polygonal surface.
2: Apply lighting model at each vertex to compute color at that position.
3: Linearly interpolate the vertex intensities over the projected area of the polygon.

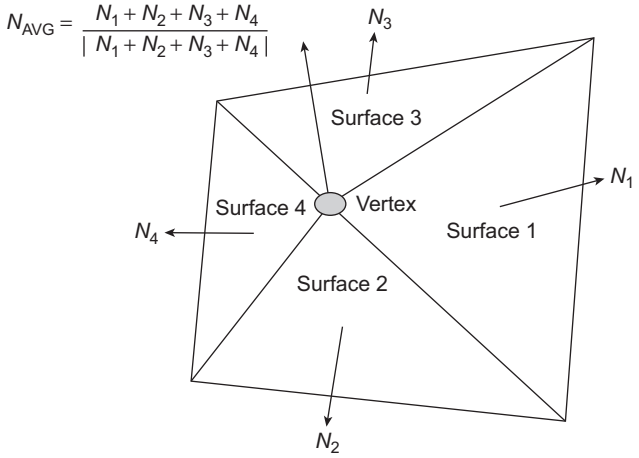


Fig. 4.8 Illustration of a shared vertex. $\hat{N}_i$ denotes the unit normal vectors for the ith surface. The shared vertex is shown with a filled circle, for which the average unit normal vector $\hat{N}_{AVG}$ is computed as shown.

[1] Also known as the *intensity interpolation surface rendering*.

vertex is computed using Eq. 4.11.

$$\hat{N}_{AVG} = \frac{\sum_{i=1}^{n} \hat{N}_i}{|\sum_{i=1}^{n} \hat{N}_i|} \tag{4.11}$$

In the second step, the $\hat{N}_{AVG}$ is used in our simple lighting model (instead of $\hat{N}$) to compute color at the vertex. This step is performed for all the vertices of the polygon. Once this step is completed, we use the vertex colors to interpolate the colors of the pixels that are part of the projected pixel in the third and final step.

Let us try to understand the third step in terms of an example. Consider Fig. 4.9. In the figure, the shaded area represents projection of a surface on the pixel grid. After the second step, the colors are computed at the three vertices 1, 2, and 3. We use these colors to interpolate colors of the intermediate points 4, 7 (two edge intersection points on a scan line) and 5 (a pixel inside the projected surface on the same scan line) using Eq. 4.12, where I_i denotes the intensity (color) at the ith point.

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2 \tag{4.12a}$$

$$I_7 = \frac{y_7 - y_2}{y_3 - y_2} I_3 + \frac{y_3 - y_7}{y_3 - y_2} I_2 \tag{4.12b}$$

$$I_5 = \frac{x_7 - x_5}{x_7 - x_4} I_4 + \frac{x_5 - x_4}{x_7 - x_4} I_7 \tag{4.12c}$$

The interpolation process can be implemented efficiently with an iterative approach. In order to understand the iterative approach, let us consider the pixels 5 and 6 in Fig. 4.9.

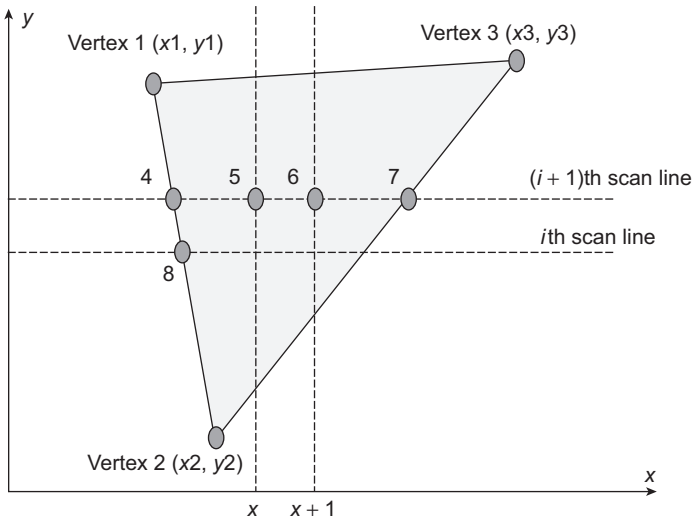


Fig. 4.9 Illustration of the interpolation process in Gauraud shading. The numbers 4, 7, and 8 denote the three points of intersection of the edges of the projected polygon surface with the scan lines. The numbers 5 and 6 denote two consecutive pixels on the same scan line, which are contained within the projected surface.

Note that they are on the same scan line with their x-coordinates differing by one. Now, we can represent the color of pixel 6 in terms of the colors of the pixels 4 and 7 in the same way as we have done for pixel 5 (see Eq. 4.12) as,

$$I_6 = \frac{x_7 - x_6}{x_7 - x_4} I_4 + \frac{x_6 - x_4}{x_7 - x_4} I_7.$$

However, $x_6 = x_5 + 1$. Thus,

$$\begin{aligned} I_6 &= \frac{x_7 - (x_5 + 1)}{x_7 - x_4} I_4 + \frac{(x_5 + 1) - x_4}{x_7 - x_4} I_7 \\ &= \frac{x_7 - x_5}{x_7 - x_4} I_4 - \frac{I_4}{x_7 - x_4} + \frac{x_5 - x_4}{x_7 - x_4} I_7 + \frac{I_7}{x_7 - x_4} \\ &= \frac{x_7 - x_5}{x_7 - x_4} I_4 + \frac{x_5 - x_4}{x_7 - x_4} I_7 - \frac{I_4}{x_7 - x_4} + \frac{I_7}{x_7 - x_4} \\ &= I_5 + \frac{I_7 - I_4}{x_7 - x_4} \end{aligned}$$

Note that the term $\frac{I_7 - I_4}{x_7 - x_4}$ is constant for a particular scan line. Let us denote this as C_i for the ith scan line. Then, the colors of the pixels along the ith scan line can be computed by simply adding the C_i to the previous color value. We start the process from the pixel next to the leftmost edge-scan line intersection point and continue to do this till we reach the rightmost edge-scan line intersection point.

Like the pixels along the scan line, we can also compute the pixel colors across scan lines in iterative manner. Consider for example the pixels 4 and 8 in Fig. 4.9. They are on two consecutive scan lines whose y-coordinates differ by one. Similar to I_4 in Eq. 4.12, we can represent the color of pixel 8 in terms of the vertex colors I_1 and I_2 as

$$I_8 = \frac{y_8 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_8}{y_1 - y_2} I_2.$$

However, $y_8 = y_4 - 1$. Thus,

$$\begin{aligned} I_8 &= \frac{(y_4 - 1) - y_2}{y_1 - y_2} I_1 + \frac{y_1 - (y_4 - 1)}{y_1 - y_2} I_2 \\ &= \frac{y_4 - y_2}{y_1 - y_2} I_1 - \frac{I_1}{y_1 - y_2} + \frac{y_1 - y_4}{y_1 - y_2} I_2 + \frac{I_2}{y_1 - y_2} \\ &= \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2 - \frac{I_1}{y_1 - y_2} + \frac{I_2}{y_1 - y_2} \\ &= I_4 + \frac{I_2 - I_1}{y_1 - y_2} \end{aligned}$$

For any edge of the polygon, the term $\frac{I_2 - I_1}{y_1 - y_2}$ is constant. Let C_j denote the constant value for the jth edge. Then, the colors of the pixels across the scan lines which are on the same edge can be computed by simply adding the C_j to the color value of the previous pixel on the edge. We start the process from the topmost pixel on the edge and proceed downwards till we

Algorithm 4.2 **Iterative algorithm for Gouraud shading**

1: Determine the average unit normal vector at each vertex of the polygonal surface.
2: Let C_R = color of the rightmost edge pixel on the ith scan line, x_r = the x-coordinate of the rightmost edge pixel, C_L = color of the leftmost edge pixel on the ith scan line and x_l = the x-coordinate of the leftmost edge pixel. Compute and store the scan line constants $C_i = \frac{C_R - C_L}{x_r - x_l}$ for each scan line that lies within the projected area of the polygon.
3: Let C_T = color of the topmost vertex pixel of the jth left edge, y_t = the y-coordinate of the topmost vertex pixel of the edge, C_B = color of the lowermost vertex pixel of the jth left edge and y_l = the y-coordinate of the lowermost vertex pixel of the edge. Compute the edge constants $C_j = \frac{C_B - C_T}{y_t - y_l}$ for all the left edges of the polygon.
4: Initialize a temporary variable $color = 0.0$.
5: **for** each scan line i within the polygon area starting from the top **do**
6: **for** each pixel on the scan line within the projected surface area, starting from the left **do**
7: **if** it is a vertex pixel **then**
$color$ = color computed using the lighting model.
8: **else if** it is a pixel on a left edge **then**
$j\ color$ = color of the pixel on the previous scan line and same edge + C_j.
9: **else**
$color$ = color of previous pixel on the same scan line + C_i.
10: **end if**
11: **end for**
12: **end for**

reach the lower most pixel on the edge. Thus, using the vertex colors we can compute colors of all the other pixels using the iterative approach. With the understanding of the iterative approach, we can rewrite Algorithm 4.1 more elaborately as shown in Algorithm 4.2.

Gouraud shading is typically implemented along with a later stage of the pipeline, namely the *hidden surface removal*.

Limitations of the model Although Gouraud shading improves photo-realism of the synthesized image to a great extent with little extra computation compared to flat shading, the method has a couple of limitations.

1. It is not good for generating specular reflection effect. This is because the linear interpolation in Gouraud shading results in a smooth change of color values between neighboring pixels unlike specular reflections where at the boundary of the reflection region, sudden change in color values take place.
2. Another problem with Gouraud shading is that of the occurrence of *Mach bands*. This is a psychological phenomenon in which we see bright bands where two blocks of solid colors meet.

4.3.3 Phong Shading

The limitations of the Gouraud shading can be overcome with a more accurate shading model, known as the Phong shading (or the *normal–vector interpolation rendering*),

although application of the model involves large compuation. Consequently, it does not have the advantage of other shading models, namely fast coloring of surface points.

The steps of the phong shading are similar to that of Gouraud shading, with the two majors differences being,

1. Here, the linear interpolation is used to determine normal vectors at each projected pixel position.
2. At each projected pixel, the lighting model is used to compute colors considering the interpolated normal vectors.

In order to understand the process, reconsider Fig. 4.9. The interpolated normal vectors at the points 4, 5, and 7 can be computed similar to that of Eq. 4.12, by replacing the intensity terms with the corresponding vector terms as shown in Eq. 4.13.

$$\hat{N}_4 = \frac{y_4 - y_2}{y_1 - y_2}\hat{N}_1 + \frac{y_1 - y_4}{y_1 - y_2}\hat{N}_2 \tag{4.13a}$$

$$\hat{N}_7 = \frac{y_7 - y_2}{y_3 - y_2}\hat{N}_3 + \frac{y_3 - y_7}{y_3 - y_2}\hat{N}_2 \tag{4.13b}$$

$$\hat{N}_5 = \frac{x_7 - x_5}{x_7 - x_4}\hat{N}_4 + \frac{x_5 - x_4}{x_7 - x_4}\hat{N}_7 \tag{4.13c}$$

Now, these interpolated vectors are used to compute colors at those points applying the lighting model. As you can see, we can formulate the interpolation as an iterative process just the way we did for Gouraud shading. Hence, we can develop the Phong shading algorithm by modifying Algorithm 4.2 slightly, which is left as an exercise (see Exercise 10).

4.4 HANDLING THE SHADOW EFFECT

When an opaque surface is positioned in front of another surface with respect to a light source, it generates a shadow of itself on the surface behind. In other words, the surface behind (or a portion of it) does not get illuminated by the light source. With *visibility detection methods* discussed in a later chapter (Chapter 8), it is possible to determine regions that are not illuminated by light sources. If we assume that the viewing position is the same as that of the light source, visibility detection methods will tell us the surfaces (or surface sections) in the scene that are not visible. These are the shadow areas.

We can have simple mechanism to create shadow effect. Once we are able to determine shadow areas in a scene, we can use only the ambient light component to illuminate those areas. Of course, this approach does not produce high quality images as there will be no variation in the shadows. We can do somewhat better by using pre-specified texture patterns (see the next chapter for discussion on texture synthesis) along with the ambient light. Depending on the number and position of light sources in a scene and their intensities, we can create a set of texture patterns (also known as *texture maps*, see the next chapter). These patterns can then be combined with the ambient light to determine colors at the shadow areas. For example, we can have one pattern for creating shadows that result due to high-intensity light sources (e.g., the sunlight in the afternoon) and another pattern for shadows due to low-intensity lights (e.g, shadows in a dimly-lit room in the night).

Example 4.1

Let us try to understand the computations involved in terms of an example. Assume a cubical object with its vertices specified as **A**(0, 0, 2), **B**(2, 0, 2), **C**(2, 2, 2), **D**(0, 2, 2), **E**(0, 0, 0), **F**(2, 0, 0), **G**(2, 2, 0), and **H**(0, 2, 0). We want to create a scene of a room, in which the object is treated as a 'shelf attached to a wall', keeping the relative positions of the corresponding vertices same as in the original object. The wall is parallel to the XZ plane, cutting the positive Y-axis at a distance of 1 unit. The length of the sides of the shelf is half that of the original object. The surface of the shelf C′D′H′G′ corresponding to the surface CDHG of the object is on the wall, with its lower left corner at (1, 1, 1). Fig. 4.10 shows the shelf as it looks like after projecting onto the pixel grid, along with some of the mapped vertices and one vertex location.

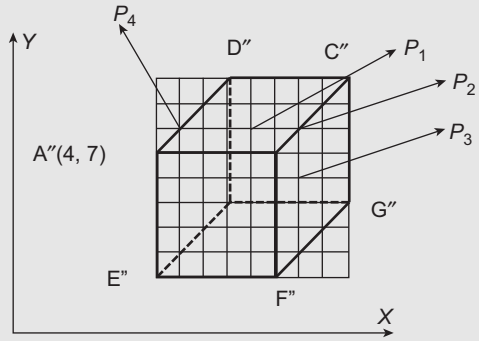


Fig. 4.10 The projection of the cube on the pixel grid, with the coordinates of one projected vertex

Let the room have a monochromatic point light source at the position (1, −1, 3) and intensity $I_s = 2$ units. Further assume that there is an ambient light with intensity of 1 unit and the object have $k_a = 0.5$, $k_d = 0.25$, $k_s = 0.25$, and n^s = specular exponent = 10. If you are looking at the shelf from a location (3, −1, 3), what would be the intensities at the pixels P_1, P_2, and P_3 assuming flat shading?

Solution The first thing to note here is that the original object has undergone transformations before it is made a part of the scene. However, we don't need to find out the vertex coordinates of the transformed object using the transformation matrices, since the final size, position and orientation are already specified in the problem. Using those specifications, we can determine the vertices of the shelf as A′ = (1, 0, 2), B′ = (2, 0, 2), C′ = (2, 1, 2), D′ = (1, 1, 2), E′ = (1, 0, 1), F′ = (2, 0, 1), G′ = (2, 1, 1), H′ = (1, 1, 1). The transformed object is shown in Fig. 4.11.

From the given specification and Fig. 4.10, we can also determine the coordinates of some of the projected vertices and the pixels of interest (P_1, P_2, and P_3), which are as follows.

$$D'' = (7, 10), P_1 = (8, 8), P_2 = (10, 8), C'' = (7, 15), P_3 = (10, 6)$$

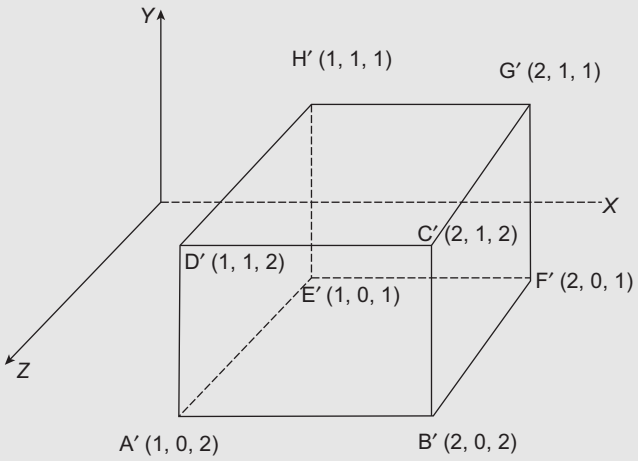


Fig. 4.11 The cube in 3D world coordinate after transformation

Note that we don't need to know anything about projection to find these coordinates, since the specifications are sufficient along with the knowledge that the distance between any two pixels on the grid is 1. Along with the coordinates of the above pixels, we also need the coordinate of the pixel P_4 (in Fig. 4.10) for ease of calculation, which is (5, 8). Further note that the light source is above the A′B′C′D′ surface and on the left side of the plane which contains the surface B′F′G′C′. Hence, it will illuminate A′B′C′D′ but will not contribute anything towards the illumination of the surface B′F′G′C′.

We can calculate the color at any point on the surface and use the same value throughout the surface. Let's calculate color at B′. We know that $\hat{N} = (0, 0, 1)$ (i.e., the unit vector parallel to the Z axis, assuming B′ to be part of the surface A′B′C′D′). The vector towards light source $\vec{L} = (1, -1, 3) - (2, 0, 2) = (-1, -1, 1)$. Thus, $\hat{L} = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$. The view vector $\vec{V}$ is $(3, -1, 3) - (2, 0, 2) = (1, -1, 1)$. Hence, $\hat{V} = \left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$. Therefore, $\hat{N}.\hat{L} = \frac{1}{\sqrt{3}} \approx 0.58$.

Now, we know that $\hat{V}.\hat{R} = \hat{V}.[(2\hat{N}.\hat{L})\hat{N} - \hat{L}]$, where $\hat{R}$ is the unit vector along the ideal specular reflection direction. From the values of the unit vectors, we get $(2\hat{N}.\hat{L})\hat{N} - \hat{L} = \frac{2}{3}(1, 1, \sqrt{3} - 1)$. Thus, $\hat{V}.\hat{R} = \frac{2}{3\sqrt{3}}(\sqrt{3} - 1) \approx 0.28$.

Therefore the color at B′ (using Eq. 4.1) is $= 0 \times 5 \times 1 + 0 \times 25 \times 2 \times (0.58) + 0 \times 25 \times 2 \times (0 \times 28) \times 10 = 0.79$ unit. Since P_1 and P_2 both are part of the same surface containing B′, color at P_1 = color at $P_2 = 0.79$ unit. However, as we have informally discussed before, the light source do not contribute in the illumination of the surface B′F′G′C′. Thus, color of any point on this surface is decided by the ambient light only. Since P_3 is part of the surface B′F′G′C′, color at $P_3 = 0.5.1 = 0.5$ unit. Note that the same results can be arrived at by considering the vectors only (which is left as an exercise).

Example 4.2

What would the intensities of the same pixels P_1, P_2, and P_3 be, assuming Gouraud shading?

Solution In order to calculate color, we need to calculate colors at the vertices A′, B′, C′, and D′, using which we can interpolate other colors. The computation of the vertex colors are shown in Table 4.1.

Table 4.1 The computation of the vertex intensities for Gouraud shading in Example 4.2

Vertex	Color computation
A′	$\hat{N_{AVG}} = \frac{(-1,0,0)+(0,-1,0)+(0,0,1)}{\lvert(-1,0,0)+(0,-1,0)+(0,0,1)\rvert} = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$ $\hat{L} = \frac{(1,-1,3)-(1,0,2)}{\lvert(1,-1,3)-(1,0,2)\rvert} = \left(0, -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ $\hat{V} = \frac{(3,-1,3)-(1,0,2)}{\lvert(3,-1,3)-(1,0,2)\rvert} = \left(\frac{2}{\sqrt{6}}, -\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}\right)$ $\hat{N_{AVG}}.\hat{L} = \frac{2}{\sqrt{6}}$ $\hat{V}.\hat{R} = \hat{V}[(2.\hat{N}.\hat{L})\hat{N} - \hat{L}] = -\frac{4}{3}\left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}}\right) < 0$ (no specular component) **Color** $(I_{A'}) = I_{amb} + I_{diff} = 0.5.1 + 0.25.2.\frac{2}{\sqrt{6}} \approx 0.91$
B′	$\hat{N_{AVG}} = \frac{(0,-1,0)+(0,0,1)+(1,0,0,)}{\lvert(0,-1,0)+(0,0,1)+(1,0,0,)\rvert} = \left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$ $\hat{L} = \frac{(1,-1,3)-(2,0,2)}{\lvert(1,-1,3)-(2,0,2)\rvert} = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$ $\hat{V} = \frac{(3,-1,3)-(2,0,2)}{\lvert(3,-1,3)-(2,0,2)\rvert} = \left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$ $\hat{N_{AVG}}.\hat{L} = \frac{1}{3}$ $\hat{V}.\hat{R} = \hat{V}[(2.\hat{N}.\hat{L})\hat{N} - \hat{L}] = \frac{4}{9}$ **Color** $(I_{B'}) = I_{amb} + I_{diff} + I_{spec} = 0.5.1 + 0.25.2.\frac{1}{3} + 0.25.2.\frac{4}{9^{10}} \approx 0.67$
C′	$\hat{N_{AVG}} = \frac{(0,0,1)+(0,1,0)+(1,0,0)}{\lvert(0,0,1)+(0,1,0)+(1,0,0)\rvert} = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$ $\hat{L} = \frac{(1,-1,3)-(2,1,2)}{\lvert(1,-1,3)-(2,1,2)\rvert} = \left(-\frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}}, -\frac{1}{\sqrt{6}}\right)$ $\hat{V} = \frac{(3,-1,3)-(2,1,2)}{\lvert(3,-1,3)-(2,1,2)\rvert} = \left(\frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}}, \frac{1}{\sqrt{6}}\right)$ $\hat{N_{AVG}}.\hat{L} = -\frac{2}{3\sqrt{2}} < 0$ (no diffuse and specular components) **Color** $(I_{C'}) = I_{amb} = 0.5.1 = 0.5$

(Contd)

Table 4.1 (*Contd*)

Vertex	Color computation
D′	$\hat{N_{AVG}} = \frac{(-1,0,0)+(0,1,0)+(0,0,1)}{\lvert(-1,0,0)+(0,1,0)+(0,0,1)\rvert} = \left(-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$ $\hat{L} = \frac{(1,-1,3)-(1,1,2)}{\lvert(1,-1,3)-(1,1,2)\rvert} = \left(0, -\frac{2}{\sqrt{5}}, \frac{1}{\sqrt{5}}\right)$ $\hat{V} = \frac{(3,-1,3)-(1,1,2)}{\lvert(3,-1,3)-(1,1,2)\rvert} = \left(\frac{2}{3}, -\frac{2}{3}, \frac{1}{3}\right)$ $\hat{N_{AVG}}.\hat{L} = -\frac{1}{\sqrt{15}} < 0$ (no diffuse and specular components) **Color** $(I_{D'}) = I_{amb} = 0.5.1 = 0.5$

Using iterative method, we can now calculate colors at P_1 and P_2.

Color at P_2 $(I_{P_2}) = \left(I_{C'} + \frac{I_{B'} - I_{C'}}{y_{C''} - y_{B''}}\right) + \frac{I_{B'} - I_{C'}}{y_{C'} - y_{B'}} = 0.67 + 0.034 + 0.034 = 0.738$ unit.

Let $C = \frac{I_{A'} - I_{D'}}{y_{D''} - y_{A''}} \approx 0.14$.

Then, color at P_1 (I_{P_1}) can be computed iteratively as $(((((I_{D'} + C) + C) + C') + C') + C')$, where $C' = \frac{I_{P_2} - I_{P_4}}{x_{P_2} - x_{P_4}} = \frac{I_{P_2} - (I_{D'} + 2C)}{x_{P_2} - x_{P_4}} = 0.0196$. Thus, $I_{P_1} \approx 0.84$.

Since the light source does not contribute directly towards the illumination of P_3, its intensity will be $0.5.1 = 0.5$ unit. The computation of the vertex intensities for Gouraud shading is shown in Table 4.1.

4.5 INTENSITY REPRESENTATION

As Examples 4.1 and 4.2 illustrate, the lighting and shading models compute color (intensities) at any surface points as a real number between 0.0 and 1.0. If we go inside a graphics system, we shall see that the computed intensity values are used to drive the mechanism to draw pictures on the screen. For example, the computed intensity values are used to set the voltages for the electron gun and the deflection arrangements of a CRT display (see Chapter 1) such that electron beams with appropriate intensity are generated and directed towards specific phosphor dots. However, due to the discrete nature of a computer, any real intensity value cannot be represented and used for the purpose. The representation depends on the design of the frame buffer.

4.5.1 Basic Idea

Consider a graphics system having a frame buffer that uses 8 bits to store the intensity values for each pixel. With 8 bits, we can represent $2^8 = 256$ values. Thus, the system can support at most 256 intensity values for each pixel. The intensity values computed for a pixel by the lighting/shading models, on the other hand, are infinite (any number between 0.0 and 1.0). Therefore, we need to map these potentially infinite intensity values to the 256 values supported by the system. We cannot use any arbitrary mapping, however, as arbitrary mapping may lead to visible distortion in the image. How can we achieve the objective?

The idea is to distribute the *computed* intensty values among the *system supported* values such that the distribution corresponds to the way our eyes perceive intensity differences. We take help of a psychological phenomenon to arrive at the distribution. Suppose you are shown two sets of intensity values. In the first set, there are two intensities: 0.10 and 0.11 (i.e., a difference of 10%). In the second set also, there are two intensities: 0.50 and 0.55 (again, a difference of 10%). In both the cases, the intensity difference will look the same although the absolute intensity values are different. In other words, we cannot perceive the absolute difference in intensity values. Only the relative difference matters. Therefore, if the ratio of two intensities is the same as the ratio of two other intensities, we perceive the difference as the same. Thus, if we want to distribute a continuous range of intensity values (the range of computed values) into a finite set of discrete values (the values supported by the device), what we have to do is to preserve the ratios in the successive intensity values. Exploiting this idea, we can map the continuous intensity range (between 0.0 and 1.0) into the set of discrete values in the following way.

Suppose the device supports N discrete values for each pixel, denoted by $I_0, I_1, \ldots I_N$. We can use a photometer to determine the minimum (I_0) and maximum (I_N) brightness of the device. This range is known as the *dynamic range*. The highest value in the range is usually

Example 4.3

An example to illustrate the idea of intensity mapping (from computed value to device-supported value)

Solution Suppose we have a display device, which supports a minimum intensity $I_0 = 0.01$ as found with a photometer. We assume the maximum intensity supported by the device $I_N = 1.0$. The device supports 8 bits for each pixel location. Then the number of intensity values N supported by the device for each pixel is $2^N = 2^8 = 256$.
Thus, the intensities values are $I_0, I_1, \ldots I_{255}$. Then, $1.0 = r^{255}0.01$. Solving this equation, we get $r = 1.0182$. Thus, $I_1 = 0.01820.01 = 0.0102$, $I_2 = rI_1 = 0.0104$, and so on.
We now need to assign bit patterns to these 256 intensity values. One mapping is shown below as an illustration, although it is possible in principle to have any arbitrary mapping.

$$\begin{bmatrix} \text{Intensity} & \text{Bit pattern} \\ I_0 = 0.0100 & 00000000 \\ I_1 = 0.0102 & 00000001 \\ I_2 = 0.0104 & 00000010 \\ \cdots & \cdots \\ I_{255} = 1.0 & 11111111 \end{bmatrix}$$

Now, suppose the lighting model computes the intensity at a pixel location as 0.01039. We try to find the nearest intensity value the pixel supports. In this case, it is 0.0104 (I_2). Hence, the pixel intensity is represented by the bit pattern 00000010 for I_2. Note that the final intensity the pixel gets is different from the computed intensity.

taken as 1.0. Thus, the intensities range between I_0 to 1.0. In order to preserve ratio between successive intensities, the following relations should hold.

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} - - - \frac{I_N}{I_{N-1}} = r$$

Here r is the common ratio. Therefore, $I_1 = rI_0$, $I_2 = rI_1 = r(rI_0) = r^2I_0$, $I_3 = rI_2 = r(r^2I_0) = r^3I_0$, and so on. In general, $I_k = r^kI_0$ for $k > 0$. Then we can say $I_N = r^NI_0$. Since we have already determined I_0 and $I_N = 1.0$, we can determine r. Using r, we can obtain N intensity values. Once we obtain the intensities, we can assign a bit pattern to them.

Now asume that the lighting model computes an intensity value I_P for a pixel. We find out the nearest value to I_P from the device-supported intensity values (say I_k). The frame buffer stores the bit pattern for I_k at the pixel location.

4.5.2 Gamma Correction

We now know the procedure to map the computed intensity to one of the device-supported intensities (and ultimately to a bit pattern). How this is going to be useful? Let us try to understand in the context of a CRT. The bit pattern is used to design circuitry that can generate an appropriate voltage V. When V is applied to the electron gun, an electrom beam of appropriate intensity is generated. The beam hits the specific phosphor dot on the screen corresponding to the pixel, which then emits light with the required brightness.

Let us assume that the phosphor dot emits light with brightness I. Intuitively, we expect $I \propto aV$. In other words, the generated brightness is linearly proportional to the voltage applied to the elctron gun (a is the proprotionality constant for a display device). However, display devices do not follow such linear behavior. What happens instead is a non-linear relationship: $I \propto aV^\gamma$. Hence, if we set V proportional to the intensity value we computed, the brightness I we get will be different than what we expect.

To overcome this problem, what is done is known as the *gamma correction*. For a display device, the non-linearity exponent γ is constant (although there may be different values for the red, green and blue components). For the desired brightness I, we compute the desired voltage V' with the inverse relation: $V' = (\frac{I}{a})^{\frac{1}{\gamma}}$. If V' is used to set the electron gun voltage, we get the required brightness.

Steps in intensity mapping

Let I be the intensity value calculated with an illumination model.

1. Compute intensity levels for the device and assign bit patterns to each level.
2. Determine nearest intensity level I_k of the device for I.
3. Compute required voltage for I_k after gamma correction.

To speed-up processing, the intensity levels computed for a device are usually kept in a table along with its bit pattern and gamma corrected voltage value. The computations in steps 1 and 3 can then be replaced with table look-ups only.

Some devices are designed to perform the gamma correction in hardware (SGIs), whereas in most PCs it has be done through software interface by the user. Consequently, images generated on one type of display devices may not look the same in the other type.

4.5.3 Halftoning and Dithering

You may have noticed in the previous discussion that the intensity we get on the screen is not the same as the intensity we calculate with a lighting model. This is so since the calculated intensity is mapped to the nearest intensity the device supports. In order to generate the desired image on the screen, we need to keep this difference as small as possible. Otherwise, the quality of the generated image may be affected.

The desired image quality can be maintained if the device supports large number of intensity levels. Now consider a bi-level device (each pixel has a single bit to represent its intensity). Any computed intensity (between 0.0 and 1.0) has to be mapped to either of these two levels. Clearly, we cannot generate good-quality images with such devices. However, we can partially overcome this problem with a technique known as *halftoning* (borrowed from the printing technology). In halftoning, a rectangular grid of *screen pixels* are considered to represent a single pixel of the image. The rectangular pixel grid is called the *halftone approximation pattern* or simply the *pixel pattern*. The total number of intensity levels supported by each pixel pattern depends on the number of levels supported by each device. Let us try to understand the concept with an example.

Suppose we have a bi-level device. We define a 2×2 pixel pattern on this device. Thus, 4 screen pixels constitute each image pixel. Ordinarily, each screen pixel supports only two intensity levels (as it is a bi-level device). Let us denote these two levels as ON and OFF. However, with each pixel pattern (image pixel), we can have 5 intensity levels, as shown in Fig. 4.12. In Fig. 4.12(a), all screen pixels are in the OFF state. We set one of the four pixels in ON state (Fig. 4.12(b)-(d)) till we have all four pixels in the ON state (Fig. 4.12(e)). Hence, we can determine five intensity levels and map them to the five states of the pixel pattern. Any computed intensity can then be mapped to the nearest of the five intensity levels instead of two. In general, for a bi-level device having an $N \times N$ pixel grid, we can have $N^2 + 1$ intensity levels.

It is obvious that the halftoning technique, although helps to generate good-quality images on devices that support few intensity levels, reduces resolution of the image. For a bi-level device having 512×512 screen resolution, application of 2×2 halftoning technique reduces the effective screen resolution to 256×256. Another problem with the approach is that the subgrid patterns become visible as the grid size increases. In other words, we cannot decide the pixel-grid size arbitrarily. It has to be decided depending on the size of the

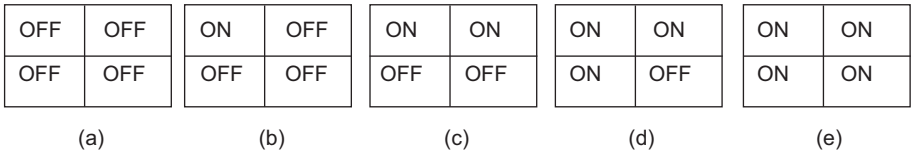


Fig. 4.12 The 5 intensity levels generated with the 2×2 pixel pattern on a bi-level device

Algorithm 4.3 **Floyd–Steinberg error diffusion**

```
1: Input: M[m][n], the matrix of computed intensity values.
2: Output: The same matrix M[m][n] after modifying its elements through error diffusion.
3: for i = 0; i < n; i++ do
4:    for j = 0; i < m; j++ do
5:       e = M[i][j]-nearest intensity value supported by the device.
6:       M[i][j+1] += αe. /*α = 7/16, j ≠ m − 1*/
7:       M[i+1][j-1] += βe. /*β = 3/16, i ≠ n − 1, j ≠ 0*/
8:       M[i+1][j] += γe. /*γ = 5/16, i ≠ n − 1*/
9:       M[i+1][j+1] += δe. /*δ = 1/16, i ≠ n − 1, j ≠ m − 1*/
10:   end for
11: end for
```

screen pixels. For low-resolution devices (i.e., fewer screen pixels per centimeter implying larger pixel size), we have to be satisfied with fewer intensity levels. Otherwise, there may be distortion in the generated image.

We can approximate halftoning without reducing resolution through *dithering* techniques. One such technique is the *Floyd–Steinberg error diffusion*. Let us assume that we have a screen with resolution $m \times n$. For each pixel location, we have computed intensity values with a lighting model. Thus, we have a matrix M (of size $m \times n$) of computed intensity values. In the approach, we start with the top-left element of M and proceed from left to right and top to bottom manner. For each element M_{ij} in M (i.e., the jth value in the ith row), we determine the nearest intensity supported by the device. The difference between these two intensities is the error term e. We now disperse (diffuse) this error to the neighbouring pixels as shown in Algorithm 4.3. In this way, we try to minimize the quantization error due to intensity mapping. Of course, the error at the bottom-right pixel cannot be dispersed.

Example 4.4

Let us reconsider Example 4.2. Suppose the screen on which the scene is displayed has 2 bits/pixel. Each pixel location can emit light with an intensity of atleast 0.05 unit. What would be the intensity of P_2 assuming Gouraud shading?

Solution Each pixel has 2 bits, implying that the device supports 4 intensity levels. As stated in the problem description, the minimum brightness supported by the device is 0.05 unit We assume the maximum screen brightness to be 1.0 unit.

Thus, $0.05 \times r^3 = 1$ or $r = 20^{\frac{1}{3}} = 2.69$.

With the value of r, we can determine the four intensity levels as $I_0 = 0.05$, $I_1 = 0.05 \times 2.69 = 0.1345$, $I_2 = 0.05 \times 2.69^2 = 0.361805$ and $I_4 = 1.0$.

Now, the intensity value at P_2 computed for Gouraud shading is 0.738 unit (see worked out example 1). This value is closer to I_4. Thus, the intensity for the corresponding pixel will be set as $I_4 = 1.0$ unit instead of the computed value (0.738).

SUMMARY

In this chapter, we have learnt many important and useful concepts. First, we need to artificially illuminate a scene to generate the 3D effect. Illumination of a scene implies coloring of the points on the object surfaces. Color of a point, which is a psychological phenomenon, is determined by the intensity of the light reflected from the point that reaches our eye. The reflected intensity has three components: reflection due to ambient light, diffucse reflection due to direct light source and specular reflection due to direct light source.

The computation of color at a point is done through the lighting models. We have learnt the simple lighting model, which makes several simplistic assumptions about the light source and surface properties. Notable among these are: point light source, a simple ambient light source that illuminates all surfaces in the same way, ideal diffuse reflection or Lambertian surfaces and an approximation of the specular effect in terms of a specular reflection coefficient. On the basis of these assumptions, the model can compute color of a point in the world coordinate in the presence of single or multiple light source(s). For monochromatic light sources, the model computes single intensity values whereas for colored light sources, it computes three components of the intensity, corresponding to the primary colors red, green and blue. Moreover, by taking into account the radial and angular intensity attenuation, the model allows us to generate realistic as well as spotlight effects.

In order to reduce complexities of color computations, shading models are proposed. The simplest of these is the flat shading, which works well only under some constraints. The limitations of the flat shading are alleviated to a great extent with Gouraud shading, although the speed of compuation is affected a little. Very realistic images can be generated with Phong shading, although it is very expensive in terms of computation time and required resources.

The lighting model computes a real number, which we term as color. We saw in this chapter how this number is mapped to a bit pattern. The mapping involves several stages which include the determination of the intensity levels for the device (by preserving ratio), assigning bit patterns to each level, mapping the computed intensity to the nearest intensity the device support and finally, performing the gamma correction. We have also learnt two techniques to reduce errors due to mapping, namely the halftoning and the Floyd-Steinberg error diffusion.

In the next chapter, we shall get introduced to the concept of *color models* to learn more about the way colors are handled in computer graphics. In addition, we shall learn about texture synthesis to increase the photo-realism of the generated image, which the simple lighting model in itself is incapable of doing.

BIBLIOGRAPHIC NOTE

A good starting point for further readings on the simple lighting model and shading models are the original works by Gouraud [1971] and Phong [1975]. A fast implementation of Phong shading is discussed in Bishop and Wiemer [1986]. More information on lighting and shading models can be found in Birn [2000] and Akenine-Moller and Haines [2002]. The *graphics gems* series of books (Glassner [1990], Arvo [1991], Kirk [1992], Heckbert [1994] and Paeth [1995]) contains implementation algorithms for lighting and shading models. Halftoning methods are discussed in Velho and Gomes [1991]. Knuth [1987] can be referred for more information on dithering and error diffusion methods.

KEY TERMS

Ambient light – light that comes from indirect source (i.e., after reflection from surrounding surfaces) that illuminates a point
Angular attenuation factor – the effect of reduction of light energy over angular limits
Attenuation factor – mathematical technique to incorporate attenuation in lighting calculations
Color – the intensity of the light that reaches a viewer's eye and gives rise to the perception of color
Diffuse reflection – reflection of incident light from a surface in all directions
Dithering – halftoning without reduction of screen resolution
Flat shading – the simplest shading model, in which the same color is assigned to all pixels belonging to a surface
Floyd–Steinberg error diffusion – one type of dithering technique
Gamma correction – the method used to overcome the problem of non-linear relationship between the input voltage and the output intensity for a display device
Global lighting model – the model that considers direct as well as indirect (ambient) sources in color computation
Gouraud shading – a shading model in which color of a pixel belonging to a surface is assigned after interpolating vertex colors of the surface
Halftoning – displaying more than two intensity levels on a bi-level device
Idea diffuse reflectors/Lambertian reflectors – the surfaces that follow the Lambert's cosine law
Illumination – the transport of light energy from a light source to a point
Intensity attenuation – loss of reflected light intensity due to the the distance from the light source
Lambert's cosine law – the reflected light energy to a direction is proportional to the cosine of the angle between the surface normal and the reflection direction
Lighting – the process of computing the luminous intensity of the outgoing (reflected) light from a point
Local lighting model – the model that emphasize on direct light sources for color computation
Mach bands – a psychological phenomena in which we see bright bands when two solid blocks of colors meet
Opacity – the amount of opaqueness; represented in terms of the opacity factor
Phong model/Phong specular reflection model – an empirical model to compute the specular reflection component of color
Phong shading – an advanced shading model used for realistic rendering of surfaces
Point light source – a light source that emits light from a point with equal intensity in all directions
Radial attenuation factor – the effect of reduction of light energy over distance
Reflection coefficient/reflectivity – the surface property that determines the fraction of incident light that gets reflected
Refraction – the phenomenon of light passing through a surface
Shading/Surface rendering – the process of assigning colors to pixels through interpolation techniques
Specular reflection exponent – a numerical value to generate different specular effects
Specular reflection – reflection of incident light from a surface in a specific direction
Spotlight – a light source that emits light only in a specific direction in the form of a (conical) light beam
Translucent surface – surfaces that allow objects behind them to be seen but the objects appear blurred
Transparency index – the degree of transparency
Transparent surfaces – surfaces that allow objects behind them to be seen

EXERCISES

4.1 We can simply assign some colors to each of the surfaces in a scene. Then why do we need the illumination stage in the graphics pipeline?

4.2 Suppose you saw that the pen you use has golden color. How can you explain this observation?

4.3 We can see color on the surface of an object even if it is not illuminated by a direct light source. Explain why and how this is possible.

4.4 Derive the simple lighting model for monochromatic single light source without attenuation. List all the assumptions made in the model.

4.5 Discuss the problem we are likely to face if we do not consider radial intensity attenuation in the lighting model. Explain how we can implement the spotlight effect with the model.

4.6 How we can extend the simple lighting model to color light sources?

4.7 Discuss the purpose of using shading models. What are the key differences between shading and lighting models?

4.8 Write the pseudo code for flat shading. Discuss its limitations.

4.9 Derive the iterative Gouraud shading algorithm. What are its limitations?

4.10 Write the pseudo code for the Phong shading. As we can see, the Phong shading requires huge computations unlike other shading models. What, do you think, is the advantage of this model?

4.11 Mention the steps involved in transforming a computed intensity value to discrete bit pattern.

4.12 Why do we need gamma correction? How it is done?

4.13 How does halftoning help in image generation? Calculate the number of intensity levels supported by a 2×4 pixel pattern on a 5-level device.

4.14 Discuss the problems with halftoning. Explain one technique to overcome those limitations.

4.15 Consider an object made of two rectangular surfaces ABCD and EFCD. The vertices are A(0, 0, 2), B(1, 0, 2), C(1, 1, 1), D(0, 1, 1), E(0, 0, 0) and F(1, 0, 0). There is a light source located at (3/2, 2, 2), emitting light with an intensity of 0.5 unit and the object is being seen from (5/2, 3, 3). After projection, the line CD is on the fifth scan line, as shown in Fig. 4.13. We want to display the object on a device having 3 bits/pixel. Each pixel location can emit light with an intensity of atleast 0.01 unit. Calculate the intensity at P (midway point between C and D) for the device assuming (a) flat shading and (b) Phong shading. (Assume an ambient light with intensity of 0.5 unit, $k_a = k_d = k_s = 0.5$, $n_s = 100$, ignore attenuation).

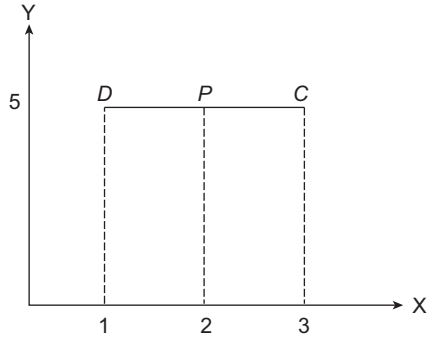


Fig. 4.13 Projected line CD

CHAPTER 5

Color Models and Texture Synthesis

Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the physiological process behind the perception of color
- Learn about the idea of color representation through the use of color models
- Understand additive color models and learn about RGB and XYZ models
- Understand subtractive color models and learn about the CMY model
- Learn about the HSV color model, which is popularly used in interactive painting/drawing software applications
- Get an overview of the three texture synthesis techniques-projected texture, texture mapping, and solid texturing

INTRODUCTION

The fourth stage of the graphics pipeline, namely the coloring of 3D points, has many aspects. One aspect is the use of the lighting models to compute color values. This has been discussed in Chapter 4. In the related discussions, we mentioned the basic principle governing color computation, namely that color of a point is a psychological phenomenon resulting from the intensities of the reflected light incident on our eyes. In this chapter, we shall see in some detail what happens inside our eye that gives us a sensation of color. Along with that, we shall also have a look at different *color models*, that are essentially alternative representations of color aimed at simplifying color manipulation. Color models are the second aspect of understanding the fourth stage. In addition, we shall have some discussion on texture synthesis, which acts as an improvement of the simple lighting models to generate more realistic effects.

5.1 PHYSIOLOGY OF VISION

We came across the fact that color is a psychological phenomenon. The physiology of our visual system gives rise to this psychological behavior. Figure 5.1 shows a schematic of the physiology of our visual system, marking the important components.

The light rays incident on the eye pass through the cornea, the pupil, and the lens in that sequence to finally reach the retina. In between, the rays get refracted (by the cornea and

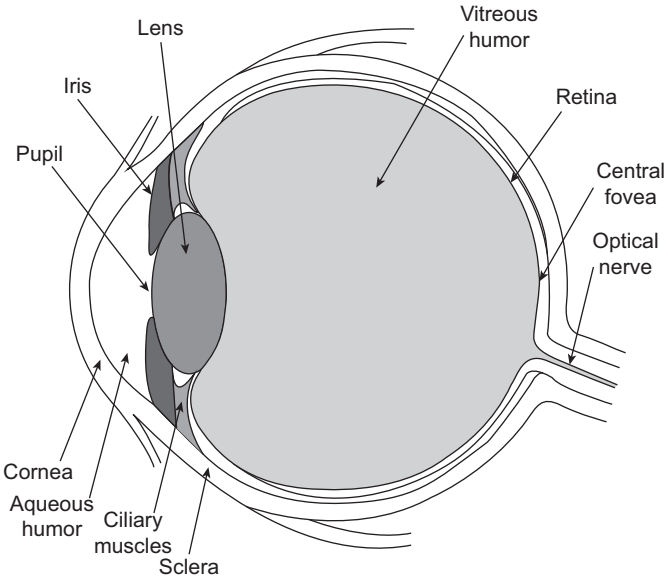


Fig. 5.1 Schematic diagram of our visual system

the lens) so as to focus the images on the retina. The amount of light entering the eye is controlled by the iris (by dilating or constricting the pupil size). The retina is composed of optical nerve fibres and photoreceptors that are sensitive to light. There are two types of photoreceptors—*rods* and *cones*. The concentration of rods is more in the peripheral region of the retina, whereas the cones are mainly present in a small central region of the retina known as the *fovea*. More than one rod can share an optic nerve. In such cases, the nerve pools the stimulation by all the connected rods and aids sensitivity to lower levels of light. In contrast, there are more or less one optic nerve fibre for each cone, which aids in image resolution or *acuity*. Vision accomplished mainly with cones is known as *photopic* vision, whereas the vision accomplished mainly by rods is called *scotopic* vision. Only in photopic vision, we can perceive colors; in scotopic vision, weak lights are visible as a series of grays.

As we know, what we call *visible light* actually refers to a spectrum of frequencies of electromagnetic (light) waves. At the one end of the spectrum is the red light (frequency: 4.3×10^{14} Hz, wavelength: 700 nm) and at the other end is the violet light (frequency: 7.5×10^{14} Hz, wavelength: 400 nm). Light waves within this range are able to excite the cones in our eye, giving the photopic vision (i.e., perception of color). There are three types of cones present in the eye: **L** or **R**, which are most sensitive to the red light; **M** or **G**, which are most sensitive to the green light (wavelength: 560 nm); and **S** or **B**, which are most sensitibe to blue light (wavelength: 430 nm). Perception of color results from the stimulation of all the three cone types together (thus, this is also referred to as the *tristimulus* theory of vision).

5.2 COLOR MODELS

In computer graphics, we are interested in synthesizing colors so as to create realistic scenes. The presence of metamers (see the following boxed text) makes this possible in principle. Metamers imply that we can create any color without actually worrying about the optics

What is metamerism? How is the idea helpful in computer graphics?

Light incident on our eye is composed of different frequencies (light spectrum). The component frequencies excite the three cone types L, M, and S in different ways, giving us the sensation of a particular color. However, a particular color perception can result from different spectra. That means, sensation of a color C resulting from an incident light spectrum S_1 can also result from a different light spectrum S_2. This optical behavior is known as metamerism and the different spectra that result in the sensation of the same color are known as metamers.

Metamers imply that we don't need to know the exact physical process behind the perception of a particular color. Instead, we can always come up with an artificial way to generate that sensation (metamers). Thus, it is possible to generate realistic scenes artificially, without knowing the actual optical process.

behind it (how it is perceived in our eye). What we can do is to come up with a set of *basic* (or *primary*) colors. We then mix these colors in appropriate amounts to synthesize the desired color. This gives rise to the notion of *color models* (i.e., ways to represent and manipulate colors).

5.2.1 RGB Color Model

As we saw before, there are three cone types in the retina: L, M, and S. Cones of type L get excited mostly by red light, M by green light, and S by blue light. Therefore, the incident light excites these three cones in different ways, which results in the photopic vision (i.e., color perception). Therefore, we can think of color as a mixture of the three primary colors: red, green, and blue. We need to mix the primary colors in appropriate amounts to synthesize a desired color. This model of color, where we assume that any color is a combination of the red, green, and blue colors in appropriate amount, is known as the RGB model (see Fig. 5.2a). This is an *additive* model in the sense that any color is obtained by *adding* proper amounts of red, green, and blue colors.

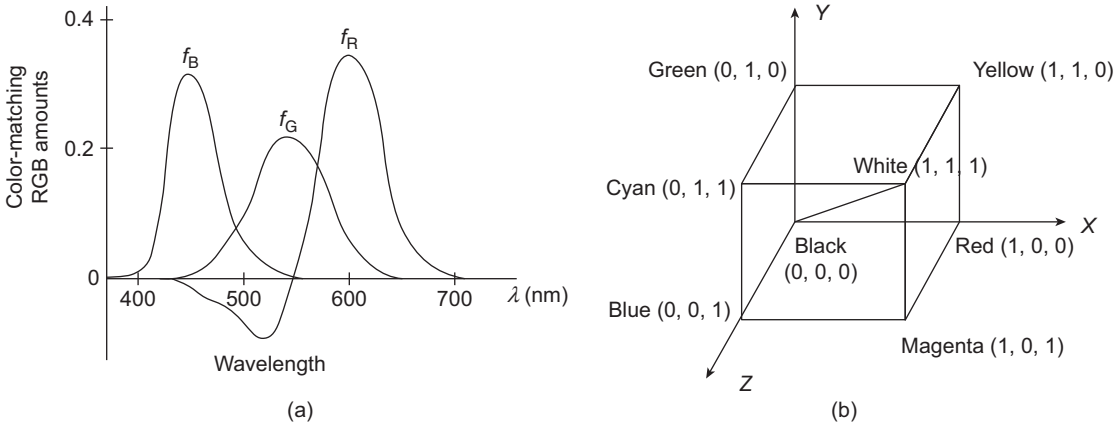


Fig. 5.2 The RGB model. In (a), the basic idea is to illustrate with the primary light waves and the amount of the light required to generate colors. The 3D cube due to the RGB model is shown in (b). Color is represented as a point in the cube, corresponding to specific amount of red, green, and blue.

Since there are three primaries, we can think of a color as a point in a three dimensional color space. The three axes of the space correspond to the three primary colors. If we are using the normalized values of the colors (within the range [0, 1]), then we can visualize the RGB model as a 3D color cube, as shown in Fig. 5.2(b). The cube is the *color gamut* (i.e., set of all possible colors) that can be generated by the RGB model. The origin or the absence of the primaries represent the black color whereas we get the white color when all the primaries are present in equal amount. The diagonal connecting the black and white colors represents the shades of gray. The yellow color is produced when only the red and green colors are added in equal amounts in the absence of blue. Addition of only blue and green in equal amounts with no red produces cyan, while the addition of red and blue in equal amounts without green produces magenta.

5.2.2 XYZ Color Model

Although the RGB model is very intuitive owing to its direct correspondence to the tristimulus color theory, it has its limitations. For some of the colors in the visible light spectrum (colors around the 500 nm wavelength, see Fig. 5.2a), the additive model fails. The colors in this region can not be obtained by adding any amounts of the three primary colors. In fact, this is a problem for any color model based on naturally occurring primary colors: no model can account for all the colors in the visible spectrum. In order to overcome this problem, the Commission Internationale de l'Eclairage (CIE) proposed three hypothetical lightwaves (usually denoted by the letters X, Y, and Z) for use as primary colors. The idea is, we should be able to generate any wavelength λ in the visible range by positive combinations of the three primaries, as illustrated in Fig. 5.3. This model is known as the XYZ model due to the name given to the primary colors.

By definition, the XYZ model is additive. Therefore, any color C can be represented in the XYZ color space (which is three-dimensional like the RGB color space) as an additive combination of the primaries using the unit vectors along the axes (corresponding to the

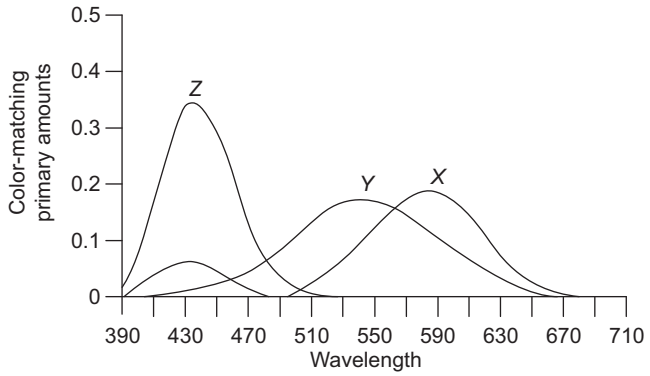


Fig. 5.3 The figure illustrates the three hypothecial lightwaves designed for the XYZ model and the amounts in which they should be mixed to produce a color in the visible range. Compare this figure with Fig. 5.2(a) and notice the difference.

three primaries), as shown in Eq. 5.1.

$$C = X\hat{X} + Y\hat{Y} + Z\hat{Z} \tag{5.1}$$

For convenience, the amounts X, Y, and Z of the primary colors used to generate C are represented in normalized forms. Calculations of the normalized forms are shown in Eq. 5.2.

$$x = \frac{X}{X + Y + Z} \tag{5.2a}$$

$$y = \frac{Y}{X + Y + Z} \tag{5.2b}$$

$$z = \frac{Z}{X + Y + Z} \tag{5.2c}$$

Since $x + y + z = 1$, we can represent any color by specifying just the x and y amounts. The normalized x and y are called the *chromaticity values*. If we plot the chromaticity values, we get a tongue-shaped curve as shown in Fig. 5.4. This curve is known as the *CIE chromaticity diagram*. The spectral (pure) colors are represented by points along the curve. However, the line joining the violet and red spectral points, called the

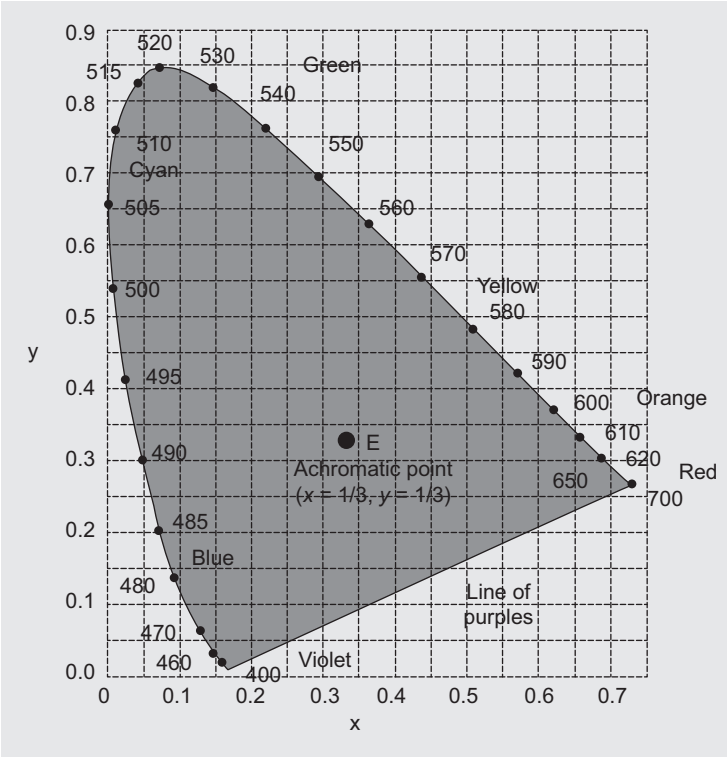
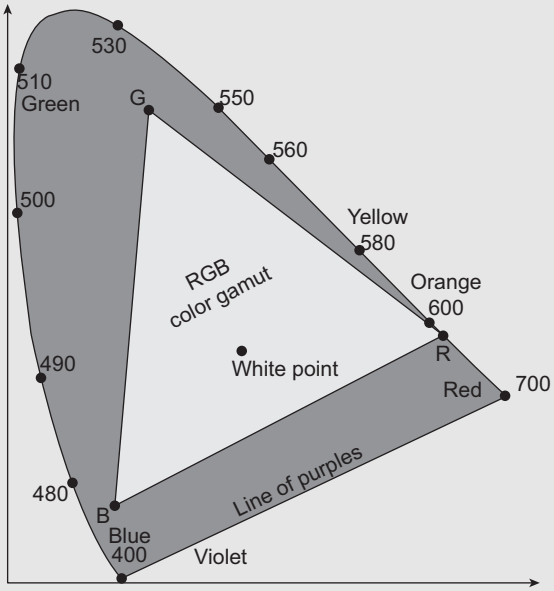


Fig. 5.4 CIE chromaticity diagram for the visible color spectrum

Importance of the CIE chromaticity diagram

The CIE chromaticity diagram represents the whole range of perceptible colors in 2D. As we have already mentioned, no other set of natural primaries (representing a color model) can generate all possible colors. Thus, any sets of primaries (such as red, green, and blue) generates only a subset of the colors represented by the CIE chromaticity diagram. Given the *x* and *y* values, we can plot the primaries on the chromaticity diagram and then join those points using lines to visualize the color gamut represented by the set of primaries. In this way, we can visualize and compare the color gamuts generated by different models (sets of primaries). Thus, the chromaticity diagram allows us to compare different color models. As an example, the following figure shows the RGB color gamut within the chromaticity diagram.



purple line, is not part of the spectrum. Interior points in the diagram represent all possible visible colors. Therefore, the diagram is a 2D representation of the XYZ color gamut. The point E represents the white-light position (a standard approximation for average daylight).

5.2.3 CMY Color Model

As we saw before, the RGB color model is additive. That means, a color is generated by adding appropriate amounts of the primaries red, green, and blue. The video display devices such as the CRT are designed using this color model. In those devices, each pixel location contains three *dots*, each corresponding to one of the primary colors. These dots are excited individually, resulting in the emission of the corresponding primary lights with appropriate intensity. When we see it, they appear to represent the color. However, in many hard-copy devices such as printers and plotters, a different process is used. In such cases, a color picture is produced by coating a paper with color pigments (inks). When we look at the paper,

reflected lights from the points containing the pigments comes to our eye giving us the perception of color. This process is, however, not additive. The color perception results from the *subtraction* of primaries.

We can form a subtractive color model with the primaries cyan, magenta, and yellow. Consequently, the model is called the CMY model. The primary cyan is a combination of blue and green (see Fig. 5.2b). Thus, when white light is reflected from a cyan pigment on a paper, the reflected light contains these two colors only and the red color is absorbed (subtracted). Similarly, green component is subtracted by the magenta pigment and the primary yellow subtracts the blue component of the incident light. Therefore, we get the color due to the subtraction of the red, green, and blue components from the white (reflected) light by the primaries.

We can depict the CMY model as a color cube in 3D in the same way we did it for the RGB model. The cube is shown in Fig. 5.5. Note how we can find the location of the corner points. Clearly origin (absence of the primaries) represents the white light. When all the primaries are present in equal amounts, we get black color since all the three components of the white light (red, green, and blue) are absorbed. Thus, the points on the diagonal joining white and black represents different shades of gray. When only cyan and magenta are present in equal amount without any yellow color, we get blue because red and green are absorbed. Similarly, presence of only cyan and yellow in equal amount without any magenta color results in green and the red color results from the presence of only the yellow and magenta in equal amounts without cyan.

The CMY model for hardcopy devices is implemented using an arrangement of three ink dots, much in the same way three phosphor dots are used to implement the RGB model on a CRT. However, sometimes four ink dots are used instead of three, with the fourth dot representing black. In such cases, the color model is called the CMYK model with K being the parameter for black. For black and white or gray-scale images, the black dot is used.

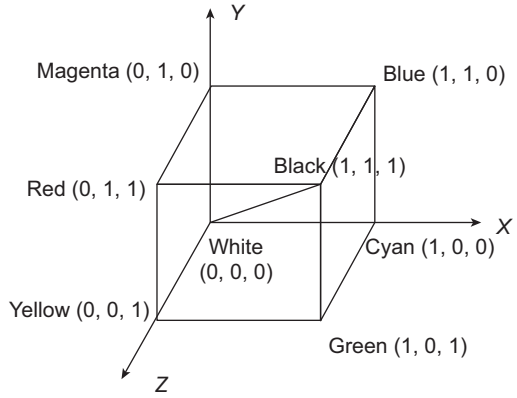


Fig. 5.5 CMY color cube—Any color within the cube can be described by subtracting the corresponding primary values from white light

We can convert the CMY color representation to the RGB representation and vice versa, through simple subtraction of column vectors. In order to convert CMY representation to RGB, we use the following subtraction.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The opposite conversion (from RGB to CMY representation) is done in a likewise manner, as given here.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

5.2.4 HSV Color Model

If you are using a drawing application, you need some interface to color the pictures you draw. The interface can be built based on the color models we discussed in the preceding sections. For example, there can be an interface through which you enter the R, G, and B values in some format and the system generates the corresponding color using the additive model. Clearly, this is not very intuitive from a users' perspective: in order to generate a color, you *need to know* the amount of the primary colors.

Artists follow a different approach to color their pictures. In order to generate a specific color, they first choose a saturated (pure) color (e.g., red, green, blue, yellow, etc.). Next, they add *white* to the pure color to decrease saturation. This is known as *tint*. In order to generate *shades*, black is added to the pure color. Both white and black can also be added to the pure color to generate *tones*. Thus, any color is generated by the process of tinting, shading, or toning. From an artist's (user's) perspective, this approach is more intuitive than the one where we need to know the amount of the primary colors required to generate a color.

The intuitive color generation process is replicated with the HSV color model. The acronym HSV stands for *Hue*, *Saturation*, and *Value*. The color space represented by the model is the hexcone shown in Fig. 5.6. The pure colors are represented by the boundary of the hexagon on top. These colors are identified by the hue (H) value, which is the angle the color make with the axis that connects the white (at the center) and red colors on the hexagon (white to red is the positive axis direction about which the hue angle is measured). Thus, red have a hue value of 0 whereas cyan has a value of 180. The saturation (S) value determines the purity of the color: any color on the boundary has the highest S value of 1.0; as we move towards the center of the hexcone, the saturation value decreases towards the minimum (0.0). The parameter value (V) indicates the brightness (intensity) of the color. On the hexagonal plane, the brightness is the maximum (1.0). As we move towards the apex of the hexcone, V decreases till it reaches 0.0.

We can relate the operations on the HSV model to that of the intuitive coloring process. When we are on any cross-sectional plane (same or parallel to the top hexagon) of the hexcone (fixed V) and moving from any boundary color (fixed H) towards the center of the

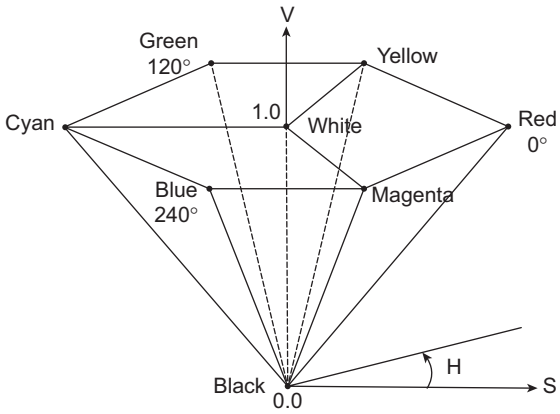


Fig. 5.6 HSV color space. Movement from boundary towards center on the same plane represents tinting; movement across planes parallel to the V axis represents shading; and movement across planes from boundary towards center represent toning.

plane, we are reducing S. Reduction in S value is equivalent to the tinting process. Alternatively, when we move from a point on any cross-sectional plane (same or parallel to the top hexagon) towards the hexcone apex, we are only changing V. This is equivalent to the shading process. Any other movement (from boundary to the center across planes) in the hexcone represents the toning process.

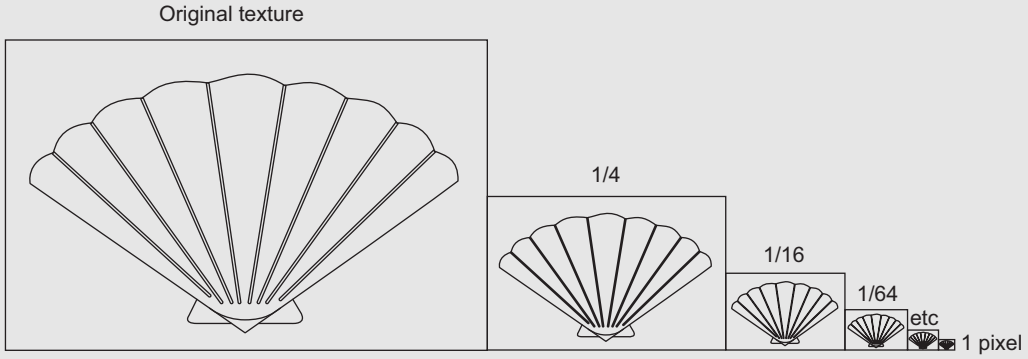
5.3 TEXTURE SYNTHESIS

When we apply lighting models to assign colors (or shades of gray), we get objects with *smooth* surfaces. However, the surfaces that we see around us contain complex geometric patterns or textures on top of the surface colors and are usually rough. As an example, consider the textured surface shown in Fig. 5.7. Such textures or roughness cannot be synthesized with the lighting model alone. Various techniques are used to make the synthesized surfaces look realistic. In this section, we shall discuss the various texture synthesis techniques.

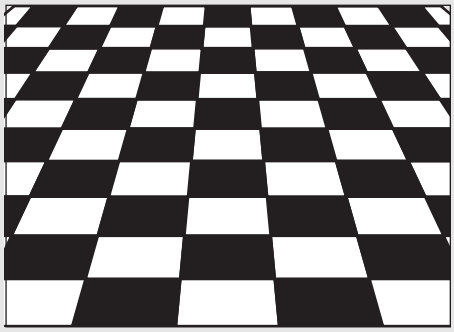
Fig. 5.7 A block made from wood. Notice the patterns (texture) on the surface. It is not posible to generate such patterns with only the lighting model.

MIPMAP

MIPMAPs are special types of projected texturing method. MIP stands for **M**ultum **I**n **P**arvo or *many things in a small space*. In this technique, a series of texture maps with decreasing resolutions, for the same texture image, are stored, as illustrated in the following figure.



These different maps are used to *paste* textures at different places of the surface, so as to get a realistic effect. The method is useful for imposing textures on surfaces with perspective projection, as shown in the following figure. As we can see in the following figure, the texture on the *near* part is synthesized with the largest texture map; smaller sized maps are subsequently used to synthesize textures on the *far* side. Obviously, MIPMAP takes more storage space for storing the different versions of the same texture image.



We can broadly categorize the various texture synthesis techniques into the following three types[1].

1. Projected texture
2. Texture mapping
3. Solid texture

5.3.1 Projected Texture

As we know, a computer screen can be viewed as a 2D array of pixels. The scenes are generated by assigning appropriate color values to these pixels. Thus we have a 2D array of pixel values. On the surface, we want to generate a particular texture. What we can do is somehow *create* the texture pattern and *paste* it on the surface.

[1]The names of the categories, however, are not standard. You may find different names used to denote the same concepts.

Example 5.1

Texture mapping method

Consider the situation shown in Fig. 5.8. On the left side is a (normalized) texture map defined in the (u, w) space. This map is to be 'pasted' on a 50×50 square area in the middle of the object surface as shown in the right side figure. What are the linear mappings we should use?

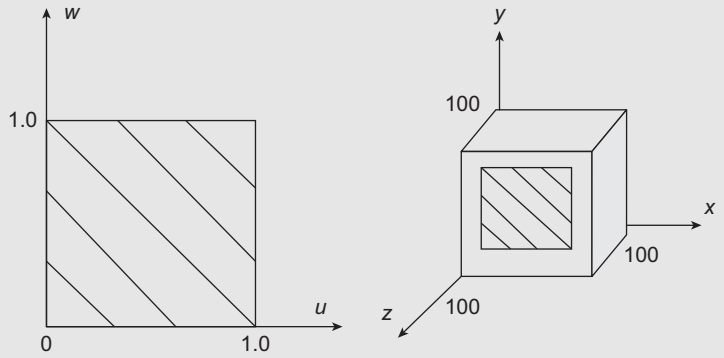


Fig. 5.8

Solution As we can see, the target surface is a square of side 100 units, on a plane parallel to the XY plane. Therefore, the parametric representation of the target area (middle of the square) is,

$$x = \theta \text{ with } 25 \leq \theta \leq 75$$
$$y = \phi \text{ with } 25 \leq \phi \leq 75$$
$$z = 100$$

Now let us consider the relationships between the parameters in the two spaces with respect to the corner points.

The point $u = 0, w = 0$ in the texture space is mapped to the point $\theta = 25, \phi = 25$.
The point $u = 1, w = 0$ in the texture space is mapped to the point $\theta = 75, \phi = 25$.
The point $u = 0, w = 1$ in the texture space is mapped to the point $\theta = 25, \phi = 75$.
The point $u = 1, w = 1$ in the texture space is mapped to the point $\theta = 75, \phi = 75$.

We can substitute these values into the linear mappings $\theta = Au + B$, $\phi = Cw + D$ to determine the constant values. The values thus determined are: A = 50, B = 25, C = 50, and D = 25 (left as an exercise for the reader). Therefore, the mappings we can use to synthesize the particular texture on the specified target area on the cube surface are $\theta = 50u + 25$, $\phi = 50w + 25$.

This idea is implemented in the projected texture method. We create a *texture image*, also known as *texture map* from synthesized or scanned images. The map is a 2D array of color values. Each value is called a *texel*. There is a one-to-one correspondance between the texel array and the pixel array. We now replace the pixel color values with the corresponding texel values to mimic the 'pasting of the texture on the surface'. The replacement can be done in one of the following three ways.

1. We can replace the pixel color value on a surface point with the corresponding texel value. This is the simplest of all.
2. Another way is to *blend* the pixel and texel values. Let C be the color after blending the pixel value C_{pixel} and the texel value C_{texel}. Then, we can use the following equation for smooth blending of the two: $C = (1 - k)C_{pixel} + kC_{texel}$ where $0 \leq k \leq 1$.
3. Sometimes a third approach is used in which we perform a *logical* operation (AND, OR) between the two values (pixel and texel) represented as bit strings. The outcome of the logical operation is the color of the surface point.

Projected texture method is suitable when the target surfaces are relatively flat and facing the reference plane (roughly related to the screen, as we shall see later). However, for curved surfaces, it is not very useful and we require some other method, as discused in the next section.

5.3.2 Texture Mapping

On a curved surface, simple *pasting* of pre-synthesized texture patterns does not work. We go for a more general definition of the texture map. Now, we asume the texture map to be defined in a 2D *texture space*, whose principle axes are typically denoted by the letters **u** and **w**. The object surface is represented in parametric form, usually denoted by the symbols θ and ϕ. Two mappings are then defined from the texture space to the object space as: $\theta = f(u, w), \phi = g(u, w)$.

In the simplest case, the mapping functions are assumed to be linear. Therefore, we can write: $\theta = Au + B, \phi = Cw + D$. **A**, **B**, **C**, and **D** are constants, whose values can be determined from the relationships between known points in the two spaces (for example, the corner points of the texture map and the corresponding surface points as illustrated in the example).

5.3.3 Solid Texture

While texture mapping is useful to deal with curved surfaces, it is still difficult to use in many situations. In texture mapping, we need to define the mapping between the two spaces. However, for complex surfaces, it is difficult to determine the mapping. Also in situations where there should be some 'continuity' of the texture between adjacent surfaces (see Fig. 5.9), the methods we discused so far are not suitable. In such cases, we use the solid texturing method.

In this method, a texture is defined in a 3D texture space (note that in the previous methods, texture maps were defined in 2D), whose principle axes are usually denoted by the letters **u**, **v**, and **w**. When rendering an object, the object is placed in the texture space through *transformations*. Thus a point P(x,y,z) on the object surface is tranformed to a point P′(u,v,w) in the texture space. The color value associated with P′ is then used to color the corresponding surface point.

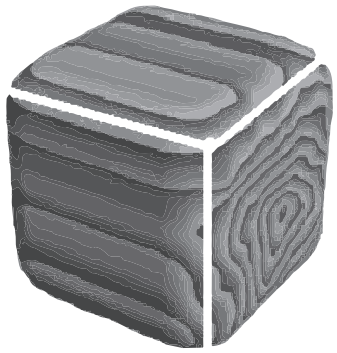


Fig. 5.9 An example situation where solid texturing method is required. The white lines show the surface boundaries. Note the continuation of the texture patterns across adjacent surfaces.

SUMMARY

In this chapter, we have discussed the fundamental idea behind the sensation of color, through a brief discussion on the physiology of vision. We learnt that the three cone type photoreceptors are primarily responsible for our perception of color, which gives rise to the Tristimulus theory of color. We also learnt that the existence of metamers, which are the various spectra related to the generation of a color, makes it possible to synthesize colors artificially, without mimicking the exact natural process.

Next, we discussed about the idea of representing colors through the color models. These models typically use a combination of three primary colors to represent any arbitrary color. Two types of combinations are used in practice: additive and subtractive. We discussed two additive color models, namely the RGB and the XYZ models. The CMY model is discussed to illustrate the idea of the subtractive model. Finally, we discussed the HSV model, which is primarily used to design user interfaces for interactive painting/drawing applications.

The third topic we learnt about in this chapter is synthesis of textures/patterns on a surface, to make the surfaces look realistic. Three types of textures synthesis techniques are introduced. In the projected texture method, a texture pattern (obtained from a synthesized or scanned image) is imposed on a surface through the use of blending functions. In the texture mapping technique, a texture map/pattern defined in a two-dimensional space is mapped to the object space. The solid texturing method extends the texture mapping idea to three dimensions, in which a 3D texture pattern is mapped to object surfaces in three dimensions.

Once the coloring is done, we transform the scene defined in the world coordinate system to the eye/camera coordinate system. This transformation, known as the view transformation, is the fourth stage of the 3D graphics pipeline, which we shall learn in Chapter 6.

BIBLIOGRAPHIC NOTE

More discussion on human visual system and our perception of light and color can be found in Glassner [1995]. Wyszecki and Stiles [1982] contains further details on the science of color. Color models and its application to computer graphics are described in Durrett [1987], Hall [1989] and Travis [1991]. Algorithm for various color applications are presented in the *computer gems* series of books (Glassner [1990], Arvo [1991], Kirk [1992], Heckbert [1994] and Paeth [1995]). More on texture-mapping can be found in Demers [2002].

KEY TERMS

Additive model – a color model that represents arbitrary colors as a sum (addition) of primary colors
CIE chromaticity diagram – the range of all possible colors in two-dimension
CMY color model – a subtractive color model in which the cyan, magenta, and yellow are the primary colors
Color gamut – the set of all possible colors that can be represented by a color model
Color models – the ways to represent and manipulate colors
Cones – one type of photoreceptors that help in image resolution or acuity
HSV color model – a color model typically used to design the user interfaces in painting applications
Metamerism – the phenomenon that color perception can result from different spectra
Metamers – different spectra that gives rise to the perception of the same color
MIPMAP – *Multum In Parvo* Mapping, which is a special type of projected texturing technique
Photoreceptors – the parts of the eye that are sensitive to light
Primary colors – the basic set of colors in any color model, which are combined together to represent arbitrary colors
Projected texture – the technique to synthesize texture on a surface by blending the surface color with the texture color
RGB color model – an additive color model in which the red, green, and blue colors are the three primary colors
Rods – one type of photoreceptors that are sensitive to lower levels of light
Solid texturing – the texture mapping technique applied in three-dimension
Subtractive model – a color model that represents arbitrary colors as a difference (subtraction) of primary colors
Texel – the color of each pixel in the texture map grid
Texture image/texture map – a grid of color values obtained from a synthesized/scanned image
Texture mapping – the technique to synthesize texture on a surface by mapping a texture defined in the texture space to the object space
Tristimulus theory of vision – the theory that color perception results from the activation of the three cone types together
Visible light – a spectrum of frequencies of the electromagnetic light wave (between 400 nm to 700 nm wavelength)
XYZ color model – an additive standardized color model in which there are three hypothetical primary colors denoted by the letters X, Y, and Z.

EXERCISES

5.1 Explain the process by which we sense colors.
5.2 It is not possible to exactly mimic the lighting process that occurs in nature. If so, how does computer graphics work?
5.3 What is the basis for developing models with three primary colors such as RGB?
5.4 Discuss the limitation of the RGB model that is overcome with the XYZ model.
5.5 Briefly discuss the relationship between the RGB and the CMY models. When is the CMY model useful?
5.6 Explain the significance of using the HSV model instead of the RGB or CMY models.
5.7 Mention the three broad texture synthesis techniques.
5.8 Explain the key idea of the projected texture method. How are the texels and pixels combined?
5.9 Explain the concept of MIPMAP. How is it different from projected texture methods?

5.10 Discuss how texture mapping works. In what respect is it different from projected texture methods? When do we need it?

5.11 Discuss the basic idea behind solid texturing. When do we use it?

5.12 Consider a cube ABCDEFGH with side length 8 units; E is at origin, EFGH on the XY plane, AEHD on the YZ plane and BFEA on the XZ plane (defined in a right-handed coordinate system). The scene is illuminated with an ambient light with an intensity of 0.25 units and a light source at location (0,0,10) with an intensity of 0.25 unit. An observer is present at (5,5,10). Assume $k_a = k_d = k_s = 0.25$ and $n_s = 10$ for the surfaces. We want to render a top view of the cube on the XY plane. A texture map is defined in the uw space as a circle with center at (1.0,1.0) and radius 1.0 unit. Any point p (u,w) within this circle has intensity $\frac{u}{u+w}$. We need to map this texture on a circular region of radius 3 units in the middle of the top of the cube. What would be the color of the points P1(4,2,8) and P2(3,1,8), assuming Gouraud shading (ignore attenuation)? [Hint: See Example 4.1 in Chapter 4 and Example 5.1].

CHAPTER 6 3D Viewing

Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the 3D viewing transformation stage and its importance in computer graphics
- Set-up the 3D viewing coordinate reference frame
- Understand the mapping from the world coordinate frame to the viewing coordinate frame
- Get an overview of the parallel and perspective projections with subcategories
- Learn to perform parallel projection of a 3D scene in the view coordinate frame to the view plane
- Learn to perform perspective projection of a 3D scene in the view coordinate frame to the view plane
- Understand the concept of canonical view volumes
- Learn to map objects from clipping window to viewport

INTRODUCTION

Let us recollect what we have learnt so far. First, we saw how to represent objects of varying complexities in a scene (Chapter 2). Then, we saw how to put those objects together through modeling transformations (Chapter 3). Once the objects are put together to synthesize the scene in the world coordinate system, we learnt how to apply colors to make the scene realistic (Chapters 4 and 5). All these discussions up to this point, therefore, equipped us to synthesize a *realistic* 3D scene in the world coordinate system. When we show an image on a screen, however, we are basically showing a *projection* of a *portion* of the 3D scene.

The process is similar to that of taking a photograph. The photo that we see is basically a projected image of a portion of the 3D world we live in. In computer graphics, this process is simulated with a set of stages. The very first of these stages is to *transform* the 3D world coordinate scene to a *3D view* coordinate system (also known as the *eye* or *camera* coordinate system). This process is generally known as the *3D viewing transformation*. Once this transformation is done, we then *project* the transformed scene onto the *view plane*. From the view plane, the objects are projected onto a *viewport* in the device coordinate system. In this

chapter, we shall discuss about the 3D viewing, projection, and viewport transformation stages.

6.1 3D VIEWING TRANSFORMATION

Let us go back to our photograph analogy and try the understand the process in a little more detail. What is the first thing we do? We point our camera to a particular direction with a specific orientation, so as to capture the desired part of the scene. Then, we set our *focus* and finally click the picture. Focusing is the most important part here. Through this mechanism, we get to know (or at least estimate) the quality and coverage of the picture taken. To set focus, we look-at the scene *through* the viewing mechanism provided in the camera. Note the difference: instead of looking at the scene directly, we are looking at it through the camera. In the former case, we are looking at the scene in its world coordinate system. In the latter case, we are looking at a different scene, one that is changed by the arrangement of lenses in the camera, to aid us in our estimation. Therefore, in taking a photograph with a camera, we actually change or transform the 3D world coordinate scene to a description (in another coordinate system) characterized by the camera parameters (position and orientation). The latter coordinate system is generally called the view coordinate system and the transformation is known as the *viewing transformation*.

In order to simulate the viewing transformation in computer graphics, we need two things. First, we need to define the view coordinate system and then, we perform the transformation. Let us first see how we can define (or setup) the view coordinate system.

6.1.1 Setting up a View Coordinate System

The process that we are trying to simulate is visualized in Fig. 6.1. Note the three axes x_{view}, y_{view}, and z_{view} orthogonal to each other. These three are the basis vectors essentially defining the *view coordinate system*.

Although we have used the common notation (x, y, and z) in Fig. 6.1 to denote the three basis vectors of the view coordinate system, they are usually denoted by the three vectors $\vec{u}$, $\vec{v}$, and $\vec{n}$. In the subsequent discussion, we shall use the latter notation.

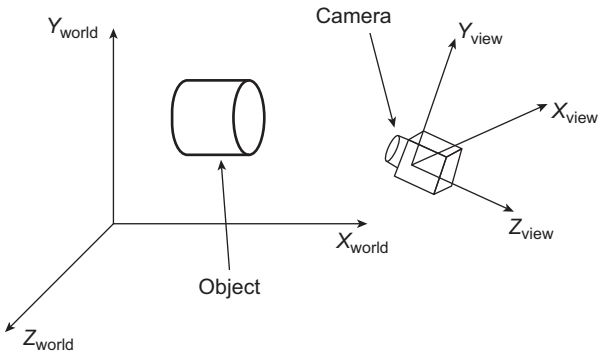


Fig. 6.1 Visualization of the view coordinate system, defined by the mutually orthogonal x_{view}, y_{view}, and z_{view} axes. The x_{world}, y_{world}, and z_{world} axes define the world coordinate system.

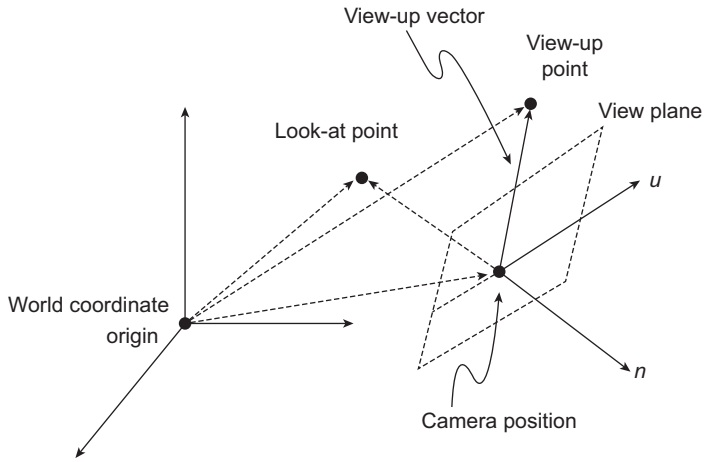


Fig. 6.2 Illustration of the determination of the three basis vectors for the view coordinate system

How do we set-up the view coordinate system. The first thing is to determine the *origin*, where the three axes meet. This is easy. We assume that the camera is represented as a point in the 3D world coordinate system. We simply choose this point as our origin (denoted by $\vec{o}$). However, determining the vectors $\vec{u}$, $\vec{v}$, and $\vec{n}$ that meet at this origin is tricky.

When we try to bring something into focus with our camera, the first thing we do is to *choose a point* (in the world coordinate). This is the *center of interest* or the *look-at point* (denoted by $\vec{p}$). Then, using simple vector algebra, we can see that $\vec{n} = \vec{o} - \vec{p}$, as depicted in Fig. 6.2. Finally, we normalize $\vec{n}$ as $\hat{n} = \dfrac{\vec{n}}{|\vec{n}|}$ to get the unit basis vector.

Next, we specify an arbitrary point (denoted by $\vec{p_{up}}$) along the direction of our head while looking through the camera. This is the *view-up* direction. With this point, we determine the *view-up* vector $\vec{V_{up}} = \vec{p_{up}} - \vec{o}$ (see Fig. 6.2). Then, we get the unit basis vector $\hat{v} = \dfrac{\vec{V_{up}}}{|\vec{V_{up}}|}$.

We know that the unit basis vector $\hat{u}$ is perpendicular to the plane spanned by $\hat{n}$ and $\hat{v}$ (see Fig. 6.2). Hence, $\hat{u} = \hat{v} \times \hat{n}$ (i.e., the vector cross-product assuming a right-handed coordinate system). Since both $\hat{n}$ and $\hat{v}$ are unit vectors, we do not need any further normalization.

Example 6.1

Consider Fig. 6.3. We are looking at the square object with vertices A(2,1,0), B(2,3,0), C(2,3,3) and D(2,1,3). The camera is located at the point (1,2,2) and the look-at point is the center of the object (2,2,2). The up direction is parallel to the positive z direction. What is the coordinate of the center of the object, after its transformation to the viewing coordinate system?

Solution First, we determine the three unit basis vectors for the viewing coordinate system.

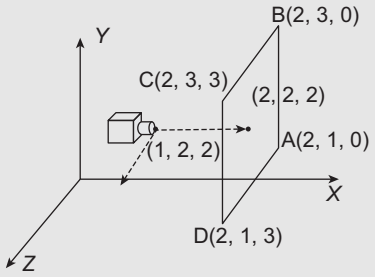


Fig. 6.3

The camera position $\vec{o}$ is (1,2,2) and the look-at point is (2,2,2). Therefore $\vec{n} = \vec{o} - \vec{p} = (-1, 0, 0) = \hat{n}$.

Since it is already mentioned that the up direction is parallel to the positive z direction, we can directly determine that $\hat{v} = (0, 0, 1)$ without any further calculations. Note that this is another way of specifying the up vector (instead of specifying a point in the up direction and computing the vector).

Finally, the cross product of the two vectors (i.e., $\hat{v} \times \hat{n}$) gives us $\hat{u} = (0, 1, 0)$.

After the three basis vectors are determined, we compute the transformation matrix M_{w2v} which is a composition of translation and rotation (i.e., $T_{w2v} = R.T$).

Since the camera position is (1,2,2), we have

$$T = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 1 & 0 & 0 & -2 \\ 1 & 0 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the unit basis vectors that we have already derived [i.e., $\hat{n}(-1, 0, 0)$, $\hat{u}(0, 1, 0)$, $\hat{v}(0, 0, 1)$], we have

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Therefore,

$$M_{w2v} = R.T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -1 \\ 1 & 0 & 0 & -2 \\ 1 & 0 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -2 \\ 1 & 0 & 0 & -2 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, the new coordinates of the object center (2,2,2) is,

$$C' = M_{w2v}C = \begin{bmatrix} 1 & 0 & 0 & -2 \\ 1 & 0 & 0 & -2 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

In other words, the object center gets transformed to the point $(0,0,-1)$ in the view coordinate system.

Determination of the viewing coordinate system

We are given the camera parameters:

1. Camera position (origin of the coordinate) $\vec{o}$
2. View-up point $\vec{p_{up}}$
3. Center of interest or the look-at point $\vec{p}$

From these parameters, the three unit basis vectors are determined as follows.

1. Determine $\hat{n}$ (unit basis vector opposite to the looking direction): $\vec{n} = \vec{o} - \vec{p}; \hat{n} = \frac{\vec{n}}{|\vec{n}|}$
2. Determine $\hat{v}$ (unit basis vector along the view-up direction): $\vec{V_{up}} = \vec{p_{up}} - \vec{o}; \hat{v} = \frac{\vec{V_{up}}}{|\vec{V_{up}}|}$
3. Determine $\hat{u}$ (the third unit basis vector): $\hat{u} = \hat{v} \times \hat{n}$

6.1.2 Viewing Transformation

Once we set up the coordinate system, the next task is to transform the scene description from the world coordinate to the view coordinate system. In order to understand the process, let us consider Fig. 6.4. The point **P** is an arbitrary point in the world coordinate, which we need to transform to the view coordinate. Assume that the view coordinate origin (the camera position) has the coordinates (o_{vx}, o_{vy}, o_{vz}) and the three basis vectors are represented as $\hat{u}(u_x, u_y, u_z)$, $\hat{v}(v_x, v_y, v_z)$, and $\hat{n}(n_x, n_y, n_z)$. We have to set up the transformation matrix

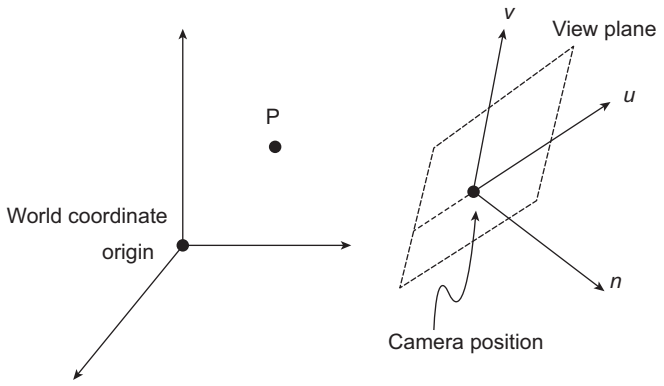


Fig. 6.4 Visualization of the transformation process. **P** is any arbitrary point, which has to be transformed to the view coordinate system. This requires translation and rotation.

How can we generate different 3D viewing effects?

We have learnt about the process of tranforming a world coordinate description to the view coordinate. There are several quantities involved in this process. By manipulating one or more of these quantities, we can generate different viewing effect.

1. If we wish to generate a composite display consisting of multiple views from a fixed camera position, we can keep the camera position fixed but systematically change the basis vector $\hat{n}$ (by choosing different look-at points).
2. We can keep $\hat{n}$ fixed but change the camera position (hypothetically by *moving* it along $\hat{n}$) in order to generate panning effects seen in animations.
3. In order to view an object from different positions, we can *move* the camera position around the object (note that $\hat{n}$ also changes).

M_{w2v}, which, when multiplied to **P**, gives the transformed point $\mathbf{P}'$ in the view coordinate system (i.e., $P' = M_{w2v}.P$).

In order to do so, we need to find out the sequence of transformations required to *align* the two coordinate systems. We can achieve this by two operations: *translation* and *rotation*. We first translate the view coordinate origin to the world coordinate origin. The necessary translation matrix T is (in homogeneous form),

$$T = \begin{bmatrix} 1 & 0 & 0 & -o_{vx} \\ 1 & 0 & 0 & -o_{vy} \\ 1 & 0 & 0 & -o_{vz} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then, we rotate the view coordinate frame to align it with the world coordinate frame. The rotation matrix R is (in homogeneous form),

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This sequence is then applied in reverse order on **P**. Thus, we get $M_{w2v} = R.T$. Hence, $P' = (R.T)P$.

6.2 PROJECTION

When we see an image on a screen, it is two-dimensional (2D). The scene in the view coordinate system, on the other hand, is three-dimensional (3D). Therefore, we need a way to *transform* a 3D scene to a 2D image. The technique to do that is *projection*. In general, projection allows us to transform objects from n dimensions to $(n - 1)$ dimensions. However, we shall restrict our discussion on projections from 3D to 2D.

In computer graphics, we project the 3D objects onto the 2D *view plane* (see Fig. 6.2). In order to do that, we define an area (usually rectangular) on the view plane that contains the projected objects. This area is known as the *clipping window*. We also define a 3D *volume*

in the scene, known as the *view volume*. Objects that lie inside this volume are projected on the clipping window. Other objects are discarded (through the clipping process that we shall discuss in Chapter 7). Note that the entire scene is not projected; instead, only a portion of it enclosed by the view volume is projected. This approach gives us flexibility to synthesize images as we want. The trick lies in choosing an *appropriate* view volume, for which we require an understanding of the different types of projections.

6.2.1 Types of Projections

The basic idea of projection is as follows: we want to project a 3D object on a 2D view plane. From each point on the object, we generate *straight lines* towards the view plane. These lines are known as *projectors*. These lines intersect the view plane. The intersection points together give us the projected image.

Depending on the nature of the projectors, projections can be broadly classified into two types: *parallel* and *perspective*. In parallel projection, the projectors are parallel to each other. This is not the case in perspective projection, in which the projectors are not parallel and converge to a *center of projection*. The idea is illustrated in Fig. 6.5. Note that the center of projection is at *infinity* for parallel projection.

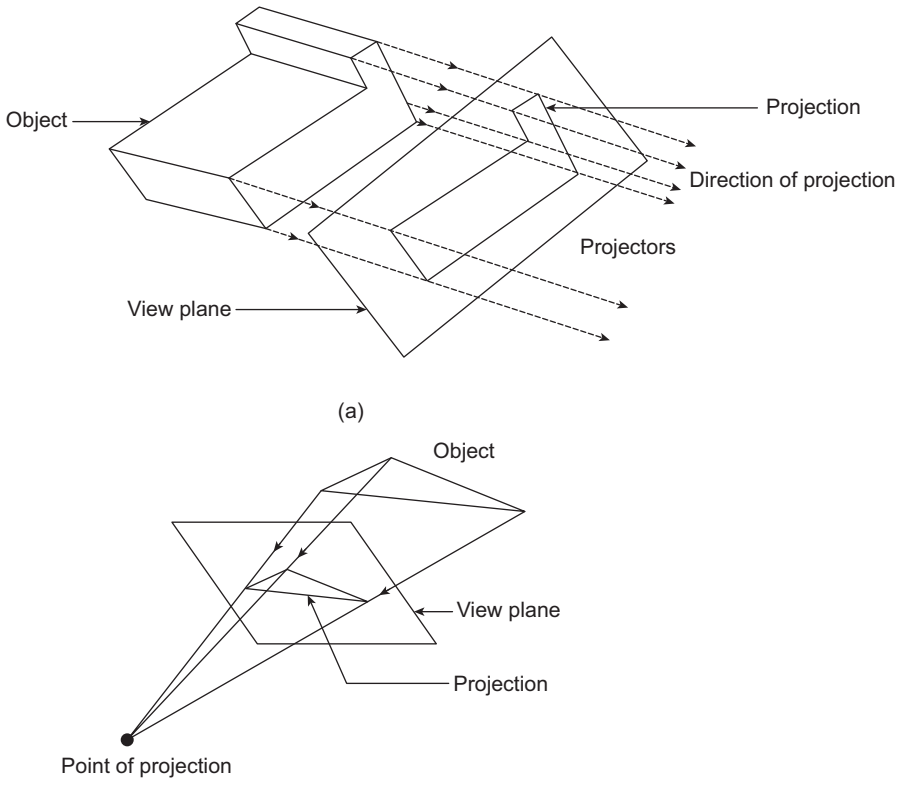


Fig. 6.5 Two types of projection (a) Parallel (b) Perspective

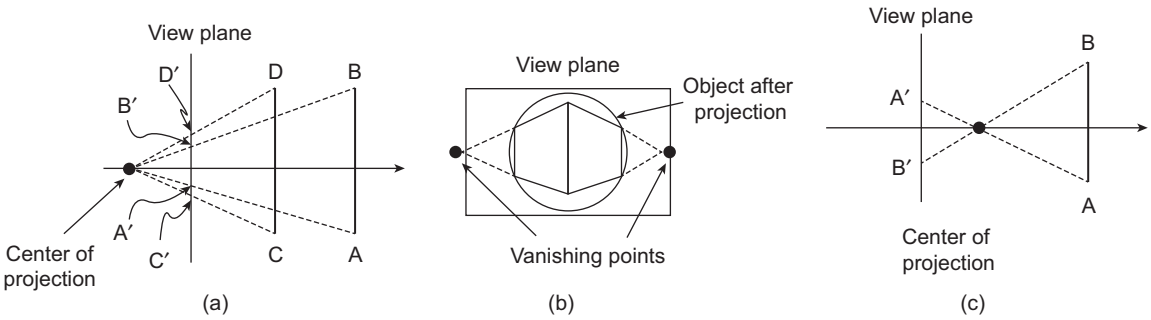


Fig. 6.6 The different types of anomalies associated with perspective projection. Foreshortening is depicted in (a). Note the projected points (A′,B′) for object AB and (C′,D′) for object CD, although both are of the same size. In (b), the concept of vanishing points is illustrated. View confusion is illustrated in (c).

Since the projectors converge at a point, perspective projection gives rise to several *anomalies* (i.e., the appearance of the object in terms of shape and size gets changed).

Perspective foreshortening If two objects of the same size are placed at different distances from the view plane, the distant object *appears* smaller than the near objects (see Fig. 6.6(a)).

Vanishing points Lines that are not parallel to the view plane *appear* to meet at some point on the view plane after projection. The point is called *vanishing point* (see Fig. 6.6(b)).

View confusion If the view plane is *behind* the center of projection, objects in front of the center of projection appear upside down on the view plane after projection (see Fig. 6.6(c)).

As you can see, the anomalies actually help in generating realistic images since this is the way we perceive objects in the real world. In contrast, the shape and size of objects are preserved in parallel projection. Consequently, such projections are not used to generate

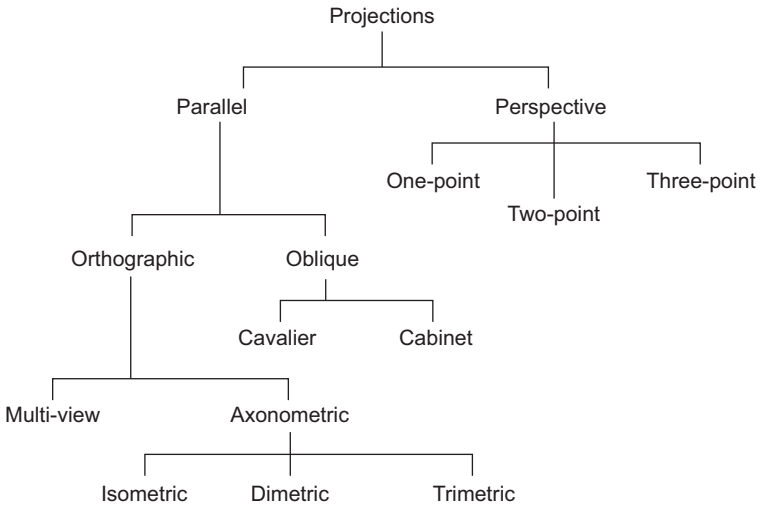


Fig. 6.7 Taxonomy of different projection types

realistic scenarios (such as in computer games or animations). Instead, they are more useful for graphics systems that deal with engineering drawings (such as CAD packages).

Although we have mentioned two broad categories of projection, there are many subcategories under parallel and perspective projection. The complete taxonomy of projections is shown in Fig. 6.7.

When projectors are *perpendicular* to the view plane, the resulting projection is called *orthographic*. There are broadly two types of orthographic projections. In *multiview orthographic projection*, *principle object surfaces* are parallel to the view plane. Three types of principle surfaces are defined: top (resulting in *top view*), front (resulting in *front view*), and side (resulting in *side view*). The three views are illustrated in Fig. 6.8.

In contrast, no principal surface (top, front, or side) is parallel to the view plane in *axonometric* orthographic projection. Instead, they are at certain angles with the view plane (see Fig. 6.9). Note that the principal faces can make *three* angles with the view plane as

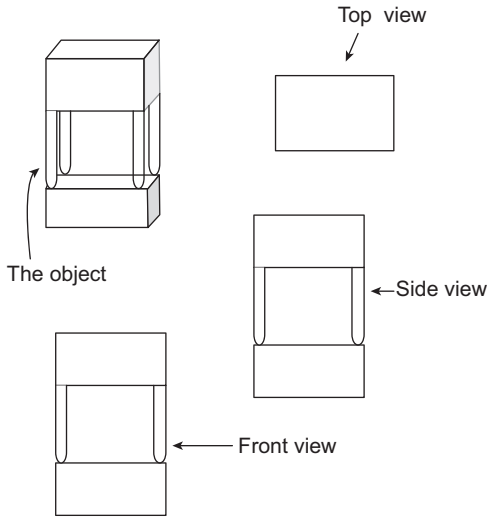


Fig. 6.8 Three types (top, side, and front view) of multiview orthographic projections

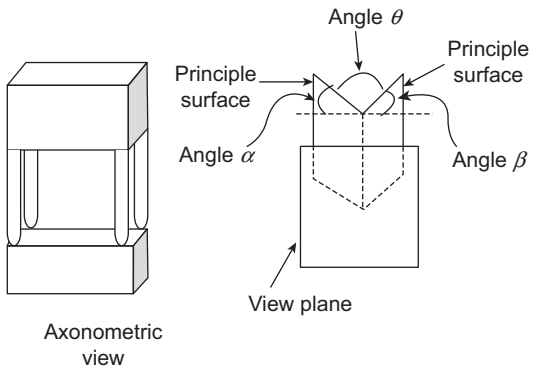


Fig. 6.9 Axonometric orthographic projection where principal surfaces are at certain angles with the view plane

illustrated in Fig. 6.9. Depending on how many of these angles are equal to each other, three types of axonometric projections are defined: *isometric*, when all the three angles are equal to each other; *dimetric*, when two of the three angles are equal to each other; and *trimetric*, when none of the angles is equal to the other.

When the projectors are not perpendicular to the view plane (but parallel to each other), we get *oblique* parallel projection, as illustrated in Fig. 6.10. In oblique projection, if lines

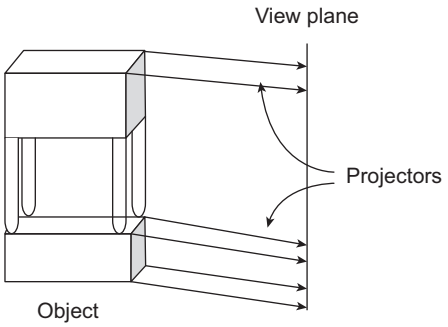


Fig. 6.10 Oblique parallel projection where projectors are not perpendicular to the view plane

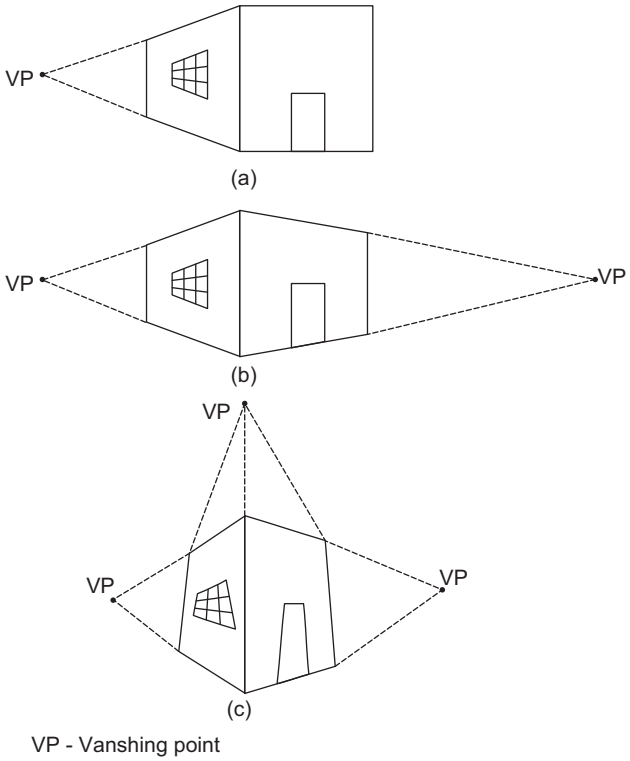


Fig. 6.11 Three types of perspective projections, depending on the number of vanishing points (a) One-point (b) Two-point (c) Three-point

perpendicular to the view plane are foreshortened by *half* after projection, it is known as *cabinet* projection. When there is no change in the perpendicular lines after projection, it is called *cavalier* projection.

Recall that in perspective projection, we have the idea of *vanishing points*. These are basically *perceived* points of convergence of lines that are not parallel to the view plane. Depending on the orientation of the object with respect to the view plane, we can have between one to three vanishing points in the projected figure. Depending on the number of vanishing points, we define perspective projections as *one-point* (see Fig. 6.11a), *two-point* (see Fig. 6.11b) or *three-point* (see Fig. 6.11c).

In the next section, we shall outline the fundamental concepts involved in computing projected points on the view plane. However, we shall restrict our discussion to the two broad classes of projections. Details regarding the individual projections under these broad classes will not be discussed, as such details are not necessary for our basic understanding of projection. *In all subsequent discussions, the term parallel projection will be used to refer to parallel orthographic projections only*.

6.2.2 Projection Transformation

Let us come back to our camera analogy. What we capture on film is 2D, after projection of the objects on the film plane (view plane). The same idea is implemented in computer graphics. After the objects are transformed to the view coordinate system, those are projected on the view plane. This is achieved through the *projection transformations*. Similar to all the other transformations we have encountered so far (modeling and view transformations), projection transformations are also performed with matrix multiplication (between the coordinate vectors and the projective transformation matrices). In this section, we shall see how the matrices are derived.

Recall that projection requires us to define a view volume, which is the region in the 3D view coordinate system that encloses all the objects to be projected. The shape of the view volume depends on the type of projection we want. For parallel projection, the view volume takes the shape of a *rectangular parallelepipe* as shown in Fig. 6.12(a). The shape is defined by its six planes (near, far, top, bottom, right and left). The *near* plane is the view plane on which the clipping window is present. A frustum is used to represent the view volume for perspective projection, as shown in Fig. 6.12(b), defined by the six planes.

In terms of the view volumes, let us try to understand the derivation of the projection matrices. We begin with parallel projection. Consider Fig. 6.13. The point **P** is in the view volume with coordinate (x, y, z). It is projected as the point P′ (x', y', z') on the clipping window. Assuming that the near plane is at a distance d along the $-z$ direction, the coordinate of P′ can be derived simply as $x' = x$, $y' = y$ and $z' = -d$.

Therefore, the transformation matrix for parallel projection is,

$$T_{par} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

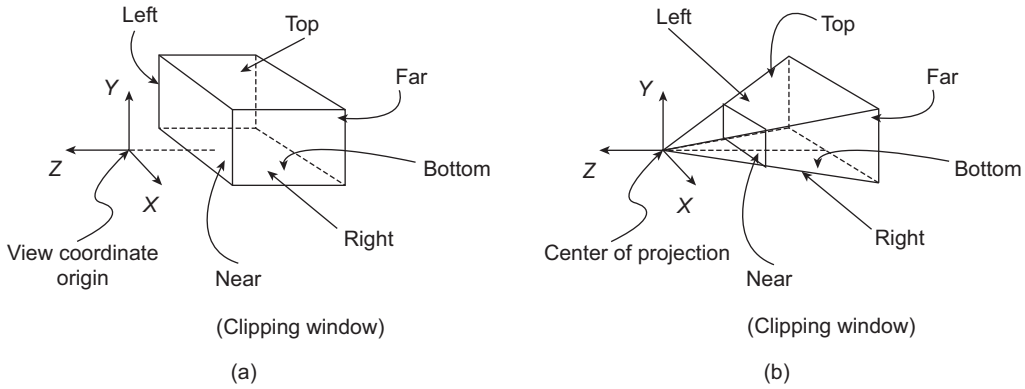


Fig. 6.12 The shape of the view volumes for the two basic projection types. The parallelepipe in (a) is used for parallel projection and the frustum in (b) is used for perspective projection.

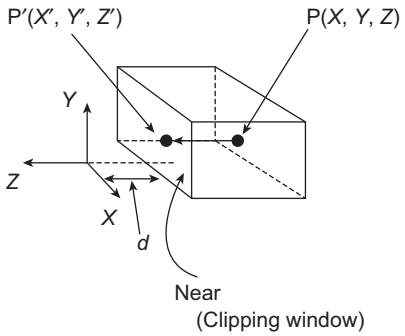


Fig. 6.13 Illustration for derivation of the transformation matrix for parallel projection

Hence, in terms of matrix multiplication, we can write,

$$P'' \begin{bmatrix} x'' \\ y'' \\ z'' \\ w \end{bmatrix} = T_{par}P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Note that the multiplication is performed in homogeneous coordinate system. Therefore, the real coordinates of P′ are $x' = \frac{x''}{w}$, $y' = \frac{y''}{w}$, and $z' = \frac{z''}{w}$.

Obviously, derivation of the transformation matrix for perspective projection involves a little more calculation. Consider Fig. 6.14, which shows a side view seen along the $-x$ direction. We need to derive the transformation matrix that projects the point $\mathbf{P}\ (x, y, z)$ to the $\mathbf{P}'(x', y', z')$.

The original and the projected points are part of two similar triangles. As a result, we can say $\frac{y}{y'} = \frac{-z}{-d}$ or $y' = y\frac{d}{z}$. In a similar way, we can derive the projected x coordinate

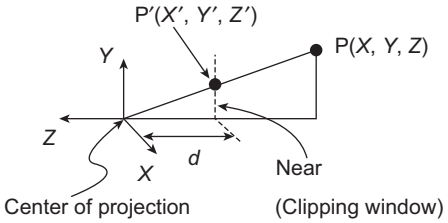


Fig. 6.14 Illustration for the derivation of the transformation matrix for perspective projection

x' as $x' = x\frac{d}{z}$. It is obvious that $z' = -d$. We can use these expressions to construct the transformation matrix in homogeneous coordinate form as,

$$T_{psp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

Finally, in terms of matrix multiplication, we can write,

$$P'' \begin{bmatrix} x'' \\ y'' \\ z'' \\ w \end{bmatrix} = T_{psp}P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

As in the case of parallel projection, we need to compute the coordinates of the projected point as $x' = \frac{x''}{w}$, $y' = \frac{y''}{w}$, and $z' = \frac{z''}{w}$, since the transformation matrix is in homogeneous form.

Example 6.2

What would be the coordinates of the object center in Example 6.1 on a view plane $z = -0.5$, if we want to synthesize images of the scene for parallel projection. Assume that the view volume is sufficiently large to encompass the whole transformed object.

Solution The coordinates of the point in the view coordinate system is $(0, 0, -1)$. We also know that the transformation matrix for parallel projection

$$T_{par} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

.

Hence, the coordinates of the point after projection will be,

$$T_{par}P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -0.5 \\ 1 \end{bmatrix}$$

In other words, the point will be projected to $(0,0,-0.5)$ on the view plane.

Example 6.3

What would happen to the point if we perform a perspective projection on the view plane $z = -0.5$ with the view coordinate origin to be the center of projection. Assume that the view volume is sufficiently large to encompass the whole transformed object.

Solution We proceed as before. The transformation matrix for perspective projection

$$T_{psp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & \frac{1}{0.5} & 0 \end{bmatrix}.$$

The point before projection is at $(0,0,-1)$. The new coordinate of the point after projection will be,

$$T_{psp}P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & \frac{1}{0.5} & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ -2 \end{bmatrix}$$

The derived point is in homogeneous form as before. However, we have -2 as the homogeneous factor. Thus, the projected point is $\left(\frac{0}{-2}, \frac{0}{-2}, \frac{1}{-2}\right)$. In other words, the point will be projected to $(0,0,-0.5)$ on the view plane.

6.2.3 Canonical View Volume and Depth Preservation

An important stage in the graphics pipeline is clipping, which we shall discuss in Chapter 7. The objective of this stage is to *remove* all the objects that lie outside the view volume. As we shall see, such removal often requires lots of calculation to determine object surface-view volume boundary intersection points. If such intersection calculations are to be performed with respect to any arbitrary view volume, computation time may increase significantly. Instead, what we can do is to define the clipping procedures with

respect to a *standard* view volume. This standard volume is known as the *canonical view volume* (CVV).

Figure 6.15 shows the CVV for parallel projection. Note that the CVV is a cube within the range [−1,1] along the x, y, and z directions[1]. As you can see, any arbitrary view volume can be *transformed* to the CVV using the scaling operations along the three axial directions.

Canonical view volume for perspective projection is a little tricky. For ease of clipping computations, perspective view frustums are also transformed to parallel CVV (i.e., the cube within the range [−1,1] along the x, y, and z directions). Clearly, this transformation is not as straightforward as in the case of parallel projection and involves composition of two types of modeling transformations: shearing and scaling. The idea is illustrated in Fig. 6.16.

With the idea of CVV, let us now try to understand the sequence of transformations that take place when a point **P** in the world coordinate is projected on the view plane. First, it gets transformed to the view coordinate system. Next, the view volume in which the point lies is transformed to CVV. Finally, the point in the CVV is projected. In matrix notation, we can write this series of steps as: $\mathbf{P}' = T_{proj}.T_{cvv}.T_{VC}\mathbf{P}$, where T_{proj} is the projection transformation matrix, T_{cvv} is the matrix for transformation to the canonical view volume and T_{VC} is the matrix for transformation to the view coordinate.

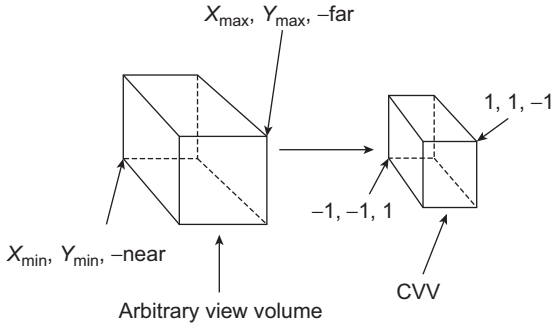


Fig. 6.15 Canonical view volume for parallel projection

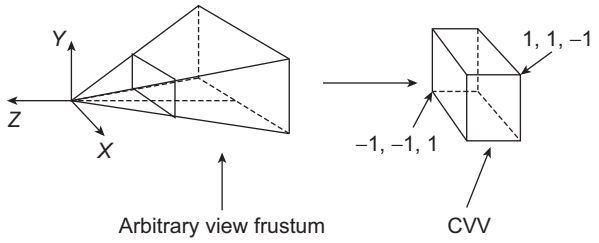


Fig. 6.16 The canonical view volume for perspective projection. Note that the frustum should be sheared along x and y directions and scaled along z direction to obtain the CVV.

[1]Another variation is also used for parallel projection in which the CVV is a unit cube that lies within the range [0,1] along each of the three axial directions.

There is one more thing that we need to remember in projection. We mentioned that in projection, 3D points are mapped to 2D. This is achieved by removing the z (depth) component of the points. However, in the implementation of graphics pipeline, the depth component is not removed. In other words, while computing the transformed coordinates of a point, its original z (depth) value is preserved in a separate storage known as the *z(depth)-buffer*. The depth information is required to perform a later stage of the graphics pipeline, namely *hidden surface removal*, which we shall discuss in Chapter 8.

6.3 WINDOW-TO-VIEWPORT TRANSFORMATION

So far, we have discussed the steps involved in transforming a point in the world coordinate to the clipping window on the view plane. Note that the clipping window is the near plane of the canonical view volume (see Fig. 6.15). For the ease of computations, it is assumed that the window is at zero depth (i.e., $z = 0$). Moreover, since the x and y extents of the window are fixed [between -1 to 1], the coordinates of the points in the window have a fixed range, irrespective of their actual position in the world coordinate. For this reason, the clipping window on the CVV is often called the *normalized* window.

Clipping window is an abstract and intermediate concept in the process of image synthesis. The points on the clipping window constitute the objects that we want to show on the screen. However, the scene may or may not occupy the whole screen. The idea is illustrated in Fig. 6.17, in which the content of the clipping window is displayed on a portion of the screen. Hence, we should distinguish between the following two concepts.

Window This is the same as (normalized) clipping window. The world coordinate objects that we want to display are projected on this window.

Viewport The objects projected on the window may be displayed on the whole screen or a portion of it. The rectangular region on the screen, on which the content of the window is rendered, is known as the *viewport*.

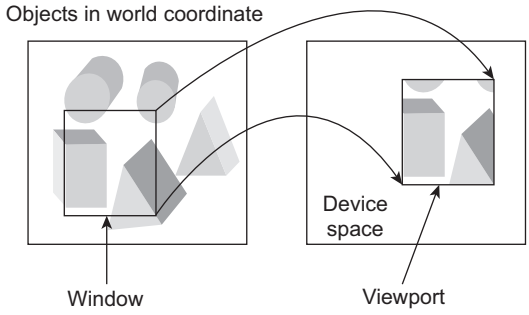


Fig. 6.17 The difference between window and viewport. Window contains the objects to be displayed (left figure). Viewport is the region on the screen, where the window contents are displayed (right figure).

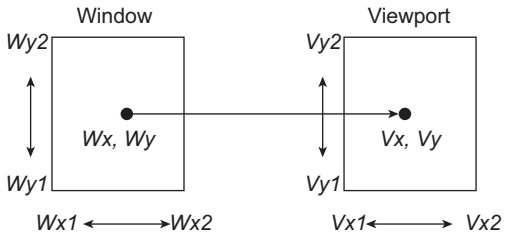


Fig. 6.18 Window-to-viewport mapping. Note that the window coordinates are generalized in the illustration instead of the normalized coordinates.

Note that the viewport is defined in the *device space*. In other words, it is defined with respect to the screen origin and dimensions. So, one more transformation is required to transfer points from the window (in the view coordinate system) to the viewport (in the device coordinate system). Let us try to understand the derivation of the transformation matrix.

Consider Fig. 6.18. The point (Wx, Wy) in the window is transformed to the viewport point (Vx, Vy). The window lies within [$Wx1$, $Wx2$] along the X axis and [$Wy1$, $Wy2$] along the Y axis. The viewport ranges between [$Vx1$, $Vx2$] and [$Vy1$, $Vy2$] along the X and Y directions, respectively. In order to maintain the relative position of the point in the viewport, we must have,

$$\frac{Wx - Wx1}{Wx2 - Wx1} = \frac{Vx - Vx1}{Vx2 - Vx1}$$

This relation can be rewritten as,

$$Vx = sx.Wx + tx$$

where,

$$sx = \frac{Vx2 - Vx1}{Wx2 - Wx1} \text{ and } tx = sx.(-Wx1) + Vx1.$$

Maintenance of relative position of the point in the viewport also implies that,

$$\frac{Wy - Wy1}{Wy2 - Wy1} = \frac{Vy - Vy1}{Vy2 - Vy1}$$

From this relation, we can derive the y-coordinate of the point in viewport Vy in a way similar to that of Vx as,

$$Vy = sy.Wy + ty$$

where,

$$sy = \frac{Vy2 - Vy1}{Wy2 - Wy1} \text{ and } ty = sy.(-Wy1) + Vy1.$$

From the expressions for Vx and Vy as derived here, we can form the viewport transformation matrix T_{vp} as,

$$T_{vp} = \begin{bmatrix} sx & 0 & tx \\ 0 & sy & ty \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, to get the point on the viewport $P_{vp}(x', y')$ from the window point $P_w(x, y)$, we need to perform the following matrix multiplication.

$$\begin{bmatrix} x'' \\ y'' \\ w \end{bmatrix} = T_{vp}P_w = \begin{bmatrix} sx & 0 & tx \\ 0 & sy & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The coordinates of P_{vp} are computed as $x' = \frac{x''}{w}, y' = \frac{y''}{w}$, since the matrices are in homogeneous form.

Example 6.4

Let us assume that the point is projected on a normalized clipping window (as you can see, the projected point in either parallel or perspective projection is (0,0,−0.5), which lies at the center of the normalized window). We want to show the scene on a viewport having lower left and top right corners at (4,4) and (6,8) respectively. What would the position of the point be in the viewport?

Solution Since the clipping window is normalized, we have $Wx1 = -1$, $Wx2 = 1$, $Wy1 = -1$ and $Wy2 = 1$. Also, from the viewport specification, we have $Vx1 = 4$, $Vx2 = 6$, $Vy1 = 4$ and $Vy2 = 8$. Therefore, $sx = \frac{6-4}{1-(-1)} = 1$, $sy = \frac{8-4}{1-(-1)} = 2$, $tx = 1. - (-1) + 4 = 5$ and $ty = 2. - (-1) + 4 = 6$. Thus, the viewport transformation matrix is, $T_{vp} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 2 & 6 \\ 0 & 0 & 1 \end{bmatrix}$. The new coordinate of the point after viewport transformation will be,

$$\begin{bmatrix} 1 & 0 & 5 \\ 0 & 2 & 6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -2.5 \\ -3 \\ -0.5 \end{bmatrix}$$

Since the derived point is in homogeneous form with −0.5 as the homogeneous factor, the transformed point is $\left(\frac{-2.5}{-0.5}, \frac{-3}{-0.5}\right)$. In other words, the viewport coordinates of the projected point will be (5,6).

Steps performed in the 3D viewing stage

In the 3D viewing stage of the graphics pipeline, objects from the world coordinate are tranfomed to the viewport objects through a series of transformations. These are,

View transformation The world coordinate objects are transformed to the view coordinate system. This step involves setting-up of the view coordinate system and transforming the objects to it.

Canonical view volume We define a view volume and transform it to the canonical view volume.

Projection transformation The objects within the canonical view volume are projected on the (normalized) clipping window (the near face of the canonical view volume).

Viewport transformation From the clipping window, the projected objects are transformed to the viewport defined in the device coordinate system.

Thus, there are altogether four transformations that take place in this stage. Along with those, this stage also involves setting-up of the view coordinate system and the view volume. Two other important components of this stage are *clipping* in which objects outside the view volume are removed (before projection) and *hidden surface removal*, both of which we shall discuss in Chapters 7 and 8.

SUMMARY

In this chapter, we learnt about the process of transforming objects from world coordinate to viewport, which is an important and essential part of the image synthesis process. The transformation takes place in distinct stages, which are analogous to the process of capturing a picture with your camera. The very first step is to set-up the view coordinate system. In this stage, the three orthogonal unit basis vectors of the view coordinate system are determined from three input parameters: the camera position or the view coordinate origin, the look-at point, and the view-up point (or view-up vector). After the formation of the view coordinate system, we transform the object to the view coordinate system.

The transformed objects are still in 3D. We transform them to 2D view plane through projection. We learnt about the two basic types of projections: parallel and perspective. In parallel projection, the projectors are parallel to each other. The projectors converge to a center of projection in perspective projection. Before projection, we first define a view volume, a 3D region that encloses the objects we want to be part of the image. For parallel projection, the view volume takes the shape of a rectangular parallelepipe. It takes the shape of a frustum for perspective projection. For computational efficiency in subsequent stages of the graphics pipeline, the view volumes are transformed to canonical view volumes which is a cube with all its points lying within the range $[-1,-1,-1]$ to $[1,1,1]$. Transformation to canonical view volume requires scaling for parallel view volume and a combination of shear and scale for perspective view volume.

The objects in the canonical volume is projected on its near or view plane, which acts as the normalized clipping window. The window is in view coordinate system. A final transformation is applied on the points in the window to transform it to the points in the viewport, which is defined in the device coordinate system.

As we have seen, totally four transformations take place in this stage: world to view coordinate transformation, view volume to canonical view volume transformation, projection transformation, and window-to-viewport transformation. When we define a view volume, the objects that lie outside neeed to be removed. This is done in the clipping stage, that we shall discuss in Chapter 7. Moreover, to generate realistic images, we need to perform hidden surface removal, which we shall learn in Chapter 8.

BIBLIOGRAPHIC NOTE

Please refer to the bibliographic note of Chapter 7 for further reading.

KEY TERMS

Axonometric projection – a type of parallel projection in which the principal object surfaces are not parallel to the view plane
Canonical view volume – a standardized view volume
Center of interest/Look-at point – the point in the world coordinate frame with respect to which we focus our camera while taking a photograph
Center of projection – the point where the projectors meet in perspective projection
Clipping window – a region (usually rectangular) of the view plane
Oblique projection – a type of parallel projection in which the projectors are not perpendicular to the view plane
Orthographic projection – a type of parallel projection in which projectors are perpendicular to the view plane.
Parallel projection – a type of projection in which the projectors are parallel to each other
Perspective foreshortening – an effect due to perspective projection in which the closer objects appear larger
Perspective projection – a type of projection in which the projectors meet at a point
Projection – the process of mapping an object from an n-dimensional space to an $n - 1$ dimensional space
Projectors – lines that originate from the object points to be projected and intersect the view plane
Vanishing points – an effect due to perspective projection in which lines that are not parallel appear to meet at a point on the view plane
View confusion – an effect due to perspective projection in which the objects appear upside down after projection
View coordinate – the coordinate reference frame used to represent a scene with respect to camera parameters
View plane – the plane on which a 3D object is projected
View volume – a 3D region in space (in the view coordinate system) that is projected on the view plane
View-up vector – a vector towards the direction of our head while taking a photograph with a camera
Viewing transformation – the process of mapping a world coordinate object description to the view the coordinate frame
Viewport – a rectangular region on the display screen where the content of the window is rendered
Window – a term used to denote the clipping window on the view plane

EXERCISES

6.1 Discuss the similarities between taking a photograph with a camera and transforming a world coordinate obejct to view plane. Is there any difference?
6.2 Mention the inputs we need to construct a view coordinate system.
6.3 Explain the process of setting-up of the view coordinate system.
6.4 How do we transform objects from world coordinate to view coordinate? Explain.
6.5 What are the broad categories of projections? Discuss their difference(s).

6.6 Why are perpective projections preferable over parallel projections to generate realistic effects? When do we need parallel projection?
6.7 Illustrate with diagrams the view volumes associated with each of the two broad projection types.
6.8 Why do we need canonical view volumes? Discuss how we can transform arbitrary volumes to canonical forms.
6.9 Derive the transformation matrices for parallel and perspective projections.
6.10 How is viewport different from window? Derive the window-to-viewport transformation matrix.
6.11 Does the projection transformation truly transform objects from 3D to 2D? Discuss.
6.12 Consider a spherical object centered at (1,1,1) with a radius of 1 unit. A camera is located at the point (1,1,4) and the look-at point is (1,1,2). The up direction is along the negative Y direction. Answer the following.

(a) Determine the coordinates of the point P_V in the view coordinate system to which the point P(1,1,2) is transformed.
(b) Assume a sufficiently large view volume that encloses the entire sphere after transformation. Its near plane is $z = -1$ and the clipping window is defined between [−10,−10] to [10,10]. What would the position P_W of the point P_V on the clipping window be after we perform (i) parallel and (ii) perspective projection? Ignore canonical transformations.
(c) Assume a view port defined between [2,3] (lower left corner) and [5,6] (top right corner) in the device coordinate system. Determine the position of P_W on this viewport after window-to-viewport transformation.

CHAPTER 7 Clipping

Learning Objectives

After going through this chapter, the students will be able to

- Understand the idea of clipping in two and three dimensions
- Learn about point clipping in two and three dimensions
- Learn the Cohen–Sutherland line clipping algorithm in two and three dimensions
- Understand the working of the parametric line clipping algorithm with the Liang–Barsky algorithm
- Know about the fill area clipping issues
- Learn about the Sutherland–Hodgeman fill area clipping algorithm for two and three dimensions
- Understand the steps of the Weiler–Atherton fill area clipping algorithm
- Learn the algorithm to convert a convex polygon into polygonal meshes, which is required for three-dimensional fill area clipping

INTRODUCTION

In Chapter 6, we discussed the concept of view volume. As you may recall, before projection on the view plane, we define a 3D region (in the view coordinate system) that we call as view volume. Objects within this region are projected on the view plane while objects that lie outside the volume boundary are discarded. An example is shown in Fig. 7.1. The process of discarding objects that lie outside the volume boundary is known as *clipping*. How does the computer *discard* (or *clip*) objects? We employ some programs or algorithms for this purpose, which are collectively known as the clipping algorithms. In this chapter, we shall learn about these algorithms.

Two things are to be noted here. Recall the important concept we learnt in Chapter 6, namely the *canonical view volume* (the cube). The algorithms we discuss in this chapter shall assume that the clipping is done against canonical view volumes only. Moreover, for the ease of understanding the algorithms, we shall first discuss the clipping algorithms in 2D. Clipping in 3D is performed by extending the 2D algorithms, which we shall discuss next.

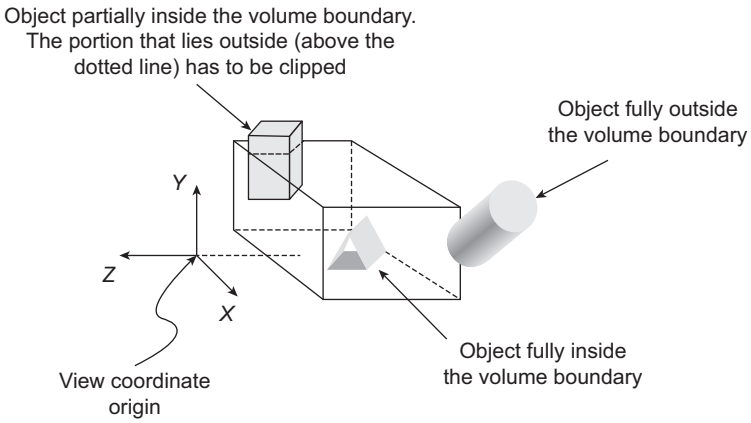


Fig. 7.1 Concept of clipping. The objects that lie outside the boundary, either fully or partially, are to be discarded or clipped. This is done with the clipping algorithms.

7.1 CLIPPING IN 2D

Unlike the view volume which is a 3D concept, we shall assume a *view window*, which is a square-shaped region on the view plane, to discuss 2D clipping algorithms. This is equivalent of assuming that the view volume and all the objects are already projected on the view plane. The view volume is projected to form the window. Other objects are projected to form points, lines, and fill-areas (e.g., an enclosed region such as a polygon). Thus, our objective is to clip points, lines, and fill-areas with respect to the window.

The simplest is point clipping. Given a point with coordinate (x,y), we simply check if the coordinate lies within the window boundary. In other words, if $wx_{\min} \leq x \leq wx_{\max}$ AND $wy_{\min} \leq y \leq wy_{\max}$, we keep the point; otherwise, we clip it out. $(wx_{\min}, wx_{\max})$ and $(wy_{\min}, wy_{\max})$ are the minimum and maximum x and y coordinate values of the window, respectively.

Line clipping is not so easy, however. We can represent any line segment with its end points. For clipping, we can check the position of these end points to decide whether to clip the line or not. If we follow this approach, either of the following three scenarios can occur, which is illustrated in Fig. 7.2.

1. Both the end points are within the window boundary. In such cases, we don't clip the line.
2. One end point is inside and the other point is outside. Such lines must be clipped.
3. Both the end points are outside. We cannot say for sure if the whole line is outside the window or part of it lies inside (see Fig. 7.2). Thus, we have to check for line–boundary intersections to decide if the line needs to be clipped or not.

As you can see in Fig. 7.2, when both the line end points are outside of the window, the line may be fully or partially outside. We cannot determine this from just the position of the end points. What we can do is to determine if there are intersection points of the line and the window boundaries (see Appendix for calculation of intersection points between two lines). Thus, given a line with both end points outside, we have to check for line–window intersection for all the four window boundary line segments. Clearly, the process

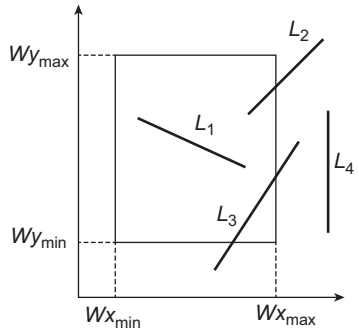


Fig. 7.2 Three scenarios for line clipping. For the line L_1, both the line endpoints are inside the window, so we don't clip the line. For L_2, one end point is inside and the other one is outside, so we clip it. In case of L_3 and L_4, both the end points are completely outside the window. However, L_3 is partially inside and needs further checking to determine the portion to be clipped.

is time-consuming. In a real-world application which may require thousands of line clipping in rendering a scene, it is not practical and we need more efficient clipping algorithms.

7.1.1 Cohen–Sutherland Line Clipping Algorithm

The Cohen–Sutherland line clipping algorithm is one efficient way of performing line clipping. In this algorithm, the world space (the window and its surrounding) is assumed to be divided into *nine* regions. The regions are formed by extending the window boundaries, as shown in Fig. 7.3. Each of these regions has a 4-bit unique code as identifier. Each bit in the code indicates the position (above, below, right, or left, in that order from left to right) of the region with respect to the window, as shown in Fig. 7.3. For example, a code of 1001 indicates that the corresponding region is situated above left of the window.

Above left	Above	Above right
Left	Window	Right
Below left	Below	Below right

1001	1000	1010
0001	0000	0010
0101	0100	0110

Above	Below	Right	Left

The 4-bit code and significance of each bit

Fig. 7.3 The nine regions of the Cohen–Sutherland algorithm. The left figure on top shows the regions while the right figure shows the region codes. The four bit code is explained in the bottom figure. Note that the window gets a code 0000.

Given the two end points of a line, the algorithm first assigns region codes to the end points. Let an end point be denoted by $P(x, y)$ and the window be specified by $(x_{min}, x_{max}, y_{min}, y_{max})$ (i.e., the x and y extents of its boundary). Then, we can determine region code of P through the following simple check.

$$\begin{aligned}\text{Bit 3} &= \text{sign}(y - y_{max})\\ \text{Bit 2} &= \text{sign}(y_{min} - y)\\ \text{Bit 1} &= \text{sign}(x - x_{max})\\ \text{Bit 0} &= \text{sign}(x_{min} - x)\end{aligned}$$

where sign$(a) = 1$ if a is positive, 0 otherwise.

Once the region codes are assigned to both the end points, the following checks are performed and the corresponding action taken.

Algorithm 7.1 **Cohen–Sutherland line clipping algorithm**

Input: A line segment with end points PQ and the window parameters $(x_{min}, x_{max}, y_{min}, y_{max})$
Output: Clipped line segment (NULL if the line is completely outside)
for each end point with coordinate (x,y), where sign$(a) = 1$ if a is positive, 0 otherwise **do**
  Bit 3 = sign $(y - y_{max})$
  Bit 2 = sign $(y_{min} - y)$
  Bit 1 = sign $(x - x_{max})$
  Bit 0 = sign $(x_{min} - x)$
end for
if both the end point region codes are 0000 **then**
  RETURN PQ.
else if logical AND (i.e., bitwise AND) of the end point region codes $\neq$ 0000 **then**
  RETURN NULL
else
  for each boundary b_i where b_i = above, below, right, left, **do**
    Check corresponding bit values of the two end point region codes
    if the bit values are same, **then**
      Check next boundary
    else
      Determine b_i-line intersection point using line equation
      Assign region code to the intersection point
      Discard the line from the end point outside b_i to the intersection point (as it is outside the window)
      if the region codes of both the intersection point and the remaining end point are 0000 **then**
        Reset PQ with the *new* end points
      end if
    end if
  end for
  RETURN modified PQ
end if

1. If both the end point region codes are 0000, the line is completely inside the window. Retain the line.
2. If logical AND (i.e., bitwise AND) of the end point region codes is not equal to 0000, the line is completely outside the window. Discard the entire line.

However, when none of these above cases occur, the line is partially inside the window and we need to clip it. For clipping, we need to calculate the line intersection point with window boundaries. This is done by taking one end point and following some order for checking, e.g., above, below, right, and left. For each boundary, we compare the corresponding bit values of the two end point region codes. If they are not the same, the line intersects that particular boundary. Using the line equation, we determine the intersection point and assign the region code to the intersection point as before. In the process, we *discard* the line segment outside the window. Next, we compare the two *new* end points to see if they are completely inside the window. If not, we take the other end point and repeat the process. The pseudo-code of the algorithm is shown in Algorithm 7.1.

Example 7.1

Consider the line segment *AB* in Fig. 7.4.

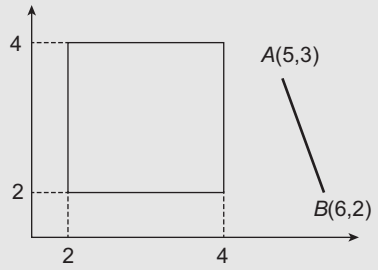


Fig. 7.4

From the figure, we see that $x_{\min} = 2, x_{\max} = 4, y_{\min} = 2$, and $y_{\max} = 4$. Also, $A(5,3)$ and $B(6,2)$. The first step is to determine the region codes of A and B (lines 3–8 of Algorithm 7.1). Let's consider A first. We can see that for A,

$$\begin{aligned}
\text{Bit 3} &= \text{sign}\,(3-4) = \text{sign}\,(-1) = 0\\
\text{Bit 2} &= \text{sign}\,(2-3) = \text{sign}\,(-1) = 0\\
\text{Bit 1} &= \text{sign}\,(5-4) = \text{sign}\,(1) = 1\\
\text{Bit 0} &= \text{sign}\,(2-5) = \text{sign}\,(-3) = 0
\end{aligned}$$

since sign(a) $= 0$ if a ≤ 0. Thus, the region code of A is 0010. Similarly the region code of B is derived as 0010. The next step (lines 9–27 of Algorithm 7.1) is the series of checks. The first check fails as both the end points are not 0000. However, the second check succeeds since the logical AND of AB is 0010 (i.e., $\neq$ 0000). Hence, we do not need to go any further. The line is totally outside the window boundary. We do not need to clip it and discard it as a whole.

Example 7.2

Consider the line segment PQ in Fig. 7.5.

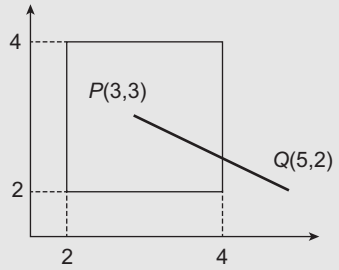


Fig. 7.5

We have $x_{\min} = 2$, $x_{\max} = 4$, $y_{\min} = 2$, and $y_{\max} = 4$. Also, $P(3,3)$ and $Q(5,2)$. We first determine the region codes of P and Q (lines 3–8 of Algorithm 7.1). Let us consider P first. We can see that for P,

$$\text{Bit } 3 = \text{sign } (3-4) = \text{sign } (-1) = 0$$

$$\text{Bit } 2 = \text{sign } (2-3) = \text{sign } (-1) = 0$$

$$\text{Bit } 1 = \text{sign } (3-4) = \text{sign } (-1) = 0$$

$$\text{Bit } 0 = \text{sign } (2-3) = \text{sign } (-1) = 0$$

Thus, the region code of P is 0000. Similarly the region code of Q is derived as 0010. The next step (lines 9–27 of Algorithm 7.1) is the series of checks. The first check fails as both the end points are not 0000. The second check also fails as the logical AND of PQ is 0000. Hence, we need to determine line–boundary intersection.

From the end points, we can derive the line equation as: $y = -\frac{1}{2}x + \frac{9}{2}$ (see Appendix for the derivation of line equation from end points). Now, we have to check for the intersection of this line with the boundaries following the order: above, below, right, left. The aforementioned bit values (bit 3) of P and Q are the same. Hence, the line does not cross above boundary (lines 16–19 of Algorithm 7.1). Similarly, we see that it does not cross the below boundary. However, for the right boundary, the two corresponding bits (bit 1) are different. Hence, the line crosses the right boundary.

The equation of the right boundary is $x = 4$. Putting this value in the line equation, we get the intersection point as $Q'(4,\frac{5}{2})$. We discard the line segment $Q'Q$ since Q is outside the right boundary (line 21 of Algorithm 7.1). Thus, the new line segment becomes PQ'. We determine the region code of Q' as 0000. Since both P and Q' have region code 0000, the algorithm resets PQ by changing Q to Q' (lines 22–23 of Algorithm 7.1). Finally, we check the left boundary. Since there is no intersection (bit 0 is same for both end points), the algorithm returns PQ' and stops.

Example 7.3

Consider the line segment MN in Fig. 7.6.

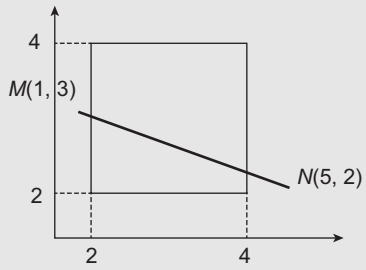


Fig. 7.6

Here we have $x_{\min} = 2, x_{\max} = 4, y_{\min} = 2$, and $y_{\max} = 4$ and the two end points $M(1,3)$ and $N(5,2)$. We determine the region code for M first as,

$$\begin{aligned}
\text{Bit 3} &= \text{sign}\,(3-4) = \text{sign}\,(-1) = 0\\
\text{Bit 2} &= \text{sign}\,(2-3) = \text{sign}\,(-1) = 0\\
\text{Bit 1} &= \text{sign}\,(1-4) = \text{sign}\,(-3) = 0\\
\text{Bit 0} &= \text{sign}\,(2-1) = \text{sign}\,(1) = 1
\end{aligned}$$

Thus, the region code of M is 0001. Similarly the region code of N is derived as 0010. The next step (lines 9–27 of Algorithm 7.1) is the series of checks. The first check fails as both the end points are not 0000. The second check also fails as the logical AND of MN is 0000. Hence, we need to determine line–boundary intersection points.

From the end points, we can derive the line equation as: $y = -\frac{1}{4}x + \frac{13}{4}$ (see Appendix for the derivation). Next, we check for the intersection of this line with the boundaries following the order: above, below, right, left. These bit values (bit 3) of M and N are the same. Hence, the line does not cross above boundary (lines 16–19 of Algorithm 7.1). Similarly, we see that it does not cross the below boundary (bit 2 is same for both). However, for the right boundary, the two corresponding bits (bit 1) are different. Hence, the line crosses the right boundary.

The equation of the right boundary is $x = 4$. Putting this value in the line equation, we get the intersection point as $N'(4,\frac{9}{4})$. We discard the line segment $N'N$ since N is outside the right boundary (line 21 of Algorithm 7.1). Thus, the new line segment becomes MN'. We determine the region code of N' as 0000.

We now have two new end points M and N' with the region codes 0001 and 0000, respectively. The boundary check is now performed for the left boundary. Since the bit values are not the same, we check for intersection of the line segment MN' with the left boundary. The equation of the left boundary is $x = 2$. Putting this value in the line equation, we get the intersection point as $M'(2,\frac{11}{4})$. We discard the line segment MM' since M is outside the left boundary (line 21 of Algorithm 7.1). Thus, the new line segment becomes $M'N'$. We determine the region code of M' as 0000.

Since both M' and N' have the region code 0000, the algorithm resets the line segment to $M'N'$ (lines 22–23 of Algorithm 7.1). As no more boundary remains to be checked, the algorithm returns $M'N'$ and stops.

7.1.2 Liang–Barsky Line Clipping Algorithm

The Cohen–Sutherland algorithm works well when the number of lines, which can be clipped without further processing, is large compared to the size of the input set of lines. However, it still has to perform some boundary–line intersection calculations. There are other faster line-clipping methods developed to reduce the intersection calculation further, based on more efficient tests. The algorithm proposed by Cyrus and Beck (see the bibliographic note for reference) was among the earliest attempts in this direction, which is based on parametric line equation. Later, a more efficient version was proposed by Liang and Barsky. The basic idea of the Liang–Barsky algorithm is as follows.

Given a line segment with endpoints $P(x_1, y_1)$ and $Q(x_2, y_2)$, we can represent the line in parametric form as,

$$x = x_1 + u\Delta x \quad \text{where } \Delta x = x_2 - x_1$$
$$y = y_1 + u\Delta y \quad \text{where } \Delta y = y_2 - y_1$$

where $0 \leq u \leq 1$ is the parameter. Given the window parameters $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$, the following relationships should hold for the line to be retained.

$$x_{\min} \leq x_1 + u\Delta x \leq x_{\max}$$
$$y_{\min} \leq y_1 + u\Delta y \leq y_{\max}$$

We can rewrite the Example 7.4 relations in a compact form as $p_k \leq q_k$ where $k = 1, 2, 3, 4$. Thus,

$$p_1 = -\Delta x, q_1 = x_1 - x_{\min}$$
$$p_2 = \Delta x, q_2 = x_{\max} - x_1$$
$$p_3 = -\Delta y, q_3 = y_1 - y_{\min}$$
$$p_4 = \Delta y, q_4 = y_{\max} - y_1$$

where $k = 1, 2, 3, 4$ corresponds to the left, right, below, and above window boundaries, in that order. If for any k for a given line, $p_k = 0$ AND $q_k < 0$, discard the line as it is completely outside the window. Otherwise, we calculate two parameters u_1 and u_2, that define the line segment within the window. In order to calculate u_1, we first calculate the ratio $r_k = \frac{q_k}{p_k}$ for all those edges for which $p_k < 0$. Then, we set $u_1 = \max\{0, r_k\}$. Similarly for u_2, we calculate the ratio $r_k = \frac{q_k}{p_k}$ for all those edges for which $p_k > 0$ and then set $u_2 = min\{1, r_k\}$. If $u_1 > u_2$, the line is completely outside, so we discard it. Otherwise, the end points of the clipped lines are calculated as follows.

1. If $u_1 = 0$, there is one intersection point which is calculated as $x_2 = x_1 + u_2\Delta x, y_2 = y_1 + u_2\Delta y$ (note that the other end point remains the same).
2. Otherwise, there are two intersection points (i.e., both the end points need to be changed). The two new end points are calculated as $x_1' = x_1 + u_1\Delta x, y_1' = y_1 + u_1\Delta y$ and $x_2 = x_1 + u_2\Delta x, y_2 = y_1 + u_2\Delta y$.

The pseudocode of the algorithm is shown in Algoritm 7.2.

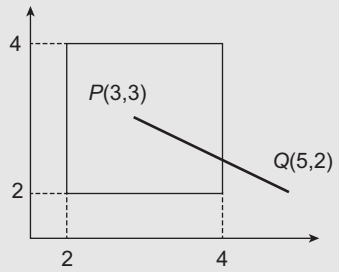
Algorithm 7.2 **Liang–Barsky line clipping algorithm**

1: **Input:** A line segment with end points $P(x_1, y_1)$ and $Q(x_2, y_2)$, the window parameters $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$. A window boundary is denoted by k where k can take the values 1, 2, 3, or 4 corresponding to the left, right, below, and above boundary, respectively.
2: **Output:** Clipped line segment
3: Calculate $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$
4: Calculate $p_1 = -\Delta x$, $q_1 = x_1 - x_{\min}$
5: Calculate $p_2 = \Delta x$, $q_2 = x_{\max} - x_1$
6: Calculate $p_3 = -\Delta y$, $q_3 = y_1 - y_{\min}$
7: Calculate $p_4 = \Delta y$ $q_4 = y_{\max} - y_1$
8: **if** $p_k = 0$ and $q_k < 0$ for any $k = 1, 2, 3, 4$ **then**
9: Discard the line as it is completely outside the window
10: **else**
11: Compute $r_k = \frac{q_k}{p_k}$ for all those boundaries k for which $p_k < 0$. Determine parameter $u_1 = \max\{0, r_k\}$.
12: Compute $r_k = \frac{q_k}{p_k}$ for all those boundaries k for which $p_k > 0$. Determine parameter $u_2 = min\{1, r_k\}$.
13: **if** $u_1 > u_2$ **then**
14: Eliminate the line as it is completely outside the window
15: **else if** $u_1 = 0$ **then**
16: There is one intersection point, calculated as $x_2 = x_1 + u_2\Delta x, y_2 = y_1 + u_2\Delta y$
17: **Return** the two end points (x_1, y_1) and (x_2, y_2)
18: **else**
19: There are two intersection points, calculated as: $x_1' = x_1 + u_1\Delta x$, $y_1' = y_1 + u_1\Delta y$ and $x_2 = x_1 + u_2\Delta x, y_2 = y_1 + u_2\Delta y$
20: **Return** the two end points (x_1', y_1') and (x_2, y_2)
21: **end if**
22: **end if**

Example 7.4

We will show here the running of the algorithm for one of the three examples (Example 7.2) used to illustrate the Cohen–Sutherland algorithm. The working of the Liang–Barsky algorithm for the other two examples can be understood in a likewise manner and left as an exercise for the reader.

Now let us reconsider the line segment PQ in Example 7.2. The figure is reproduced here for convenience.



From the figure, we see that $x_{\min} = 2, x_{\max} = 4, y_{\min} = 2$, and $y_{\max} = 4$. Also, $P(3,3)$ and $Q(5,2)$. We first calculate $\Delta x = 2, \Delta y = -1, p_1 = -2, q_1 = 1, p_2 = 2, q_2 = 1, p_3 = 1, q_3 = 1, p_4 = -1$, and $q_4 = 1$ (lines 3–7 of Algorithm 7.2). Since the condition $p_k = 0$ and $q_k < 0$ for any k is not true, the first condition (lines 8–9 of Algorithm 7.2) fails. Hence, we calculate u_1 and u_2.

Note that $p_1, p_4 < 0$. Hence we calculate $r_1 = -\frac{1}{2}$ and $r_4 = -1$. Thus, $u_1 = \max\{0, -\frac{1}{2}, -1\} = 0$ (line 11 of Algorithm 7.2). Also, $p_2, p_3 > 0$. Hence we calculate $r_2 = \frac{1}{2}$ and $r_3 = 1$. Thus, $u_2 = \min\{1, \frac{1}{2}, 1\} = \frac{1}{2}$ (line 12 of Algorithm 7.2). Since $u_1 < u_2$, the line is not eliminated (lines 13–14 of Algorithm 7.2). However, $u_1 = 0$. Thus, the next condition satisfies (line 15 of Algorithm 7.2). So, we calculate the single intersection point: $x_2 = 4, y_2 = \frac{5}{2}$. Thus, the end points of the clipped line segment returned are (3,3) and $(4,\frac{5}{2})$ (lines 16–17 of Algorithm 7.2).

7.1.3 Fill-area Clipping: Sutherland–Hodgeman Algorithm

As we saw, the previous algorithms are used for clipping lines. In many situations, we have to clip polygons with respect to the window. Although we can use the line clippers to the individual edges of the polygon, the approach is not necessarily efficient and better; there are more efficient algorithms. In the following discussion, we look-at one such algorithm, namely the Sutherland–Hodgeman polygon clipping algorithm.

The basic idea of the algorithm is as follows: we start with four *clippers* or the lines that define the window boundaries. Each clipper takes as input a list of ordered pair of vertices (i.e., edges) and produces another list as output. We impose an order to the clipper for checking. Let it be left clipper, followed by right clipper, followed by bottom clipper, followed by top clipper. The original polygon vertices are given as input to the first (i.e., left) clipper. To create the vertex list, a naming convention of the vertices is followed (either clockwise or anti-clockwise). Let us assume anti-clockwise naming of the vertices. For each clipper, the output vertex list is generated in the following way.

Let the input vertex list to a clipper be denoted by the set $V = \{v_1, v_2, \cdots, v_n\}$ where v_i denotes the the ith vertex. Then, for each edge in the list (i.e., successive vertex pair) (v_i, v_j), we do the following:

1. If v_i is *inside* and v_j *outside* of the clipper, **return** the intersection point of the clipper with the edge (v_i, v_j).
2. If both the vertices are *inside* the clipper, **return** v_j.
3. If v_i is *outside* and v_j *inside* of the clipper, **return** the intersection point of the clipper with the edge (v_i, v_j) and v_j.
4. If both the vertices are *outside* the clipper, **return** NULL.

The terms *inside* and *outside* are to be interpreted differently for different clipper. For the left clipper, if a vertex is on its right side, then the vertex is *inside*; otherwise, it is *outside*. For the right clipper, a vertex is *inside* if it is on the left side; otherwise it is *outside*. For the top clipper, a vertex below means the vertex is *inside*, else the vertex is outside. Similarly, for the bottom clipper, an *inside* vertex implies it is above the clipper, otherwise it is *outside*. In all the cases, we assume that a vertex is *inside* if it is *on* the clipper. The pseudocode of the Sutherland–Hodgeman algorithm is shown in Algorithm 7.3.

Algorithm 7.3 **Sutherland–Hodgeman fill-area clipping algorithm**

1: **Input:** Four clippers: $c_l = x_{\min}, c_r = x_{\max}, c_t = y_{\max}, c_b = y_{\min}$ corresponding to the left, right, top, and bottom window boundaries, respectively. The polygon is specified in terms of its vertex list $V_{in} = \{v_1, v_2, \cdots, v_n\}$, where the vertices are named anti-clockwise.
2: **for** each clipper in the order c_l, c_r, c_t, c_b **do**
3: Set output vertex list $V_{out} = NULL, i = 1, j = 2$
4: **repeat**
5: Consider the vertex pair v_i and v_j in V_{in}
6: **if** v_i is *inside* and v_j *outside* of the clipper **then**
7: **ADD** the intersection point of the clipper with the edge (v_i, v_j) to V_{out}
8: **else if** both the vertices are *inside* the clipper **then**
9: **ADD** v_j to V_{out}
10: **else if** v_i is *outside* and v_j *inside* of the clipper **then**
11: **ADD** the intersection point of the clipper with the edge (v_i, v_j) and v_j to V_{out}
12: **else**
13: **ADD** NULL to V_{out}
14: **end if**
15: **until** all edges (i.e., consecutive vertex pairs) in V_{in} are checked
16: Set $V_{in} = V_{out}$
17: **end for**
18: **Return** V_{out}

Example 7.5

Consider the polygon with vertices {1,2,3} (named anti-clockwise) shown in Fig. 7.7. We wish to determine the clipped polygon (i.e., the polygon with vertices {2′,3′,3″,1′,2}) following the Sutherland–Hodgeman algorithm.

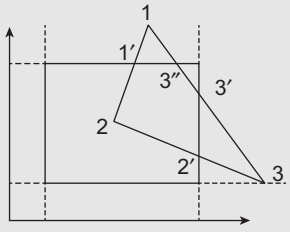


Fig. 7.7

We check the vertex list against each clipper in the order left, right, top, bottom (the outer **for** loop, line 2 of Algorithm 7.3). For the left clipper, the input vertex list $V_{in} = \{1, 2, 3\}$. The pair of vertices to be checked for the left clipper are {1,2}, {2,3}, and {3,1} (the inner loop, line 4 of Algorithm 7.3). For each of these pairs, we perform the checks (lines 6–13 of Algorithm 7.3) to determine V_{out} for the left clipper. We start with {1,2}. Since both the vertices are on the right side of the left clipper (i.e., both are *inside*), we set $V_{out} = \{2\}$. Similarly, after checking {2,3}, we set $V_{out} = \{2, 3\}$ and after checking {3,1}, the final output list becomes $V_{out} = \{2, 3, 1\}$.

In the next iteration of the outer loop (check against right clipper), we set $V_{in} = V_{out} = \{1, 2, 3\}$ and $V_{out} = NULL$. Thus the three pair of vertices to be checked are {1,2}, {2,3}, and {3,1}. In {1,2}, both the vertices are *inside* (i.e., they are on the left side of the right clipper); hence $V_{out} = \{2\}$. For the next pair {2,3}, we notice that vertex 2 is *inside* while vertex 3 is *outside*. Thus, we compute the intersection point $2'$ of the right clipper with the edge {2,3} and set $V_{out} = \{2, 2'\}$. For the remaining pair {3,1}, vertex 3 is *outside* (on the right side) and vertex 1 *inside* (on the left side). Thus, we calculate the intersection point $3'$ of the edge with the clipper and set $V_{out} = \{2, 2', 3', 1\}$. The inner loop stops as all the edges are checked.

Next, we consider the top clipper. We set $V_{in} = V_{out} = \{2, 2', 3', 1\}$ and $V_{out} = NULL$. The pair of vertices to be checked are {2,2′}, {2′,3′}, {3′,1}, and {1,2}. Since both the vertices of {2,2′} are *inside* (i.e., below the clipper), $V_{out} = \{2'\}$. Similarly, after checking {2′,3′}, we set $V_{out} = \{2', 3'\}$ as both are *inside*. In the pair {3′,1}, the vertex $3'$ is *inside* whereas the vertex 1 is *outside* (i.e., above the clipper). Hence, we calculate the intersection point $3''$ between the clipper and the edge and set $V_{out} = \{2', 3', 3''\}$. In the final edge {1,2}, the first vertex is *outside* while the second vertex is *inside*. Thus, we calculate the intersection point $1'$ between the edge and the clipper and set $V_{out} = \{2', 3', 3'', 1', 2\}$. After this, the inner loop stops.

Finally, we check against the bottom clipper. Before the checking starts, we set $V_{in} = V_{out} = \{2', 3', 3'', 1', 2\}$ and $V_{out} = NULL$. As all the vertices are *inside* (i.e., above the clipper), after the inner loop completes, the output list becomes $V_{out} = \{2', 3', 3'', 1', 2\}$ (i.e., same as the input list, check for yourself).

Thus, after the input polygon is checked against all the four clippers, the algorithm returns the vertex list {2′,3′,3″,1′,2} as the clipped polygon.

7.1.4 Fill-area Clipping: Weiler–Atherton Algorithm

The Sutherland–Hodgeman algorithm works well when the fill-area is a convex polygon to be clipped against a rectangular clipping window. The Weiler–Atherton algorithm provides a more general fill-area clipping procedure. It can be used for any type of polygon fill-area (concave or convex) against any polygonal clipping window.

In the Sutherland–Hodgeman algorithm, we processed the edges of the fill-area following only a particular order and performed clipping. However, the processing is done differently in the Weiler–Atherton algorithm. Here, we start with processing the fill-area edges in a particular order (typically anti-clockwise). We continue along the edges till we encounter an edge that crosses to the *outside* of the clip window boundary. At the intersection point, we make a detour: we now follow the edges of the clip window (along the same direction maintaining the traversal order). We continue our traversal along the edges of the clip boundary till we encounter another fill-area edge that crosses to the *inside* of the clip window. At this point, we resume our polygon edge traversal again along the same direction. The process continues till we encounter a previously processed intersection point. So, the two rules of traversal followed in the Weiler–Atherton algorithm are the following:

1. From an intersection point due to an outside-to-inside fill-area edge (with respect to a clip boundary), follow the fill-area polygon edges.

2. From an intersection point due to an inside-to-outside fill-area edge (with respect to a clip boundary), follow the window boundaries.

In both these cases, the traversal direction remains the same. At the end of the processing, when we encounter a previously processed intersection point, we output the vertex list representing a clipped area. However, if the whole fill-area polygon is not fully covered at this point, we resume our traversal along the polygon edges in the same direction from the last intersection point of an inside-outside polygon edge. The pseudocode of the Weiler–Atherton algorithm is shown in Algorithm 7.4.

Algorithm 7.4 **Weiler–Atherton fill-area clipping algorithm**

1: Start from a vertex inside the window.
2: Process the edges of the polygon fill-area in any particular order (clockwise or anti-clockwise). Continue the processing till an edge of the fill-area is found that crosses a window boundary from inside to outside. The intersection point of the edge with the window boundary is the *exit-intersection* point. Record the intersection point.
3: From the exit-intersection point, process the window boundaries in the same direction (clockwise or anti-clockwise). Continue processing till another intersection point (of a fill-area edge with a window boundary) is found.
4: **if** the intersection point is a *new* point not yet processed **then**
5: Record the intersection point
6: Continue processing the fill-area edges till a previously processed vertex is encountered
7: **end if**
8: Form the output vertex list V_{out} for this section of the clipped fill-area
9: **if** all the polygon fill-area edges have been processed **then**
10: Output V_{out}
11: **else**
12: Return to the exit-intersection point
13: Continue processing the fill-are a edges in the same order (clockwise or anti-clockwise) till another intersection point (of a fill-area edge with a window boundary) is found
14: Go to the line 4
15: **end if**

Example 7.6

Let us consider the fill-area shown in Fig. 7.8. The vertices of the polygon are named anti-clockwise. Note that this is a concave polygon, which is to be clipped against the rectangular window. Let us try to perform clipping following the steps of the Algorithm 7.4. In the figure, the traversal of the fill-area edges and window boundaries is shown with arrows.

We first start with the edge {1,2}. Both the vertices are inside the window. Since there is no intersection, we add these vertices to the output vertex list V_{out} and continue processing the fill-area edges anti-clockwise.

Next we process the edge {2,3}. As we can observe, the edge goes from inside of the clip window to outside. We record the intersection point $2'$ and add it to V_{out}. $2'$ is the

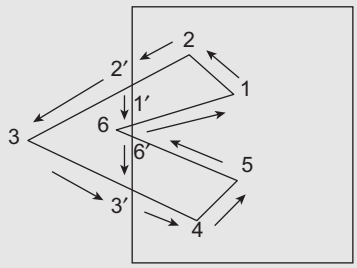


Fig. 7.8

exit-intersection point. At this point, we make a detour and proceed along the window boundary in the anti-clockwise direction.

While traversing along the boundary, we encounter the intersection point $1'$ of the fill-area edge $\{6,1\}$ with the boundary. This is a new intersection point not yet processed. We add this to V_{out}. Then, we start processing the fill-area edges again.

The fill-area edge processing takes us to the vertex 1. We have already processed this vertex. Thus, we have completed determining one clipped segment of the fill area, represented by the vertex list $V_{out} = \{1, 2, 2', 1', 1\}$. We output this list. Since some edges of the fill-area are not yet processed, we return to the exit-intersection point $2'$ and continue processing the fill-area edges.

We first check the edge segment $\{2',3\}$. Since the vertex 3 is on the left side of the left window boundary (i.e., outside the clip window), we continue processing the fill-area anti-clockwise.

The next edge processed is $\{3,4\}$. Note that the edge intersects the window boundary at $3'$. Since the vertex 4 is inside the window and $3'$ is a new intersection point, we add the intersection point $3'$ and the vertex 4 to the output vertex list V_{out}. We continue processing the fill area edges.

Both the vertices of the next edge $\{4,5\}$ are inside the clipping window. So, we simply add them to V_{out} and continue processing the fill-area edges.

The next edge $\{5,6\}$ intersects the window boundary at the intersection point $6'$. This is an exit-intersection point. We record the intersection point and add it to the output vertex list V_{out}.

From $6'$, we start processing the window boundaries again (anti-clockwise). During this processing, we encounter the intersection point $3'$. This is already processed before. So, we output $V_{out} = \{3', 4, 5, 6', 3'\}$ that represents the other clipped region of the fill area. Since now all the edges of the fill-area are processed, the algorithm stops at this stage.

7.2 3D CLIPPING

So far, we have discussed clipping algorithms for 2D. The same algorithms, with a few modifications, are used to perform clipping in 3D. A point to be noted here is that clipping is performed against the normalized view volume (usually the symmetric cube with each coordinates in the range $[-1,1]$ in the three directions). In the following, we will discuss only the extension of the 2D algorithms without further elaborations, as the core algorithms remain the same.

Point clipping in 3D is done in a way similar to 2D. Given a point with coordinate (x,y,z), we simply check if the coordinate lies within the view volume. In other words, if $-1 \leq x \leq 1$ AND $-1 \leq y \leq 1$ AND $-1 \leq z \leq 1$, we keep the point; otherwise, we clip it out.

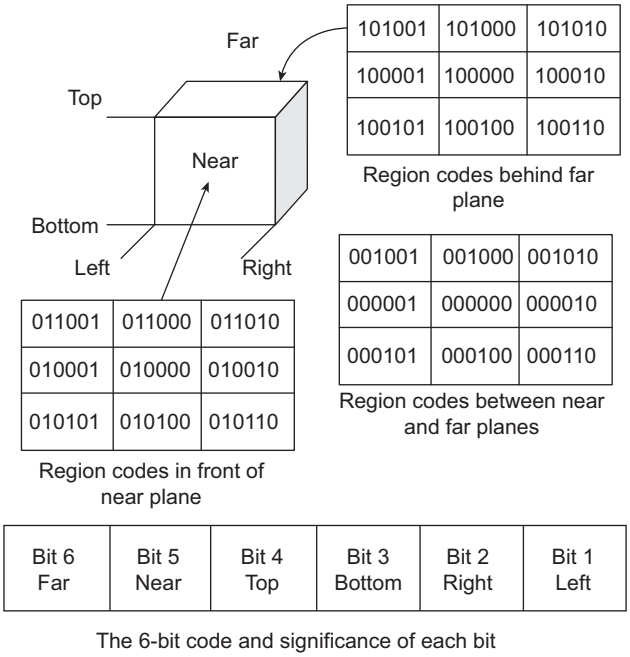


Fig. 7.9 The 27 regions of the Cohen–Sutherland algorithm for 3D. The 6-bit code is explained in the bottom figure. Note that only the view volume interior gets a code 0000.

7.2.1 3D Line Clipping

The Cohen–Sutherland can be extended for 3D line clipping. The core idea of the 3D Cohen–Sutherland algorithm remains the same: we divide the view coordinate space into regions. However, there are the following major differences between the two versions of the algorithm.

1. Unlike in 2D, we have 27 regions to consider.
2. Regions are encoded with 6 bits corresponding to the 6 planes (far, near, top, bottom, right, left) of the cube, unlike the 4 bits in 2D. The 6 bits are (from left to right):

The 27 regions and the 6-bit encoding scheme are depicted in Fig. 7.9. It is left as an exercise for the reader to modify Algorithm 7.1 taking into account these considerations.

7.2.2 3D Fill-area Clipping

In 3D fill-area (polyhedron) clipping, we first check if the *bounding volume* of the polyhedron is outside the view volume (by comparing their maximum and minimum coordinates in each of the x, y, z directions). Otherwise, we can apply the 3D extension of the Sutherland–Hodgeman algorithm to perform the clipping.

The core idea of the 3D Sutherland–Hodgeman algorithm remains the same. The main differences are outlined below.

1. A polyhedron is made up of polygonal surfaces. We take one surface at a time to perform clipping. Usually, the polygon is further divided into triangular meshes. Then, each

triangle in the mesh is processed at a time. Thus, there are two more outer loops in Algorithm 7.3. The outermost loop is for checking one surface at a time. Inside this loop, the next level loop processes each triangle in the mesh (of that surface). Then the two loops of Algorithm 7.3 are executed in sequence.

2. Instead of four clippers, we now have six clippers corresponding to the six bounding surfaces of the normalized view volume. Hence, the **for** loop in Algorithm 7.3 (lines 2–17) is executed six times.

Algorithm 7.5 shows a quick and easy way of creating a triangle mesh from a convex polygon.

Algorithm 7.5 **Algorithm to create triangle mesh from a convex polygon**

1: **Input:** Set of vertices $V = \{v_1, v_2, \cdots, v_n\}$, the triangle set $V_T = NULL$
2: **Output:** Set of triangles V_T
3: **repeat**
4: Take first three vertices from V to form the vertex set v_t representing a triangle
5: Add v_t to V_T
6: Reset V by removing from it the middle vertex of v_t
7: **until** V contains only three vertices
8: Add V to V_T and Return V_T

Let us consider an example to understand the idea of Algorithm 7.5. Suppose we want to create a triangle mesh from the polygon shown in Fig. 7.10.

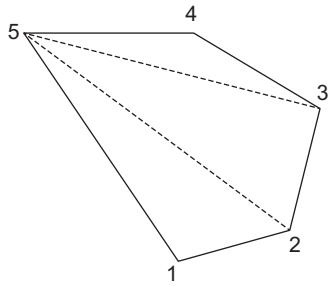


Fig. 7.10 Polygon for creating triangular mesh

The input vertex list is $V = \{1, 2, 3, 4, 5\}$ (we followed an anti-clockwise vertex naming convension). In the first iteration of the loop, we create $v_t = \{1, 2, 3\}$ and reset $V = \{1, 3, 4, 5\}$ after removing vertex 3 (the middle vertex of v_t). Then we set $V_T = \{\{1, 2, 3\}\}$. In the next iteration, $V = \{1, 3, 4, 5\}$. We create $v_t = \{1, 3, 4\}$ and reset $V = \{1, 4, 5\}$. Also, we set $V_T = \{\{1, 2, 3\}, \{1, 3, 4\}\}$. Since V now contains three vertices, the iteration stops. We set $V_T = \{\{1, 2, 3\}, \{1, 3, 4\}, \{1, 4, 5\}\}$ and return V_T as the set of three triangles.

Note that Algorithm 7.5 works when the input polygon is convex. In case of concave polygons, we first split it into a set of convex polygons and then apply Algorithm 7.5 on each member of the set. There are many efficient methods for splitting a concave polygon into a set of convex polygons such as the vector method, the rotation method and so on. However,

we shall not go into the details of these methods any further, as they are not necessary to understand the basic idea of 3D clipping.

SUMMARY

In this chapter, we learnt the basic idea behind the clipping stage of the graphics pipeline. For ease of understanding, we started with the 2D clipping process. We covered three basic clipping algorithms for line- and fill-area (polygon) clipping, namely the Cohen–Sutherland line clipping algorithm, the Liang–Barsky line clipping algorithm, and the Sutherland–Hodgeman polygon clipping algorithm.

The core idea of the Cohen–Sutherland algorithm is the division of world space into regions, with each region having its own and unique region code. Based on a comparison of region codes of the end points of a line, we decide if the line needs to be clipped or not. On the other hand, the Liang–Barsky algorithm makes use of a parametric line equation to perform clipping. Clipping is done based on the line parameters determined from the end points of the line. The algorithm reduces line-window boundary intersection calculation to a great extent. In the Sutherland–Hodgeman algorithm, polygons are clipped against window boundaries, on the basis of the *inside-outside* test.

The same 2D algorithms are applicable in 3D with some minor modifications. The first thing to note is that the clipping algorithms are designed keeping the normalized view volume in mind. There are 27 regions and a 6-bit region code to be considered for using the Cohen–Sutherland algorithm in 3D. In order to use the Sutherland–Hodgeman algorithm, we need to consider six clippers as against four in 2D.

In clipping, we discard the portion of the object that lies outside the window/view volume. However, depending on the position of the viewer, some portion of an inside object also sometimes needs to be discarded. When we want to discard parts of objects that are inside window/view volume, we apply another set of algorithms, which are known as *hidden surface removal* (or *visible surface detection*) methods. Those algorithms are discussed in Chapter 8.

BIBLIOGRAPHIC NOTE

Two-dimensional line clipping algorithms are discussed in Sproull and Sutherland [1968], Cyrus and Beck [11978], Liang and Barsky [1984], and Nicholl et al. [1987]. In Sutherland and Hodgman [1974] and Liang and Barsky [1983], basic polygon-clipping methods are presented. Weiler and Atherton [1977] and Weiler [1980] contain discussions on clipping arbitrarily shaped polygons with respect to arbitrarily shaped polygonal clipping windows. Weiler and Atherton [1977], Weiler [1980], Cyrus and Beck [1978], and Liang and Barsky [1984] also describe 3D viewing and clipping algorithms. Blinn and Newell [1978] presents homogeneous-coordinate clipping. The *Graphics Gems* book series (Glassner [1990], Arvo [1991], Kirk [1992], Heckbert [1994], and Paeth [1995]) contain various programming techniques for 3D viewing.

KEY TERMS

Clipping – the process of eliminating objects fully or partially that lie outside a predefined region
Clipping window – the predefined region with respect to which clipping is performed
Cohen–Sutherland line clipping – an algorithm used to perform line clipping

Fill area clipping – clipping procedure for fill area
Fill-area – an enclosed region in space, usually of polygonal shape
Liang–Barsky line clipping – a parametric algorithm used to perform line clipping
Region codes – a coding scheme used to identify spatial regions in the Cohen–Sutherland algorithm
Sutherland–Hodgeman algorithm – a fill area clipping procedure
Triangular mesh – representation of a polygon in terms of a set of interconnected triangles
Weiler–Atherton algorithm – a fill area clipping procedure

EXERCISES

7.1 Briefly explain the basic idea of clipping in the context of 3D graphics pipeline. In which coordinate system does this stage work?
7.2 Write an algorithm to clip lines against window boundaries using *brute force* method (i.e., intuitively what you do). What are the maximum and minimum number of operations (both integer and floating point) required? Give one suitable example for each case.
7.3 Consider the clipping window with vertices *A*(2,1), *B*(4,1), *C*(4,3), and *D*(2,3). Use the Cohen–Sutherland algorithm to clip the line *A*(−4,−5) *B*(5,4) against this window (show all intermediate steps).
7.4 Determine the maximum and minimum number of operations (both integer and floating point) required to clip lines using the Cohen–Sutherland clipping algorithms. Give one suitable example for each case.
7.5 Consider the clipping window and the line segment in Exercise 7.2. Use the Liang–Barsky algorithm to clip the line (show all intermediate steps).
7.6 Answer Exercise 7.4 using the Liang–Barsky algorithm.
7.7 In light of your answer to Exercised 7.2, 7.4, and 7.6, which is the best method (among brute force, Cohen–Sutherland, and Liang–Barsky)?
7.8 Discuss the role of the *clippers* in the Sutherland–Hodgman algorithm.
7.9 Write a procedure (in pseudocode) to perform *inside-outside* test with respect to a clipper. Modify Algorithm 7.3 by invoking the procedure as a sub-routine.
7.10 Consider a clipping window with corner points (1,1), (5,1), (5,5), and (1,5). A square with vertices (3,3), (7,3), (7,7), and (3,7) needs to be clipped against the window. Apply Algorithm 7.3 to perform the clipping (show all intermediate stages).
7.11 Modify Algorithm 7.1 for 3D clipping of a line with respect to the symmetric normalized view volume.
7.12 Modify Algorithm 7.3 for 3D clipping of a polyhedron (with convex polygonal surfaces) with respect to the symmetric normalized view volume.

CHAPTER 8

Hidden Surface Removal

Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the concept of hidden surface removal in computer graphics
- Understand the two broad categories of hidden surface removal techniques—object space method and image space method
- Get an idea about the object space method known as back face elimination
- Learn about two well-known image space methods—Z-buffer algorithm and A-buffer algorithm
- Learn about the Painter's algorithm, a popular hidden surface removal algorithm containing elements of both the object space and image space techniques
- Get an overview of the Warnock's algorithm, which belongs to a group of techniques known as the area subdivision methods
- Learn about the octree method for hidden surface removal, which is another object space method based on the octree representation

INTRODUCTION

In Chapter 7, we learnt to remove objects that are fully or partially outside the view volume. To recap, this is done using the clipping algorithms. However, sometimes, we need to remove, either fully or partially, objects that are *inside* the view volume. An example is shown in Fig. 8.1. In the figure, object B is partially blocked from the viewer by object A. For realistic image generation, the blocked portion of B should be eliminated before the scene is rendered. As we know, clipping algorithms cannot be used for this purpose. Instead, we make use of another set of algorithms to do it. These algorithms are collectively known as the *hidden surface removal* methods (often also called the *visible surface detection* methods). In this chapter, we shall learn about these methods.

In all the methods we discuss, we shall assume a right-handed coordinate system with the viewer looking at the scene along the negative Z direction. One important thing should be noted: when we talk about *hidden surface*, we assume a specific viewing direction. This is so since a surface *hidden* from a particular viewing position may not be so from another position. Moreover, for simplicity, we shall assume only objects with polygonal surfaces.

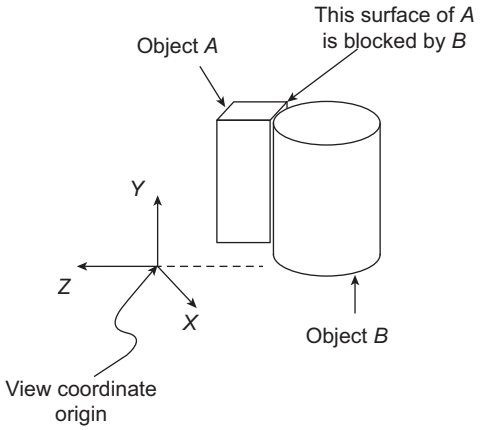


Fig. 8.1 The figure illustrates the idea of hidden surface. One surface of object *A* is blocked by object *B* and the *back* surfaces of *A* are hidden from view. So, during rendering, these surfaces should be removed for realistic image generation.

This is not an unrealistic assumption after all, since curved surfaces are converted to polygonal meshes anyway.

8.1 TYPES OF METHODS

The hidden surface removal methods are broadly of the following two types—object space and image space methods.

Object Space Methods

In these algorithms, objects and parts of objects are compared to each other to determine which surfaces are visible. The general approach can be summarized as follows:

For each object in the scene, **do**,

1. *Determine* those parts of the object whose view is unobstructed by other parts of it or any other object with respect to the viewing specification.
2. *Render* those parts with the object color.

As you can see, such methods work *before* projection and have both advantages and disadvantages. On the plus side, they are device-independent and work for any resolution. However, step 1 is computation-intensive. Depending on the complexity of the scene and the hardware resources available, the methods can even become infeasible. Usually, such methods are suitable for simple scenes with small number of objects.

Image Space Methods

In such an approach, visibility is decided point-by-point at each pixel position on the projection plane. All such methods perform the following steps.

For each pixel on the screen, **do**,

1. *Determine* objects closest to the viewer that are pierced by the projector through the pixel.
2. *Draw* the pixel with the object color.

Clearly, such methods work *after* the surfaces are projected and rasterized (i.e., mapped to pixels). The computations involved are usually less, although the methods depend on the display resolution. A change in resolution requires recomputation of pixel colors.

In this chapter, we shall have a closer look-at some of the algorithms belonging to both these classes.

8.2 APPLICATION OF COHERENCE

As the general algorithms of the two classes of methods show, hidden surface elimination is a computationally intensive process. Therefore, it is natural that we try to reduce computations. One way to do this is to use the *coherence* properties. Here, we exploit the *local similarities*, that is, making use of the results calculated for one part of the scene or image for the other *nearby* parts. There are several types of coherences.

In *object coherence*, we can check for visibility of an object with respect to another by comparing its circumscribing solids (which are usually of simple forms such as sphere or cube). Only if the solids overlap, we go for further processing.

In *face coherence*, surface properties computed for one part of a surface can be applied to adjacent parts of the same surface. For example, if the surface is small, we can sometimes assume that the surface is invisible to a viewer if one part of it is invisible.

Such coherence checking can be performed for edges also. The *edge coherence* indicates that the visibility of an edge changes only when it crosses another edge. Therefore, if one segment of a non-intersecting edge is visible, we determine without further calculation that the entire edge is also visible.

The *scan line coherence* states that a line or surface segment visible in one scan line is also *likely* to be visible in the adjacent scan lines. Therefore, we need not perform visibility computations for every scan line.

In many cases, a group of adjacent pixels in an image is often found to be covered by the same visible object. This is known as the *area and span* coherence, which is based on the assumption that a small enough region of pixels will most likely lie within a single polygon. This reduces computation effort in searching for those polygons which contain a given screen area (region of pixels).

The *depth coherence* tells us that the depth of the adjacent parts of the same surface are similar, which is very useful in determining the visibility of adjacent parts.

Finally, we have the *frame coherence* which implies that pictures of the same scene at successive points in time are likely to be similar, despite small changes in objects and viewpoint, except near the edges of moving objects. Consequently, visibility computations need not be performed for every scene rendered on the screen.

Such properties are used in one form or the other in most of the hidden surface removal methods. As we mentioned, they reduce computations to a good extent. In addition, there is another method called *back face elimination*, which is very simple and eliminates a large number of hidden surfaces.

8.3 BACK FACE ELIMINATION

For a scene consisting of polyhedrons, back face elimination is the simplest way of removing a large number of hidden surfaces. As the name suggests, the objective of this method is to detect and eliminate surfaces that are on the *back* side of objects with respect to the viewer. The process is simple and consists of the following steps.

1. From the surface vertices, determine the normal for each surface (for calculation of normal vectors, see Appendix A). Let $\vec{N}_i = (a, b, c)$ be the normal vector of the ith surface.
2. If $c < 0$, the surface cannot be seen (i.e., it is a *back* face and should be eliminated). Otherwise, if $c = 0$, then the viewing vector grazes the surface. In other words, the surface cannot be seen and should be eliminated. Otherwise, retain the surface (i.e., if $c > 0$).
3. Perform steps 1 and 2 for all surfaces.

Let us understand the idea with an example. Consider the polyhedron of Fig. 8.2, with the vertex coordinates given. As we can see, there are four surfaces, which we can represent as a list $S = [ACB, ADB, DCB, ADC]$. For each of these surfaces, we calculate the z component of the surface normal. Let us start with ACB. The z component of the normal is -12 (for details of calculation, see Appendix A). Since this is less than 0, the surface ACB is not visible. Similarly, for the surfaces ADB, DCB, and ADC, we calculate the z components of the surface normals to be -4, 4, and 2, respectively. Note that the two surfaces DCB and ADC have the z component of the surface normal greater than 0. Hence, these two are the visible surfaces.

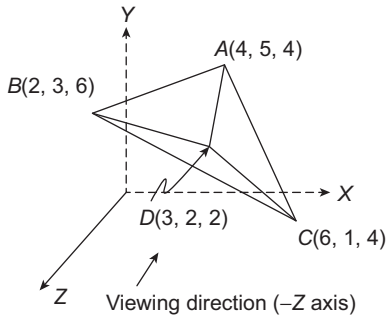


Fig. 8.2 Illustration of the back face elimination method

Since the method works on surfaces, this is an object space method. Using this simple method, about half of the surfaces in a scene can be eliminated. However, note that the method does not consider obscuring of a surface by other objects in the scene. For such situations, we need to apply other algorithms (in conjunction with this method), as we shall see next.

8.4 DEPTH (Z) BUFFER ALGORITHM

The depth (Z) buffer algorithm is an image space method, in which the comparisons are done at the pixel level. We assume the presence of an extra storage, called the *depth-buffer* (also

called *z-buffer*). Its size is the same as that of the frame buffer (i.e., one storage for each pixel). As we assume canonical volumes, we know that the depth of any point within the surface cannot exceed the normalized range; hence, we can fix the size of the depth-buffer (number of bits per pixel).

The idea of the method is simple (we assume that $0 \leq depth \leq 1$): at the beginning, we initialize the depth-buffer locations with 1.0 (the maximum depth value) and the frame buffer locations with the value corresponding to the *background color*. Then, we process each surface of the scene at a time. For each projected pixel position (i, j) of the surface s, we calculate the depth of the point in 3D d_{ij}^s. Then, we compare d_{ij}^s value with the corresponding entry in the depth-buffer (i.e., (i, j)th depth-buffer value DB_{ij}). If $d_{ij}^s < DB_{ij}$, we set $DB_{ij} = d_{ij}^s$ and the surface color value is set to the corresponding location in the frame buffer. The process is repeated for all projected points of the surface and for all surfaces. The pseudocode of the method is shown in Algorithm 8.1.

Algorithm 8.1 **Depth-buffer algorithm**

```
1: Input: Depth-buffer DB[][] initialized to 1.0, frame buffer FB[][] initialized to background color
   value, list of surfaces S, list of projected points for each surface.
2: Output: DB[][] and FB[][] with appropriate values.
3: for each surface in S do
4:   for each projected pixel position of the surface i, j, starting from the top-leftmost projected pixel
     position do
5:     Calculate depth d of the projected point on the surface.
6:     if d <DB[i][j] then
7:       Set DB[i][j]=d
8:       Set FB[i][j]=surface color
9:     end if
10:  end for
11: end for
```

We can follow an iterative procedure to calculate the depth of a surface point. We can represent a planar surface with its equation $ax + by + cz + d = 0$, where a, b, c, and d are surface constants. Thus, depth of any point on the surface is represented as $z = \frac{-ax - by - d}{c}$. Since we are assuming a canonical view volume, all projections are parallel. Thus, a point (x, y, z) is projected to the point (x, y) on the view plane. Now, consider a projected pixel (i, j) of the surface. Then, depth of the original surface point is $z = \frac{-ai - bj - d}{c}$. As we progress along the same scan line, the next pixel is at $(i + 1, j)$. Thus, the depth of the corresponding surface point is $z' = \frac{-a(i+1) - bj - d}{c} = \frac{-ai - bj - d}{c} - \frac{a}{c} = z - \frac{a}{c}$. Hence, along the same scan line, we can calculate the depth of consecutive surface pixels by adding a constant term $(-\frac{a}{c})$ from the current depth value. We can perform similar iterations across scan lines also. Assume a point (x, y) on an edge of the projected surface. If

we go down to the next scan line, the x value of the edge point will be $x' = x - \frac{1}{m}$ where $m \neq 0$ is the slope of the edge. The y value also becomes $y - 1$. Hence, depth of that point is $z' = \frac{-a(x - \frac{1}{m}) - b(y-1) - d}{c}$. Rearranging, we get $z' = z + \frac{\frac{a}{m} + b}{c}$. In other words, the depth of the starting x position of the projected points on a scan line can be found by adding a constant term to the depth of the starting x position of the previous line ($m \neq 0$). The idea is illustrated in Fig. 8.3 with the pseudocode shown in Algorithm 8.2.

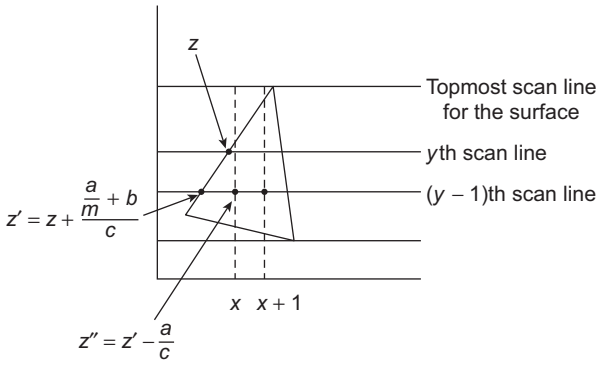


Fig. 8.3 Illustration of the iterative depth calculation procedure

Algorithm 8.2 **Iterative depth calculation**

1: **Input**: Plane constants a, b, c, d
2: **Output**: Depth values of the projected surface points stored in DV[][]
3: Initialize x = leftmost projected point, y = topmost scan line, z value of the leftmost edge point $z' = 0$, constants $c_1 = -\frac{a}{c}, c_2 = \frac{\frac{a}{m} + b}{c}$.
4: Compute depth of the projected surface point at (x, y): $z = \frac{-ax - by - d}{c}$
5: Set $z' = z$
6: **for** y = topmost scan line to the lowermost scan line **do**
7: Set DV[leftmost edge pixel][y] = z
8: **for** x = (leftmost edge pixel + 1) to rightmost edge pixel **do**
9: $z = z + c_1$
10: Set DV[x][y] = z
11: **end for**
12: Set $z = z' + c_2, z' = z$
13: **end for**

Let us try to understand Algorithm 8.1 in terms of a numerical example. For simplicity, we shall assume an arbitrary (that means, the coordinate extents are not normalized) parallel view volume. In that case, we shall initialize depth-buffer with a *very large* value (let us denote that by the symbol ∞).

Example 8.1

Assume there are two triangular surfaces s_1 and s_2 in the view voloume. The vertices of s_1 are [(0,0,6), (6,0,0), (0,6,0)] and that of s_2 are [(2,0,6), (6,0,6), (4,4,0)]. Since we are assuming parallel projection, the projected vertices of s_1 on the view plane are [(0,0), (6,0), (0,6)] (simply drop the z coordinate value). Similarly, for s_2, the projected vertices are [(2,0), (6,0), (4,4)]. The situation is depicted in Fig. 8.4.

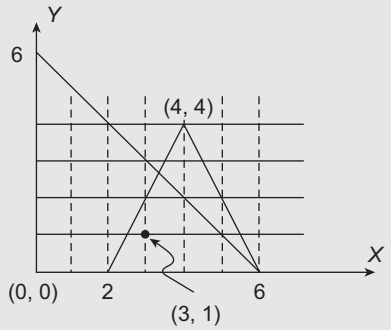


Fig. 8.4 The two surfaces after their projection

Solution Let us follow the steps of the algorithm to determine the color of the pixel (3,1). Assume that cl_1 and cl_2 are the colors of the surfaces s_1 and s_2, respectively and the background color is bg. After initialization, the depth-buffer value DB[3][1]= ∞ and the frame buffer value FB[3][1]=bg. We will process the surfaces one at a time in the order s_1 followed by s_2.

From the vertices, we can determine the surface equation of s_1 as $x + y + z - 6 = 0$ (see Appendix A for details). Using the surface equation, we first determine the depth of the leftmost projected surface pixel on the topmost scan line. In our case, the pixel is (0,6) with a depth of $z = \dfrac{-1.0 - 1.6 - (-6)}{1} = 0$. Since this is the only point on the topmost scan line, we move to the next scan line below ($y = 5$). Using iterative method, we determine the depth of the leftmost projected pixel on this scan line (0,5) to be $z' = z + \dfrac{\frac{a}{m} + b}{c}$. However, note that we have the slope of the left edge $m = \infty$. Hence, we set $\dfrac{a}{m} = 0$. Therefore, $z' = 0 + \dfrac{1}{1} = 1$. The algorithm next computes depth and determines the color values of the pixel along the scan line $y = 5$ till it reaches the right edge. At that point, it goes to the next scan line down ($y = 4$). For brevity, we will skip these steps and go to the scan line $y = 1$, as our point of interest is (3,1).

Following the iterative procedure across scan lines, we compute the depth of the left most projected surface point (0,1) as $z = 5$ (check for yourself). We now move along the scan line to the next projected pixel (1,1). Its depth can be iteratively computed as $z = z + (-\dfrac{a}{c}) = 5 - 1 = 4$. Similarly, the depth of the next pixel (2,1) is $z = 4 - 1 = 3$. In this way, we calculate depth of s_1 at (3,1) as $z = 3 - 1 = 2$. As you can see, this depth value at (3,1) is less than DB[3][1], which is ∞. Hence, we set DB[3][1]=2 and FB[3][1]= cl_1.

Afterwards, the other projected points of s_1 are computed in a likewise manner, till the rightmost edge point on the lowermost scan line. However, we shall skip those steps for brevity and move to the processing of the next surface.

From the vertices of s_2, we derive the surface equation as $3y + 2z - 12 = 0$ (see Appendix A for details). The projected point on the topmost scan line is (4,4). Therefore, depth at this point is $z = \frac{-3.4 - (-12)}{2} = 0$. Going down the scan lines (skipping the pixel processing along the scan lines for brevity, as before), we reach $y = 1$. Note that the slope of the left edge of the projected surface is $m = 2$. We can calculate the left-most projected point on $y = 1$ iteratively based on the fact that the x-coordinate of the intersection point of the line with slope m and the $(y - 1)$th scan line is $x - \frac{1}{m}$, if the x-coordinate of the intersection point of the same line with the yth scan line is x. In this way, we compute $x = 2.5$ for $y = 1$. In other words, the leftmost projected point of s_2 on $y = 1$ is (2.5,1). Using the iterative procedure for depth calculation across scan line (with $z = 0$ at $y = 4$), we compute depth at this point to be $z = 4.5$ (check for yourself). Next, we apply the iterative depth calculation along the scan line to determine depth of the projected point (3,1) (the very next projected pixel) to be $z = 4.5$.

Note that $z = 4.5 > DB[3][1]$, which is having the value 2 after the processing of s_1. Therefore, the DB value and the corresponding FB value (which is cl_1) is *not* changed. The algorithm processes all the other pixels along $y = 1$ till the right edge in a likewise manner and the pixels for $y = 0$. However, as before we skip those calculations for brevity. Thus, at the end of processing the two surfaces, we shall have DB[3][1]=2 and FB[3][1]= cl_1.

8.5 A-BUFFER ALGORITHM

In the depth-buffer algorithm, a pixel can have only one surface color. In other words, from any given viewing position, only one surface is visible. This is alright if we are dealing with opaque surfaces only. However, as we know, if the scene contains transparent surfaces visible from a pixel position, the pixel color is a combination of the surface color as well as contributions from the surfaces behind (see Chapter 4). Since in the depth-buffer method, we have only one location to store depth value for each pixel (the depth buffer), we cannot store all the surfaces contributing to the color value for a transparent surface simultaneously.

The A-buffer method is an attempt to overcome this limitation of the depth-buffer algorithm. The 'A' in the A-buffer method stands for *accumulation*. The algorithm works in the following way: as before, we have access to a depth-buffer although we now call it A-buffer. Each location of the A-buffer corresponds to a pixel position just like a depth buffer. However, unlike before, each position in the buffer is *not* a single field of depth value. Instead, it can reference a linked list of surfaces. There are the following two fields at each A-buffer location.

1. The *depth* field as in the depth-buffer. This field can store a real-number value (positive, negative, or zero).
2. The *surface data* field. This is the modification to the depth-buffer. It contains various surface data or a pointer to the next node in a linked list data structure.

The surface data includes the following:

1. RGB intensity components
2. Opacity factor value (indicating percent of transparency)
3. Depth
4. Surface identities

Depending on the value stored at the depth field of the A-buffer, we will be able to determine if the pixel color comes from a single surface (i.e., opaque surface) or a combination of multiple surface colors (i.e., a transparent surface is visible from the pixel position). The convention usually followed is this: if the depth field stores a *non-negative* value, the number indicates the depth of an opaque surface. The surface data field contains various information related to that surface. However, if the depth field contains a *negative* value, the visible surface is transparent. In that case, the surface data field contains a pointer to a linked list of data of all surfaces that contribute to the surface color. The two situations are shown schematically in Fig. 8.5 for illustration. The scenario when depth of the visible surface is non-negative is shown in Fig. 8.5(a). The case for a transparent visible surface is shown in Fig. 8.5(b).

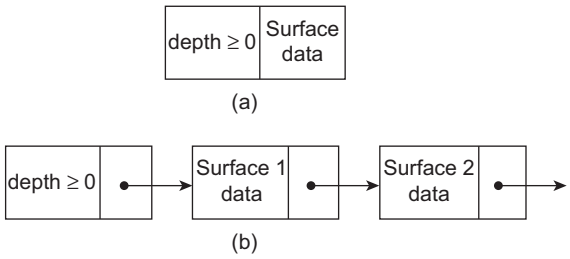


Fig. 8.5 The figure illustrates the organization of an A-buffer location for the two possible cases. In (a), organization for an opaque visible surface is shown. The case for a transparent visible surface is shown in (b).

8.6 DEPTH SORTING (PAINTER'S) ALGORITHM

As we noted before, back face elimination is an object space method, whereas the depth-buffer algorithm is performed in image space. In this section, we shall learn about the depth sorting algorithm that works at both the image and object spaces. The algorithm is often called the *painter's algorithm*, as it tries to simulate the way a painter draws a scene.

There are two basic steps in the algorithm: first, we *sort* the surfaces present in the scene on the basis of their depth (from the view plane). In order to do that, we determine the maximum and minimum depths of each surface. The list is created based on the maximum depth. Let us denote the list by $S = \{s_1, s_2, \cdots s_n\}$ where s_i denotes the ith surface and depth(s_i)<depth(s_{i+1}). Next, we render the surface on the screen one at a time starting with the surface having maximum depth (i.e., s_n in S) to the surface with the lowest depth.

During rendering of each surface s_i, we compare it with all the other surfaces of S to see if there is any *depth overlap* (i.e., the minimum depth of one surface is greater than the

maximum depth of the other surface, see Fig. 8.6). If no overlap is found, we render the surface and remove it from S. Otherwise, we perform the following checks.

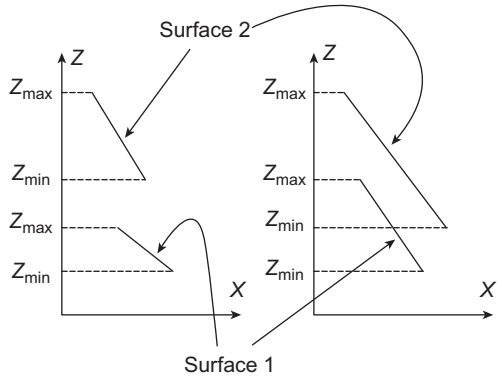


Fig. 8.6 The figure illustrates the idea of depth overlap. No depth overlap between the two surface is there on the left figure whereas in the right figure, the surfaces overlap.

1. The *bounding rectangles* of the two surfaces do not overlap.
2. Surface s_i is completely *behind* the overlapping surface relative to the viewing position.
3. The overlapping surface is completely *in front of* s_i relative to the viewing position.
4. The boundary edge projections of the two surfaces onto the view plane do not overlap.

The first check is *true* if there is *no* overlap in the x and y coordinate extents of the two surfaces. If either of these coordinates overlap, then the first condition *fails*. Figure 8.7 illustrates the idea for the case where there is an overlap of the bounding rectangles along the x direction. In order to check for the second condition in this list, we need to determine the plane equation of the overlapping surface (the plane normal should point towards the viewer). Next, we check all the vertices of s_i with the equation. If for *all* the vertices of s_i, the plane equation returns a value < 0, s_i is behind the overlapping surface. Otherwise, the second condition *fails*. The idea is illustrated in Fig. 8.8. The third condition can be checked similarly with the plane equation of s_i and the vertices of the overlapping surface (for all

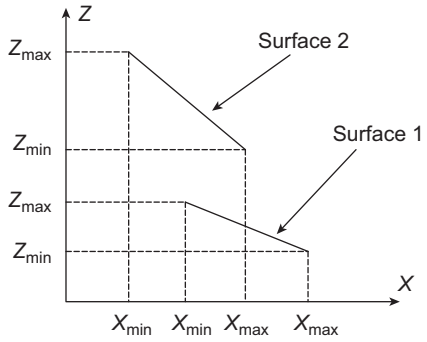


Fig. 8.7 The figure illustrates the idea of bounding rectangle overlap of two surfaces along the *x* axis

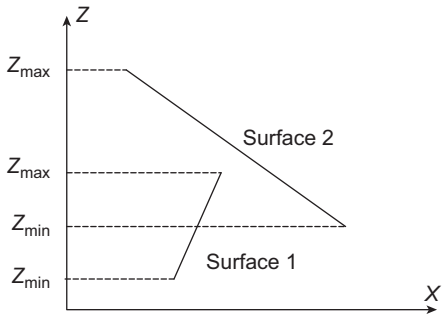


Fig. 8.8 An example showing one surface (surface 1) completely behind the other surface, viewed along the -*z* direction

vertices, the equation should return positive value). Figure 8.9 depicts the situation for two surfaces. In order to check the final condition, we need to have the set of projected pixels for each surface and then check if there are any common pixels in the two sets (se Fig. 8.10 for illustration). As you can see, the first and the last checks are performed at the pixel level, whereas the other two checks are performed at the object level. Hence, the depth sorting algorithm incorporates elements of both the object space and image space methods.

The tests are performed following the order as in our preceeding discussion. As soon as one of the checks is true, we move to check for overlap with the next surface of the list.

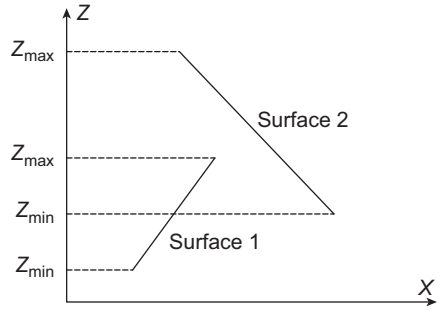


Fig. 8.9 Illustration of one surface (surface 2) completely in front of surface 1, although surface 1 is *not* completely behind surface 2

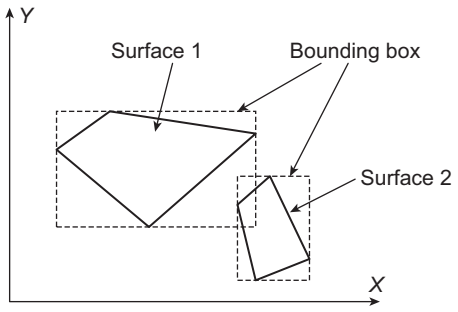


Fig. 8.10 An example where the projected surfaces do not overlap although their bounding rectangles do

If all tests fail, we swap the order of the surfaces in the list (called *reordering*) and stop. Then, we restart the whole process again. The steps of the depth sorting method, in pseudocode, are shown in Algorithm 8.3.

Algorithm 8.3 Painter's Algorithm

```
1: Input: List of surfaces S = {s1, s2, ··· sn}, in sorted order (of increasing maximum depth value).
2: Output: Final frame buffer values.
3: Set a flag Reorder=OFF
4: repeat
5:   Set s = sn (i.e., the last element of S)
6:   for for each surface si in S where 1 ≤ i < n do
7:     if zmin(s) < zmax(si) (that means, there is depth overlap) then
8:       if (bounding rectangles of the two surfaces on the view plane do not overlap then
9:         Set i = i + 1 and continue loop.
10:      else if s is completely behind si then
11:        Set i = i + 1 and continue loop
12:      else if si is completely in front of s then
13:        Set i = i + 1 and continue loop
14:      else if projections of s and si do not overlap then
15:        Set i = i + 1 and continue loop
16:      else
17:        Swap the positions of s and si in S
18:        Set Reorder = ON
19:        Exit inner loop
20:      end if
21:    end if
22:  end for
23:  if Reorder = OFF then
24:    Invoke rendering routine for s
25:    Set S = S − s
26:  else
27:    Set Reorder = OFF
28:  end if
29: until S = NULL
```

Sometimes, there are surfaces that intersect each other. As an example, consider Fig. 8.11, in which the two surfaces intersect. As a result, one part of surface 1 is at a depth larger than surface 2, although the other part has lesser depth. Therefore, we may initially keep surface 1 after surface 2 in the sorted list. However, since the conditions fail (check for yourself), we have to reorder them (bring surface 1 in front of surface 2 in the list). As you can check, the conditions shall fail again and we have to reorder again. This will go on in an infinite loop and Algorithm 8.3 will loop forever.

In order to avoid such situations, we can use an extra *flag* (a Boolean variable) for each surface. If a surface is reordered, the corresponding flag will be set *on*. If the surface needs to be reordered next time, we shall do the following.

1. *Divide* the surface along the intersection line of the two surfaces.
2. *Add* the two new surfaces in the sorted list, at appropriate positions.

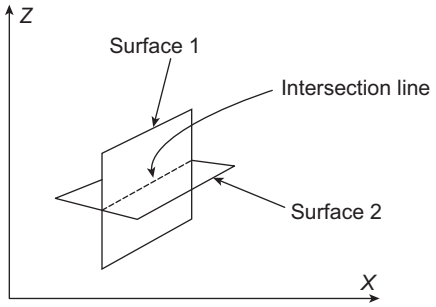


Fig. 8.11 Two surfaces that intersect each other. Algorithm 8.3 should be modified to take care of such cases

8.7 WARNOCK'S ALGORITHM

There is a group of hidden surface removal techniques collectively known as the *area subdivision methods*. All these methods work on the same general idea: we first consider an *area* of the projected image. If we can determine which (polygonal) surfaces are visible in the area, we assign those surface colors to the area. Otherwise, we *recursively* subdivide the area into smaller regions and apply the same decision logic on the subregions. The Warnock's algorithm is one of the earliest among all the subdivision methods developed for hidden surface removal. The algorithm works as follows.

First, we subdivide a screen area into *four* equal squares. Then, we check for visibility in each square to determine the color of the pixels contained in the (square) region. The algorithm proceeds as per the following cases.

1. The current square region being checked does not contain any surface. Thus, we do not subdivide the region any further and assign background color to the pixels contained in it.
2. The nearest surface *completely* overlaps the region under consideration. In that case, the square is not subdivided further. We assign the surface color to the region.
3. None of these. We recursively divide the region into four subregions and repeat the aforementioned checks. The recursion stops if either of the cases is met or the region size becomes equal to the pixel size.

The steps of the Warnock's algorithm are shown in Algorithm 8.4 (in pseudocode).

Let us consider Fig. 8.12 to understand the steps of Algorithm 8.4. In the figure, there is a surface occupying a region of the screen (the shaded region). One vertex of the polygonal surface is on the center of the screen. We shall execute the steps of Algorithm 8.4 to determine the color of the screen where the surface is.

We make the first call to the algorithm with the whole screen region as input. The algorithm then creates four subregions of equal size (denoted by P_1, P_2, P_3, and P_4 in the figure) and checks for visibility of the surface for each subregion inside the loop. We first check the region P_1. Since it does not contain any surface, background color is assigned to

Algorithm 8.4 **Warnock's Algorithm**

```
1:  Input: The screen region
2:  function Warnock (Projected region P)
3:  Divide the input region P into four equal sized subregions P1, P2, P3 and P4
4:  for each subregion Pi do
5:    if there is no surface in Pi or Pi equals the pixel size then
6:      Assign background color to Pi
7:    else if the nearest surface completely overlaps Pi then
8:      Assign the surface color to Pi
9:    else
10:     Warnock (Pi)
11:   end if
12: end for
```

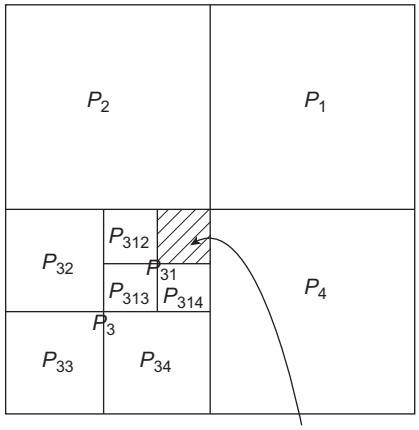


Fig. 8.12 Example illustrating Warnock's algorithm

this region. Next, we check the region P_2. Again, no surface is contained within this region. So, we assign background color to the region and proceed to the next region P_3.

We determine that P_3 contains the surface. However, it is *not* completely overlapped by the surface. Therefore, we go for dividing the region into four subregions of equal size (the recursive call in the Algorithm 8.4, line 10). The four subregions are denoted by P_{31}, P_{32}, P_{33}, and P_{34} in the figure. For each of these subregions, we perform the checks again.

We find that the subregion P_{31} contains the surface. However, the surface does not completely overlap P_{31}. Therefore, we go for subdividing the region. The four subregions of P_{31} are denoted as P_{311}, P_{312}, P_{313}, and P_{314}. We then check each of these subregions for surface visibility.

Since the surface lies in the subregion P_{311} and is completely overlapping it, we assign the surface color to the subregion P_{311}. The other three subregions of P_{31} do not contain

any surface. Therefore, all these subregions are assigned background color. This completes our processing of the subregion P_{31}.

We then retrace the recursive step and go for checking the other three subregions of the region P_3, namely P_{32}, P_{33}, and P_{34}. Since none of them contains any surface, we assign background color to them and complete our processing of the subregion P_3.

We then return from the recursive step to check the remaining subregion P_4 of the screen. We find that the region contains no surface. Therefore, background color is assigned to it. Since all the regions have been checked, the algorithm stops.

8.8 OCTREE METHODS

The depth-buffer algorithm is an image space method while the depth sorting algorithm is a combination of both image and object space methods. There are several other algorithms available for hidden surface removal, which belong to either of the methods. In the following, we discuss another hidden surface removal technique based on the octree representation of objects. As may be clear to you, the method depends on a particular object representation technique, namely the octree representation (see Chapter 2). Therefore, we categorize it as an object space method. Note that the method assumes a volumetric (or interior) representation of objects, while in the back face elimination or depth sorting methods, we assumed that objects are represented by their bounding surfaces (exterior representation).

As you may recall, in octree representation, we first divide a 3D space into eight regions (quadrants). Each region is then subdivided into eight subregions. The process continues in a recursion till we reach a predefined region size (usually unit cubes, sometimes called *voxels*, see Chapter 2). After the division, we get an octree (a tree with each node having eight children). For simplicity, we shall assume a cubical region to start with and that the divisions are performed till we reach voxels. In such case, the leaf nodes (voxels) of the octree stores attributes of the object associated with it (we shall assume only color as the object attribute for simplicity). Thus, the voxels with attributes define objects in the scene. In terms of this simplistic representation, let us try to understand the octree methods for hidden surface removal.

During the creation of the octree representation, we label each of the eight subregions of a region according to its position with respect to the viewer. Refer to Fig. 8.13. With respect to the viewer position, we may number the eight subregions of a region as {1,2,3,4} denoting *front* four quadrants and {5,6,7,8} denoting the four *back* quandrants. We follow such convention in each step of the recursive subdivision of a region. When we reach the leaf

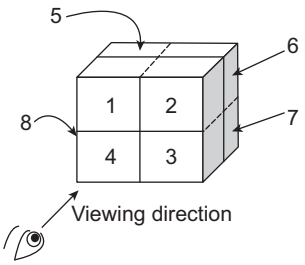


Fig. 8.13 The naming of regions with respect to a viewer in an octree method

nodes (i.e., voxels), each voxel shall have the information about its position with respect to the viewer, along with the color of the object associated with it.

In order to render the scene represented by the octree, we project the voxels on the view plane in a *front-to-back* manner. In other words, we start with the voxels nearest to the viewer, then move to the next nearest voxels, and so on. It is easy to see that the terms such as *voxels nearest to the viewer* indicates a voxel grid (having voxels at the same distance from the viewer) with a one-to-one correspondance to the pixel grid on the view plane (since both the grid sizes are same). When a color is encountered in a voxel (of a particular grid), the *corresponding pixel* in the frame buffer is painted *only if* no previous color has been loaded into the same pixel position. This we can achieve by initially assuming that the frame buffer locations contains 0 (no color). A pseudocode of the method (for our simplest octree representation) is shown in Algorithm 8.5.

Algorithm 8.5 **Octree method of hidden surface removal**

```
1: Input: The set of octree leaf nodes (voxels) OCT with each voxel having two attributes (distance from
   the viewer, color). We assume that the nearest voxel to viewer has a distance 0.
2: Output: Frame buffer with appropriate color in its locations
3: Set all frame buffer locations to 0 (to denote no color)
4: Set distance = 0
5: repeat
6:   for From the leftmost to the rightmost element of OCT do
7:     Take an element of OCT (denoted by v)
8:     if distance of v from viewer = distance then
9:       if corresponding frame buffer location has a value 0 then
10:        Set the voxel color as the corresponding frame buffer value
11:      end if
12:    end if
13:    Set OCT = OCT − v
14:  end for
15:  Set distance = distance + 1
16: until OCT = NULL
```

Example 8.2

Let us consider an example to illustrate the idea of our simplistic octree method for hidden surface removal. Assume that we have a display device with a 4×4 pixel grid. On this display, we wish to project a scene enclosed in a cubical region with each side = 4 unit. Note that, we shall have two levels of recursion to create the octree representation of the scene till we reach the voxel level. In the first recursion, we create eight regions from the original volume given, each region having a side length = 2 units. In the next level of recursion, we divide each of the subregions further, so that we reach the voxel level. The divisions are illustrated in Fig. 8.14, showing two levels of recursion for one quadrant. Other quadrants are divided similarly.

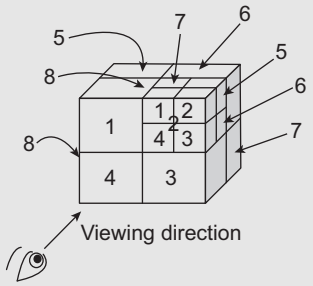


Fig. 8.14 Octree of Example 8.2

Note how the naming convention is used. In the first level of recursion, we named the quadrants as we discussed before ({1,2,3,4} are the front regions while {5,6,7,8} are the back regions with respect to the viewer). In the second level of recursion, we have done the same thing. Thus, after the recursion ends, each voxel is associated with two numbers in the form {first level number, second level number}. For example, the voxels shown in Fig. 8.14 will have numbers as {1,1}, {1,2}, {1,3}, {1,4}, {1,5}, {1,6}, {1,7} and {1,8}.

As you can see, from these numbers, we can make out the relative position of the voxels with respect to the viewer. Hence, we can easily determine the voxel grids (i.e., voxels at the same distance from the viewer). There are four such grids as follows:

Grid 1: {1,1}, {1,2}, {1,3}, {1,4}, {2,1}, {2,2}, {2,3}, {2,4}, {3,1}, {3,2}, {3,3}, {3,4}, {4,1}, {4,2}, {4,3}, and {4,4} [distance = 0 from the viewer]

Grid 2: {1,5}, {1,6}, {1,7}, {1,8}, {2,5}, {2,6}, {2,7}, {2,8}, {3,5}, {3,6}, {3,7}, {3,8}, {4,5}, {4,6}, {4,7}, and {4,8} [distance = 1 from the viewer]

Grid 3: {5,1}, {5,2}, {5,3}, {5,4}, {6,1}, {6,2}, {6,3}, {6,4}, {7,1}, {7,2}, {7,3}, {7,4}, {8,1}, {8,2}, {8,3}, and {8,4} [distance = 2 from the viewer]

Grid 4: {5,5}, {5,6}, {5,7}, {5,8}, {6,5}, {6,6}, {6,7}, {6,8}, {7,5}, {7,6}, {7,7}, {7,8}, {8,5}, {8,6}, {8,7}, and {8,8} [distance = 3 from the viewer]

It is also easy to define a mapping between voxel and pixel grids. For example, we may define that a voxel with location (i,j) in a grid maps to the pixel (i,j) in the pixel grid.

Now, let us try to execute the steps of Algorithm 8.5. Initially, all pixels have a color value 0 and OCT contains all the voxels of the four grids. Since the distance of the voxels in grid 1 is 0, all these voxels will be processed in the inner loop first. During the processing of each voxel, its color (if any) will be set as the color of the corresponding frame buffer location. Afterwards, the voxel will be removed from the list of voxels in OCT. Thus, after the first round of processing of the inner loop, OCT shall contain voxels of grids 2, 3 and 4.

In a similar manner, voxels of grid 2 will be processed during the second round of inner loop execution and the frame buffer colors modified appropriately (i.e., if a frame buffer location already contains a non zero color value and the corresponding voxel in grid 2 has a color, the frame buffer color value remains unchanged. Otherwise, the current voxel color will replace the frame buffer color) . After the second round of inner loop execution, OCT shall contain grid 3 and 4 voxels and the inner loop executes third time. The process continues in this way till the fourth round execution of the inner loop is over, after which we will have OCT = NULL and the execution of the algorithm stops.

SUMMARY

In this chapter, we learnt about the idea of hidden surface removal in a scene. The objective is to eliminate surfaces that are invisible to a viewer, with respect to a viewing positions. We learnt about the two broad types of methods—image space and object space methods. The former works at the pixel level while the later works at the level of object representations.

In order to reduce computations, coherence properties are used in conjunction with the algorithms. We mentioned seven such properties, namely (a) object coherence, (b) face coherence, (c) edge coherence, (d) scan line coherence, (e) area and span coherence, (f) depth coherence, and (g) frame coherence. We discussed the ideas in brief. In addition to these, we also saw how the back face elimination method provides a simple and efficient way for removal of a large number of hidden surfaces.

Among the many hidden surface removal algorithms available, we discussed three in detail along with illustrative examples. The first algorithm, namely the depth-buffer algorithm, is one of the most popular algorithms which works in the image space. The depth sorting algorithm that we discussed next works at both the image and object space. As we saw, it is more complex and computation-intensive compared to the depth-buffer algorithm. The third algorithm, namely the octree method, is an object space method that is based on the octree representation of objects. We illustrated the idea of octree methods considering few simplisitic assumptions.

In the next chapter, we will discuss the final stage of a 3D graphics pipeline, namely *rendering*.

BIBLIOGRAPHIC NOTE

There are a large number of hidden surface removal techniques. We have discussed a few of those. More techniques can be found in Elber and Cohen [1990], Franklin and Kankanhalli [1990], Segal [1990], and Naylor et al. [1990]. A well-known hidden surface removal technique is the A-buffer method. Works on this are presented in Cook et al. [1987], Haeberli and Akeley [1990], and Shilling and Strasser [1993]. Hidden surface removal is also important in three-dimensional line drawings. For curved surfaces, contour plots are displayed. Such contouring techniques are summarized in Earnshaw [1985]. For various programming techniques for hidden surface detection and removal, the *graphics gems* book series can be referred (Glassner [1990], Arvo [1991], Kirk [1992], Heckbert [1994], and Paeth [1995]).

KEY TERMS

(A)ccumulation buffer – a data structure to store depth and associated information for each surface to which a pixel belongs

A-buffer method – an image space technique for hidden surface removal that works with transparent surfaces also

Area subdivision – recursive subdivision of the projected area of a surface

Back face elimination – an object space method for hidden surface removal

Coherence – the property by which we can apply some results calculated for one part of a scene or image to the other parts

Depth (Z) buffer algorithm – an image space method for hidden surface removal

Depth-buffer – a data structure to store the depth information of each pixel

Depth coherence – the depth of nearby parts of a surface is similar

Depth overlap – the minimum depth of one surface is greater than the maximum depth of another surface

Depth sorting (Painter's) algorithm – a hidden surface removal technique that combines elements of both object space and image space methods
Face coherence – the property by which we can check visibility of one part of a surface by checking its properties at other parts
Frame coherence – pictures of the successive frames are likely to be similar
Hidden surfaces – object surfaces that are hidden with respect to a particular viewing position
Image space method – hidden surface removal techniques that work with the pixel level projections of object surfaces
Object coherence – determining visibility of an object surface with respect to nearby object surfaces by comparing their bounding volumes
Object space methods – hidden surface removal techniques that work with the objects, rather than their projections on the screen.
Octree method – an object space method for hidden surface removal
Scan line coherence – a line or surface segment visible in one scan line is also likely to be visible in the adjacent scan line
Visible surfaces – surfaces that are visible with respect to the viewing position
Warnock's algorithm – the earliest area subdivision method for hidden surface removal

EXERCISES

8.1 Discuss the importance of hidden surface removal in a 3D graphics pipeline. How is it different from clipping?
8.2 What are the broad classes of hidden surface removal methods? Describe each class in brief along with its pros and cons.
8.3 Briefly explain the idea of coherence. Why is it useful in hidden surface removal?
8.4 In which category of methods does the depth-buffer algorithm belong to? Justify.
8.5 We have discussed the depth-buffer algorithm (Algorithm 8.1) and the iterative depth calculation (Algorithm 8.2) separately. Write the pseudocode of an algorithm combining the two.
8.6 The depth sorting method is said to be a hybrid of the object space and image space methods. Why?
8.7 Why does Algorithm 8.3 fail for intersecting surfaces? Explain with a suitable example.
8.8 Modify Algorithm 8.3 to take into account intersecting surfaces.
8.9 Consider the objects mentioned in Example 8.1. Can Algorithm 8.3 be applied to these objects or the modified algorithm you wrote to answer Example 8.8? Show the execution of the steps of the *appropriate* algorithm for the objects.
8.10 Both the back face elimination and octree methods belong to the object space methods. What is the major difference between them?
8.11 Assume that during octree creation, we named each region as illustrated in Example 8.2. Write the pseudocode of an algorithm to determine the distance of a voxel from the viewer. Integrate this with Algorithm 8.5.
8.12 Algorithm 8.5 assumed a simplistic octree representation. Discuss ways to improve it.

CHAPTER 9 Rendering

Learning Objectives

After going through this chapter, the students will be able to

- Understand the concept of scan conversion
- Get an overview of the issues involved in line scan conversion
- Learn about the digital differential analyser (DDA) line drawing algorithm and its advantage over the intuitive approach
- Understand the Bresenham's line drawing algorithm and its advantage over the DDA algorithm
- Get an overview of the issues involved in circle scan conversion
- Learn about the mid-point algorithm for circle scan conversion
- Understand the issues and approaches for fill area scan conversion
- Learn about the seed fill, flood fill, and scan line polygon fill algorithms for fill area scan conversion
- Get an overview of character rendering methods
- Understand the problem of aliasing in scan conversion
- Learn about the Gupta-Sproull algorithm for anti-aliasing lines
- Learn about the area sampling and supersampling approaches towards anti-aliasing

INTRODUCTION

Let us review what we have learnt so far. In a 3D graphics pipeline, we start with the definition of objetcs that make up the scene. We learnt different object definition techniques. We also learnt the various geometric transformations to put the objects in their appropriate place in a scene. Then, we learnt about the lighting and shading models that are used to assign colors to the objects. Subsequently, we discussed the viewing pipeline comprising the three stages: (a) view coordinate formation, (b) projection, and (c) window-to-viewport transformations. We also learnt various algorithms for clipping and hidden surface removal.

Thus, we now know the stages involved in transforming a 3D scene to a 2D viewport in the device coordinate system. Note that a device coordinate system is continuous in nature (i.e., coordinates can be any real number). However, we must use the pixel grid to render a scene on a physical display. Clearly, pixel grids represent a discrete coordinate system, where any point **must** have integer coordinates. Thus, we need to *map* the

scene defined in the viewport (continuous coordinates) to the pixel grid (discrete coordinates). The algorithms and techniques used for perfoming this mapping are the subject matter of this chapter. These techniques are collectively known as *rendering* (often called *scan conversion* or *rasterization*). In the rest of this chapter, these terms will be used synonymously.

The most basic problem in scan conversion is to map a point from the viewport to the pixel grid. The approach is very simple: just round off the point coordinates to their nearest integer value. For example, consider the point P(2.6,5.1). This viewport point is mapped to the pixel grid point P′(3,5) after rounding off the individual coordinates to their nearest integers. However, scan conversion of more complex primitives such as line and circle are not so simple and we need more complex (and efficient) algorithms. Let us learn about those algorithms.

9.1 SCAN CONVERSION OF A LINE SEGMENT

We know that a line segment is defined by its end points. In order to scan convert the line segment, we need to *map* points on the line to the *appropriate* pixels. We can follow a simple approach: first, we shall map the end points to the appropriate pixels following the point scan conversion method (i.e., round off to nearest integer). This will give us the starting and ending pixels for the line segment. We will then take one end point having the lower x and y coordinate values. As we know that two pixels are seperated by a unit distance along x- axis, we then work out the y coordinates for successive x-coordinates differing by one. The computed y-coordinate is then mapped to its nearest integer, giving us the pixel coordinates of the point. Let us try to understand this idea in terms of an example.

Assume we have a line segment defined by the end points A(2.1,2.3) and B(6.7,5.2). We first convert the end points to pixels. Thus, the two end point pixels of the line segment are: A'(2,2) and B'(7,5) (after rounding off each coordinate value to the nearest integer). Since the coordinate values of A' are less than those of B', we take A' as our starting pixel and compute the points on the line.

First, from the end points, we compute slope $(m) = \frac{5-2}{5-2} = \frac{3}{5}$ [we know that $m = \frac{y_2-y_1}{x_2-x_1}$] and y-intercept (b) $= 2 - \frac{3}{5}.2 = \frac{4}{5}$ [from the equation $y_1 = m.x_1 + b$]. Then, for each x-coordinate separated by unit distance (starting from 2), we compute the y-coordinate using the line equation $y = \frac{3}{5}x + \frac{4}{5}$ (till we reach the other end point B'). The computation yields the following four values of y.

$$y(x = 3) = \frac{3}{5}3 + \frac{4}{5} = 2.6$$
$$y(x = 4) = \frac{3}{5}4 + \frac{4}{5} = 3.2$$
$$y(x = 5) = \frac{3}{5}5 + \frac{4}{5} = 3.8$$
$$y(x = 6) = \frac{3}{5}6 + \frac{4}{5} = 4.4$$

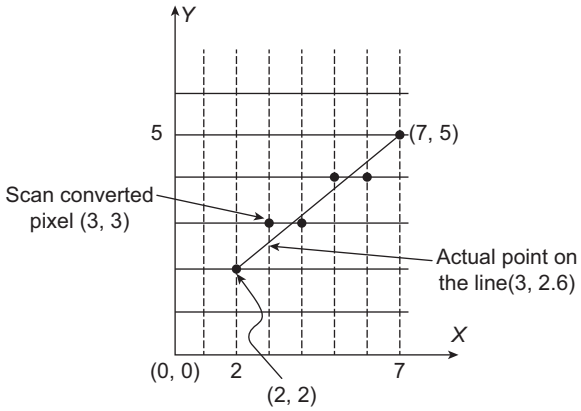


Fig. 9.1 Simple line scan conversion—Note that the actual points on the line are scan converted to the nearest pixels after rounding off

Thus, between A' and B', we obtain the four points on the line as (3,2.6), (4,3.2), (5,3.8), and (6,4.4). Following point conversion technique, we determine the pixels for these points as (3,3), (4,3), (5,4), and (6,4), respectively. The idea is illustrated in Fig. 9.1.

The approach is very simple. However, there are mainly two problems with this approach. First, we need to perform the multiplication $m.x$. Second, we need to round off the y-coordinates. Both of these may involve floating point operations, which are computationally expensive. In typical graphics applications, we need to scan convert very large number of lines within a very small time. In such cases, floating point operations make the process slow and flickers may occur. Thus we need some better solution.

Role of slopes in line scan conversion

In the simple line scan conversion we discussed, we calculated the y-coordinate for each x-coordinate. We could have done the other way round also. Let us see the result for the same example (Fig. 9.1). As before, we start with the end point A'. This time, we calculate the x-coordinates of successive points by increasing y by 1 (moving from one scan line to the next) based on the equation $x = \frac{y-b}{m}$. Therefore, we get two x values between $y = 2$ and $y = 5$ (the two end pixel y-coordinates), unlike the four y values we got before. The two x values are,

$$x(y = 3) = \frac{3 - \frac{4}{5}}{\frac{3}{5}} = 3.7$$

$$x(y = 4) = \frac{4 - \frac{4}{5}}{\frac{3}{5}} = 5.3$$

Thus, the two computed points between A' and B' are (3.7,3) and (5.3,4). These two are scan converted to the pixels (4,3) and (5,4). Recall that when we computed y, we got the pixels (including end points) (2,2), (3,3), (4,3), (5,4), (6,4), and (7,5). However, if we calculate x for this line segment we get the pixels (2,2), (4,3), (5,4), and (7,5). As you can see, the line segment rendered with the first set of pixels is much better compared to the second set of pixels. Thus, we have to decide which coordinate to calculate when.

The decision is taken based on the slope of the line. If we have $0 \leq m \leq 1$ or $-1 \leq m \leq 0$, we work out y-coordinates based on x-coordinates (as we did in the example). Otherwise, we compute x-coordinates based on the y-coordinates.

9.1.1 DDA Algorithm

DDA stands for *digital differential analyser*. In the DDA algorithm, we take an *incremental* approach to speed up line scan conversion. In order to illustrate the working of the algorithm, let us consider the previous example again.

Recall that there are four pixels we calculated between the two end pixels (2,2) and (7,5). These pixels are (3,2.6), (4,3.2), (5,3.8), and (6,4.4). We know that the slope of the line $m = \frac{3}{5} = 0.6$. Note that the successive y-coordinates are obtained by adding 0.6 to the current value (i.e., (3,2.6), (4,3.2 + 0.6), (5,(3.2 + 0.6) + 0.6), and (6,((3.2 + 0.6) + 0.6) + 0.6). Thus, instead of computing the y-coordinate by the line equation $y = m.x + b$ every time, we can simply add m to the current y value (i.e., $y_{k+1} = y_k + m$). In this way, we eliminate the floating point multiplication $m.x$ from calculation. When $m > 1$ or $m < -1$, we compute successive x-coordinates as $x_{k+1} = x_k + \frac{1}{m}$ (derivation is left as an exercise), in the process eliminating floating point operations. The algorithm is shown in Algorithm 9.1, where RoundOff(a) is a function to round off the real number a to its nearest integer.

Although the DDA algorithm is able to reduce *some* floating point operations (multiplication), it still requires floating point addition and rounding off operations. Moreover, for large line segments, the rounding off may result in pixels that are far away from the actual line. So, it is preferable to have a more efficient algorithm for line scan conversion.

Algorithm 9.1 **DDA algorithm**

1: **Input:** The two line end points $(x_1, y_1$ and $x_2, y_2)$
2: **Output:** Set of pixels P to render the line segment
3: Compute $m = \frac{y_2 - y_1}{x_2 - x_1}$
4: **if** $0 \leq m \leq 1$ or $-1 \leq m \leq 0$ **then**
5: Set $x = x_1 + 1, y = y_1$
6: **repeat**
7: $y = y + m$
8: RoundOff(y)
9: Add (x, y) to P
10: Set $x = x + 1$
11: **until** $x < x_2$
12: **else**
13: Set $x = x_1, y = y_1 + 1$
14: **repeat**
15: $x = x + \frac{1}{m}$
16: RoundOff(x)
17: Add (x, y) to P
18: Set $y = y + 1$
19: **until** $y < y_2$
20: **end if**

9.1.2 Bresenham's Algorithm

The Bresenham's algorithm is a very efficient way to scan convert a line segment. Let us first understand the idea of the algorithm. We shall discuss the idea for line segments with $0 \leq m \leq 1$ or $-1 \leq m \leq 0$.

Consider Fig. 9.2. As we know, in order to determine pixels, we move along the x-direction in unit steps from the current position (x_k, y_k). At each of these steps, we have to choose between the two y values y_k and $y_k + 1$. Clearly, we would like to choose a pixel that is closer to the original line. Let us denote the distances of $(x_k + 1, y_k + 1)$ and $(x_k + 1, y_k)$ from the actual line by d_{upper} and d_{lower}, respectively (see Fig. 9.2). At $x_k + 1$, the y-coordinate on the line is $y = m(x_k + 1) + b$, where m and b are the slope and y-intercept of the line segment, respectively. Therefore, we can determine d_{upper} and d_{lower} as,

$$\begin{aligned} d_{upper} &= (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b \\ d_{lower} &= y - y_k = m(x_k + 1) + b - y_k \end{aligned}$$

Based on these two quantities, we can take a simple decision about the pixel closer to the actual line, by taking the difference of the two.

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

If the difference is less than 0, the lower pixel is closer to the line and we choose it. Otherwise, we choose the upper pixel. Now, we substitute m in this expression with $\frac{\triangle y}{\triangle x}$, where $\triangle y$ and $\triangle x$ are the differences between the end points. Rearranging, we get,

$$\begin{aligned} \triangle x(d_{lower} - d_{upper}) &= \triangle x \left(2\frac{\triangle y}{\triangle x}(x_k + 1) - 2y_k + 2b - 1 \right) \\ &= 2 \triangle y.x_k - 2 \triangle x.y_k + 2 \triangle y + \triangle x(2b - 1) \\ &= 2 \triangle y.x_k - 2 \triangle x.y_k + c \end{aligned}$$

where the constant $c = 2 \triangle y + \triangle x(2b - 1)$

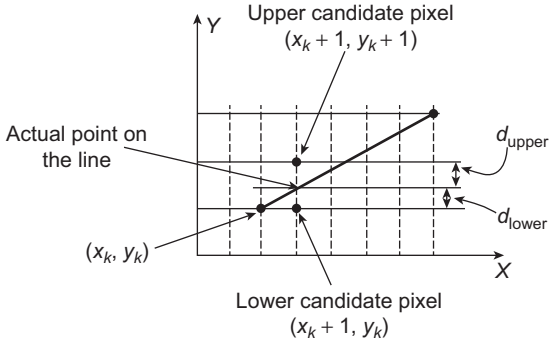


Fig. 9.2 The key idea of Bresenham's line scan conversion algorithm. The algorithm chooses one of the two candidate pixels based on the distance of the pixels from the actual line. The decision is taken entirely based on integer calculations.

Note that $\triangle x(d_{lower} - d_{upper})$ can also be used to make the decision about the closeness of the pixels to the actual line. Let us denote this by p_k, a *decision parameter* for the kth step. Clearly, the sign of the decision parameter will be the same as that of $(d_{lower} - d_{upper})$. Hence, if $p_k < 0$, the lower pixel is closer to the line and we choose it. Otherwise, we choose the upper pixel.

At step $k + 1$, the decision parameter is,

$$p_{k+1} = 2 \triangle y.x_{k+1} - 2 \triangle x.y_{k+1} + c$$

Subtracting from p_k, we get,

$$p_{k+1} - p_k = 2 \triangle y(x_{k+1} - x_k) - 2 \triangle x(y_{k+1} - y_k)$$

We know that $x_{k+1} = x_k + 1$. Substituting and rearranging, we get Eq. 9.1.

$$p_{k+1} = p_k + 2 \triangle y - 2 \triangle x(y_{k+1} - y_k) \tag{9.1}$$

Note that in Eq. 9.1, if $p_k < 0$, we set $y_{k+1} = y_k$, otherwise we set $y_{k+1} = y_k + 1$. Thus, depending on the sign of p_k, the difference $(y_{k+1} - y_k)$ in this expression becomes either 0 or 1. The first decision parameter at the starting point is given by $p_0 = 2 \triangle y - \triangle x$.

What is the implication of this? We are choosing pixels at each step, depending on the sign of the decision parameter. The decision parameter is computed entirely with integer operations only. All floating point operations are eliminated. Thus, the approach is a huge improvement, in terms of speed of computation, over the previous approaches we discussed. The pseudocode of the Bresenham's algorithm for line scan conversion is given in Algorithm 9.2.

Algorithm 9.2 **Bresenham's line drawing algorithm**

```
1:  Input: The two line end points (x1, y1 and x2, y2)
2:  Output: Set of pixels P to render the line segment
3:  Compute Δx = x2 − x1, Δy = y2 − y1, p = 2 Δ y − Δx
4:  Set x = x1, y = y1
5:  Add (x1, y1) to P
6:  repeat
7:    if p < 0 then
8:      Set x = x + 1
9:      Set p = p + 2 Δ y
10:   else
11:     Set x = x + 1, y = y + 1
12:     Set p = p + 2 Δ y − 2 Δ x
13:   end if
14:   Add (x, y) to P
15: until x < (x2 − 1)
16: Add (x2, y2) to P
```

Example 9.1

In order to understand Algorithm 9.2, let us execute its steps for the line segment defined by the end points A(2,2) and B(7,5) in our previous example. Following line 3 of Algorithm 9.2, we compute $\triangle x = 5$, $\triangle y = 3$, and $p = 1$. The two variables x and y are set to the end point A' as $x = 2$, $y = 2$ (line 4). Also, the end point A' (2,2) is added to P (line 5).

Note that $p = 1 \geq 0$. Therefore, we execute the ELSE part of the loop (lines 10–12) and we get $x = 3, y = 3, p = -3$. The pixel (3,3) is added to the output list P (line 14). Since $x = 3 < 6$ (the loop termination condition), loop is executed again.

In the second execution of the loop, we have $p = -3 < 0$. Thus, the IF part (lines 7–9) is executed and we get $x = 4$, $y = 3$ (no change), and $p = 3$. The pixel (4,3) is added to the output list P. Since $x = 4 < 6$, the loop is executed again.

Now we have $p = 3 \geq 0$. Therefore, in the third execution of the loop, the statements in the ELSE part are executed. We get $x = 5, y = 4, p = -1$. The pixel (5,4) is added to the output pixel list P. Since $x = 5 < 6$, the loop is executed again.

In the fourth loop execution, $p = -1 < 0$. Hence, the IF part is executed with the result $x = 6$, $y = 4$ (no change), and $p = 5$. The pixel (6,4) is added to the output list P. As the loop termination condition ($x < 6$) is no longer true ($x = 6$), the loop stops.

Finally, we add the other end point B'(7,5) to the output list P and the algorithm stops.

Thus, we get the output list $P = \{(2, 2), (3, 3), (4, 3), (5, 4), (6, 4), (7, 5)\}$ after the termination of the algorithm.

Algorithm 9.2 works for line segments with $0 \leq m \leq 1$ or $-1 \leq m \leq 0$. For other line segments, minor modification to the algorithm is needed, which is left as an exercise for the reader.

9.2 CIRCLE SCAN CONVERSION

Similar to lines, we can also use a very simple approach to scan convert circles. Let us consider a circle centered at origin. Assuming a radius r, the equation for the circle is $x^2 + y^2 = r^2$. Using this equation, we can solve for y after every unit increment of x as $y = \pm\sqrt{r^2 - x^2}$. Obviously, this solution is not good as it involves inefficient computations such as square root and multiplications (remember, r need not be integer). Also, we may need to round off computed values. Moreover, the pixels obtained may not generate a smooth circle just as in the case of lines (the gap between actual points on the circle and choosen pixels may be large). In the following section, we discuss a much more efficient algorithm, known as the *midpoint* algorithm, to scan convert circles. For simplicity, we shall restrict our discussion to circles about origin. We shall follow an approach similar to the previous section: first, we shall discuss the derivation of the steps of the algorithm; then, the pseudocode of the algorithm will be presented, followed by an illustrative example.

9.2.1 Midpoint Algorithm

A circle about origin has an important property that we shall exploit in the algorithm—the eight-way symmetry. We can divide a circle into eight quadrants, as shown in Fig. 9.3. If we can determine one point on any of these quadrants, seven other points on the circle belonging to the seven quadrants can be trivially derived, as illustrated in Fig. 9.3. In the algorithm, we shall compute pixels for the top right eighth quadrant (the marked quadrant of Fig. 9.3) and determine the pixels for other quadrants following the property.

Now assume that we have just determined the pixel (x_k, y_k), as shown in Fig. 9.4. The next pixel is a choice between the two pixels $(x_k + 1, y_k)$ and $(x_k + 1, y_k - 1)$ [note that here we are going down the scan lines for the top right eighth quadrant, unlike the line scan conversion where we were going up]. Clearly, the pixel that is closer to the actual circle should be chosen. How do we decide that?

We perform some mathematical tricks here as we did for Bresenham's line drawing algorithm. We first restate the circle equation as $f(x, y) = x^2 + y^2 - r^2$.

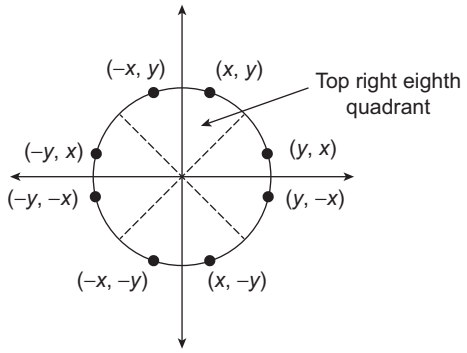


Fig. 9.3 The eight way symmetry and its use to determine points on a circle centered at origin. We compute pixels for the marked quadrant. The other pixels are determined based on the property.

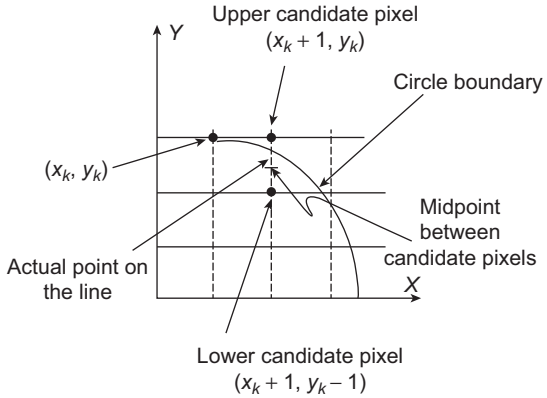


Fig. 9.4 Idea of candidate pixels and midpoint decision variable

The equation evaluates as follows.

$$f(x,y)\begin{cases} < 0 & \text{if the point } (x,y) \text{ is inside the circle} \\ = 0 & \text{if the point } (x,y) \text{ is on the circle} \\ > 0 & \text{if the point } (x,y) \text{ is outside the circle} \end{cases}$$

We evaluate this function at the *midpoint* of the two candidate pixels to make our decision (see Fig. 9.4). In other words, we compute the value $f(x_k + 1, y_k - \frac{1}{2})$. Let us call this the *decision variable* p_k after the kth step. Thus, we have

$$\begin{aligned} p_k &= f\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

Note that if $p_k < 0$, the midpoint is inside the circle. Thus, y_k is closer to the circle boundary and we choose the pixel $(x_k + 1, y_k)$. Otherwise, we choose $(x_k + 1, y_k - 1)$ as the midpoint is outside the circle and $y_k - 1$ is closer to the circle boundary.

To come up with an efficient algorithm, we perform some more tricks. First, we consider the decision variable for the $(k + 1)$th step p_{k+1} as,

$$\begin{aligned} p_{k+1} &= f\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \\ &= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

After expanding the terms and rearranging, we get Eq. 9.2.

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \tag{9.2}$$

In Eq. 9.2, y_{k+1} is y_k if $p_k < 0$. In such a case, we have $p_{k+1} = p_k + 2x_k + 3$. If $p_k > 0$, we have $y_{k+1} = y_k - 1$ and $p_{k+1} = p_k + 2(x_k - y_k) + 5$. Thus, we can choose pixels based on an incremental approach (computing the next decision parameter from the current value).

One thing remains, that is the *first* decision variable p_0. This is the decision variable at $(0, r)$. Using the definition of p_0, we can compute it as follows.

$$\begin{aligned} p_0 &= f\left(0 + 1, r - \frac{1}{2}\right) \\ &= 1 + \left(r - \frac{1}{2}\right)^2 - r^2 \\ &= \frac{5}{4} - r \end{aligned}$$

The pseudocode of the algorithm is shown in Algorithm 9.3, where RoundOff(a) rounds off the number a to its nearest integer.

With very simple modifications to Algorithm 9.3, we can determine pixels for circles about any arbitrary center. The modifications are left as an exercise for the reader.

Algorithm 9.3 **Midpoint circle drawing algorithm**

1: **Input:** The radius of the circle r
2: **Output:** Set of pixels P to render the line segment
3: Compute $p = \frac{5}{4} - r$
4: Set $x = 0, y = RoundOff(r)$
5: Add the four axis points $(0, y)$, $(y, 0)$, $(0, -y)$ and $(-y, 0)$ to P
6: **repeat**
7: **if** $p < 0$ **then**
8: Set $p = p + 2x + 3$
9: Set $x = x + 1$
10: **else**
11: Set $p = p + 2(x - y) + 5$
12: Set $x = x + 1, y = y - 1$
13: **end if**
14: Add (x, y) and the seven symmetric points $\{(y, x), (y, -x), (x, -y), (-x, -y), (-y, -x), (-y, x), (-x, y)\}$ to P
15: **until** $x \geq y$

Example 9.2

Let us consider a circle with radius $r = 2.7$. We will execute the steps of Algorithm 9.3 to see how the pixels are determined.

First, we compute $p = -1.05$ and set $x = 0, y = 3$ (lines 3–4). Also, we add the axis pixels $\{(0,3), (3,0), (0,-3), (-3,0)\}$ to the output pixel list P (line 5). Then, we enter the loop.

Note that $p = -1.45 < 0$. Hence, the IF part (lines 7–9) is executed and we get $p = 1.55$ and $x = 1$ (y remains unchanged). So, the pixels added to P are (line 14): $\{(1,3), (3,1), (3,-1), (1,-3), (-1,-3), (-3,-1), (-3,1), (-1,3)\}$. Since $x = 1 < y = 3$, the loop is executed again.

In the second run of the loop, we have $p = 1.55 > 0$. Hence, the ELSE part is now executed (lines 10–12) and we get $p = 2.55$, $x = 2$, and $y = 2$. Therefore, the pixels added to P are $\{(2,2), (2,2), (2,-2), (2,-2), (-2,-2), (-2,-2), (-2,2), (-2,2)\}$. Since now $x = 2 = y$, the algorithm terminates.

Thus, at the end, P consists of the 20 pixels $\{(0,3), (3,0), (0,-3), (-3,0), (1,3), (3,1), (3,-1), (1,-3), (-1,-3), (-3,-1), (-3,1), (-1,3), (2,2), (2,2), (2,-2), (2,-2), (-2,-2), (-2,-2), (-2,2), (-2,2)\}$.

Note that the output set P contains some duplicate entries. Before rendering, we perform further checks on P to remove such entries.

9.3 FILL AREA SCAN CONVERSION

What we discussed so far is concerned with the determination of pixels that define a line or a circle boundary. Sometimes, however, we may know the pixels that are part of a region and we want to apply a specific color to that region (i.e., color the pixels that are part

of the region). In other words, we want to *fill* the region with a specified color. For example, consider an interactive painting system. You draw an arbitrary shape and color it (both boundary and interior). Now, you want to change the color of the shape interactively (e.g., select a color from a menu and click in the interior of the shape to indicate that the new color be applied to the shape). There are many ways to perform such *region filling*. The techniques depend on how the regions are *defined*. There are broadly the following two types of definitions of a region.

Pixel level definition A region is defined in terms of its boundary pixels (known as *boundary-defined*) or the pixels within its boundary (called *interior defined*). Such definitions are used for regions having complex boundaries or in interactive painting systems.

Geometric definition A region is defined in terms of geometric primitives such as edges and vertices. Primarily meant for defining polygonal regions, such definitions are commonly used in general graphics packages.

In the following section, we shall discuss algorithms used to fill regions defined in either of the ways.

9.3.1 Seed Fill Algorithm

In the seed fill algorithm, we start with one interior pixel and color the region progressively. The algorithm works based on the boundary definition of a region (i.e., pixel level definition with the boundary pixels specified). It further assumes that we know atleast one interior pixel called the **seed** (which is easy to obtain from the boundary pixels). Moreover, it is assumed that an interior pixel is connected to either four (*four-connected*) or eight (*eight-connected*) of its neighbouring pixels. In the former case, the neighbours are: top, bottom, left, and right pixels. In the latter case, the neighbours are: top, top left, top right, left, right, bottom, bottom left, and bottom right pixels. The algorithm works as follows.

We maintain a stack for the algorithm. The seed is first pushed to the stack. While the stack is not empty, we *pop* the stack top pixel and color it. Then, for four-connected convention, we check each of the four connected pixels to the current pixels. If the connected pixel is a boundary pixel (i.e., having the boundary color) or already has the specified color, we ignore it. Otherwise, we push it into the stack. A similar step is done for eight-connected convention also, with the only difference that the eight neighbouring pixels are checked instead of four. The steps are shown in Algorithm 9.4.

9.3.2 Flood Fill Algorithm

In the flood fill algorithm, we assume an interior definition (i.e., interior pixels of a region are known). We want to recolor the region with a specified color. The algorithm is similar to the seed fill algorithm, with the difference that now we take decisions based on the original interior color of the current pixels instead of the boundary pixel color. Other things remain the same. The pseudocode of the procedure is shown in Algorithm 9.5.

Algorithm 9.4 **Seed fill algorithm**

```
1: Input: Boundary pixel color, specified color, and the seed (interior pixel) p
2: Output: Interior pixels with specified color
3: Push(p) to Stack
4: repeat
5:    Set current pixel = Pop(Stack)
6:    Apply specified color to the current pixel
7:    for Each of the four connected pixels (four-connected) or eight connected pixels (eight-connected)
      of current pixel do
8:       if (connected pixel color ≠ boundary color) OR (connected pixel color ≠ specified color) then
9:          Push(connected pixel)
10:      end if
11:   end for
12: until Stack is empty
```

Algorithm 9.5 **Flood fill algorithm**

```
1: Input: Interior pixel color, specified color, and the seed (interior pixel) p
2: Output: Interior pixels with specified color
3: Push(p) to Stack
4: repeat
5:    Set current pixel = Pop(Stack)
6:    Apply specified color to the current pixel
7:    for Each of the four connected pixels (four-connected) or eight connected pixels (eight-connected)
      of current pixel do
8:       if (Color(connected pixel) = interior color then
9:          Push(connected pixel)
10:      end if
11:   end for
12: until Stack is empty
```

9.3.3 Scan Line Polygon Fill Algorithm

Unlike the seed fill or flood fill algorithms, we assume that a polygonal region is defined in terms of its vertices and edges (i.e., a geometric definition) in the scan line polygon fill algorithm. We shall further assume that the vertices are rounded off to the nearest pixels. The pseudocode is shown in Algorithm 9.6.

Let us illustrate the idea with an example. Consider Fig. 9.5. The polygon is specified with the four vertices A, B, C, and D (note that we are following an anti-clockwise vertex naming convention). Therefore, the edges are AB, BC, CD, and DA. Let us execute the steps of the algorithm.

Algorithm 9.6 **Scan line polygon fill algorithm**

1: **Input:** Set of vertices of the polygon
2: **Output:** Interior pixels with specified color
3: From the vertices, determine the maximum and minimum scan lines (i.e., maximum and minimum y values) for the polygon.
4: Set scanline = minimum
5: **repeat**
6: **for** Each edge (pair of vertices (x_1, y_1) and (x_2, y_2)) of the polygon **do**
7: **if** $(y_1 \leq \text{scanline} \leq y_2)$ OR $(y_2 \leq \text{scanline} \leq y_1)$ **then**
8: Determine edge–scanline intersection point
9: **end if**.
10: **end for**
11: Sort the intersection points in increasing order of x coordinate
12: Apply specified color to the pixels that are within the intersection points
13: Set scanline = scanline + 1
14: **until** scanline = maximum

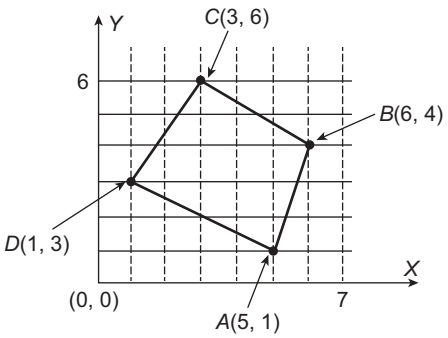


Fig. 9.5 Illustrative example of the scan line polygon fill algorithm

First, we determine the maximum and minimum scanlines (line 3) from the coordinate of the vertices as: maximum = max{1,4,6,3} (i.e., maximum of the vertex y-coordinates) = 6, minimum = min{1,4,6,3} (i.e., minimum of the vertex y-coordinates) = 1.

In the first iteration of the outer loop, we first determine the intersection points of the scan line $y = 1$ with all the four edges in the inner loop (lines 6–10). For the edge AB, the IF condition is satisfied and we determine the intersection point as the vertex A (lines 7–8). For BC and CD, the condition is not satisfied. However, for DA, again the condition is satisfied and we get the vertex A again. Thus, the two intersection points determined by the algorithm are the same vertex A. Since this is the only pixel between itself, we apply specified color to it (lines 11–12). Then we set scanline = 2 (line 11). Since $2 \neq maximum = 6$, we reenter the outer loop.

In the second iteration of the outer loop, we check for the intersection points between the edges and the scanline $y = 2$. For the edge AB, the IF condition is satisfied. So there is an intersection point, which is $(5\frac{1}{3}, 2)$. The edges BC and CD do not satisfy the condition, hence

there are no edge–scanline intersections. The condition is satisfied by the edge *DA* and the intersection point is (3,2). After sorting (line 11), we have the two intersection points (3,2) and $(5\frac{1}{3}, 2)$. The pixels in between them are (3,2), (4,2), and (5,2). We apply the specified color to these pixels (line 12) and set scanline = 3. Since $3 \neq maximum = 6$, we reenter the outer loop.

The algorithm works in a similar way for the the remaining scanlines $y = 3$, $y = 4$, $y = 5$, and $y = 6$ (the execution is left as an exercise for the reader). There are two things in the algorithm that require some elaboration. First, how do we determine the edge–scanline intersection point? Second, how do we determine pixels within two intersection points?

We can use a simple method to determine the edge–scanline intersection point. First, from the vertices, determine the line equation for an edge. For example, for the edge *AB* in Fig. 9.5, we compute $m = \frac{4-1}{6-4} = 3$. Thus, the line equation is $y = 3x + b$. Now, evaluating the equation at the end point *A* (i.e., $x = 5, y = 1$), we get $b = 1 - 3.5 = -14$. Therefore the equation for *AB* is $y = 3x - 14$. Now, to determine the edge–scanline intersection point, simply replace the scanline (y) value in the equation and compute the x-coordinate. Thus, to get the x-coordinate of the intersection point between the scanline $y = 2$ and *AB*, we evaluate $2 = 3x - 14$ or $x = 5\frac{1}{3}$.

Given two intersection points (x_1, y_1) and (x_2, y_2) where $x_1 < x_2$, determination of the pixels between them is easy. Increment x_1 by one to get the next pixel and continue till the current x value is less than x_2. If either or both the intersection points are pixels themselves, they are also included. As an illustration, consider the two intersection points (3,2) and $(5\frac{1}{3}, 2)$ of the polygon edges (*AB* and *DA*, respectively) with the scanline $y = 2$ in Fig. 9.5. Here $x_1 = 3, x_2 = 5\frac{1}{3}$. The first pixel is the intersection point (3,2) itself. The next pixel is $(3 + 1, 2)$ or (4,2). We continue to get the next pixel as $(4 + 1, 2)$ or (5,2). Since $5 + 1 = 6 > x_2 = 5\frac{1}{3}$, we stop.

An important point to note here is that Algorithm 9.6 works for convex polygons only. For concave polygons, an additional problem needs to be solved. As we discussed before, we determine pixels between the pair of edge–scanline intersection points. However, all these pixels may not be *inside* the polygon in case of a concave polygon, as illustrated in Fig. 9.6. Therefore, in addition to determining pixels, we also need to determine which pixels are *inside*.

In order to make Algorithm 9.6 work for concave polygons, we have to perform an *inside–outside* test for each pixel between a pair of edge–scanline intersection points (an additional overhead). The following are the steps for a simple inside–outside test for a pixel p.

1. Determine the bounding box (maximum and minimum x and y coordinates) for the polygon.
2. Choose an arbitrary pixel p_o outside the bounding box (This is easy. Simply choose a point whose x and y coordinates are outside the minimum and maximum range of the polygon coordinates).
3. Create a line by joining p and p_o (i.e., determine the line equation).
4. If the line intersects the polygon edges an *even* number of times, p is an outside pixel. Otherwise, p is inside the polygon.

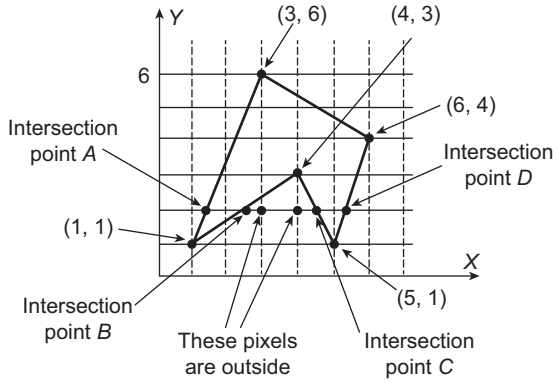


Fig. 9.6 The problem with concave polygons—two pixels (3,2) and (4,2), which are inside the pair of intersection points *B* and *C*, are not *inside* the polygon

9.4 CHARACTER RENDERING

An important issue in scan conversion is the way alphanumeric and non-alphanumeric characters are rendered. These are the building blocks of any textual content displayed on the screen. Efficient rendering of characters is necessary since we often need to display a large amount of text in a short time span. For example, consider scrolling up/down a text document. With each scroll action, a whole new set of characters needs to be displayed on the screen quickly.

When we deal with characters, the term *font* or *typeface* is used to denote the overall design style of the characters. For example, we have fonts such as Times New Roman, Courier, Arial, and so on. Each of these fonts can be rendered with varying appearance such as **bold**, *italic*, or both ***bold and italic***. The size of a character on screen is denoted by *point* (e.g., 10-point, 12-point), which is a measure of the character height in inches. Although the term is borrowed from typography (like the other terms), we do not use the original point measure as used in typography. Instead, we use the definition that 'a point equals to $\frac{1}{72}$ of an inch or ≈ 0.0139 inch'. This is also known as the DTP (desk top publishing) or postscript point.

There are broadly two ways to render characters—bitmapped fonts and outlined fonts. In *bitmapped font*, a pixel grid definition is maintained for each character. In the grid, those pixels that are part of the character are marked as *on pixels* and the others are marked as *off pixels*. The idea is illustrated in Fig. 9.7. In contrast, a character is defined in terms of geometric primitives such as points and lines in the *outlined* definition of font. Before rendering, the characters are scan converted (i.e., pixels are determined using scan conversion methods such as those we discussed for points, lines, and circles). The idea is illustrated in Fig. 9.8.

Clearly, bitmapped fonts are simple to define and fast to render (since we do not need to *compute* pixels). However, they require large storage and do not look good after resizing or reshaping. Also, the size of the bitmapped font depends on screen resolution. For example, a 12-pixel high bitmap will produce a 12-point character in a 72 pixels/inch resolution. However, the same bitmap will produce 9-point character in a 96 pixels/inch resolution.

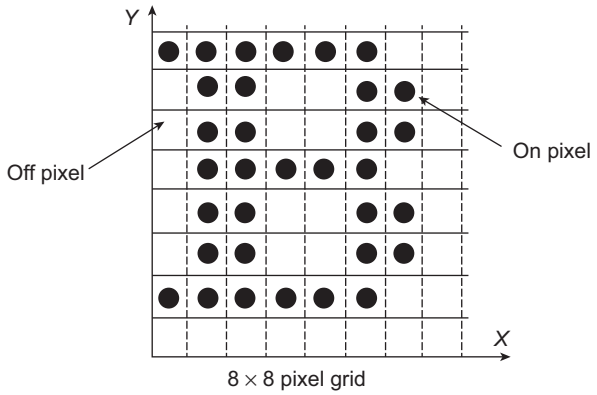


Fig. 9.7 Bitmap font definition for the character *B*

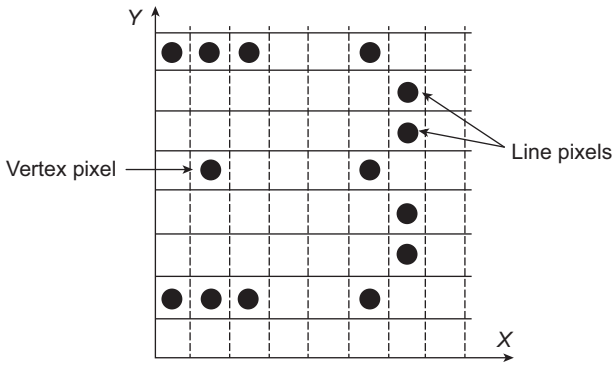


Fig. 9.8 The same character *B* of Fig. 9.7 is defined in terms of vertices and edges, as outline definition. The intermediate pixels are computed using scan conversion procedure during rendering.

Outline fonts, on the other hand, require less storage and we can perform geometric transformations with satisfactory effect to reshape or resize such fonts. Moreover, they are not resolution-dependent. However, rendering is slow since we have to perform scan conversion procedures before rendering.

9.5 ANTI-ALIASING

Let us consider Fig. 9.9. This is basically a modified form of Fig. 9.1, where we have seen the pixels computed to render the line. As shown in Fig. 9.9, the scan converted line (shown as a continuous line) does not look exactly like the original (shown as a dotted line). Instead, we see a stair-step like pattern, often called the *jaggies*. This implies that, after scan conversion, some distortion may occur in the original shape. Such distortions are called *aliasing* (we shall discuss in the next section why it is called so). Some additional operations are performed to remove such distortions, which are known as *anti-aliasing* techniques.

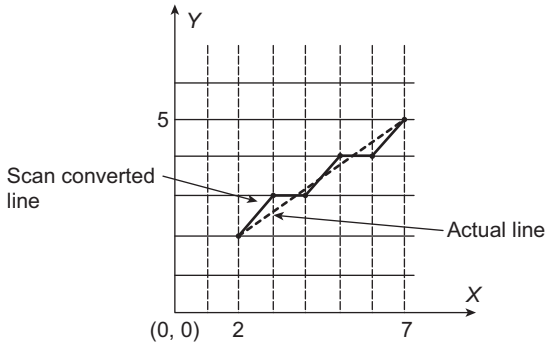


Fig. 9.9 Problem of aliasing—Note the difference between actual line (dotted) and scan converted line

9.5.1 Aliasing and Signal Processing

Aliasing is usually explained in terms of concepts borrowed from the field of signal processing. We know that in computer graphics, we are concerned about synthesizing images. Think of it as a problem of rendering a *true* image (on window in view coordinate system) to device. The image is defined in terms of intensity values, which can be any real number. Therefore, the intensity of a true image represents a *distribution* of *continuous* values. In other words, a true image can be viewed as a *continuous signal*. The rendering process then can be viewed as a two stage process: *sampling* of the continuous signal (i.e., computing pixel intensities) and then *reconstructing* the original signal as the set of colored pixels on the display. When we reconstruct an original signal, the reconstructed signal is clearly a *false representation* of the original. In English, when a person uses a false name, that is known as an *alias*, and so it was adapted in signal analysis to apply to falsely represented signals. As we have already seen, aliasing usually results in visually distracting artifacts. Additional efforts go into trying to reduce or eliminate its effect, through techniques that we call *anti-aliasing*.

A continuous intensity signal can be viewed as a composition of various frequency components (i.e., primary signal of varied frequencies). The uniform regions of constant intensity values correspond to the low frequency components, whereas values that change abruptly and correspond to sharp edges are at the high end of the frequency spectrum. Clearly, such abrupt changes in the intensity signal result in aliasing effects, which we need to smoothen out. In other words, we need to remove (filter) high frequency components from the (reconstructed) intensity signal. Consequently, we have the following two broad groups of anti-aliasing techniques.

Pre-filtering It works on the true signal in the continuous space to derive proper values for individual pixels. In other words, it is *filtering before sampling*. There are various pre-filtering techniques, often known as *area sampling*.

Post-filtering In post-filtering, we try to filter high frequency components of the signal from the sampled data (i.e., modify computed pixel values). In other words, it

is filtering after sampling. The post-filtering techniques are often known as *super sampling*.

Let us now get some idea about the working of these two types of filtering techniques.

9.5.2 Pre-filtering or Area Sampling

In pre-filtering techniques, we assume that a pixel has an *area* (usually square or circular with unit radius), rather than being a dimensionless point. Lines passing through those pixels have some finite *width*. In other words, each line has some area. In order to compute pixel intensity, we first determine the percentage p of pixel area occupied by the line. Assume the original line color (either preset or computed from the stages of the graphics pipeline) is c_l and the background color is c_b. Then, we set pixel intensity $I = p.c_l + (1 - p)c_b$. The idea is illustrated in Fig. 9.10.

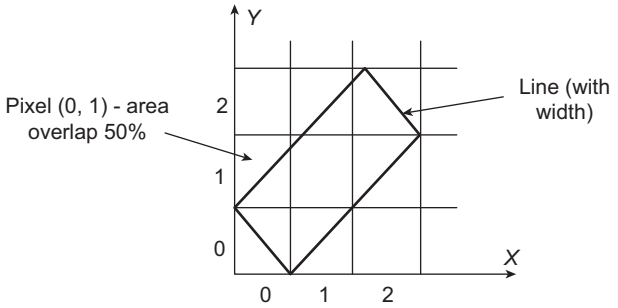


Fig. 9.10 Illustration of area sampling technique. Each square represents a pixel area. Depending on the area overlap of the line with the pixels, pixel intensities are set.

9.5.3 Gupta–Sproull Algorithm

The Gupta–Sproull algorithm is a pre-filtering technique for anti-aliased line drawing. In this algorithm, the intensity of a pixel is set based on the *distance* of a line center from the pixel center. The algorithm is based on the midpoint line drawing algorithm. Let us first understand this algorithm.

Consider Fig. 9.11. Let us assume that we have just determined the pixel (x_k, y_k). There are two candidates for the next pixel: $E(x_k + 1, y_k)$ and $NE\ (x_k + 1, y_k + 1)$. This is similar to the Bresenham's algorithm. However, instead of the decesion parameter based on the distance of the line from the candidate pixels, here we consider the midpoint $M(x_k + 1, y_k + \frac{1}{2})$ between the candidate pixels. We know that a line can be represented as $F(x, y) = \mathrm{a}x + \mathrm{b}y + \mathrm{c}$, where a, b, and c are integer constants. We can restate the same as $F(x, y) = 2(\mathrm{a}x + \mathrm{b}y + \mathrm{c})$ without changing the nature of the equation. We require this matematical manipulation to avoid some floating point operations. We set our decision variable as,

$$\begin{aligned} d_k &= F(M) \\ &= F\left(x_k + 1, y_k + \frac{1}{2}\right) \\ &= 2(\mathrm{a}(x_k + 1) + \mathrm{b}\left(y_k + \frac{1}{2}\right) + \mathrm{c}) \end{aligned}$$

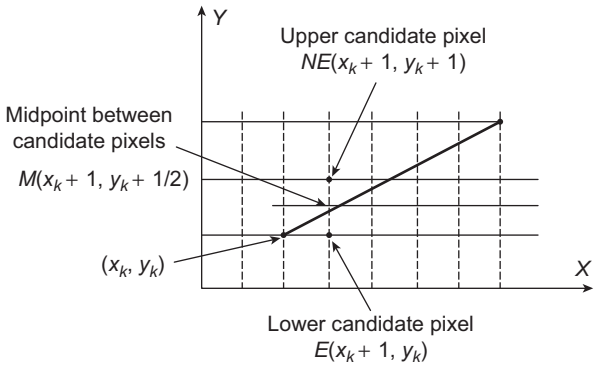


Fig. 9.11 Midpoint line drawing algorithm. The decision variable is based on the midpoint between the candidate pixels.

If $d > 0$, the midpoint is *below* the line. Thus, the pixel NE is closer to the line and we choose it. In such a case, the next decision variable is,

$$\begin{aligned} d_{k+1} &= F\left((x_k+1)+1, (y_k+1)+\frac{1}{2}\right) \\ &= 2\left(\mathrm{a}((x_k+1)+1)+\mathrm{b}\left((y_k+1)+\frac{1}{2}\right)+\mathrm{c}\right) \\ &= \left[2(\mathrm{a}(x_k+1)+\mathrm{b}\left(y_k+\frac{1}{2}\right)+\mathrm{c}\right]+2(\mathrm{a}+\mathrm{b}) \\ &= d_k + 2(\mathrm{a}+\mathrm{b}) \end{aligned}$$

Similarly, if $d_k \leq 0$, we choose the next pixel as E. In that case, we have

$$d_{k+1} = F\left((x_k+1)+1, \left(y_k+\frac{1}{2}\right)\right)$$

Expanding and rearranging as before, we get $d_{k+1} = d_k + 2a$ (check for yourself). The initial decision variable is defined as $d_0 = F(x_0+1, y_0+\frac{1}{2})$. Expanding, we get

$$\begin{aligned} d_0 &= 2\left(\mathrm{a}(x_0+1)+\mathrm{b}\left(y_0+\frac{1}{2}\right)+\mathrm{c}\right) \\ &= 2(\mathrm{a}x_0+\mathrm{b}y_0+\mathrm{c})+2\left(\mathrm{a}+\frac{1}{2}\mathrm{b}\right) \\ &= F(x_0,y_0)+2\mathrm{a}+\mathrm{b} \\ &= 2\mathrm{a}+\mathrm{b} \text{ since } F(x_0,y_0)=0. \end{aligned}$$

Algorithm 9.7 shows the pseudocode of this algorithm.

In the Gupta–Sproull algorithm, the basic midpoint algorithm is modified a little. Consider Fig. 9.12. Suppose the current pixel is (x_k, y_k). Based on the midpoint, we have chosen pixel E in the next step. D is the perpendicular distance of E from the line. Using analytical

Algorithm 9.7 **Midpoint line drawing algorithm**

1: **Input:** The two line end points (x_1, y_1 and x_2, y_2)
2: **Output:** Set of pixels P to render the line segment
3: Determine the line constants a, b, and c from the end points
4: Determine the initial decision value $d = 2\text{a} + \text{b}$
5: Set $x = x_1, y = y_1$
6: Add (x_1, y_1) to P
7: **repeat**
8: **if** $d > 0$ **then**
9: Set $x = x + 1, y = y + 1$
10: Set $d = d + 2(\text{a} + \text{b})$
11: **else**
12: Set $x = x + 1$
13: Set $d = d + 2\text{a}$
14: **end if**
15: Add (x, y) to P
16: **until** $x < x_2 - 1$
17: Add (x_2, y_2) to P

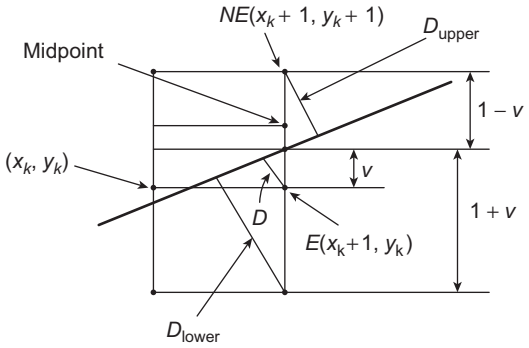


Fig. 9.12 Illustration of distance calculation in the Gupta–Sproull algorithm

geometry, we can determine D as (left as an exercise for the reader),

$$D = \frac{d + \triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}} \tag{9.3}$$

In the expression, d is the midpoint decision variable and $\triangle x$ and $\triangle y$ are the differences in x and y coordinate values of the line endpoints, respectively. Note that the denominator is a constant.

The intensity of E will be a *fraction* of the original line color. The fraction is determined based on D. This is unlike Algorithm 9.7 where the line color is simply assigned to E. In order to determine the *fraction*, a *cone filter function* is used. In other words, the more the distance of the line from the chosen pixel center, the lesser will be the intensity. The function

is implemented in the form of a table. In the table, each entry represents the fraction with respect to a given D.

In order to increase the line smoothness, the intensity of the two vertical neighbours of E, namely the points $(x_k + 1, y_k + 1)$ and $(x_k + 1, y_k - 1)$ are also set in a similar way according to their distances D_{upper} and D_{lower} respectively from the line. We can analytically derive the two distances as (derivation is left as an exercise),

$$D_{upper} = 2\frac{(1-v)\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}} \tag{9.4a}$$

$$D_{lower} = 2\frac{(1+v)\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}} \tag{9.4b}$$

If instead of E, we have chosen the NE pixel, the corresponding expressions would be (derivation is left as an exercise for the reader),

$$D = \frac{d - \triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}} \tag{9.5a}$$

$$D_{upper} = 2\frac{(1-v)\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}} \tag{9.5b}$$

$$D_{lower} = 2\frac{(1+v)\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}} \tag{9.5c}$$

Note that in Eq. 9.5(b), D_{upper} is the perpendicular distance of the pixel $(x_k + 1, y_k + 2)$ and D_{lower} denotes the distance of the pixel E $(x_k + 1, y_k)$ from the line.

Thus in the Gupta–Sproull algorithm (Algorithm 9.8), we perform the following additional steps in each iteration of the midpoint line drawing algorithm.

Algorithm 9.8 **Gupta–Sproull algorithm**

1: **if** The lower candidate pixel E is chosen **then**
2: Compute $D_E = \frac{d+\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}}$
3: **else**
4: Compute $D_{NE} = \frac{d-\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}}$
5: **end if**
6: Update d as in the regular midpoint algorithm
7: Set intensity of the current pixel (E or NE) according to D, determined from the table (the higher the value, the lower the intensity)
8: Compute $D_{upper} = 2\frac{(1-v)\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}}$
9: Compute $D_{lower} = 2\frac{(1+v)\triangle x}{2\sqrt{\triangle x^2 + \triangle y^2}}$
10: Set the intensity of the two vertical neighbours of the current pixels according to D_{upper} and D_{lower}, determined from the table (the higher the value, the lower the intensity)

Example 9.3

Let us understand the working of the Gupta–Sproull algorithm in terms of an example. Consider the line segment shown in Fig. 9.13 between the two end points $A(1,1)$ and $B(4,3)$. Our objective is to determine the following two things.

1. The pixels that should be colored to render the line.
2. The intensity values to be applied to the chosen pixels (and its vertical neighbours) to reduce aliasing effect.

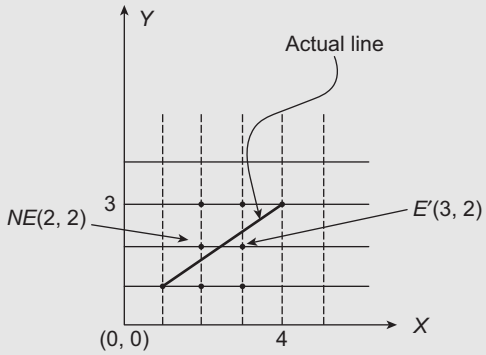


Fig. 9.13 Gupta–Sproull algorithm example

Let us first determine the pixels to be choosen to render the line following the midpoint algorithm (Algorithm 9.7). From the line end points, we can derive the line equation as $2x - 3y + 1 = 0$ (see Appendix A for derivation of line equation from two end points). Thus, we have a $= 2$, b $= -3$, and c $= 1$. Hence the initial decision value is: $d = 1$ (lines 3–4, Algorithm 9.7).

In the first iteration of the algorithm, we need to choose between the two pixels: the upper candidate pixel *NE* (2,2) and the lower candidate pixel *E* (2,1) (see Fig. 9.11). Since $d > 0$, we choose the *NE* pixel (2,2) and reset $d = -1$ (lines 8–10, Algorithm 9.7). In the next iteration, the two possibilities are: the upper candidate pixel NE' (3,3) and the lower candidate pixel E' (3,2). Since now $d < 0$, we choose E' (3,2) as the next pixel to render and reset $d = 3$ (lines 11–13, Algorithm 9.7). However, since now $x = 3$, the looping condition check fails (line 16, Algorithm 9.7). The algorithm stops and returns the set of pixels {(1,1), (2,2), (3,2), (4,3)}. These are the pixels to be rendered.

Next, we determine the intensity values for the choosen pixels and its two vertical neighbours according to the Gupta–Sproull algorithm (Algorithm 9.8). We know that $\triangle x = 4 - 1 = 3$ and $\triangle y = 3 - 1 = 2$. Let us start with the first intermediate pixel. Note that the first intermediate pixel chosen is the upper candidate pixel *NE* (2,2). For this pixel, we have $d = 1$. Therefore, we compute the perpendicular distance D from the line as,

$D_{NE} = \frac{-1}{\sqrt{13}}$ (line 4, Algorithm 9.8)

Next, we have to compute the distances of the vertical neighbours of the choosen pixels D_{upper} and D_{lower}. The line equation is $2x - 3y + 1 = 0$. At the chosen pixel position, $x = 2$. Putting this value in the line equation, we get $y = \frac{5}{3}$. Therefore, $v = \frac{5}{3} - 1 = \frac{2}{3}$ (see Fig. 9.12). Hence,

$D_{upper} = \frac{1}{\sqrt{13}}$ (line 8, Algorithm 9.8)

$D_{lower} = \frac{1}{\sqrt{13}}$ (line 9, Algorithm 9.8)

We perform a table lookup to determine the fraction of the original line color to be applied to the three pixels based on the three computed distances.

The next chosen point is the lower candidate pixel E' (3,2). For this pixel, we have $d = -1$. Hence, the perpendicular distance of the pixel from the line is computed as,

$D_E = \frac{1}{\sqrt{13}}$

As in the previous case, we compute D_{upper} and D_{lower} for the two vertical neighbours of E'. For E', $x = 3$. We put this value in the line equation $2x - 3y + 1 = 0$ to obtain $y = \frac{7}{3}$. Therefore, $v = \frac{7}{3} - 2 = \frac{1}{3}$ (see Fig. 9.12). Hence,

$D_{upper} = \frac{2}{\sqrt{13}}$

$D_{lower} = \frac{4}{\sqrt{13}}$

As before, we perform the table lookup to determine the fraction of the line color to be assigned to these three pixels based on the three distances.

9.5.4 Super Sampling

In super sampling, each pixel is assumed to consist of a grid of sub-pixels (i.e., we effectively increase the display resolution). In order to draw an anti-aliased line, we count the number of sub-pixels through which the line passes in each pixel. This number is then used to determine pixel intensity. The idea is illustrated in Fig. 9.14.

Another approach is to use a finite line width. In that case, we determine which sub-pixels are *inside* the line (a simple check for this is to consider sub-pixels, whose lower left corners are inside the line, to be *inside*). Then, pixel intensity is determined as the *weighted average* of the sub-pixel intensities, where the weights are the fraction of sub-pixels that are inside or

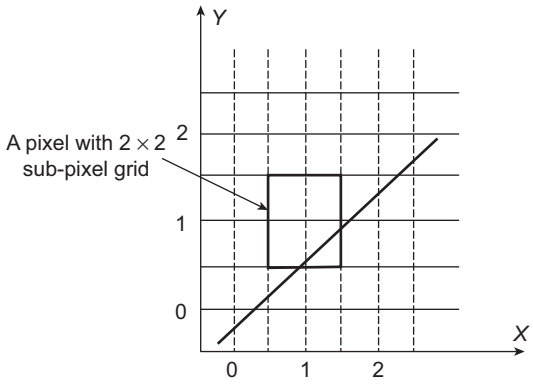


Fig. 9.14 The idea of super sampling with a 2 × 2 sub-pixel grid for each pixel. The pixel intensity is determined based on the number of sub-pixels through which the line passes. For example, in the pixel (0,0), the line passes through 3 sub-pixels. However, in (1,0), only one sub-pixel is part of the line. Thus, intensity of (0,0) will be more than (1,0)

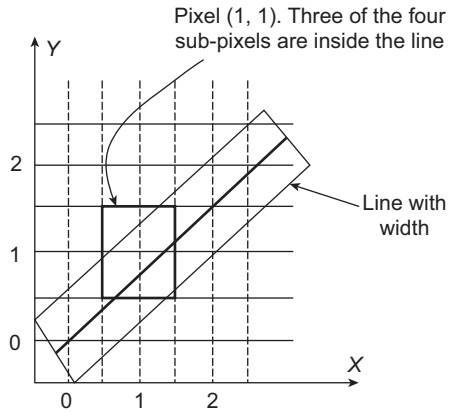


Fig. 9.15 The idea of super sampling for lines with finite width

outside of the line. For example, consider Fig. 9.15 where each pixel is divided into a 2×2 sub-pixel grid.

Assume that the original line color is red ($R = 1$, $G = 0$, $B = 0$) and background is light yellow ($R = 0.5$, $G = 0.5$, $B = 0$). Note that three sub-pixels (top right, bottom left, and bottom right) are inside the line in pixel (1,1). Therefore, the fraction of sub-pixels that are inside is $\frac{3}{4}$ and the outside sub-pixel fraction is $\frac{1}{4}$. The weighted average for individual intensity components (i.e., R, G, and B) for the pixel (1,1) therefore are,

$$Average_R = 1 \times \frac{3}{4} + 0.5 \times \frac{1}{4} = \frac{7}{8}$$
$$Average_G = 0 \times \frac{3}{4} + 0.5 \times \frac{1}{4} = \frac{1}{8}$$
$$Average_B = 0 \times \frac{3}{4} + 0 \times \frac{1}{4} = 0$$

Thus, the intensity of (1,1) will be set as ($R = \frac{7}{8}$, $G = \frac{1}{8}$, $B = 0$).

Sometimes, we use *weighting masks* to control the amount of contribution of various sub-pixels to the overall intensity of the pixel. The size of the mask depends on the sub-pixel grid size. For example, for a 3×3 sub-pixel grid, we shall have a 3×3 mask. How do we determine pixel intensity from a given mask? Let us consider an example.

Assume we have the following 3×3 mask.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Note that the intensity contribution of a sub-pixel is its corresponding mask value divided by 16 (sum of all the values). For example, the contribution of the center sub-pixel is $\frac{4}{16}$. Now suppose a line passes through (or encloses) the sub-pixels top, center, bottom left, and bottom of a pixel (x, y). Thus, if the line intensity is $c_l(r_l, g_l, b_l)$ and the background color

is $c_b(r_b, g_b, b_b)$, then the pixel intensity can be computed as: Intensity = (total contribution of sub-pixels) × line color + (1 − total contribution of sub-pixels) × background color for each of the R, G, and B color components as we have done before.

SUMMARY

In this chapter, we learnt about the last stage of the 3D graphics pipeline, namely the rendering of objects on the screen (also known as scan conversion or rasterization). We discussed rendering for geometric primitives such as lines and circles. In line rendering, we started with the simple and intuitive algorithm and saw its inefficiency in terms of the floating point operations it requires. Only some and not all these operations can be eliminated in the DDA algorithm, which thus offer little improvement. Bresenham's algorithm is the most efficient as it renders a line using integer operations only. The midpoint circle rendering algorithm similarly increases the efficieny by performing mostly integer operations. However, unlike the Bresenham's line drawing, some floating point operations are still required in midpoint circle drawing.

An issue in interactive graphics is to render a fill area (i.e. an enclosed region). We discussed the two ways to define a fill area, namely the pixel-level definition and geometric definition. Depending on the definition, we discussed various fill area rendering algorithms such as seed fill, flood fill, and scanline polygon fill. The first two rely on pixel-level definitions while the third algorithm assumes a geometric definition of fill area.

A frequent activity in computer graphics is to display characters. We discussed both the bitmap and outlined character rendering techniques along with their pros and cons.

Finally, the problem of distortion in original shapes (known as aliasing) that arises during the rendering process is discussed, along with the various techniques (called anti-aliasing) to overcome it. Following an explanation of the origin of the term *aliasing* with the signal processing concepts, we mentioned and briefly discussed the two broad groups of anti-aliasing techniques: pre-filtering and post-filtering. The pre-filtering or area sampling is discussed including the Gupta–Sproull algorithm. We also learnt about the idea of various post-filtering or super sampling techniques with illustrative examples.

BIBLIOGRAPHIC NOTE

Bresenham [1965] and Bresenham [1977] contain the original idea on the Bresenham's algorithm. More on the midpoint methods can be found in Kappel [1985]. Fill area scan conversion techniques are discussed in Fishkin and Barsky [1984]. Crow [1981], Turkowski [1982], Fujimoto and Iwata [1983], Korien and Badler [1983], Kirk and Arvo [1991], and Wu [1991] can be referenced for further study on anti-aliasing techniques. The *Computer Gems* book series ((Glassner [1990], Arvo [1991], Kirk [1992], Heckbert [1994] and Paeth [1995]) contain additional discussion on all these topics.

KEY TERMS

Aliasing – the distortions that may occur to an object due to scan conversion
Anti-aliasing – techniques to eliminate/reduce the aliasing effects
Bitmapped fonts – a character representation scheme in which each character is represented in terms of on and off pixels in a pixel grid
Boundary defined – defining a fill area in terms of its boundary pixels

Bresenham's algorithm – a more efficient line scan conversion algorithm that works based on integer operations only
DDA algorithm – a line scan conversion algorithm
Decision parameter – a parameter used in the Bresenham's algorithm
Eight-connected – an interior pixel is connected to eight of its neighbours
Flood fill algorithm – a fill area scan conversion algorithm that works with the interior defined regions
Font/Typeface – overall design style of a character
Four-connected – an interior pixel is connected to four of its neighbours
Geometric definition – defining a fill area in terms of geometric primitives such as edges and vertices
Gupta-Sproull algorithm – a pre-filtering anti-aliasing technique for lines
Interior defined – defining a fill area in terms of its interior pixels
Midpoint algorithm – an algorithm for circle scan conversion
Outlined font – a character representation scheme in which each character is represented in terms of some geometric primitives such as points and lines
Pixel level definition – defining a fill area in terms of constituent pixels
Point (of font) – size of a character
Post-filtering/Super sampling – anti-aliasing techniques that work on the pixels to modify their intensities
Pre-filtering/Area sampling – anti-aliasing techniques that work on the actual signal and derive appropriate pixel intensities
Scan conversion/Rasterization/Rendering – the process of mapping points from continuous device space to discrete pixel grid
Scan line polygon fill algorithm – a fill area scan conversion algorithm that works with geometric definition of fill regions
Seed – an interior pixel inside a fill area
Seed fill algorithm – a fill area scan conversion algorithm that works with the boundary defined regions
Sub pixel – a unit of (conceptual) division of a pixel for super sampling techniques

EXERCISES

9.1 Discuss the role played by the rendering techniques in the context of the 3D graphics pipeline.
9.2 Derive the incremental approach of the Bresenham's line drawing algorithm. Algorithm 9.2 works for lines with slope $0 \leq m \leq 1$ or $-1 \leq m \leq 0$. Modify the algorithm for slopes that are outside this range.
9.3 The midpoint line drawing algorithm is shown in Algorithm 9.7. Is there any difference between Algorithms 9.2 and 9.7? Discuss with respect to a suitable example.
9.4 Derive the incremental computation on which the midpoint circle algorithm is based. Explain the importance of the eight-way symmetry in circle drawing algorithms.
9.5 Algorithm 9.3 works for circles having the origin as center. Modify the algorithm so as to make it work for circles with any arbitrary center.
9.6 Explain the different definitions of an enclosed region with illustrative examples. Calculate the pixels for the scanlines $y = 3$, $y = 4$, $y = 5$, and $y = 6$ in the example mentioned in Section 9.3.3.
9.7 Algorithm 9.6 works for convex polygons only. Modify the algorithm, by incorporating the steps for the simple inside-outside test, so that the algorithm works for concave polygons also.

9.8 Discuss the advantanges and disadvantages of the bitmapped and the outlined font rendering methods. Suppose we have a $20'' \times 10''$ display with resolution 720×360. What would be the bitmap size (in pixels) to produce a 12-point font on this display?

9.9 Explain the term *aliasing*. Why it is called so? How is the concept of *filtering* related to *anti-aliasing*?

9.10 Discuss the basic idea of area sampling with an illustrative example.

9.11 Derive the expressions of Eqs. 9.3, 9.4, and 9.5 using analytical geometry. Modify Algorithm 9.7 to include the Gupta–Sproull anti-aliasing algorithm.

9.12 Explain the basic idea of super sampling. Discuss, with illustrative examples other than the ones mentioned in the text, the three super sampling techniques we learnt, namely (a) super sampling for lines without width, (b) super sampling for lines with finite width, and (c) super sampling with weighting masks. Do you think the use of masks offers any advantage over the non-mask-based methods? Discuss.

CHAPTER 10 Graphics Hardware and Software

Learning Objectives

After going through this chapter, the students will be able to

- Review the generic architecture of a graphics system
- Get an overview of the input and output devices of a graphics system
- Understand the basics of the flat panel displays including the plasma panels, thin-film electroluminescent displays, light-emitting diode (LED) displays, and liquid crystal displays (LCDs)
- Get an overview of the common hardcopy output devices-printers and plotters
- Know about the widely used input devices including keyboards, mouse, trackballs, spaceballs, joysticks, data gloves, and touch screen device.
- Learn the fundamentals of the graphics processing unit (GPU)
- Get an overview of shaders and shader programming
- Know about graphics software and software standards
- Learn the basics of OpenGL, a widely used open source graphics library

INTRODUCTION

We are now in a position to understand the fundamental *process* involved in depicting an image on a computer screen. In very simple terms, the process is as follows: we start with the abstract representation of the objects in the image, using points (vertices), lines (edges), and other such geometric primitives in the 3D world coordinate; the pipeline stages are then applied to convert the abstract representation to a sequence of *bits* (i.e., a sequence of 0's and 1's); the sequence is stored in the *frame buffer* and used by the video controller to *activate* appropriate pixels on the screen, so that we *perceive* the image. So far, we have discussed only the theoretical aspects of this process; how it works conceptually without elaborating on the implemetation issues. Although we touched upon the topic of displaying images on a CRT screen Chapter 1, it was very brief. In this chapter, we shall learn in more detail the implementation aspect of the fundamental process. More specifically, we shall learn about the overall architecture of a graphics system, the technology of few important display devices, introductory concepts on the graphics processing unit (GPU), and how the rendering process (the 3D pipeline) is actually implemented on the hardware. The chapter

will also introduce the basics of OpenGL, an open source graphics library widely used to write computer graphics programs.

10.1 GENERIC ARCHITECTURE

Let us begin by reviewing the generic architecture of a graphics system we learnt in Chapter 1 (Refer to Fig. 1.7). Recall that there are five major hardware components apart from the *computer* (which basically represents the *system memory*, CPU, and the *system bus* together), which are specific to any graphics system: the *display controller*, the *video controller*, the *frame buffer* (part of the *video memory*), the *input devices*, and the *display screen*.

Earlier, we understood the working of the system in terms of broad concepts. However, after learning the pipeline stages, let us now try to understand the relationship between these hardware components and the pipeline stages. Assume that we have written a program to display two objects (a ball and a cube) on the screen. Once the CPU detects that the process involves graphics operations, it transfers control to the display controller (thus, freeing itself up for other activities). The controller contains its own processing unit (GPU). The pipeline stages (geometric transformations, illumination, projection, clipping, hidden surface removal, and scan conversion operations) are then performed on the object definitions

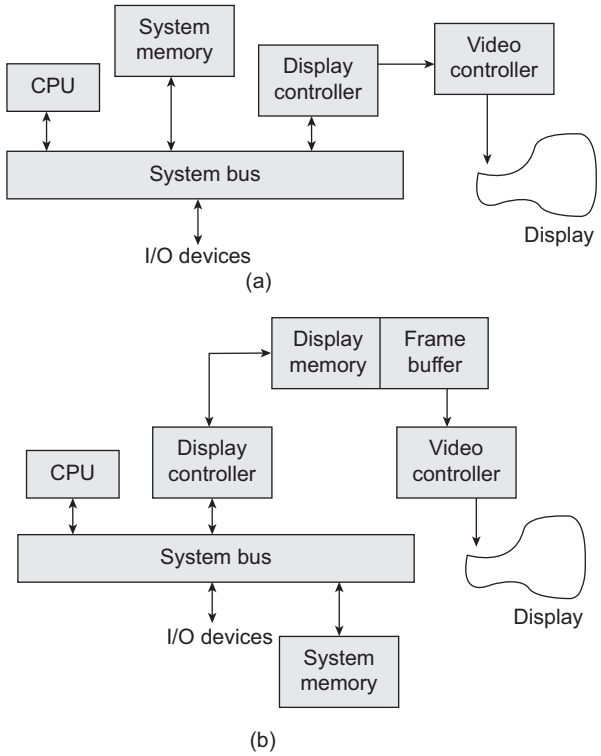


Fig. 10.1 Different ways to integrate memory in the generic graphics system (a) No separate graphics memory is present and the system memory is used in shared mode by both the CPU and the graphics controller (b) Controller has its own dedicated memory

(as specialized instructions executed in the GPUs) to convert them to the sequence of bits. The bit sequence gets stored in the frame buffer. In the case of interactive systems, the frame buffer content may be changed depending on the input coming from the input devices such as a mouse.

The video controller acts based on the frame buffer content (the bit sequence). The job of the video controller is to *map* the value represented by the bit(s) in each frame buffer location to the *activation* of the *corresponding pixel* on the display screen. For example, in the case of CRT devices, such activation refers to the *excitation* (by an appropriate amount) of the corresponding phosphor dots on the screen. Note that the amount of excitation is determined by the intensity of the electron beam, which in turn is determined by the voltage applied on the electron gun, which in turn is determined by the frame buffer value.

The frame buffer is only a part of the video memory required to perform graphics operations. Along with the frame buffer, we also require memory to store object definitions and instructions for graphics operations (i.e., to store code and data as in any other program). The memory can be integrated in the generic architecture either as *shared* system memory (shared by CPU and GPU) or *dedicated* graphics memory (part of graphics controller organization). The two possibilities are illustrated in Fig. 10.1. Note that when the memory is shared, the execution will be slower since the data transmission takes place through the common system bus.

10.2 INPUT AND OUTPUT OF GRAPHICS SYSTEM

Whenever we talk of a graphics system, the primary output device that comes to our mind is a video monitor. Sometimes, instead of monitors, outputs are *projected* using projectors. As we know, both can be present together in a graphics system. In addition, the system may have a third mode of output: hardcopy output through printers or plotters. Head-mounted displays are also another way of displaying output to a viewer.

10.2.1 Video Monitors

Till very recently, the ubiquitous video monitor was the cathode ray tube or CRT monitor. We have already seen in Chapter 1 the basic idea behind the working of a CRT. Of late, the *flat panels* have started replacing the CRTs. Such screens are much thinner and lighter than CRTs. As a result, they are useful for both non-portable and portable systems. Consequently, we can see them almost everywhere *around* us, in desktops, laptops, palmtops, calculators, advertising boards, pocket video-game console, wrist-watch, and so on.

Flat Panel Displays

Let us have a look-at the technology behind flat panel displays. The very first thing we should remember is that *flat panel* is a generic term indicating a display monitor having a (much) reduced volume, weight, and power consumption compared to a CRT. Technology-wise, they are of two types—*emissive displays* and *non-emissive displays*. Emissive displays (or *emitters*) are those that convert electrical energy into light on the screen. Prominent examples of such devices are the plasma panels, thin-film electroluminescent displays, and light-emitting diodes (LEDs). In the case of non-emissive displays (or *non-emitters*), no

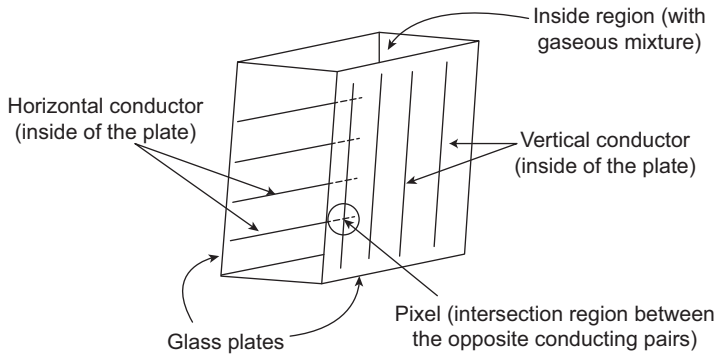


Fig. 10.2 Schematic illustration of the basic plasma panel design

electrical-to-optical energy conversion takes place. Instead, such devices convert light (either natural light or light from some other sources) to a graphics pattern on the screen through some optical effects. A widely used non-emissive display is the liquid crystal display (LCD).

Plasma Panels

The schematic of a plasma panel is shown in Fig. 10.2. As the figure illustrates, there are two glass plates placed parallelly. The region between the plates is filled with a mixture of gases (xeon, neon, and helium). The inner walls of each glass plate contains a set of parallel conductors (very thin and shaped like ribbons). One plate has a set of vertical conductors while the other contains a set of horizontal conductors. The region between each corresponding pair of conductors on the two plates (e.g., two consecutive horizontal and opposite vertical conductors) form a *pixel*. The screen side wall of the pixel is coated with phosphors like in CRT (three phosphors corresponding to RGB for color displays). With the application of appropriate *firing* voltage, the gas in the pixel cell breaks down into electrons and ions. The ions rush towards the electrodes and collide with the phosphor coating emitting lights. Separation between pixels is achieved by the electric fields of the conductors.

LED Displays

Thin-film electroluminescent displays are similar in construction to plasma panels; they also have mutually perpendicular sets of conducting ribbons on the inside walls of two parallel glass plates. However, instead of gases, the region between the glass plates is filled with a phosphor, such as zinc sulphide doped with manganese. The phosphor becomes a conductor at the point of intersection when a sufficiently high voltage is applied to a pair of crossing electrotodes. The manganese atom absorbes the electrical energy and releases photons, generating the perception of a glowing spot or pixel on the screen. Such displays, however, require more power than plasma panels. Also, good color displays with this technology is difficult to achieve.

Light emitting diode or LED displays are another type of emissive devices, which are becoming popular nowadays. In such devices, each pixel position is represented by an LED. Thus, the whole display is a grid of LEDs corresponding to the pixel grid. Based on the frame buffer information, suitable voltage is applied to each diode to make it emit appropriate amount of light.

Liquid Crystal Displays

The most well-known non-emissive display is the liquid crystal display or LCD. In an LCD, there are two parallel glass plates as before. Each contains a light polarizer that is aligned in a perpendicular way to the other (i.e., one plate contains the vertical polarizer while the other contains the horizontal polarizer). Rows of horizontal, transparent conductors are placed on the (inside) surface of one plate (having the vertical polarizer) while columns of vertical, transparent conductors are placed on the other. A *liquid crystal* material is put in between the plates. The term *liquid crystal* refers to materials having crystalline molecular arrangement, though they flow like liquids. The material used in flat-panel LCDs typically contain *nematic* or thread-like crytalline molecules. The rod-shaped molecules tend to align along

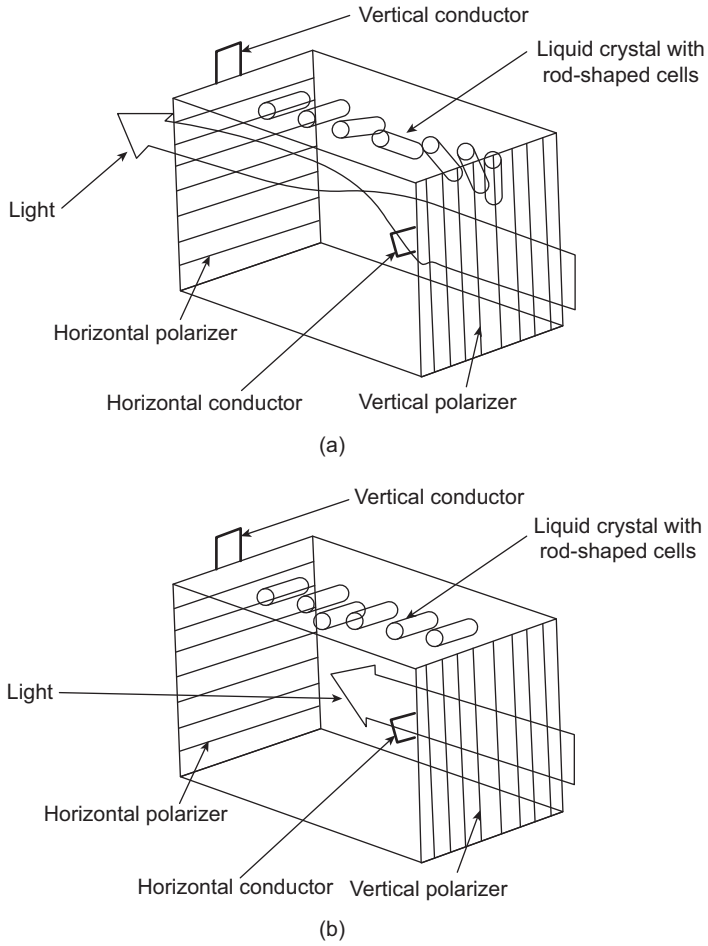


Fig. 10.3 The working of a *transmissive* LCD. A light source at the back sends light through the polarizer. The polarized light gets twisted and passes through the opposite polarizer to the viewer in the *active* pixel state, shown in (a). (b) shows the molecular arrangement of the liquid crystal at a pixel position, after voltage is applied to the perpendicular pair of conductors on the opposite glass plates. The arrangement prevents light from passing between polarizers, indicating *deactivation* of the pixel.

their long axes. The intersection points of each pair of mutually perpendicular conductors define pixel positions. When a pixel position is *active*, the molecules are aligned as shown in Fig. 10.3(a). In a *reflective* display, external light enters through one polarizer and gets polarized. The molecular arrangement of the liquid crystal ensures that the polarized light gets twisted so that it can pass through the opposite polarizer. Behind the polarizer, a reflective surface is present and the light is reflected back to the viewer. In a *transmissive* display, a light source is present on the back side of the screen. Light from the source gets polarized after passing through the back side polarizer, twisted by the liquid crystal molecules and passes through the screen-side polarizer to the viewer. In order to *de-activate* the pixel, a voltage is applied to the intersecting pair of conductors. This leads to the molecules in the pixel region (between the conductors) getting arranged as shown in Fig. 10.3(b). This new arrangement prevents the polarized light to get twisted and pass through the opposite polarizer. The technology, both reflective and trasmissive, described here is known as *passive-matrix* LCD. Another method for constructing LCDs is to place thin-film transistors at each pixel location to have more control on the voltage at those locations. The transistors also help prevent charges to be leaking out gradually to the liquid crystal cells. Such types of LCDs are called *active matrix* LCDs.

10.2.2 Printers and Plotters

The primary means of hardcopy output, that is the printers, can be of two types—*impact* and *non-impact* printers. In impact printers, pre-formed character faces are present. These are pressed against an inked ribbon on the paper. An example of impact printers is a line printer, in which the typefaces are mounted on bands, chains, drums, or wheels. In line printers, the whole line gets printed at a time. Character printers, on the other hand, print one character at a time. One widely used character print device, till very recent times, was the *dot-matrix* printer. As the name suggests, the print head contained a rectangular array (i.e., matrix) of portruding wire pins (the dots). The number of pins in the matrix determines the print quality: higher the number, better the quality. Each of these pins can be retracted *inwards*. During printing of a character or graphics pattern, some pins are retracted. The remaining pins then press against the ribbon on the paper, printing the character or pattern.

Non-impact printers and plotters use laser techniques, ink-jet sprays, electrostatic, and electrothermal methods to get images onto paper. In a laser device, a laser beam is applied on a rotating drum. The drum is coated with photo-electric material such as selenium. Consequently, a charge distribution is created on the drum. The toner is then applied to the drum, which gets transferred to the paper. In *ink-jet* printers, an electrically charged ink stream is sprayed in horizontal rows across a paper wrapped around a drum. Using electrical fields that deflect the charged ink stream, dot-matrix patterns of ink are created on the paper. An electrostatic device places a negative charge on the paper (at selected dot positions determined by the frame buffer values), one row at a time. The paper is then exposed to a positively charged toner, which gets attracted to the negatively charged areas, producing the desired output. Another printing technique is the electrothermal method. In it, heat is applied to a dot-matrix print head (on selected pins). The print head is then used to put patterns on a heat-sensitive paper.

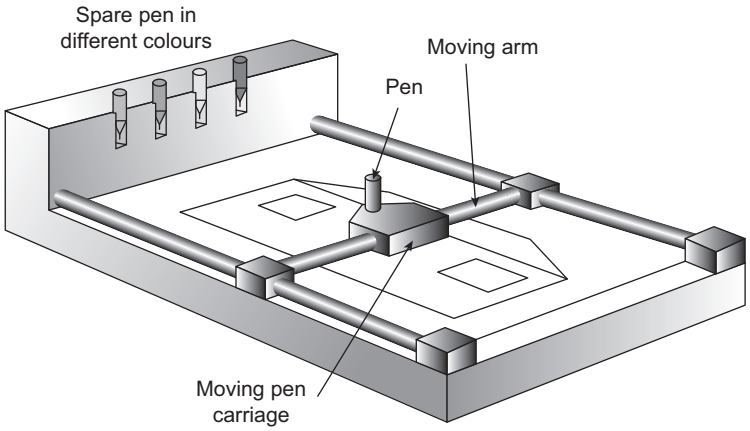


Fig. 10.4 Flatbed pen plotter device

In order to get colored printouts, impact printers use different-colored ribbons. However, the range of color produced and the quality are usually limted. Non-impact printers, on the other hand, are good at producing color images. In such devices, color is produced by combining the three color pigments (cyan, magenta, and yellow) (see the discussion on the CMY model in Chapter 5). Laser and electrostatic devices deposit the three pigments on *separate* passes; the three colors are shot together on a single pass along each line in ink-jet printers.

Plotters are another class of hardcopy graphics output devices, which are typically used to generate drafting layouts and other drawings. In a pen plotter (see Fig. 10.4), one/more pens are mounted on a carriage, or crossbar, that spans a sheet of paper. The paper can lie flat or rolled onto a drum or belt and held in place with clamps, a vaccum or an electrostatic charge. In order to generate different shading and line styles, pens with varying colors and widths are used. Pen-holding crossbars can either move or remain stationary. In the latter case, the pens themselves move back and forth along the bar. Instead of pen, ink-jet technology is also used to design plotters.

10.2.3 Input Devices

Graphics systems usually provide data input facilities for the user, through which users can manipulate the screen image. The ubiquitous keyboards and mouse, found with any computer, are two examples of such graphics input devices. There are a variety of other devices also, some of which are designed specifically for interactive computer graphics systems. Most popular of these devices are discussed in this section.

Keyboards We are all familiar with the alphanumeric keyboards (physical or virtual) that comes with most graphics systems. The alphanumeric keys in a keyboard are used for text and command inputs. In addition, keyboards typically contain some *function* and cursor-control keys also. With the function keys, frequently performed operations can be selected with a single key. Cursor-control keys allow the user to set the screen cursor position or select menu items.

Mouse A mouse is primarily used to position the cursor on the screen, select on-screen items, and perform a host of menu operations. Wheels or rollers at the bottom of the mouse record the amount and direction of mouse movement which is converted to screen-cursor movement. Sometimes, instead of a roller, optical sensing techniques are used to determine the amount and direction of mouse movement. There are between one to three buttons present on a mouse, although the two-button mouse is more common. Along with the buttons, it may be equipped with a *wheel* to perform positioning operations more conveniently. A mouse may be attached to the main computer with wires or it may be a wireless mouse.

Trackballs and spaceballs Similar to a mouse, trackball devices (Fig. 10.5) are used to position screen-cursors. A trackball device contains a ball. When the ball is rotated by a finger/palm/hand, a screen-cursor movement takes place. A potentiometer is connected to the ball and measures the amount and direction of the ball rotation, which is then mapped to the screen-cursor movement. While the term *trackball* typically denotes devices used to control cursor in a 2D-space (screen), cursor-control in 3D-space is done through *spaceballs*. A spaceball provides six degrees of freedom. However, in a spaceball, there are no actual ball movements. Instead, the ball is pushed and pulled in various directions, which is mapped to cursor positioning in 3D space.

Joysticks Another positioning device is the joystick (Fig. 10.6). It contains a small, vertical lever (called *stick*) attached to a base. The stick can be moved in various directions to move the screen-cursor. The amount and direction of cursor movement is determined by the

Fig. 10.5 Trackball device

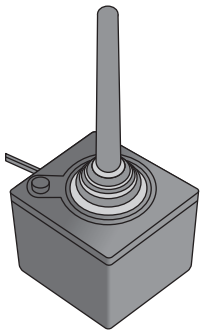


Fig. 10.6 Illustrative example of a joystick device

amount and direction of stick movement from its center position (measured with a potentiometer mounted at the base). In *isometric joysticks*, however, no movable sticks are present. Instead, the sticks are *pushed* or *pulled* to move on-screen cursor.

Data gloves Commonly used in virtual reality systems, data gloves are devices that allow a user to position and manipulate *virtual objects* in a more natural way—through hand or finger gestures. It is a glove-like device, containing sensors. These sensors can detect the finger and hand movements of the glove-wearer and map the movement to actions such as *grasping a virtual object*, *moving a virtual object*, *rotating a virtual object*, and so on. The position and orientation information of the finger or hand are obtained from electromagnetic coupling of transmitter and receiver antennas. Each of the transmitting and receiving antennas are constructed as a set of mutually perpendicular coils, thereby creating a 3D cartesian reference frame.

Touch screens Touch input systems are the preferred mode of input in most consumer-grade graphics systems nowadays. As the name suggests, on-screen elements are selected and manipulated through the touch of fingers or stylus (a special pen-like device) in touch screen systems. Touch input can be recorded using electrical, optical, or accoustic methods. In *optical* touch screens, an array of infrared LEDs are placed along one vertical and one horizontal edge. Light detectors are placed along the opposite horizontal and vertical edges. If a position on the screen is touched, lights coming from the vertical and horizontal LEDs get interrupted and recorded by the detectors, giving the touch location. In an *electrical* touch screen, there are two transparent plates separated by a small distance. One plate is coated with conducting material while the other is coated with resistive material. When the outer plate is touched, it comes into contact with the inner plate. This creates a voltage drop across the resistive plate, which is converted to the coordinate value. Less common are the *acoustic* touch screens devices, in which high-frequency sound waves are generated in horizontal and vertical directions across a glass plate. Touching the screen results in reflecting part of the waves (from vertical and horizontal directions) back to the emitters. The touch position is computed from the time interval between transmission of each wave and its reflection back to the emitter.

Apart from these, many more techniques are used to input data to a graphics system. These include image scanners (to store drawings or photographs in a computer), digitizers (used primarily for drawing, painting, or selecting positions), light pens (pencil shaped device used primarily for selecting screen positions), and voice-based input system (using speech-recognition technology).

10.3 GPU AND SHADER PROGRAMMING

A characteristic of graphics operations is that they are *highly parallel* in nature. For example, consider the modeling transformation stage of the pipeline. In this stage, we need to apply transformations (e.g., rotation) to the vertices. A transformation, as we have seen, is nothing but the multiplication of the transformation matrix with the vertex vector. The same vector-matrix multiplication is required to be performed for all the vertices which are part of the scene that we want to transform. Instead of going in for *serial* multipliclication of one

matrix-vector pair at a time, if we can apply the operation on all vectors *at the same time*, there will be significant gain in performance. The gain becomes critical in real-time rendering, where millions of vertices need to be processed per second. CPUs, owing to their design, cannot take advantage of this inherent parallelism in graphics operations. As a result, almost all graphics systems nowadays come with a separate graphics card containing its own processing unit and memory elements. The processing unit is known as the graphics processing unit or GPU.

10.3.1 Graphics Processing Unit

A GPU is a *multicore* system: it contains a large number of *cores* or unit processing elements. Each of these cores is a *stream* processor—it works on data streams. The cores are capable of performing simple integer and floating point arithmetic operations only. Multiple cores are grouped together to form *streaming multiprocessors* (SM). To understand the working of SMs, let us revisit our previous example of geometric transformation of vertices. Note that the instruction (multiplication) is same; the data (vertex vectors) varies. Consequently, what we have is known as single instruction multiple data (SIMD). The idea is illustrated in Fig. 10.7. Each SM is designed to perform SIMD operations. Organization of GPU cores and their interconnection to various storage elements (local and global) are schematically depicted in Fig. 10.8.

Let us try to understand how the 3D graphics pipeline is implemented with the GPU. Most real-time graphics systems assume that everything in a scene is made of triangles.

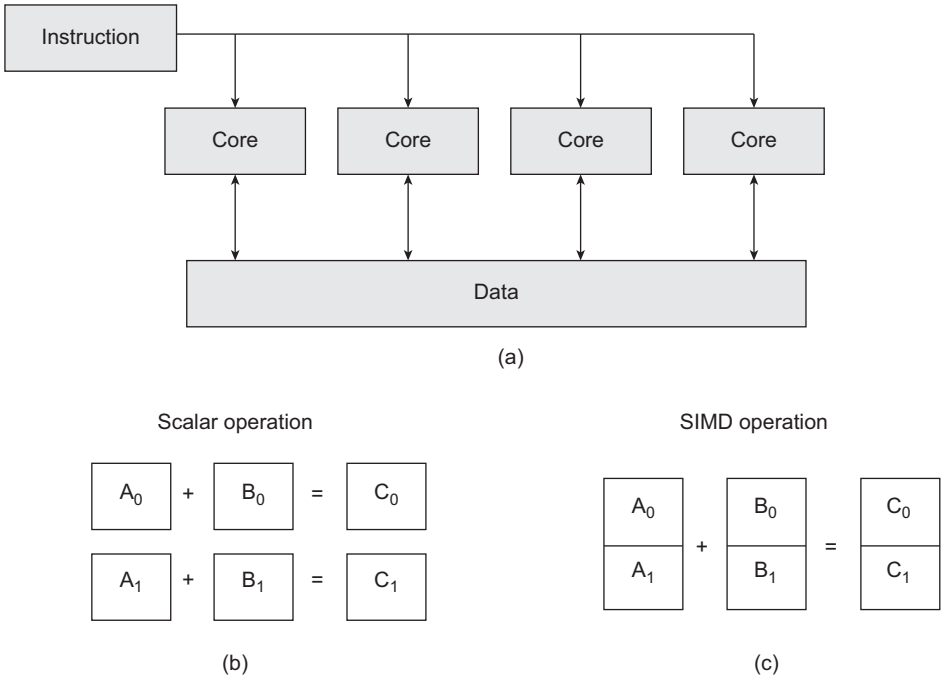


Fig. 10.7 Single instruction multiple data (SIMD) (a) The idea (b) Serial additions performed on inputs (c) Same output obtained with a single addition applied on data streams

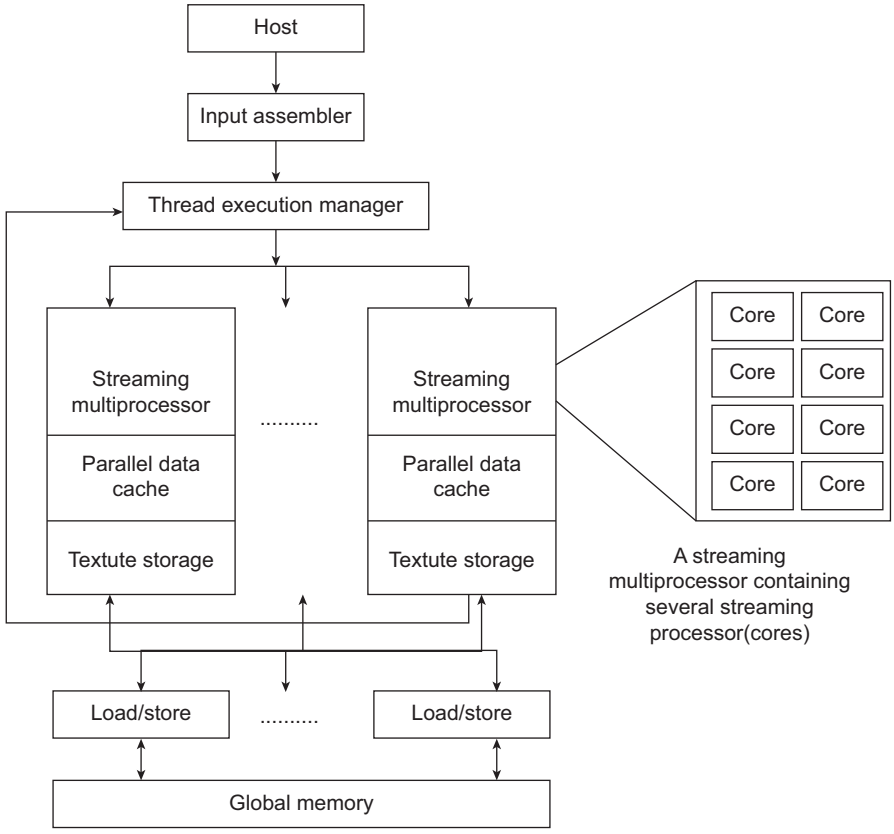


Fig. 10.8 An illustrative GPU organization. Each core is a very simple processing unit capable of performing simple floating point and integer arithmetic operations only.

Surfaces that are *not* expressed in terms of triangles, such as quadrilaterals or curved surface patches (see Chapter 2), are converted to triangular meshes. Through the APIs supported in a computer graphics library, such as OpenGL or Direct3D, the triangles are sent to the GPU one vertex at a time. The GPU assembles vertices into triangles as needed.

The vertices are expressed with homogeneous coordinates (see Chapter 3). The objects they define are represented in local or modeling coordinate system. After the vertices have been sent to the GPU, it performs modeling transformations on these vertices. The transformation (single or composite), as you may recall, is acheived with a *single* matrix-vector multiplication: the matrix represents the transformation while the vector represents the vertex. The multicore GPU architecture can be used to perform multiple such operations *simultaneously*. In other words, multiple vertices can be simultaneously transformed. The output of this stage is a stream of triangles, all represented in a common (world) coordinate system in which the viewer is located at the origin and the direction of view is aligned with the z-axis.

In the third stage, the GPU computes the color of each vertex based on the light defined for the scene. Recall the structure of the simple lighting equation we discussed in Chapter 4. The color of any vertex can be computed by evaluating vector dot products and a series of

add and multiply operations. In a GPU, we can perform these operations simultaneously for multiple vertices.

In the next stage, each colored 3D vertex is projected onto the view plane. Similar to the modeling transformations, the GPU does this using matrix-vector multiplication (see Chapter 6 for the maths involved), again leveraging efficient vector operations in hardware. The output after this stage is a stream of triangles in screen or device coordinates, ready to be converted to pixels.

Each device space triangle, obtained in the previous stage, overlaps some pixels on the screen. In the rasterization stage, these pixels are determined. GPU designers over the years have incorporated many rasterization algorithms, such as those we discussed in Chapter 9. All these algorithms exploit one crucial observation: each pixel can be treated independently from all other pixels. This leads to the possibility of handling all pixels in parallel. Thus, given the device space triangles, we can determine the color of the pixels for all pixels simultaneously.

During the pixel processing stage, two more activities take place—surface *texturing* and *hidden surface removal*. In the simplest surface texturing method, texture images are draped over the geometry to give the illusion of detail (see Chapter 5). In other words, the pixel color is replaced or modified by the texture color. GPUs store the textures in high-speed memory, which each pixel calculation must access. Since this access is very regular in nature (nearby pixels tend to access nearby texture image locations), specialized memory caches are used to reduce memory access time. For hidden surface removal, GPUs implement the depth(Z)-buffer algorithm. All modern-day GPUs contain a depth-buffer as a dedicated region of its memory, which stores the distance of the viewer from each pixel. Before writing to the display, the GPU compares a pixel's distance with the distance of the pixel that is already present. The display memory is updated only if the new pixel is closer (see the depth-buffer algorithm in Chapter 8 for more details).

10.3.2 Shaders and Shader Programming

Note that in the previous discussion, we covered all the pipelines stages in two broad group of activities—vertex (or geometry) processing and pixel processing. During its early years of evolution, GPUs used to come with *fixed-function* hardware pipeline, that is all the stages are pre-programmed and embedded into the hardware. In other words, a GPU contained dedicated components for specific tasks. The idea is illustrated in Fig. 10.9. However, in order to leverage the power of GPUs in a better way, modern day GPUs are designed to be *programmable*. Fixed-function units for transforming vertices and texturing pixels have been replaced by unified grid of processors, known as *shaders*. All these processing units can be used to do the calculations for *any* pipeline stage, as illustrated in Fig. 10.10.

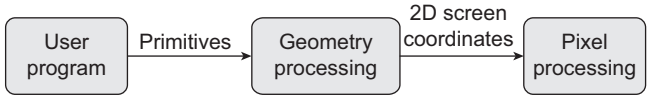


Fig. 10.9 Schematic of a fixed-function GPU stages—the user has no control on how it should work and what processing unit performs which stage of the pipeline

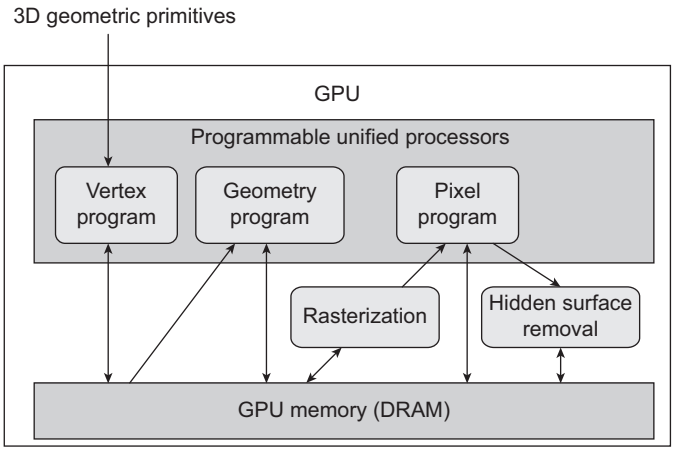


Fig. 10.10 The idea of programmable GPU. The GPU elements (processing units and memory) can be reused through user programs.

With a programable GPU, it is possible for the programers to modify how the hardware processes the vertices and shades pixels. They can do so by writing *vertex shaders* and *fragment shaders* (also known as *vertex programs* and *fragment programs*). This is known as *shader programming* (also known by many other names that include *GPU programming* and *graphics hardware programming*). As the names suggest, vertex shaders are used to process vertices (i.e., geometry)—modeling transformations, lighting, and projection to screen coordinates. Fragment shaders are programs that perform the computations in the pixel processing stage and determine how each pixel is shaded (rendering), how texture is applied (texture mapping), and if a pixel should be drawn or not (hidden surface removal). The term *fragment shader* is used to denote the fact that a GPU at any instant can process a subset (or fragment) of all the screen pixel positions. These shader programs are small pieces of codes that are sent to the graphics hardware from the user programs, but they are executed on the graphics hardware. The ability to program GPUs gave rise to the idea of a *general purpose GPU* or GPGPU; we can use the GPU to perform tasks that are not related to graphics at all.

10.4 GRAPHICS SOFTWARE AND OPENGL

The software used for computer graphics are broadly of two types—special-purpose packages and general programming packages. Special-purpose packages are designed for the non-programmers. These are complete software systems with their own graphical user interface. An example of such a package is a painting system, where an artist can select objects of various shapes, color them, place them at the desired screen position, change their size, shape, and orientation, and many more just by interacting with the interface. No knowledge of the graphics pipeline is required. Other examples include various CAD (computer-aided design) packages used in architectural, medical, business, and engineering domains. A general programming package, in contrast, provides a library of graphics functions that are to be used in a programming language such as C, C++, or Java. The graphics functions are

Graphics software standards

When programs are written with graphics functions, they may be moved from one hardware platform to another. Without some standards (i.e., a commonly agreed syntax), this is not possible and we need to rewrite the whole program. In order to avoid such problems and ensure portability, efforts were made to standardize computer graphics software. The first graphics software standard was developed in 1984, known as the *graphics kernel system* or GKS. It was adopted by the ISO (International Standards Organization) and many other national standards bodies. A second standard was developed by extending GKS, known as the PHIGS (*programmer's hierarchical interactive graphics standard*), and adopted by standards organizations worldwide.

While GKS and PHIGS were being developed, the Silicon Graphics Inc. (SGI) started to ship their graphics workstations with a set of routines called graphics library (GL). The GL became very popular and eventually evolved as the OpenGL (in the early 1990s), a de-facto graphics standard. It is now maintained by the OpenGL Architecture Review Board, a consortium of representatives from many graphics companies and organizations.

designed to perform various tasks that are part of the graphics pipeline such as object definition, modeling transformation, color assignment, projection, and display. Examples of such libraries include OpenGL (Open source Graphics Library), VRML (Virtual-Reality Modeling Language), and Java 3D. The functions in a graphics library are also known as the *computer graphics application programming interface* (CG API) since the library provides a software interface between a programming language and the hardware. So when we write an application program in C, the graphics library functions allow us to construct and display a picture on an output device.

Graphics functions in any package are typically defined independent of any programming language. A *language binding* is then defined for a particular high-level programming language. This binding gives the syntax for accessing various graphics functions from that language. Each language binding is designed to make the best use of the capabilities of the particular language and to handle various syntax issues such as data types, parameter passing, and errors. The specifications for language bindings are set by the International Standards Organization. In the following, we learn the basic idea of a graphics library with an introduction to OpenGL, a widely-used open source graphics library, with its C/C++ binding.

10.4.1 OpenGL: An Introduction

Consider Fig. 10.11, showing the program for displaying a straight line on the screen written using the OpenGL library functions in C. As you can see, there are several functions in the program. Let us try to understand each line of the code.

GLUT Library

The first thing we do is to include the header file containing the graphics library functions. Thus, the very first line in our program is

```
#include<GL/glut.h>
```

```
#include<GL/glut.h>
void init (void){
  glClearColor (1.0, 1.0, 1.0, 0.0);
  glMatrixMode (GL_PROJECTION)
  gluOrhto2D (0.0, 800.0, 0.0, 600.0)
}
void createLine (void){
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (0.0, 1.0, 0.0);
  glBegin (GL_LINES);
     glVertex2i (200, 100);
     glVertex2i (20, 50);
  glEnd ();
  glFlush ();
}
void main (int argc, char** argv){
  glutInit (& argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowPosition (0, 0);
  glutInitWindowSize (800, 600);
  glutCreateWindow ("The OpenGL example");
  init ();
  glutDisplayFunc (createLine);
  glutMainLoop ();
}
```

Fig. 10.11 Example OpenGL program

The OpenGL core library does not provide support for input and output, as the library functions are designed to be device-independent. However, we have to show the line on the display screen.Thus, auxiliary libraries are required for the output, on top of the core library, which is provided in the GLUT or OpenGL Utility Toolkit library. GLUT provides a library of functions for interacting with *any* screen-windowing system. In other words, the functions in the GLUT library allow us to set up a *display window* on our video screen (a rectangular area on the screen showing the picture, in this case the line). The library functions are prefixed with *glut*. Since GLUT functions provide interface to other device-specific window systems, we can use GLUT to write device independent programs. Note that GLUT is suitable for graphics operations only. We may require to include other C/C++ header files such as <stdio.h> or <stdlib.h> along with GLUT.

Managing the Display Window: The `main()` Function

Inclusion of GLUT allows us to create and manage the display window, that is, the region on the screen where we see the line. The first thing required is to initialize GLUT. This we do with the following statement.

```
glutInit (& argc, argv);
```

After initialization, we can set various options for the display window, using the *glutInitDisplayMode* function. This function takes symbolic GLUT constants as arguments.

For example, the following line of code written for the example program specifies that a single refresh buffer is to be used for the display window and the RGB color mode is to be used for selecting color values. Note the syntax used to represent symbolic GLUT constants: the prefix GLUT is added followed by an underscore ('_') to each constant name and is written in capital letters. The two constants are combined using a logical OR operation.

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
```

Although GLUT provides for some default position and size of the display window, we can change those. The following two lines in the example are used for the purpose. As the names suggest, the *glutInitWindowPosition* function allows us to specify the window location. This is done by specifying the top-left corner position of the window (supplied as argument to the function). The position is specified in integer screen coordinates (the X and Y pixel coordinates, in that order), assuming that the origin is in the top-left corner of the screen. The *glutInitWindowSize* function is used to set the window size. The first argument specifies the width of the window. The window height is specified with the second argument. Both are specified in pixels.

```
glutInitWindowPosition (0, 0);
glutInitWindowSize (800, 600);
```

Next, we create the window and set a caption (optional) with the following function. The argument of the function, that is, the string within the quotation, is the caption.

```
glutCreateWindow ("The OpenGL example");
```

Once the window is created, we need to specify the picture to be displayed in the window. In our example, the picture is simply the line. We create this picture in a separate function *createLine*, which contain OpenGL functions. The *createLine* function is passed as an argument to the *glutDisplayFunc* indicating that the line is to be displayed on the window. However, before the picture is generated, certain initializations are required. We perform these initializations in the *init* function (to make our code look *nice and clean*). Hence, the following sequence of lines are added to our main program.

```
init ();
glutDisplayFunc (createLine);
```

The display window, however, is not yet on the screen. What we need is to *activate* it, once the window content is decided. Thus, we add the following statement. This statement activates all display windows we have created along with their graphic contents.

```
glutMainLoop ();
```

This function must be the last one in our program. Along with displaying the initial graphics, it puts the program into an infinite loop. In this loop, the program waits for inputs from devices such as mouse or keyboard. Even if no input is available (like in our example), the loop ensures that the picture is displayed till we close the window.

Basic OpenGL Syntax

In the main body of the example program, we have mostly the GLUT library functions. However, in the two functions *init* and *createLine*, core OpenGL library functions are used. The syntax followed by these functions is different from those of the GLUT functions, and is described as follows.

Each openGL function is prefixed with *gl*. Also, each component word within the function name has its first letter capitalized. The naming convention is illustrated in the following examples.

```
glClear        glPolygonMode
```

Sometimes, some functions require that one or more of their arguments be assigned symbolic constants. Example of such constants include a parameter name, a parameter value, or a particular mode. All such constants begin with the uppercase letter GL. Each component of the name is written in capital letters and are separated by the underscore ('_') symbol. A few examples are shown here for illustration.

```
GL_RGB        GL_AMBIENT_AND_DIFFUSE
```

The openGL functions also expect specific data types, a 32 bit integer as a parameter value, for example. For the purpose, OpenGL uses built-in data type names. Each name begins with the capital letters GL. This is followed by the data type name (the standard designations for various data types) written in lower-case letters, such as

```
GLbyte        GLdouble
```

Initialization: The `init()` Function

In the *init* function, initializations and one-time parameter settings are done. There are three OpenGL library routines called in this function. The first routine is,

```
glClearColor (1.0, 1.0, 1.0, 0.0);
```

This OpenGL routine is used to set a background color to our display window. The color is specified with the red, green, and blue (RGB) components. The RGB component values are supplied through the first three arguments, in that order. Thus, in our example, we are setting the window background to be white having $R = 0.1$, $G = 0.1$ and $B = 0.1$ values. If we set all components to 0.0, we will get the color black. If the components are set to any other value between 0.0 and 1.0, we get some shade of gray. The fourth parameter in the function is called the *alpha* value for the specified color. It is used as a *blending* parameter: specifying the way to color two overlapping objects. A value of 0.0 implies the objects are totally transparent and a value of 1.0 indicates totally opaque objects.

Although we are displaying a line, which is a 2D object, OpenGL does not treat 2D picture generation separately. It treats 2D pictures as a special case of 3D viewing. So the entire 3D pipeline stages has to be performed. In our example, therefore, we need to specify the projection type and other viewing parameters. These are done with the following two functions.

```
glMatrixMode (GL_PROJECTION)
gluOrhto2D (0.0, 800.0, 0.0, 600.0)
```

Note that although the first function is an OpenGL routine (prefixed with *gl*), the second function is pefixed with *glu*. This indicates that the second function is not part of the core OpenGL library. Instead, it belongs to the GLU or the OpenGL Utility, an auxiliary library that provides routines for complex tasks including setting up of viewing and projection matrices, describing complex objects with line and polygon approximations, processing the surface-rendering operations and displaying splines with linear approximations. Together the two functions specify that an orthogonal projection is to be used to map the line from the view plane to the screen. The view plane window is specified in terms of its lower-left (0.0, 0.0) and top-right corners (800.0, 600.0). Anything outside this boundary will be clipped out.

Creating the Picture: The `createLine()` Function

This is the function that actually creates the line. The first line of the function is,

```
glClear (GL_COLOR_BUFFER_BIT);
```

With this OpenGL function, the display window with the specified background color is put on the screen. The argument, as you can see, is an OpenGL symbolic constant. It indicates that the bit values in the color (refresh) buffer are to be set to the background color values specified in the *glClearColor* function.

While we can set the background color, OpenGL also allows us to set the object color. This we do with the following function.

```
glColor3f (0.0, 1.0, 0.0);
```

The three arguments are used to specify the *R*, *G*, and *B* components of the color, in that order. The suffix *3f* in the function name indicates that the three components are specified using floating-point (*f*) values. These values can range between 0.0 and 1.0. The three values we used in the example denote the green color (as the other two components have values 0.0 each).

Finally, we need to call the appropriate OpenGL routines to create the line segment. The following piece of code performs just that: it specifies a line segment between the end points (200, 100) and (20, 50).

```
glBegin (GL_LINES);
  glVertex2i (200, 100);
  glVertex2i (20, 50);
glEnd ();
```

The two line end points (vertices) are specified using the OpenGL function *glVertex2i*. The suffix *2i* indicates that the vertices are specified by two integer (i) values denoting their *X* and *Y* coordinates. The first and second end points are determined depending on their ordering in the code. Thus, in the example, the vertex (200, 100) is the first end point while the vertex (20, 50) acts as the second line end point. The function *glBegin* with its symbolic OpenGL constant GL_LINES along with the function *glEnd* indicate that the vertices are line end points.

With all these functions, the basic line creation program is ready. However, the functions we used may be stored at different locations in the computer memory, depending on the implementation of OpenGL. We need to *force* the system to process all these functions. This we do with the following OpenGL function, which should be the last line of our picture generation procedure.

```
glFlush ();
```

SUMMARY

In this chapter, we learnt about the underlying hardware in a graphics system. There are three components of the hardware—the input devices, the output devices, and the display controller.

The most common graphics output devices are the video monitors. Various technologies are used to design monitors. The earliest of those are the CRT, which we have already discussed in Chapter 1. In this chapter, we learnt about flat panel displays. Broadly, they are of two types. In the emissive displays, elctrical energy is converted to light energy, similar to a CRT. Examples include plasma panels, LEDs, and thin-film electroluminescent displays. In non-emissive displays, external light energy is used to draw pictures on the screen. The most popular example of such displays is the LCD. Hardcopy devices are another mode of producing graphics output. Such devices are of two types: printers are used to produce any image including alphanumeric characters on paper, while plotters are used for specific drawing purpose. Input devices are mainly used for interactive graphics. Many such devices exist. Most common are the mouse and keyboards. Other input devices include joystick, trackball, data gloves, and touch screens.

The display controller or the *graphics card* contains a special-purpose processor and a memory tailor-made for graphics. The processor is called the graphics processing unit or GPU. It consists of a large number of simple processing units or *cores*, organized in the form of *streaming multiprocessors*. The organization allows a GPU to perform parallel processing in SIMD (single instruction multiple data) mode. Modern-day GPUs allow general-purpose programming of its elements. This is done through *shader* programming. The vertex shaders are programs that allow us to process vertices while the fragment shaders allow us to process pixels the way we want.

We also got introduces to the graphics software. We learnt about the role played by graphics libraries in the developement of a graphics program and learnt the basics of a popular graphics library, the OpenGL, through an example line-drawing program.

BIBLIOGRAPHIC NOTE

Sherr [1993] contains more discussions on electronic displays. Tannas [1985] can be used for further reading on flat-panel displays and CRTs. Raster graphics architecture is explained in Foley et al. [1995]. Grotch [1983] presents the idea behind the 3D and stereoscopic displays. Chung et al. [1989] contains work on head-mounted displays and virtual reality environments. More on GPU along with examples on programming the vertex and fragment processors can be found in the *GPU Gems* series of books (Fernando [2004], Pharr and Fernando [2005]). For additional details on writing program using a shading language, refer to the OpenGL^{TM} Shading Language (Rost [2004]). The website `www.gpgpu.org` is a good source for more information on GPGPU. A good starting point for learning OpenGL is the OpenGL Programming Guide (Shreiner et al. [2004]).

KEY TERMS

Computer graphics application programmers interface (CG API) – a set of library functions to perform various graphics operations
Data glove – an input device, typically used with virtual reality systems, for positioning and manipulation
Dot-matrix printer – a type of impact printer
Emissive display – a type of display that works based on the conversion of electric energy into light on screen
Fixed-function hardware pipeline – all the pipeline stages are pre-programmed and embedded into the hardware
Flat panel – a class of graphics display units
Fragment shaders (programs) – hardware programs to assign colors to pixels
GPU – the graphics processing unit, which is typically employed for graphics-related operations
Graphical kernel standard (GKS) – an early standard for graphics software
Impact printer – a printer that works by pressing character faces against inked ribbons on a paper
Joystick – an input device for positioning
Keyboard – an input device for characters
LCD – a type of flat panel non-emissive display
LED display – a type of flat panel emissive display
Mouse – an input device for pointing and selecting
Multicore – multiple processing units connected together
Non-emissive display – a type of display that works based on the conversion of light energy to some onscreen graphical patterns
Non-impact printer – a printing device that uses non-impact methods such as lasers, ink sprays, electrostatic, or electrothermal methods for printing
OpenGL – an open source graphics library, which has become the de-facto standard for graphics software
PHIGS (Programmer's Hierarchical Interactive Graphics Standard) – a standard for graphics software
Plasma panel – a type of flat panel emissive display
Plotter – a type of hardcopy output device
Printer – a type of hardcopy output device.
Programmable GPU – a GPU where pipeline stages are not fixed and can be controlled programmatically
Shader – a grid of GPU processors to perform specific stages of graphics pipeline
Shader programming/GPU programming/Graphics hardware programming – programs to manipulate shaders
Spaceball – an input device for positioning
Stream processor – a processor that work on data streams
Streaming multiprocessor – a group of stream processors
Thin-film electroluminiscent display – a type of flat panel emissive display
Touch screen – gestural input systems
Trackball – an input device for positioning
Vertex shaders (programs) – hardware programs to process vertices

EXERCISES

10.1 What are the major components of a graphics system?
10.2 Discuss the difference between emissive and non-emissive displays.

10.3 Explain, with illustrative diagrams, the working of the plasma, LED, and thin-film electroluminescent displays?
10.4 How do LCDs work? Explain with schematic diagrams.
10.5 Mention any five input devices.
10.6 Why si GPU better suited for graphics operations than CPU? Discuss with a suitable illustration.
10.7 Explain the implementation of 3D graphics pipeline on GPU.
10.8 Explain the concept of shader programming. Why is it useful?
10.9 Why do we need graphics standards? Mention any two standards used in computer graphics.
10.10 In Fig. 10.11, the code for drawing a line segment on the screen is shown. Assume that we have a square surface with four vertices. The line displayed on the screen is a specific orthogonal view of the surface (may be top view). Modify the code to define the surface and perform the specific projection. Modify the code further so that the Gouraud shading is used to color the surface.

CHAPTER 11 Computer Animation

Learning Objectives

After going through this chapter, the students will be able to

- Know about the broad issues in animation
- Understand the traditional animation techniques and their relationship to computer animation
- Get an overview of the four broad classes of computer animation techniques
- Learn about the principles of animation
- Understand the key framing technique
- Get an overview of the animation techniques that use motion capture
- Get introduced to the physics-based and procedural animation techniques

INTRODUCTION

In computer graphics, we are concerned about synthesizing a static image, often known as a *frame*. All the stages and algorithms in each stage, which we have learnt so far, are designed to meet that objective. However, sometimes we are interested in generating a *sequence* of frames in a particular way so as to synthesize the *perception* of motion of the objects in a frame. Since computer graphics deals with the generation of single frames, it does not help us decide on the particular sequence. For that, we need additional methods and techniques, which are dealt with in *computer animation*. In this chapter, we shall learn about these methods and techniques.

The word *animation* is derived from the Latin word *anima*, meaning *the act, process, or result of imparting life, interest, spirit, motion, or activity*. Therefore, computer animation deals with the methods and techniques to display *motion* using computers. There are broadly two issues in computer animation—how to impart motion to an object and how to model *collision* when multiple objects are in motion in different directions? In this chapter, we shall learn about methods and techniques used to address the former (i.e., how to impart motion). The latter issue (about collision detection and modeling) is very involved and an extensive field of study in itself and lies outside the scope of this book.

11.1 TRADITIONAL ANIMATION TECHNIQUES

Animation, as we know, came before the advent of modern-day computers. So, what is the procedure traditional animators followed? Recall that an animation (whether computer or not) is a sequence of static images or frames. The sequence is basically a rough sketch of the layout of the objects in each frame, so that when the sequence is seen by a viewer at an appropriate speed (i.e., number of frames per second), the viewer *perceives* that the objects are moving. The sequence is preplanned by the animator, which is known as *storyboard layout* or simply *storyboarding*. Each frame in the sequence is then hand-drawn by the animator. This process is often known as the *straight ahead* method. Clearly, it requires *solid drawing skills* to make the animation look *appealing* and *realistic*.

Planning of the storyboard layout and then drawing each frame is clearly a very tedious and time-consuming affair. Computer animation techniques try to reduce the animators, time and effort by helping to *automatically* generate *most* of the frames in the sequence. In order to do that, several methods and techniques are used, all geared towards making the animation look *realistic*. We can classify all of them into the following broad classes.

Keyframing-based animation This idea is borrowed from traditional animation techniques with certain modifications

Animation using motion capture Motion is synthesized from real-world data captured using special devices

Physics-based methods Differential equations inspired by the laws of physics are solved to generate motion

Procedural animation techniques Procedures that approximate complex physical processes

In the remainder of this chapter, we shall learn about the basic idea behind each of these techniques; but before that, we shall have a look-at the basic principles that any animation technique should follow to create realistic animations.

11.2 PRINCIPLES OF ANIMATION

With the advent of *powerful* computers and computer animation techniques, creating animation has become much easier than traditional techniques. Consequently, we see numerous animations around us in the form of games, movies, embedded videos in web pages, and so on. However, some of these we like while some animations we do not want to watch the next time. Why is it so? What differentiates a *good* animation from a *bad* one?

Typically, animations that have *realistic* motions are found to be rated *good* by most viewers. Traditional animators for long have been following some rules to make animations look realistic. In a paper in 1987 (see bibliographic notes), John Lasseter summarized them in the form of the following broad groups of principles for use in computer animation.

11.2.1 Timing

The most critical component of any animation is the *speed* or *timing* of action of the objects. It actually makes us perceive *realism* through the perception of weight of the

objects, their emotional state (if any), and meaning of action. For example, we perceive a slow moving object to be heavier than a fast moving object. Thus, in a realistic animation, objects with different weights should have different movement behavior. Similarly, same head movement (may be up-down) with different speed can be used to express two different actions: a fast movement may indicate expressing agreement over something whereas a slow movement may denote searching for an item vertically in a bookshelf.

11.2.2 Action Planning and Layout

A good or realistic animation should be able to make it clear to a viewer the idea (such as action, mood, or meaning) being presented at any instant of motion. This requires a high-level planning or *staging* of the action. Such planning helps to draw the viewer's attention to the place in the current frame where the important action is taking place. Since our eyes are good at perceiving relative changes, contrasting motions often help achieve the goal. For example, a slow moving object draws attention quickly among a group of objects moving in high speed. Often we are required to put objects in suitable orientations for the purpose, as illustrated in Fig. 11.1.

At the level of individual objects, each action should be split into three parts: *anticipation*, the action itself, and *follow-through*. No realistic action can start suddenly. The preparations involved before the start of an action is the *anticipation*. Similar to how it starts, no action ends suddenly. The gradual termination of an action is the *follow-through*. For example, a batsman in a cricket match cannot hit a ball with his bat all of a sudden and then freeze. Certain movements of the batsman's body, hands, legs, and head precedes the actual hitting of the ball, which is the anticipation part. Similarly, the batsman again makes certain motion *after* hitting the ball, which is the follow-through part.

Sometimes, the main action often causes one or more *overlapping* actions. For example, a fielder may start moving towards the batsman as the batsman prepares to hit the ball. Also, one action may sometime lead to some *secondary* action. An example of such secondary action is the movement of the cloth of a person who is in motion. Both of these are important to make the motion look natural. However, it is important that the secondary action is not allowed to dominate the main action.

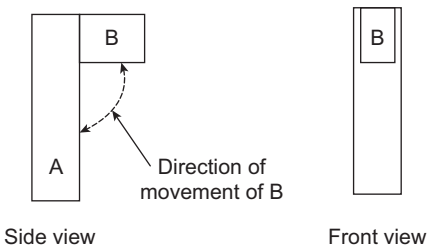


Fig. 11.1 The idea of performing staging by managing object arrangment and orientation. In the left figure (side view), the particular arrangement ensures that any movement of object B is noticeable, which is not possible in the case of the arrangement in the right figure (front view).

11.2.3 Animation Techniques

While timing and action planning are two important aspects of a realistic animation, they are not all. Unless we use suitable animation techniques, the realistic effect may not be evident. Several such techniques are there. One of the important ones is called the *squash and stretch* technique. In this technique, object shape should be changed or deformed to reflect the nature of motion it is subjected to. Generally, objects tend to get stretched along the direction of motion and squashed when external forces are applied to it. The classic example of a bouncing ball, to illustrate the idea, is shown in Fig. 11.2. Note that the more the speed or external force, the more should be the amount of stretching or squashing.

Another important technique to consider is the motion trajectory. In reality, it is very rare to find objects moving in straight lines. Therefore, the motions in a realistic animation should ideally happen along curved paths or *arcs*. Also, instantaneous change in speed is not a realistic phenomenon. Therefore, we should avoid doing the same in animation. Motions should start and end in a gradual manner. This is known as *slow in and out* principle.

As we mentioned before, an animation is nothing but a series of *frames* or static images. Traditional animators typically draw all these frames in a *straight ahead* fashion—starting with the first frame and drawing the last one after drawing all the intermediate frames in sequence. In contrast, computer animators can leverage the power of the computer by going for the *pose-to-pose* or *keyframing* technique. In this technique, which we shall learn in more detail in the next section, only a few frames (known as the *keyframes*) in the sequence are created/specified by the animator. Other *in-between* frames are generated from those keyframes. Clearly, the technique reduces much effort of the animator.

In order to achieve greater artistic effect, all the aforementioned techniques such as squash and stretch, arc, and keyframing are sometimes used with some *exaggeration* (some effects that typically are not observed in reality). The goal of exaggeration is to make the animation more *appealing* to the viewer. Too much complexity or symmetry tends to be less appealing and realistic animations should avoid those. Moreover, just as a traditional animator should have *solid drawing skills* to be able to create realistic good quality animation, a computer animator should have a good understanding of computer graphics and the animation tools he uses.

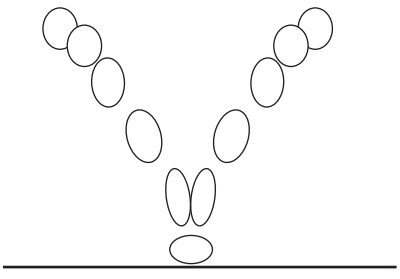


Fig. 11.2 The idea of squash and stretch, illustrated with a ball bouncing from a wall (the horizontal line below). Notice the change in shape as the ball moves towards the wall and then bounces back.

11.3 KEYFRAMING

Early computer animation systems were mainly designed based on the *keyframing* technique, which is still used widely. As we have seen in Section 11.2, the technique involves two stages. In the first stage, the animator *specifies* some specific frames in the whole sequence of frames that constitutes the animation. These selected set of frames are known as keyframes. In the second stage, the computer *determines* the intermediate or *in-between* frames. The idea is illustrated in Fig. 11.3.

In fact, the term *keyframe* is a misnomer in the context of computer animation. What the animator specifies is not a complete *frame* or image. Instead, it is basically a set of parameter values (e.g., current position of object centers, color values, light source intensity, modeling transformations between different object parts, etc.). Such parameters are often called *articulation variables*. Articulation variable values are typically specified and manipulated through some interactive interfaces that are part of computer animation tools. From the articulation variables, corresponding values for the in-between frames are computed using *interpolation* techniques. Starting from the simplest, that is linear interpolation, many types of interpolation techniques including the spline interpolation (discussed in Chapter 2) can be and are used. With interpolation, we can implement the principles of moving in arcs and slow in and out. The idea of interpolation of motion in the in-between frames from the keyframe values is illustrated in Fig. 11.4, where the motion of a point is interpolated using linear interpolation.

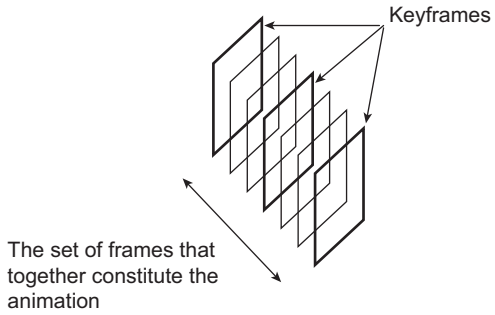


Fig. 11.3 The idea of keyframing technique. Frames in between the keyframes (called in-between frames) are generated by the computer.

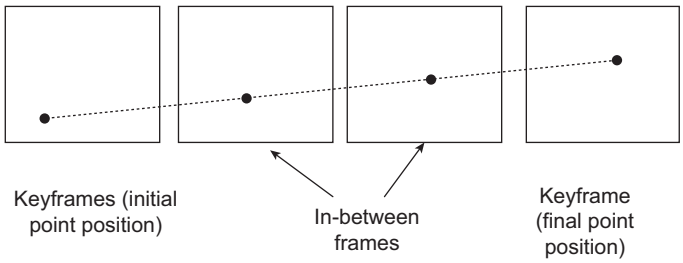


Fig. 11.4 The idea of interpolation-based in-between frame generation. The motion path of the point is interpolated from the keyframe positions with linear interpolation (the dotted line).

11.3.1 Character and Facial Animation

Although we can use most of the object-representation techniques we discussed in Chapter 2 in computer animation, the preferred choice to represent human-like characters (also called *articulated figures*) is the *skeleton*[1]. In skeleton-based articulated figure representation, atleast two layers are present. The *skin* or outer layer and the skeleton itself underneath the skin. A skeleton is a hierarchical tree structure of *bone joints* modeling the kinematic behavior of the character. The hierarchy is created by considering each of the skeleton's joints as a parent node. The idea is illustrated in Fig. 11.5.

In order to synthesize motion of a skeleton, we can apply the *forward kinematics* method. In this method, we actually specify the exact motion parameters for *each* joint in the keyframes. Such parameters may include position, velocity, and accelaration. From these parameter values, the computer determines the position of the joints in the in-between frames. Let us try to understand the concept in terms of a simple example.

Consider Fig. 11.6. It shows a portion of an articulated figure (actually, one hand of a human) having three joints. We want to synthesize the motion of lifting the hand. Let us ignore the velocity and accelation parameters (i.e., it moves with a constant speed). Then, we can specify a *motion path* (position) of all the three joints in the keyframes. Next, we apply interpolation to determine the position of the joints in the in-between frames, as explained in Fig. 11.4.

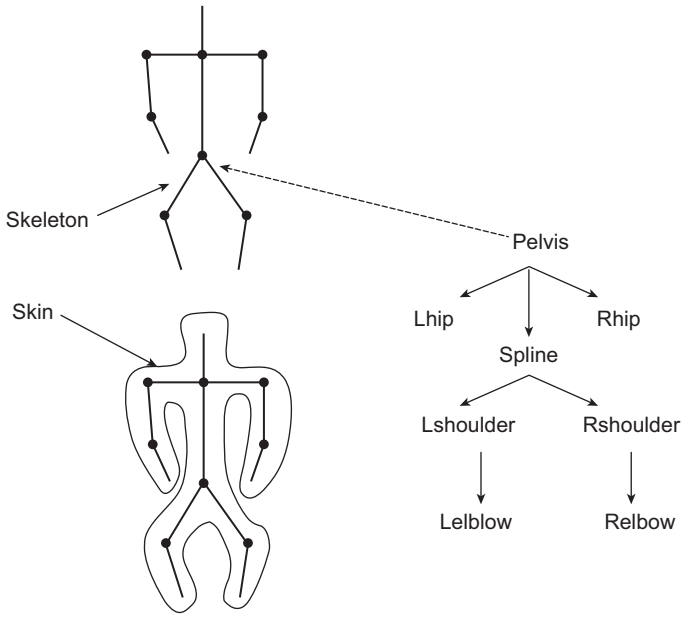


Fig. 11.5 The skeleton representation of an articulated figure. In the left, the two layers of the representation (skin and skeleton) are illustrated. The right figure shows the hierarchy representing the skeleton.

[1]In fact, we can represent any animate or inanimate character other than human using skeleton.

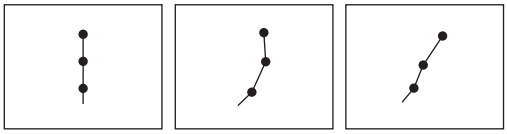


Fig. 11.6 The forward kinematics illustration. In the keyframes, we specify the positions (motion path) of *all* the three joints. Using interpolation, the joint position in the in-between frames are computed.

Usually, the motion parameters for the root node in the skeleton hierarchy are specified with respect to the world coordinate system. Parameters for the other joints are specified relative to the root node parameters.

Specifying motion behavior at all the joints is sometimes inconvenient and also unnecessary for the animator—most of the joints are internal to the skeleton and the animator does not want to be bothered about those. Instead, the animator may feel it more convenient to specify motion behavior of only a few of the joints (typically end points of a joint chain such as the *Lelbow* in Fig. 11.5) in the keyframes. It is then left to the computer to determine the movement behavior of all the other joints in the chain to obtain the particular motion of the end point. *Inverse kinematics* techniques are used to address this issue. Although the use of inverse kinematics frees the animator from specifying too many parameters, it has one important drawback—the determination of intermediate joint movements, from the end postion specification of a few selected joints, need not always give a unique solution. For example, an animator may specify in the keyframes the motion paths of the pelvis and two ankle joints of an articulated human leg and use inverse kinematics techniques to determine the motion paths of the other joints of the leg (hip and knee joints). However, there may be multiple ways to move the joints so as to get to the final position and all of them need not be realistic (although mathematically correct). Consequently, the output (i.e., what the viewer sees) may be affected.

How is the skeleton linked to its skin? Typically, each skin point is *associated* with the nearest joint in the skeleton. Whenever there is a joint movement, the skin point is moved by the same amount. It should be remembered that a viewer does not get to see the skeleton. only the changes in the skin points are visible. Thus, skeleton is essentially a data structure which is used to render the skin.

Similar to articulated figures, special techniques are used to animate human faces. The static shape of a face can be characterized by a relatively small set of *conformational parameters* such as the nose length, width of the jaws, eye to forehead distance, and so on. In order to represent the dynamic face shape, another small set of *expressive parameters*, such as rigid head rotation amount, width of the open eyes, movement of feature points from static position, etc., can be used. By combining these two sets of parameters in different possible ways, we can generate various expressive faces in an animation. The use of parameters also makes it suitable for keyframing, since the in-between parameter values can be interpolated from the keyframe values. Another widely-used approach for facial animation is known as the *facial action coding system* or FACS. In this approach, a facial expression is considered to have resulted from the combination of a set of elementary motions known as the *action*

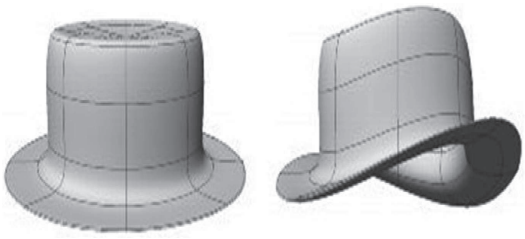
units (AU). Examples of AU include stretching the lips, raising the inner brow, wrinkling the nose, and so on. The AUs were determined from extensive psychological research. When combined, a subset of AUs can generate a specific facial expression.

11.3.2 Deformation

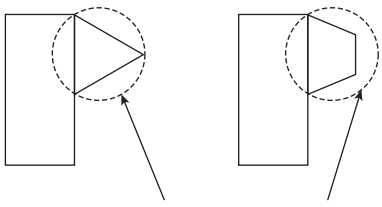
In most animations, we can see that the shape of objects gets changed. Sometimes the whole object shape gets changed; sometimes only a part of it changes. As an illustration, consider Fig. 11.7. In Fig. 11.7(a), the whole object shape changes whereas in Fig. 11.7(b), only a portion gets changed. Such change in shape is called *deformation*. Another popular term is *morphing*, which is a shortened form of the word *metamorphosing*.

Deformation can be implemented with a simple idea. Any shape can be thought of as a set of points (e.g., vertices of triangular mesh or control points of spline surfaces). A *deformation function* (implemented as *transformation matrices*) is applied on this set of points, which moves those points in the in-between frames to different positions (changing the relative postion between them). This results in the deformed object in the next keyframe. As an example, consider Fig. 11.8. We want to deform the cube shown in the left to the object shown in the right side of the figure. This is known as *tapering*. The particular tapering can be achieved if we perform the following transformation of each point $P(p_x, p_y, p_z)$ in the cube.

$$P' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & S(p_x) & 0 \\ 0 & 0 & S(p_x) \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$



(a)



The triangular part of the object gets deformed to a trapezoid. The other part remains the same.

(b)

Fig. 11.7 Illustration of deformation in computer animation. In (a), the whole object is deformed. However, the deformation is done locally in one portion of the object in (b).

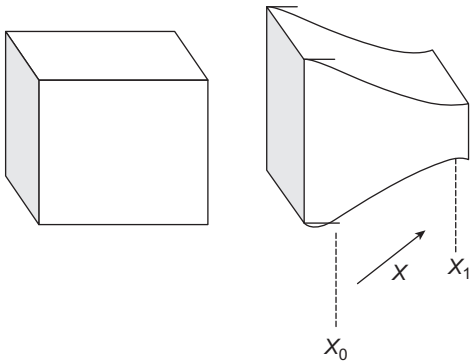


Fig. 11.8 Example of deformation by applying deformation function on the object points. In the example, we defined a function (having three components) to deform (taper) the object on the left side to the object on the right side

where

$$S(x) = \begin{cases} 1 & x \geq x_0 \\ 1 - 0.5\frac{x - x_0}{x_1 - x_0} & x_0 < x < x_1 \\ 0.5 & x \geq x_1 \end{cases} \tag{11.1}$$

There are two problems with this simple idea: first, it is very difficult to find a proper function for any arbitrarily complex deformation and second, such a deformation applies to the whole object. In order to have better control on deformation, we can use a more general technique known as the *free form deformation* or FFD. In this, we define a coordinate grid (or lattice) encapsulating the portion of the object to be deformed. The coordinates of the object points lying in this grid are recomputed with respect to the grid. We then distort the lattice points in the desired way (by applying some transformation function on the grid). The distorted lattice points are then used as control points to interpolate new object points from the old object point coordinates computed using the undistorted lattice. The idea is illustrated in Fig. 11.9.

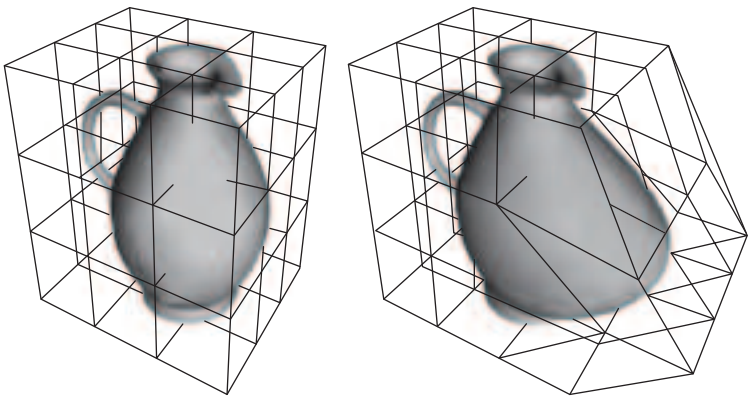


Fig. 11.9 The idea of FFD

11.4 MOTION CAPTURE

The objective of the techniques such as keyframing, forward and inverse kinematics are to help in generating realistic motion. However, even with such techniques, our abilities to impart realism to computer animation are limited. This is particularly so when we are animating articulated figures. There are certain aspects of physical motion of such articulated figures, which are subtle and difficult to model, but a small deviation from which makes the motion looks *unrealistic*. It takes a good amount of animator skills to model such subtlity. It is sometimes easier to record and map the motion of a real object (called *subject* or *talent*) to a synthetic object than to synthesize the whole motion itself. This is known as *motion capture*. It involves sensing, digitizing, and recording an object's motion and mapping it to the object model (usually skeleton). Although typically used to capture human motion, such techniques can also be used to synthesize non-human articulated figures also, which can be represented using skeletons. Presently, there are broadly two ways to capture the motion of an object—*electromagnetic sensors* and *optical markers*.

In electromagnetic tracking, often called *magnetic tracking*, sensors are placed at the joints of the subject that transmit their positions and orientations back to a central processor to record their movements. However, in the presence of other objects, it is posssible that the transmitted information is not accurate due to *interference*. Also, the transmission of sensory information can be performed through cables, which requires the subject to attach many cables to his/her body, a very inconvenient arrangement. An alternative is to wirelessly transmit the information. In such case, subjects need to carry a battery pack as a power source for the wireless transmitter; not very convenient either. Nonetheless, if such inconveniences are overlooked and the data collected in a noise-free environment, the data obtained can be used to capture motion with a very high degree of accuracy.

In the second approach, namely motion capture using optical markers, the instrumentation requirement is very convenient for the subject. A subject has to wear only some reflective markers (brightly colored objects such as table tennis balls or colored tapes) on his cloth. The markers are usually placed near the joints. For better accuracy, additional markers may also be placed at mid-joint (middle of two joints) positions. Then, the subject's movement is videographed, using either two or more cameras. The image frames in each video are then analyzed to locate the markers and their dispalcement across frames. Also, 3D marker positions are reconstructed from the multiple views. From all these information pertaining to the markers' movement, the joint movements in a skeleton model of the subject are determined (since each marker is associated with a joint) for specific motions (e.g., slow walking). If the number of markers or the frame rate of the video capturing instrument is less, inverse kinematics techniques are used to interpolate joint movements from the joint positions captured in the video. Although the subject does not require any instrumentation (other than wearing the markers) in this technique, the technique is less accurate than the magnetic tracking methods. There are many reasons for low accuracy including the limitations of image and video processing techniques and the *marker occlusion* problem (i.e., some markers are occluded by others during the subject's motion). In spite of that, the method is low-cost and easier to implement, making it the method of choice for many computer animators.

11.5 PHYSICALLY BASED METHODS AND PROCEDURAL TECHNIQUES

In the previous sections, we discussed ways to determine the position and orientation of each object in each frame of a computer animation (through interpolation). We assumed that the relative movement of objects with respect to each other in a frame is automatically taken care of. However, such relative movements are sometimes very important to perceive *realism*; some animators may be more concerned about such things than the exact position and orientation of objects. By maintaining a relationship among objects in a frame, what the animator achieves is *physical realism*. Physical realism is achieved through the use of physical forces on objects, governed by the laws of physics. The animation techniques, which model the process of application of forces to generate a motion, is known as the *physically based animation*. It should be remembered, however, that the term need not always refer to some real physical process. Unrealistic forces may be used to generate some creative animation effect. For example, a cricket ball may move towards the batsman on its own, rather than after being thrown by a bowler.

The physical laws are typically modelled using partial or, in some simple cases, ordinary differential equations. Therefore, in order to model motion that is produced by the application of the laws of physics, we need to solve the differential equations. In fact, it was one of the original goals of computers to solve such equations and various efficient numerical methods have been developed over the years. We can make use of those methods in the physically based animation. Nonetheless, such methods are expensive in terms of computation time and required resources. Consequently, these methods are typically used in situations where all other methods have failed to produce sufficiently good realistic effects or we have very good hardware in terms of processing power.

In fact, it is very important to determine when to use physically based animation as things may sometimes become hard to decide. Consider for example the modeling of a wrinkled cloth. We can actually characterize common types of wrinkles and where they occur on a piece of cloth by studying several examples. We can use this information to determine suitable wrinkles at appropriate places on an animated cloth. This *imposition* of wrinkles on the cloth, clearly, is less computationally expensive. However, it lacks flexibility. We can generate only the wrinkles that we have modeled. On the other hand, we can actually model the physics behind the formation of wrinkles: the stresses and strains of the individual threads of the cloth. If we apply the physical process on the synthetic cloth model, wrinkles will appear automatically, as the strands of the cloth moves according to the underlying physics. Thus, the wrinkles are not imposed in this case. Clearly, this physics-based approach is more flexible since any type of wrinkles can be generated. However, computation of individual threads of a cloth, after they are subjected to stress and strain due to the outside forces, is very expensive. Therefore, if we are looking for high-degree of realism, we should go for the latter. Otherwise, the former approach is alright since it produces an acceptable output at a much lesser cost.

It is obvious that the imposition of wrinkle patterns on a cloth is actually an approximation of the complex physical process underneath. We can implement such approximation with a procedure or a set of rules. In our example, such rules may indicate where to place

the wrinkles, how many wrinkes, length of each wrinkle, and so on. Consequently, such approximations are often called procedural techniques.

SUMMARY

In this chapter, we learnt about the basic idea of computer animation. An animation is a sequence of static images or frames. In traditional animation, all these frames have to be hand-drawn by the animator, which is a tedious and time-consuming process. In contrast, in computer animation, an animator merely specifies a small subset of *keyframes* (in reality, a set of parameter values rather than a complete image). The other *in-between* frames are generated by the computer by *interpolating* the keyframe values. This is known as *keyframing*, which saves much time and effort of the animator.

Although much work of a traditional animator is taken care of by the computer in computer animation, the guiding principles to generate a realistic animation remains the same. Such principles include *timing*, *staging*, *anticipation*, *follow-through*, *overlapping and secondary actions*, *squash and stretch*, *arcs*, *slow in and out*, *straight ahead and pose-to-pose*, *exaggeration*, *appeal*, and *solid drawing skill*. Specification of keyframes and algorithms to generate in-between frames should take into account these principles; otherwise the animations may not look realistic.

An important concept in computer animation is the idea of *deformation*—changing the shape of objects across frames (either wholly or partially). The idea is simple: apply some deformation *function* on the object definition (a set of points) so as to move the points to appropriate position in the in-between frame, so that we get the deformed object. However, finding such functions are often difficult and also they deform the whole object. Free form deformation techniques are used to *locally* deform objects through the use of an encapsulating lattice (or grid of points).

Animation of characters or articulated figures, primarily human, are done through the use of special representation technique called *skeleton*. A skeleton is a hierarchy of *joints* connecting the *bones*. A layer of *skin* encapsulates the skeleton. Using forward and inverse kinematics techniques, movement of skeleton joints is synthesized. A mapping technique maps such movement to the movement of outer skin, which is what the viewer gets to see. Similarly, special data structures and operations are used to generate (human) facial animation: either through the use of a set of *conformational* and *expressive* parameters or a set of *action units*. Generation of facial animation using the latter technique is known as the *facial action coding system* or FACS.

Apart from keyframing, many other techniques are used to create computer animation. Motion capture is another widely used method, in which the motion of a *subject* is captured (using electromechanical or optical sensors) and then mapped to a representation of the subject (usually a skeleton) to synthesize motion. The electromechanical sensor-based motion capture performs well in terms of accuracy in a noise-free environment. However, such methods are costly in terms of equipment and are inconvenient for the subject (since various instruments need to be attached to them). On the other hand, optical sensor (marker)-based systems are easy to implement and do not inconvenience the subject; however, their accuracy is less.

Sometimes, in order to synthesize realism, physics-based animation techniques are used. Such techniques require solving differential equations. Consequently, they are computation-intensive and require special hardware. Many of these techniques (rather their approximtions), however, can be implemented with procedural methods, which require less computation although the flexibility (in terms of the effect generated) is less.

BIBLIOGRAPHIC NOTE

Parent [2008] contains a more detailed introduction to computer animation. Principles of computer animation are summarized in Lasseter [1987]. More on key-framing, which is used in most of the early computer animation systems, can be found in Burtnyk and Wein [1971], Burtnyk and Wein [1976], Catmull [1978], and Reeves [1981]. Free-form deformation is presented in Sederberg [1986]. For more information on the use of forward and inverse kinematics in articulated figure animation, see Ribble [1982] and Welman [1993]. The facial action coding system is described in Ekman and Friesen [1978] (also see the web site http://www.cs.cmu.edu/~face/facs.htm). Menache [2000] discusses motion capture in computer animation. For various physically based models and procedural techniques, refer to Parent [2008].

KEY TERMS

Action units (AU) – a set of elementary motions which forms the basis of FACS
Anticipation – the preparation before the start of an action in an animation
Arcs – object motion paths in realistic animations
Articulation variables – a set of parameter values used to define keyframes
Conformational parameters – a set of parameters used for animation of human faces
Deformation – change in object shape in animation
Deformation function – a function to deform object/object portion
Expressive parameters – a set of parameters used for facial animation
Facial action coding system (FACS) – a widely used facial animation technique
Follow through – the gradual termination of an action in an animation
Forward kinematics – determination of skeleton joint positions in the in-betweens from the motion parameter values at keyframes
Free form deformation (FFD) – a technique for deformation in computer animation
In-betweens – the frames between keyframes
Inverse kinematics – estimating motion path of intermediate joints from the end joint positions in the keyframes
Keyframes – a set of impotant frames in the sequence of frames that constitute an animation
Morphing – generating a new shape from another shape
Motion capture – capturing motion of a real object for use in computer animation
Overlapping action – multiple actions that take part simultaneously in an animation
Secondary action – an action that is the result of another action
Skeleton – a hierarchical data structure of *bone joints* modelling the kinematic behaviour of an object
Slow in and out – gradual start and end of a motion in an animation
Squash and stretch – an animation technique in which an object shape is changed or deformed to reflect the kind of motion it is subjected to
Staging – high-level action planning
Storyboarding – the process of designing the sequence of frames for animation
Straight ahead method – the hand-drawing of frames in an animation sequence

EXERCISES

11.1 Mention the stages of animation. How is it related to computer graphics?
11.2 Explain the idea of keyframing. Does the animator *draw* a set of frames in this technique?
11.3 Mention the principles of animation. How do the principles help computer animators?
11.4 Explain the basic idea of deformation. Why is FFD advantageous?
11.5 Mention and explain the steps in articulated figure animation.
11.6 Discuss, with an illustrative example, the idea of inverse kinematics.
11.7 What are the action units in FACS. How do they help in facial animation?
11.8 Mention the two types of motion capture techniques. Can we say optical sensor-based tracking is better than magnetic tracking?
11.9 Discuss, with an example other than the one given in the text, when we require physics-based animation.
11.10 Describe an illustrative example of procedural technique. Why are such techniques useful in computer animation?

CHAPTER 12 Multimedia and Hypermedia

Learning Objectives

After going through this chapter, the students will be able to

- Get an overview of the field of multimedia and its broader aspects
- Understand the concept of hypermedia
- Learn about multimedia authoring tools and techniques
- Understand the techniques for storage and manipulation of audio and video data by a computer
- Get an overview of the data compression—both lossy and lossless
- Learn about the JPEG image compression standard
- Understand the H.261 digital video compression standard
- Learn about the MPEG standard

INTRODUCTION

In Chapter 11 we discussed one of the most important applications of computer graphics: computer animation. In this chapter, we shall learn the fundamental concepts of *multimedia*, another important application area of computer graphics. The term *multimedia* is used to refer to many things. Factually, it refers to some *content* or *media* represented in various *forms* combined together. The various forms include text, image, drawing, animation, video, and sound (including speech). This is in contrast to contents that use a single form of representation such as text display on the computer screen or video without audio. However, when we use the term, we are implicitly referring to *applications* that use multiple modalities such as text, image, drawings, animation, video, and audio to their advantage along with interactivity of some kind.

Since the media consists of various forms, we require multiple information processsing devices to record, play, display, or access the content. Together, these devices form a multimedia system. In fact, most computing systems we see around us today are examples of multimedia systems. These systems represent *convergence* of multiple technologies developed over a period of time: PCs, DVDs, computer games, digital TV, set-top web surfing, wireless networks, and so on. This convergence, in a sense, points to the coming together

of various formerly disparate and unrelated fields of study such as computer graphics, visualization, human-computer interaction, computer vision, data compression, graph theory, computer networking, database systems, and so on. Note that although we mentioned multimedia to be an application area of computer graphics, it is much more than that. Computer graphics is only one of the many components. In that sense, multimedia is different from animation.

There are many aspects of a multimedia system. We can group them into four major categories.

Content creation As we saw, multimedia involves many types of media put together to create single content. How we create the various forms and *link* them to get multimedia content is a major concern. The *multimedia authoring tools* play an important role here.

Representation In order to store multimedia contents, we need to represent them. There are various representation standards (file formats) for graphics and image data such as the BMP. Similarly, we have standards for audio and video contents as well. However, multimedia data is typically very large (owing to the presence of various media). In order to store the data using minimum space and transmit it using low bandwidth, the data is usually represented in a *compressed* form. Many compression standards are available for images as well as video and audio (e.g., JPEG, MPEG).

Search and retrieval Often, we need to retrieve a particular content from a huge database, for example, searching for specific images from web pages on the Internet. Obviously, we need to have a proper database management system for managing searching and retrieval.

Transmission and delivery In most multimedia applications, the objective is to transfer multimedia content over a network. A typical example is the Internet. Whenever we open a web page, we get to see text, image, embedded animations, and video links. In other words, we access multimedia contents from a server. Consequently, it is important to deal with the network issues (e.g., protocols, streaming, server design, QoS, etc.) arising out of this transmission and delivery requirements.

The broad issues mentioned here are very specialized and involve complete topics in themselves. A detailed discussion on these topics is beyond the scope of this book. Instead, we shall focus on a few important aspects of multimedia systems including the basic concepts in digital video and audio, data compression techniques, and some popular compression standards, and multimedia authoring. We shall begin by first discussing the meaning of the term *hypermedia*, which is closely associated with multimedia.

12.1 HYPERMEDIA

Ted Nelson was the first to use the term *hypertext* in 1965. It basically refers to the way we interact with textual content. For example, consider a book. We are supposed to read it from the begining to end, in the sequence. We cannot randomly move between pages. In other words, we access and read the content *linearly*. Hypertext is different in the sense that it allows us to access textual content non-linearly. There are *links* in the text. Whenever we select a link, it takes us to another part of the text (or another document). If the

content consists of other types of media such as audio, video, image, and/or animation in addition to text (i.e., multimedia content) and the access to various parts of the content is non-linear, we call it *hypermedia*. Apparently, this term was also proposed by Ted Nelson. Thus, hypermedia can be considered to be a particular multimedia application.

The most well known example of hypermedia is the World Wide Web (which we popularly refer to as the Internet). The web pages we access through a web browser contain multiple media. Through links, we can access various parts of those contents. A specific *protocol* (i.e., standard means of communication) was developed to transfer this multimedia content over the Internet, which is called the hypertext transfer protocol (HTTP). Although it was originally designed to transfer hypertext as the name suggests, it supports transmission of any hypermedia content over the Internet.

In order to publish a hypermedia content on the web (i.e., make it *displayable* through web browsers), the hypertext markup language or HTML was developed. With the HTML, we can put together multiple media, provide links for non-linear access, and perform various formatting operations of the content, all using the notion of the HTML *tags*. There is a pre-defined set of tags in HTML, each dedicated for a specific purpose. The idea of the tag-based mark-up of content has been extended in the extensible markup language or XML. It allows a user to define his own tags, rather than forcing him to always use the predefined tag set. We shall have a closer look-at the HTML in the next section, where we discuss multimedia authoring.

12.2 MULTIMEDIA AUTHORING

As the name suggests, the objective of multimedia authoring is to create multimedia content (also known as *multimedia production*). Any multimedia production begins with a *storyboard* - the initial idea represented by a series of sketches. These sketches are equivalent to *keyframes* of a video. A flowchart organizes the storyboards by inserting *navigational information*—the user interaction through which the scene changes. The flowchart phase is followed by a detailed *functional specification*—a walkthrough of each scenario of the production, frame by frame, including all screen actions and user interactions. Finally, the production is prototyped and tested before the final version is implemented and released for deployment. Clearly, it is a team effort involving a host of people with specialized skills such as art director, graphic designer, production artist, producer, project manager, writer, user interface designer, sound designer, videographer, and animators, in addition to programmers. Presentation of the multimedia content is also a very important aspect of production. The key concerns include graphics style (color combination and lettering), color schemes to be used, fonts (size, style, spacing, etc.), maintenance of proper color contrast and video transition design.

Various tools are available to create multimedia content. They may be simple video creation tools such as Macromedia Flash or a complete production environment such as Macromedia Director. These tools help us take into account various presentation issues and also allow us to implement various stages of a production cycle. Multimedia production with these tools follow some *metaphor*—abstraction of real world processes/methods. The following are some of the common authoring metaphors.

Scripting language metaphor We use a special scripting language, such as the OpenScript language in the Asymmetrix Learning Systems' toolbook program, for adding interactivity (buttons, mouse, etc.) and incorporate conditionals to the multimedia content.

Slide show metaphor Slide shows are one of the popular presentation methods of multimedia content. They are, by default, linear. Although it is possible to design presentation in a non-linear way, few practitioners use the option. Example programs are Microsoft PowerPoint and ImageQ.

Hierarchical metaphor Sometimes, user-controllable elements are organized into a tree structure. These are typically found in menu-driven applications.

Iconic flow-control metaphor Authoring systems that are based on flow-control metaphor provide icon libraries in the form of toolboxes. Authoring proceeds by creating a flowchart with the icons. An example of such authoring systems is the Macromedia Authorware.

Frames metaphor This is similar to the flow-control metaphor. However, the links in the flowchart are more conceptual than simply representing the program flow. An example is Quest by Allen Communications.

Card/Scripting metaphor Many multimedia production systems are designed around the concept of *card stacks*. A simple index-card structure is used for the content creation. Since links are available, it is easier to create hypermedia content. An early example of such systems was the HyperCard by Apple. Another popular example is the hyperStudio by Knowledge Adventure.

Cast/Score/Scripting metaphor In some multimedia production systems, time is depicted in a horizontal fashion with rows or tracks representing instantiation of characters. This is similar in nature to a music *score*. Multimedia elements are drawn from a *cast* of characters. A *script* is used to represent the overall flow of events. Macromedia director is an example of such systems.

An important issue in hypermedia authoring is the *link management*: how to create non-linearly navigable content. This is typically done using some form of markup language such as HTML. HTML is a language for publishing hypermedia on the World Wide Web (i.e., making them viewable through the web browsers). Any HTML document is divided into two parts: HEAD and BODY. A typical HTML document structure is,

```
<HTML>
<HEAD>
...
</HEAD>
<BODY>
...
</BODY>
</HTML>
```

Note the use of the ASCII strings inside parenthesis such as <HTML>. These are known as *tags*. Tags specify the structure of the content as well as the way the content is to be displayed and accessed. Apart from the overall structure, the document elements—text, image, and video—are also described using tags. In general, the tags are in the form <token param> to define the start point of a document element and </token> to indicate end of the element. Some elements may not require the ending tag. As an illustration, consider the following HTML document source.

```
<HTML>
<HEAD>
<TITLE>
An illustrative HTML document.
</TITLE>
<META NAME = author name CONTENT = foo bar>
</HEAD>
</BODY>
<P>
This is an example text to demonstrate the idea of a paragraph
element.
</P>
</BODY>
</HTML>
```

What we have demonstrated above is a simple example to illustrate the idea. The basic idea is to use a *predefined* set of tags. In XML, tags are not predefined. Instead, the programmer can define his/her own tag. This helps in creating logical separation between contents of different nature though they may be rendered in the same way. Another popular example of a markup language is the rich text format (RTF).

12.3 COMPONENTS OF MULTIMEDIA

As we have already seen, multimedia components include image, text, and animation. In the previous chapters, we saw how these are represented and manipulated in a computer. Apart from the image, text, and animation, the other two major multimedia components are video and audio. Let us learn how we can store and manipulate video and audio media in a computer.

12.3.1 Basics of Digital Audio

We can define audio (or sound) to be a *wave*, resulting from the compression and expansion of air molecules under the action of some physical device. For example, a speaker in an audio system vibrates back and forth and produces a longitudinal pressure wave that we perceive as sound. Thus, two things are to be noted. First, sound requires the presence of air (we cannot have the effect in space). Second, we can represent sound as a pressure wave, that is, a continuous or analog signal denoting the variation of pressure values (amplitude) over time. Typically, pressure is converted to voltage levels using some transducer. Therefore, we

can think of sound also as a variation of voltage level over time. The second point is very significant from the point of view of storage and manipulation of audio data in a computer.

Since an audio signal is continuous (or analog), we need some way to *digitize* it, to be able to store the signal in a computer. Digitization means converting the continuous-valued signal to a string of discrete numbers (preferably integers). This we can achieve through a technique called *sampling*—measuring values at evenly spaced intervals. Usually, the term *sampling* refers to the measurement of amplitude values at evenly spaced time intervals. The rate (i.e., number of samples per unit time) at which the values are measured is known as *sampling frequency*. Typical sampling frequencies are between 8 kHz (8000 samples per second) to 48 kHz (48000 samples per second). The human ear can perceive sound between the range of about 20 Hz to as much as 20 kHz (above this level, we enter the ultrasound range). Apart from keeping in mind the human perceptual ability (the audible range), we also need to keep in mind the problem of *reconstruction* of the signal from the samples. This requires us to sample at an appropriate rate called the *Nyquist rate* (named after the mathematician Harry Nyquist). We can think of any signal as a sum of sinusoids. Each of these constituent signals has its own frequency. Nyquist rate tells us that, in order to be able to reconstruct a signal from the samples, our sampling rate should be atleast *twice the maximum frequency content in the signal*.

Once the sampling is done, the values need to be quantized (i.e., represented in terms of a sequence of 0s and 1s). The number of bits used to denote each sample determines the precision of quantization. Typically, 8 or 16 bits per sample are used for quantization.

For playback of the quantized (or digital) sound, we have to convert it back to an analog signal. There are two broad methods to do so. One is called the *frequency modulation* or FM technique and the other is called the *wave table synthesis* or just the *wave* method. In the FM method, sound is synthesized by adding a modulating frequency to a carrier sinusoid. In the latter method, real sounds are stored in digitized form in memory (called *wave tables*). These quantized values are used to recreate sound. Clearly, wave method produces more realistic sounds; however, the storage requirement is high compared to the FM method.

12.3.2 Digital Video Fundamentals

Recall that we can think of a video as a sequence of images (frames). When this sequence is played with a specific speed (i.e., number of frames per second), we get the perception of motion. Therefore, in order to play a video, we need the information about each image in the sequence and the speed with which we need to display the images one after the other. Representation of an image is simple—we specify the color values for various pixels. We then need to specify the change in color values at each pixel position over time, to capture the speed aspect. This is equivalent to *sampling* of color values from *signals* at the specific time intervals. In other words, we can think of a video as a sampled data from a continuous signal. Depending on the composition, we can have three types of video signals—*component video*, *composite video*, and *S-video*.

When the video signal comprises three separate signals for the red, green, and blue colors, it is called *component video*. In general, it is not necessary to have the component video signals for the red, green, and blue components. Instead, we can use any three

components equivalent to the red, green, and blue color representation (i.e., other color models, see Chapter 5). Performance wise, component video gives the best color reproduction ability. However, it requires more bandwidth to transmit the signal. In *composite video*, color (*chrominance*) and intensity (*luminance*) signals are mixed into a single carrier wave. Chrominance is represented as a composition of two color components, usually denoted by I and Q, or U and V. The first two are taken from the YIQ color model, whereas the last two are part of the YUV color model. When being transmitted from a source to a destination, composite video uses only one wire and the color signals are mixed (unlike component video, where three separate wires are used to transmit the three components). Consequently, the quality of reproduction can degrade due to interference between the component signals. However, the channel bandwidth requirement is less. The *S-video* (separated video or super-video) is used to balance between the bandwidth requirement and interference effect. Two wires are used to carry S-video signal—one for luminance and the other for the composite chromatic signal.

Early digital videos, such as those produced by the Sony or Panasonic recorders, used to be composite video. Modern digital video generally uses component video. For digital videos, the red, green, and blue components are typically converted to the equivalent components in another color space (model), which is known as the YCbCr color space. We can represent the relationship between the RGB color space and the YCbCr space in the following matrix form.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \\ 0.5 \end{bmatrix}$$

If the red, green, and blue values are specified within the range [0,1], we use the following transformation matrix to get the Y, Cb, Cr values in the range [0,255].

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112 \\ 112 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

It has been found that humans see color with much less spatial resolution than black and white. This implies that, while transmitting the Y, Cb, Cr values for each pixel, it is not necessary to send all the values. Instead, we can *subsample* and transmit a lesser number of values without affecting the visual quality. This is known as *chromatic subsampling*. Numeral labels of the form 'm:n:p' are used to denote various subsampling schemes, each scheme defined in terms of a group of four pixels. As an example, consider a subsampling scheme denoted by '4:2:2'. We can interpret the label as follows: we take a group of four pixels. we number them sequentially from 0 to 3 (i.e., 0, 1, 2, 3). Among these four pixels, we transmit the Y color value for all four and the Cb and Cr values for the pixels 0 and 2 only. In other words, we are sending a total of 8 color values (4 Y, 2 Cb, and 2 Cr values) instead of the total 12 values (4 Y, 4 Cb, and 4 Cr values). Such sampling schemes are commonly used in the JPEG and MPEG compression standards that we are going to discuss in the next section.

12.4 DATA COMPRESSION STANDARDS

Since multimedia involves image, audio, video, and animation, in addition to text, the size of the data is usually very large. Therefore, it requires more storage space. Moreover, transmitting the data takes up lot of time and bandwidth. In order to overcome these problems, multimedia data is usually represented in *compressed* form. We can roughly define compression as *the techniques to reduce the number of bits used to represent data items*. A general scheme for data compression is shown in Fig. 12.1. The original data is used as input to an *encoder*, which converts the data to compressed form. The encoder outputs are typically termed as *codes* or *codewords*. Before we want to use the data again, it needs to be passed through a *decoder* to decompress it.

If the total number of bits required to represent data before compression is b_o and b_c denotes the number of bits required to represent the same data after compression, we define the *compression ratio* as $\frac{b_o}{b_c}$. Our objective is to have a *codec* (encoder/decoder scheme) with a compression ratio much larger than 1.0. This can be achieved in either of the two ways: *lossy* and *lossless*.

In lossless compression, as the name suggests, we do not lose any information. There are various techniques to perform lossless compression that include *variable-length coding* schemes such as the *Huffman coding* and *adaptive Huffman coding*, *dictionary-based coding* such as the *Lempel-Ziv-Welch (LZW) algorithm*, *arithmetic coding*, and *differential coding*. However, the compression ratio is usually low. In multimedia applications where we look for higher compression ratio, we usually go for *lossy* compression, that is, we lose some information during compression.

Lossy compression is typically applied on quantized data. Recall that in multimedia, we assume signal forms for various data items such as image, video, and audio. We deal with *samples* of these signals. These samples are real values, which are quantized, that is, mapped to a small set of values. The idea is similar to the one we use for intensity representation, discussed in Chapter 4. We mostly use vector quantization to represent the samples. In a vector quantization scheme, the compression works on groups of samples or vectors, rather than individual samples. Each vector is called a *code vector* and a group of code vectors representing the signal is called the *codebook*.

There are broadly two groups of lossy compression techniques: *transform coding* and *predictive coding*. In the *transform coding* techniques, samples are first transformed into a new basis space. The transformed samples are then quantized, which is followed by the *entropy coding* (lossless compression techniques such as Huffman coding) of the quantized values.

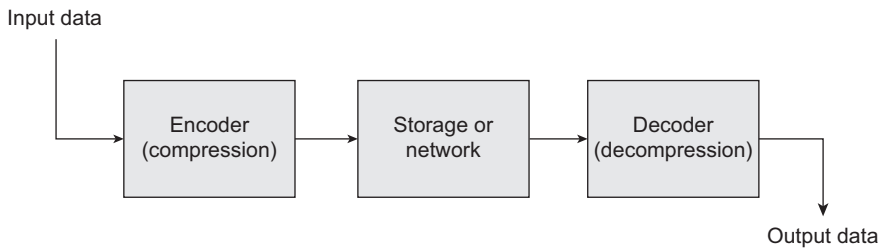


Fig. 12.1 A general data compression scheme

Examples of transform coding techniques include discrete cosine transform (DCT), inverse DCT, discrete fourier transform (DFT), and the wavelet transforms. In *predictive coding*, previous and/or subsequent decoded data is used to predict the current data. The error between the predicted data and the real data, together with any extra information needed to reproduce the prediction, is then quantized and coded. It should be remembered that in lossy compression, we never get back the original data after decompression. Instead, we get a *close perceptual approximation* of the original data. The closeness is determined based on some *distortion measure* such as the *mean square error* measure.

The compression techniques are used to design *standards* for multimedia video, audio, and image data. As we shall see, such standards use a combination of lossy and lossless compressions. In the following, we shall learn about three very popular standards, namely JPEG (image compression standard), MPEG (audio and video compression standard), and H.261 (video compression standard).

12.4.1 JPEG Image Compression Standard

As we know, an image is a grid of pixels. Each pixel has its own color values. There is an interesting property associated with these values—they change relatively slowly across the image. This property is called *spatial redundancy*. Image data compression works due to this property of images. There are various image data compression techniques developed over the years. The most popular are the JPEG standards including the basic JPEG, JPEG2000, JPEG-LS, and JBIG. All are based on similar ideas represented by the basic JPEG, which we shall discuss in this section.

The JPEG is an image compression standard developed by the *Joint Photographic Experts Group*. It was formally accepted as an international standard in 1992. The basic steps involved in the JPEG encoding process are shown in Fig. 12.2.

The encoder works as follows. Assume that we are given an image of any size. That means, the RGB values of each pixel are specified for the whole image. We first convert it

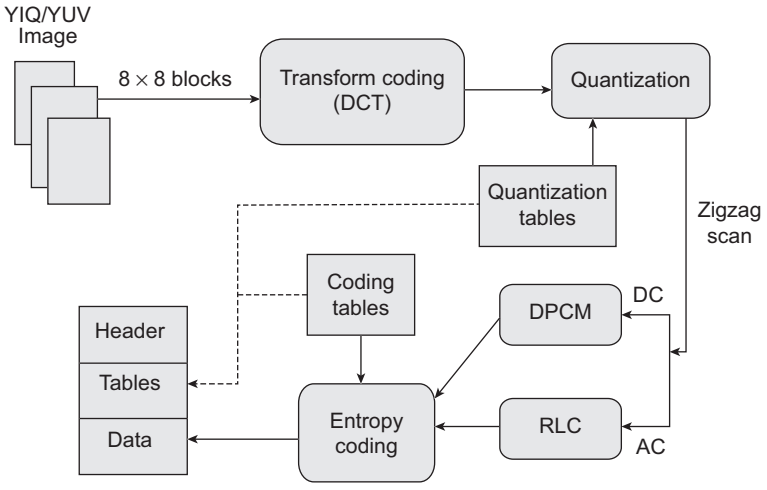


Fig. 12.2 Schematic of JPEG encoder

to YIQ or YUV color space, using the corresponding mapping shown here.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.595879 & -0.274133 & -0.321746 \\ 0.211205 & -0.523083 & -0.311878 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The JPEG encoder works on each color component separately, using the same procedure. Each converted image is divided into 8×8 blocks. The block size came from the empirical observation that *it is unusual for intensity values to vary widely within the small area of size* 8×8. On each image block, 2-dimensional (2D) DCT is applied. The general form of 2D DCT is shown in Eq. 12.1. The transformation is from spatial domain (i, j) to another (frequency) domain (u, v), with integers u and v running over the same range as i and j. The constant $C(u)$ or $C(v)$ takes the value $\frac{\sqrt{2}}{2}$ if the argument (u or v) is 0. Otherwise, the constants take the value 1.

$$F(u, v) = \frac{2C(u)C(v)}{\sqrt{MN}} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \cos \frac{(2i+1)u\pi}{2M} \cos \frac{(2j+1)v\pi}{2N} f(i, j) \tag{12.1}$$

In Eq. 12.1, $f(i, j)$ is the color value (YIQ or YUV) at the (i, j)th location of the image block. Note that in each block, $M = 8$ and $N = 8$. Thus, i and j vary from 0 to 7. Therefore, u and v, which varies with i and j, also vary between 0 to 7. Hence, the DCT equation in our case is given by Eq. 12.2.

$$F(u, v) = \frac{C(u)C(v)}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} f(i, j) \tag{12.2}$$

After the application of DCT on an image block, we get the 8×8 matrix of values known as the DCT coefficients (let us denote them by $F(u, v)$). These coefficients are conceptually of two types—the DC coefficient (the element at $u = 0$, $v = 0$) and the AC coefficients (remaining elements). The next step in JPEG compression is to *quantize* the coefficients. This is done by simply dividing each entry in the coefficient matrix by an integer and rounding off the result, as shown in Eq. 12.3.

$$\hat{F}(u, v) = \text{round}\left(\frac{F(u, v)}{Q(u, v)}\right) \tag{12.3}$$

There are standard 8×8 quantization matrices available for luminance and chrominance components (Table 12.1). These matrices are determined from psychophysical studies, which aim to maximize compression ratio while minimizing perceptual losses.

After quantization, we get the quantized matrix $\hat{F}(u, v)$. The matrix represents the quantized DC (row 0, column 0) and AC (remaining elements) coefficients. We encode the AC

Table 12.1 Standard quantization matrices (a) For luminance component (b) For chrominance component

(a)

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(b)

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

coefficients using the *run length coding* or RLC. We first perform a *zigzag* scan of the AC coefficients in the quantized matrix and encode the motion path with RLC. In RLC, we replace the matrix elements with the pairs (RUNLENGTH, VALUE), where RUNLENGTH indicates the number of zeroes in the motion path (*run of zeroes*) and VALUE denotes the next non-zero value after the run of zeroes. Likewise, the DC components of the image (one for each block) are encoded using the differential pulse code modulation or DPCM. In DPCM, a *predictor* maps the DC values to the encoded value. For example, a simple DPCM *predictor* for the ith block can be $d_i = dc_{i+1} - dc_i$, in which dc_{i+1} is the DC value of the $(i+1)$th block and dc_i is the DC value of the ith block, with $d_0 = dc_0$. After the coeffcients are encoded in this manner, an entropy coding scheme (either of Huffman coding and arithmetic coding), is applied on the coefficients. In order to decode a compressed JPEG image, we simply perform inverse operations of all the steps involved.

The organization of a JPEG image file is shown schematically in Fig. 12.3. The *frame* denotes the image. Each *scan* in the frame indicates a *pass* through the pixels (e.g., the red component). Each scan consists of *segments*. A segment is a group of 8×8 image blocks. The frame header contains various information that include bits per pixel, width and height of image, number of components, unique ID for each component, horizontal/vertical sampling

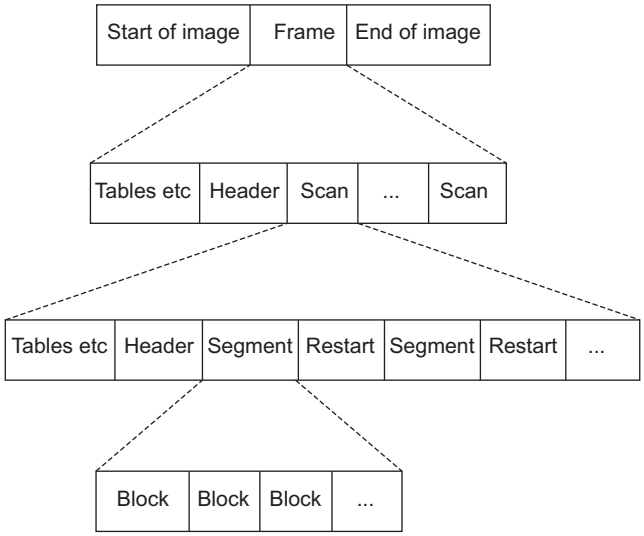


Fig. 12.3 Organization of JPEG image file

factors for each component, and quantization tables for each component. The scan header contains information such as the number of components in scan, component ID for each component, and Huffman/arithmetic coding tables for each component.

12.4.2 H.261 Digital Video Compression Standard

The volume of video data is usually huge—even a modest CIF video with picture resolution of 352 × 288 can have a size of more than 35 Mbps. Such large size has obvious implications for storage and network communication of multimedia data, necessitating the need for video data compression. As we discussed before, a video is a sequence of frames. Since the frame rate of a video is often relatively high and the camera parameters (focul length, position, viewing angle, etc.) usually do not change rapidly between frames, the contents of consecutive frames are usually similar. In other words, the video has *temporal redundancy*. Video compression works based on this idea of temporal redundancy. There are several standards available exclusively for video data compression. The eariest of those is the H.261, on which the subsequent standards (H.263 and H.264) are based. In this section, we shall learn the basics of H.261. The H.261 standard was developed by the International Telecommunication Union - Telecommunication Standardization sector (ITU-T) in 1990. It uses *motion compensation* for data compression. Let us first try to understand the idea of motion compensation in video data.

In most videos, the main cause of difference in data values between frames is the motion of camera and/or objects. This implies that the pixel values between frames do not change abruptly. There is a gradual change in values that results from the *displacement* of a group of pixels from one part of a frame to another part in another frame in the frame sequence. The idea is illustrated in Fig. 12.4. If we can predict this displacement (amount and direction) of the corresponding pixels in the frames, we can *compensate* for the motion (i.e., *regenerate* frames with fewer information). Video compression algorithms that use this idea are

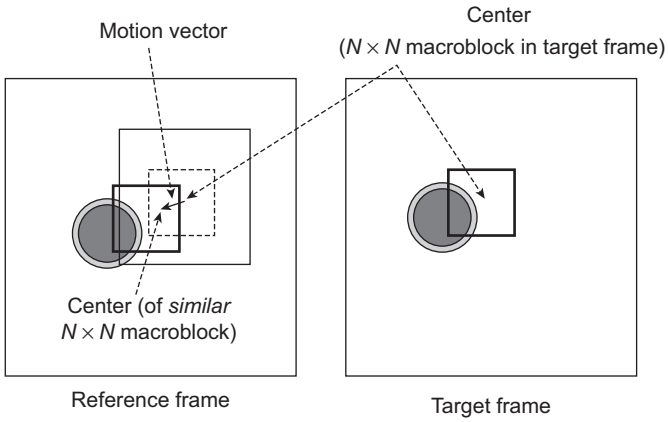


Fig. 12.4 The basic idea of motion compensation. Groups of pixels are assumed to be getting displaced across frames. We need to identify the amount and direction of this displacement (the motion vector).

said to be based on motion compensation (MC). In an MC-based technique, each image frame is divided into *macroblocks* of size $N \times N$ (typically, we use $N = 8$ or 16). The MC is performed at the macroblock level. The current image frame is referred to as the *target frame*. For a given target macroblock, we need to determine a *source* macroblock in a *previous* frame, which resulted in the target macroblock after *displacement*. In other words, we need to determine a *similar* macroblock in a previous frame and the *displacement* amount and direction. The *displacement* amount and direction of the reference macroblock to the target macroblock is known as the *motion vector*. Determination of the motion vector is a complex process and various algorithms are used to search for it. When we try to determine the motion vector from previous reference frame(s), it is called *forward prediction*. If the derivation is made from future frame(s), the technique is known as *backward prediction*. The *difference* of the target (predicted) and reference macroblocks is the prediction error. For MC-based compression, we need to encode only the motion vectors and the difference macroblocks after the first frame, rather than all the frames; they are sufficient for the decoder to regenerate all the frames.

With the basic idea and terminology of MC discussed here, let us now try to understand the H.261. The typical frame sequence in H.261 is shown in Fig. 12.5. The frames are divided

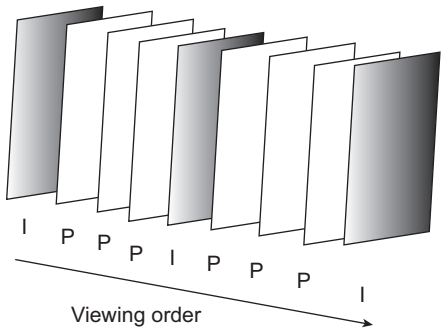


Fig. 12.5 H.261 frame sequence

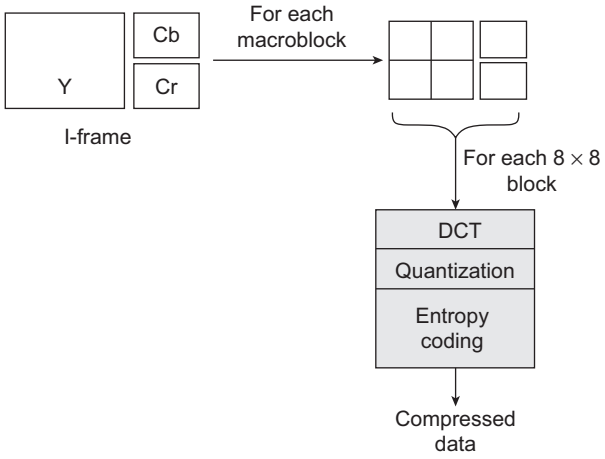


Fig. 12.6 Schematic illustration of I-frame coding in H.261

into two categories, intra-frames (I-frames) and inter-frames (P-frames). I-frames are treated as independent images. Each of these frames are encoded using a transform coding method similar to the methods used in JPEG. P-frames, however, are not considered to be independent. They are coded by forward predictive coding method in which the current macroblock is predicted from similar macroblocks in the preceding I- or P-frames. Also, the difference between the macroblocks are coded. In other words, temporal redundancy is exploited only in the P-frame coding, whereas I-frame coding exploits the spatial redundancy as in JPEG. The interval between pairs of I-frames is a variable and is determined by the encoder. Ordinary digital videos usually have two I-frames per second. Motion vectors are measured in pixels and have a limited range of ± 15.

The encoding of the I-frame is depicted in Fig. 12.6. Recall that the video color space is usually represented with the YCbCr model. For I-frame coding, we assume 16×16 Y-frame macroblocks and 8×8 Cb and Cr macroblocks. Also, we typically use 4:2:0 chroma sampling. Therefore, an I-frame macroblock consists of four Y block, one Cb block, and one Cr block. Each 8×8 sub-block of these component blocks is encoded using the same sequence of steps as in JPEG (i.e., DCT, quantization, zigzag-scan, and entropy coding).

The P-frames are encoded based on motion compensation. A motion vector is determined for each macroblock in the target frame. The prediction error is determined in terms of a *difference macroblock*. Each difference macroblock is composed of four Y, one Cb block, and one Cr block as in the case of I-frames. Each 8×8 sub-block is encoded following the steps of DCT, quantization, zigzag-scan, and entropy coding. If the prediction error turns out to be large (more than some acceptance level), the macroblock itself is encoded rather than the difference. The macroblock in that case is known as the *non-motion compensation macroblock*. Along with the difference macroblocks, we also encode the motion vector information. We first take the difference of the motion vectors of the preceding and the current macroblocks. This difference is then entropy-coded. The P-frame encoding is illustrated in Fig. 12.7.

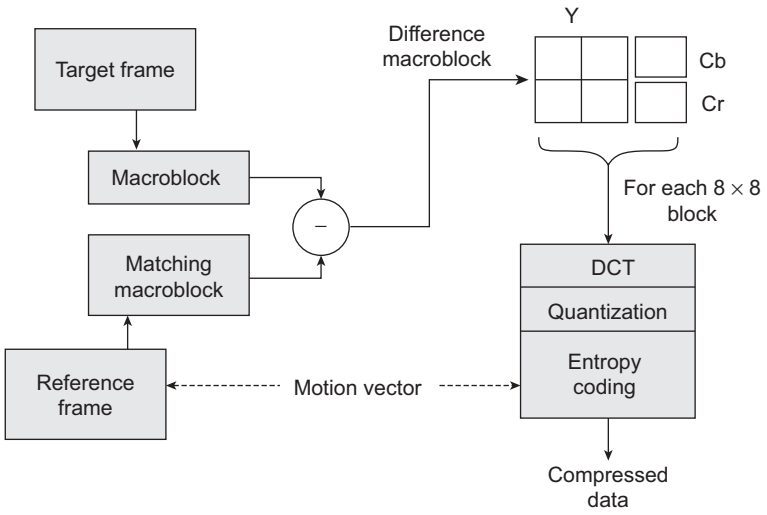


Fig. 12.7 Schematic illustration of P-frame coding in H.261

Unlike in JPEG, the quantization step in H.261 does not use values from a quantization matrix. Instead, a constant known as *step_size* is used for all DCT coefficients within a macroblock. For the DC coefficients in intra (I)-mode, we use $step\_size = 8$. In all the other cases, we can use any of the 31 *even* values in the integer range [2, 62]. Equation 12.4 shows the quantization step, where Q_H is the quantized value, DCT is the DCT coefficient and *scale* is any integer in the range [1, 31].

$$Q_H = \begin{cases} \text{round}\left(\frac{\text{DCT}}{8}\right) & \text{for DC coefficients in intra(I)-mode} \\ \text{round}\left(\frac{\text{DCT}}{2\times\text{scale}}\right) & \text{for all other DCT coefficients} \end{cases} \tag{12.4}$$

The H.261 video file (bitstream) structure is shown in Fig. 12.8. The bitstream consists of four layers: *picture*, *group of blocks or GOB*, *macroblock* and *block*. The picture layer information includes *picture start code* (PSC), a timestamp for the picture (*temporal reference* or TR), and *picture type* (pType) such as CIF or QCIF. Each picture is divided into regions of 11 × 3 macroblocks. These regions are called group of blocks or GOB. Each GOB has its *start code* (GBSC) and *group number* (GN). The quantizer is specified in GQ. Each macroblock has its address (indicating its position in the GOB), quantizer (MQ), and six 8 × 8 image blocks (4 Y, 1 Cb, and 1 Cr). *Type* indicates whether the macroblock is inter- or intra-, motion compensated or non-motion compensated. MVD is the motion vector data, determined from the difference of preceding and current macroblocks. The bitmask *coded block pattern* (CBP) is used to indicate the goodness of match to motion estimation. Only well-matched blocks will have their coefficients transmitted. Each 8 × 8 block starts with the DC value. This is followed by pairs of length of zero-run (*run*) and the next non-zero value after the run (*level*) for AC coefficients. The range of *run* is [0,63] while the *level* has range [−127, 127] and $level \neq 0$. Each block bitstream terminates with an *end of block*.

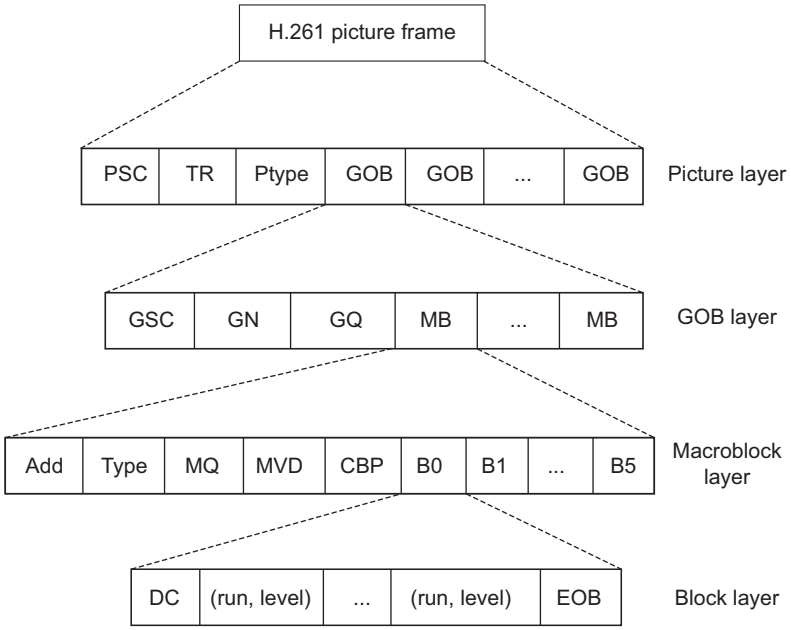


Fig. 12.8 Schematic illustration of H.261 video file (bitstream)

12.4.3 MPEG Standard

The Moving Pictures Experts Group (MPEG), established in 1988, recommended a series of compression standards for digital audio and video data. Their first proposal, approved by the International Organization for Standardization (ISO) in 1991, is called the MPEG-1. This was followed by subsequent standards, which are improvements of the MPEG-1. These include MPEG-2, MPEG-4, and MPEG-7. In this section, we shall discuss MPEG-1 to learn about the basics of the MPEG standards.

The MPEG-1, also referred to as ISO/IEC 11172, was proposed for coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbits/s. Out of the specified 1.5 Mbits/s, 1.2 MBits is intended for coded video and 256 kbps can be used for coded stereo audio data. The standard has five parts.

1. **Systems (11172-1):** This part takes care of dividing outputs into packets of bitstreams, multiplexing, and synchronization of video and audio streams.
2. **Video (11172-2):** The part is concerned with video compression standard.
3. **Audio (11172-3):** This part deals with the audio compression standard.
4. **Conformance (11172-4):** It specifies the design of tests to verify if the bitstream or decoder conforms to (complies with) the standard.
5. **Software (11172-5):** A complete software implementation of the MPEG-1 standard decoder and a sample software implementation of the encoder are also included as part of the standard.

The video encoding standard in MPEG-1 is based on motion compensation, as in H.261. Recall that in H.261, *forward prediction* is used for motion estimation. In forward prediction,

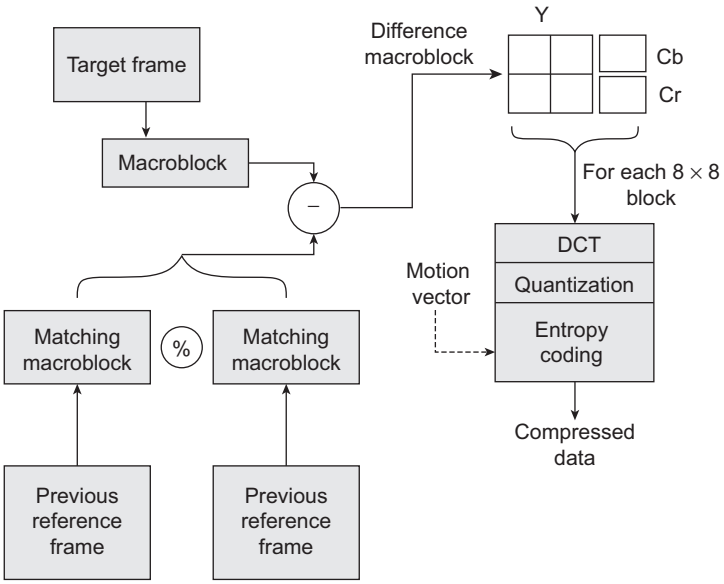


Fig. 12.9 The B-frame coding in MPEG-1. The averaging of two macroblocks is denoted by '%'.

each macroblock in the target P-frame is assigned the best matching macroblock from the previously coded I- or P-frame. However, it is possible that the target macroblock may not have any matching entity in the previous frame, due to unexpected movements and occlusions in a real scene. In such cases, we can get a match in the next frame. This is known as *backward prediction*. MPEG-1 supports both forward and backward prediction for motion estimation.

For the bidirectional (forward and backward) motion compensation, a third frame type - *B-frame* - is introduced in MPEG-1, in addition to the I- and P-frames. The idea is illustrated in Fig. 12.9. Each macroblock in a B-frame specifies up to *two* motion vectors - one from the forward prediction and one from the backward prediction. If matching in both directions (forward and backward) is successful, both the motion vectors are sent. The two corresponding matching macroblocks are averaged before comparing to the target macroblock for genertting the prediction error. If an *acceptable* match is found in only one of the reference frames, only one motion vector and its corresponding macroblock are used from either the forward or backward prediction.

A typical frame sequence of an MPEG video is illustrated in Fig. 12.10. The actual frame pattern is determined at encoding time and is specified in the frames' header. In MPEG specification, M is used to denote the interval between a P-frame and its preceding I- or P-frame. The number of B-frames in this interval is M-1. The interval between two consecutive I-frames is denoted by N. In Fig. 12.10, $M = 3$ and $N = 9$. Note that due to the use of backward prediction, MPEG encoder or decoder cannot work for any B-frame macroblock *without* its succeeding P- or I-frame. Thus, coding order of the frame is different from the display order, as illustrated in Fig. 12.10. Clearly, *buffering* or temoprary storage is an important issue in MPEG display.

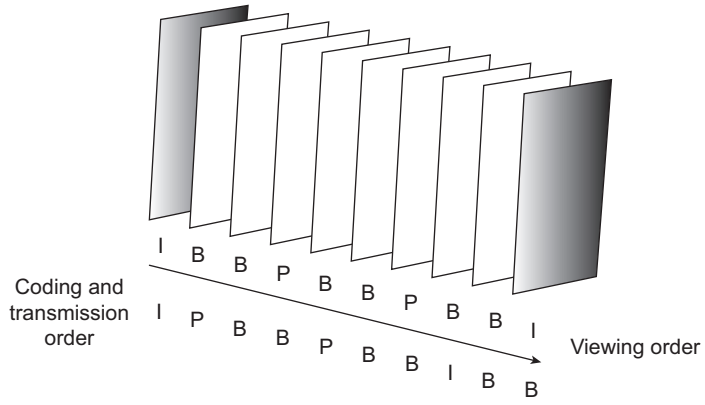


Fig. 12.10 A typical MPEG-1 frame sequence for display and coding

Table 12.2 MPEG-1 constrained parameter set

Parameter	Value
Horizontal size of picture	≤ 768
Vertical size of picture	≤ 576
Number of macroblocks/picture	≤ 396
Number of macroblocks/second	$\leq 9,900$
Frame rate	≤ 30 fps
Bitrate	$\leq 1,856$ kbps

In addition to the bidirectional prediction, there are a few more major differences between MPEG-1 video coding standard and the H.261. While H.261 supports only CIF (352×288) and QCIF (176×144) source formats, MPEG-1 supports various formats including SIF, provided a *constrained parameter set* shown in Table 12.2 is satisfied.

The MPEG-1 picture is also divided into one or more *slices* rather than GOBs as in H.261. Slices are more flexible and can contain a variable number of macroblocks. Quantization for intra (I)-coding is done using a quantization table (Table 12.3), unlike in H.261. The *step_size* in such case becomes $Q[i, j] \times scale$ where $Q[i, j]$ is the corresponding table entry and *scale* is an integer in the range $[1, 31]$. For inter (P)-coding, the *step_size* for quantization is $16 \times scale$.

The organization of MPEG-1 video file (bitsream) is shown in Fig. 12.11. It consists of six layers. The *sequence layer* at the top of the hierarchy consists of one or more groups of pictures (GOPs). The *header* contains picture information such as horizontal and vertical size, aspect ratio, frame rate, bit rate, quantization matrix, and so on. There can be an optional sequence header between GOPs to indicate parameter change. Each GOP (*GOP layer*) contains one or more pictures, one of which must be an I-picture. The GOP header contains information such as the hour-minute-second frame from the start of sequence. In the

Table 12.3 Standard quantization table for intra-coding in MPEG-1

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

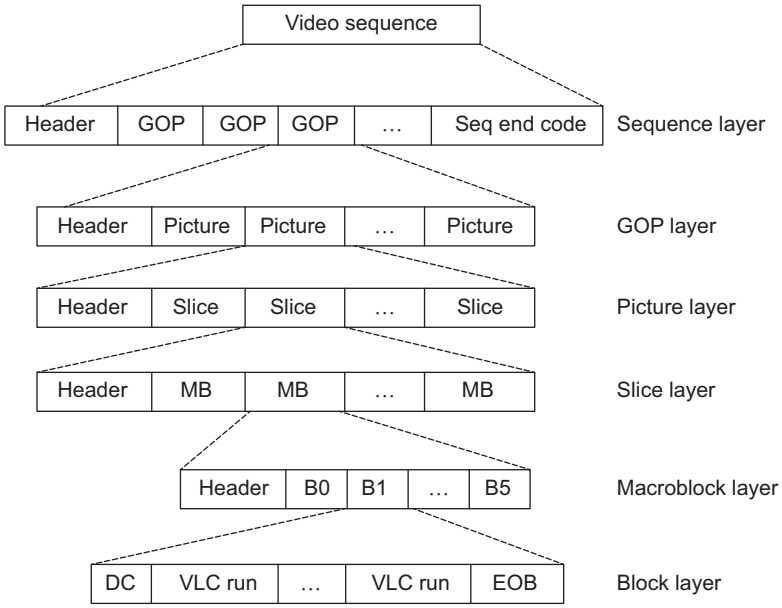


Fig. 12.11 Schematic of the MPEG video file (bitstream) organization

picture layer, the picture-level information is stored. MPEG-1 contains four picture types—there are the three common types; I-picture (intra-coding), P-picture (predictive coding), and B-picture (bidirectional predictive coding). In addtion, MPEG-1 also has a D-picture (DC-coded), in which only the DC coefficients are retained. Below the picture layer is the *slice layer*. In this layer, information such as the length (number of macroblocks) and position of slices are provided. Each macroblock in the macroblock layer consists of four Y blocks, one Cb block, and one Cr block (total six blocks). All blocks are of the size 8×8. If the blocks are intra-coded, the differential DC coefficient (DPCM of DCs, as in JPEG) is stored first, followed by variable-length codes (VLC) for the AC coefficients. Otherwise, DC and AC coefficients are both coded using VLCs.

SUMMARY

In this chapter, we introduced an important application area of computer graphics, namely the multimedia. As we saw, multimedia is a vast field. Broadly, there are four aspects in multimedia system design—multimedia content creation (with authoring tools), multimedia data representation (data compression techniques and standards), search and retrieval (involves data base management issues, file system design, and content-based retrieval), and transmission and delivery (all network-related issues including protocol, streaming server design, and QoS). Among them, we mainly discussed the first two (authoring and representation) in this chapter.

Multimedia refers to content made from different media such as text, image, audio, video, and animation put together. A closely related concept is hypermedia—multimedia content that can be accessed non-linearly. The most popular examples of hypermedia are the web pages. Web pages contain links that allow us to jump to different places/pages. In order to transfer hypermedia content over the Internet, the HTTP protocol is used. Also, markup languages such as HTML and XML are primarily used to create and publish hypermedia contents on the web.

Authoring of multimedia content is not an easy task. It typically involves a team of experts from different disciplines. The production cycle starts with storyboarding, followed by functional specification, prototyping, and testing. In order to design a production and implement various stages of the production cycle, multimedia authoring tools are used. The range of functionalities of these tools is wide—from simple video production facility to complete production environment. Authoring approaches for multimedia contents follows some *metaphors*. These include scripting language, slide show, hierarchy, flow-control, frames, card, and cast/score/scripting metaphors.

Among the various types of media, we already discussed about the basic representation techniques of texts, images, and animation in the previous chapters. In this chapter, we introduced the fundamentals of digital audio and video. Both these media are assumed to be represented using *signals*, although their nature are different. Ideas from signal processing domains are borrowed to represent and manipulate audio and video signals.

An important issue in multimedia is to have efficient representation of data, such that the storage, transmission, and manipulation become efficient. Typically, multimedia data is stored in compressed form. We briefly discussed about the fundamentals of data compression. There are two stages in any data compression method: compression or encoding and decompression or decoding. Two types of data compressions are possible. In lossless compression (e.g., entropy coding), no data element is lost during encoding. Thus, we can regenerate the exact original. In lossy compression (e.g., DCT coding), some data is lost during compression. Hence, we can generate only an approximation of the original. Using data compression techniques, many compression standards are designed and used in multimedia. We discussed three of those: one image (JPEG) and two video (H.261 and MPEG-1) data compression standards. We saw that all these standards are a combination of lossy and lossless compression techniques.

BIBLIOGRAPHIC NOTE

For general introduction to the various topics of multimedia, see Li and Drew [2004]. Hypermedia concepts are introduced in Nielsen [1995]. Digital video fundamentals are discussed in Birn [2000], Poynton [1996], and Tekalp [1995]. More on digital audio basics can be found in Pohlmann [2000]. Nielsen [1995] is a standard book on data compression. Also refer to Sayood [2000] for further reading on data compression. See also Gonzalez and Woods [2002] for more on mathematical transforms and image procesing and Stark and Woods [2001] for stochastic processes, which are required to understand data compression techniques. For details on JPEG

and JPEG2000 standards, refer to Pennebaker and Mitchell [1993] and Taubman and Marcellin [2002], respectively. For various image and video compression standards, refer to Bhaskaran and Konstantinides [1997].

KEY TERMS

B-frame – a special frame type used in MPEG
Backward prediction – prediction of motion path from next frames
Code/Codeword – a term used to refer to encoder outputs
Component video – video data comprising red, green, and blue components
Composite video – video data having a single wave combining luminance and chrominance values
Compression – a technique to reduce the number of bits required to represent data items
Compression ratio – the ratio representing the gain through compression
Decoder – a system to decompress and get back the data
Encoder – a system to convert original data to compressed form
Forward prediction – prediction of motion path from previous frames
Frequency modulation (FM) – a signal reconstruction technique
H.261 – a digital video data compression standard
HTML – hypertext markup language, used to publish hypermedia content on the web
HTTP – the hypertext transfer protocol used to transfer hypertexts over a network
Hypermedia – non-liner representation of multimedia content.
Hypertext – non-linear representation of text
I-frames – frames that are considered to be independent images in a video frame sequence
JPEG – a digital image compression standard
Lossless compression – compression techniques that do not lose any information during compression
Lossy compression – compression techniques in which some original information is lost
Metaphor – abstraction of real world process/methods
Motion compensation – video compression technique that tries to predict object motion path between frames
MPEG – a multimedia data compression standard
Multimedia – multiple contents or media combined together; also refers to systems that deal with multimedia content
Multimedia authoring/multimedia production – creation of multimedia contents
Nyquist rate – a special sampling frequency for signal reconstruction
P-frames – frames that are obtained through prediction preceding or next P-/I-frames
Predictive coding – the coding technique that makes use of prediction based on previous/subsequent decoded data and the error between real and predicted data
S-video – video data comprising luminance and composite chromatic values
Sampling – measuring signal amplitude values at specific time intervals
Sampling frequency – the rate (number of samples per unit time) of sampling
Transform coding – samples are transformed to a new basis space and then encoded with a lossless compression
Wave table synthesis (or wave method) – a signal reconstruction technique
World Wide Web (WWW) – the most popular hypermedia, which we often refer to as the Internet
XML – extensible markup language, an improved form of HTML
YCbCr – a color model used for video data representation
YIQ – a color model used in JPEG
YUV – a color model used in JPEG

EXERCISES

12.1 What is meant by the term 'multimedia'? How is computer graphics related to it?
12.2 Discuss the broad issues in multimedia.
12.3 Explain the term hypermedia. What differentiates it from multimedia?
12.4 Discuss the stages of a typical multimedia production cycle.
12.5 Explain the idea of *authoring metaphor*. Discuss any three metaphors used for multimedia content creation.
12.6 Write a short note on the relation between signal processing and handling of multimedia audio and video contents.
12.7 What is Nyquist rate? Why do we need to know it?
12.8 Discuss the main stages of JPEG compression, with illustrative diagrams. How is a JPEG file structured?
12.9 What are the main steps involved in H.261 video coding? Describe the format of a H.261 compliant video file.
12.10 Discuss the main differences of MPEG-1 video standard from H.261, with illustrative diagrams. How is the MPEG-1 video file structured?

APPENDIX

A Mathematical Background

Various mathematical concepts are involved in understanding the theories and principles of computer graphics. They include the idea of vectors and vector algebra, matrices and matrix algebra, tensors, complex numbers and quaternions, parametric and non-parametric representations and manipulations of curves, differential calculus, numerical methods, and so on. In order to explain the fundamental concepts of graphics in this book, we used some of those. The mathematics used in this book mostly involved vectors and matrices and how those are manipulated. In addition, we also used concepts such as *reference frames* and *line equations* for calculating intersection points between two lines. The backgrounds for these mathematical concepts are discussed in this appendix.

A.1 COORDINATE REFERENCE FRAMES

In the discussion on the pipeline stages, we mentioned about different (coordinate) *reference frames*. We primarily used the concepts of the Cartesian reference frames in our discussion. The Cartesian frames are characterized by the mutually perpendicular (orthogonal) axes, which are straight lines. Both 2D and 3D Cartesian frames are used to represent points at the different stages of the graphics pipeline.

Usually, the device-independent commands within a graphics package assume that a screen region represents the *first quadrant* of a 2D Cartesian reference frame in *standard* position (see Fig. A.1(a)), The lower-left corner of the screen is the coordinate origin. However, scan lines are numbered from top to bottom with the topmost scan line numbered 0. This means that the screen positions are represented *internally* assuming the upper-left corner of the screen to be the origin and the positive Y-axis points downwards, as shown in Fig. A.1(b). In other words, we make use of an *inverted* Cartesian frame. Note that the horizontal (X) coordinate values in both the systems are the same. The relationship between the vertical (Y) values of the two systems are shown in Eq. A.1, where y is the Y-coordinate value in the standard frame and y_{inv} is the Y-coordinate value in the inverted frame.

$$y = y_{max} - y_{inv} \tag{A.1}$$

In order to represent points in the three-dimensional space, usually the *right-handed* Cartesian frame is used (Fig. A.2). We can imagine the system as if we are trying to *grasp* the Z-axis with our right hand in a way such that the thumb points towards the positive

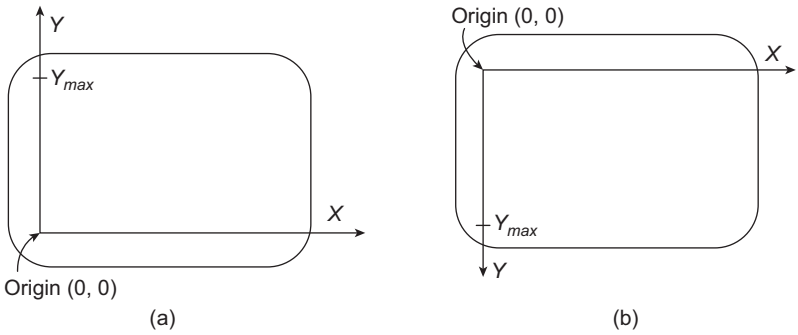


Fig. A.1 Standard and inverted 2D Cartesian reference frames used in computer graphics (a) Points are represented with respect to the origin in the lower-left corner (b) Points are represented with respect to the origin at the upper-left corner

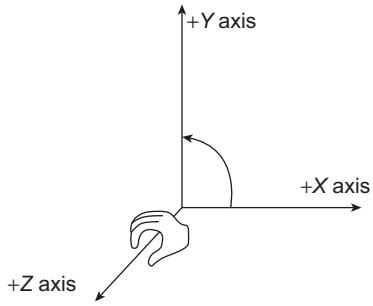


Fig. A.2 Right-handed 3D Cartesian coordinate system

Z direction. Hence the fingers are *curling* from the positive X direction to the positive Y direction (through 90°).

When a view of a 3D scene is displayed on a 2D screen, the 3D point for each 2D screen position is sometimes represented with the *left-handed* reference frame shown in Fig. A.3. Unlike the right-handed system, here we assume to *grasp* the Z axis with our left hand. Other things remain the same. That is, the left-hand thumb points towards the positive Z direction and the left-hand fingers *curl* from the positive X direction to the positive Y direction.

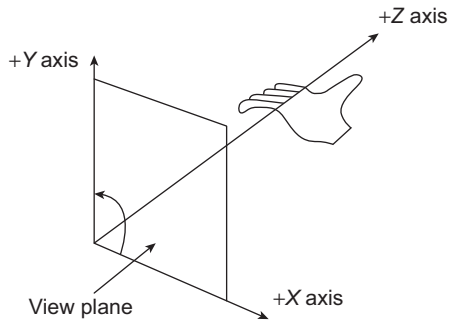


Fig. A.3 Left-handed 3D Cartesian coordinate system

The XY plane represents the screen. Positive Z values indicate positions *behind* the screen. Thus, the larger positive Z values indicate points *further* from the viewer.

A.2 VECTORS AND VECTOR OPERATIONS

A vector is a mathematical entity with two fundamental properties—*magnitude* and *direction*. In a chosen coordinate reference frame, we can use two points to define a vector. For example, consider Fig. A.4. It shows a vector in two dimensions in terms of the two points P_1 and P_2. Let us denote the coordinates of the two points as (x_1, y_1) and (x_2, y_2), respectively. Then, we can define the vector $\vec{V}$ as,

$$\begin{aligned}\vec{V} &= P_2 - P_1 \\ &= (x_2 - x_1, y_2 - y_1) \\ &= (V_x, V_y)\end{aligned}$$

The quantities V_x and V_y are the *projections* of the vector **V** onto the X- and Y-axis, respectively. They are called the Cartesian *components* (or Cartesian *elements*). The magnitude (denoted by $|\vec{V}|$) of the vector is computed in terms of these two components as,

$$|\vec{V}| = \sqrt{V_x^2 + V_y^2}$$

The direction can be specified in terms of the angular displacement α from the horizontal as,

$$\alpha = \tan^{-1}\left(\frac{V_x}{V_y}\right)$$

The idea of a 3D vector is similar. Suppose we have two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$. We now have three Cartesian components instead of two for the three axes: $V_x = (x_2 - x_1)$, $V_y = (y_2 - y_1)$, and $V_z = (z_2 - z_1)$ for the $X-$, $Y-$ and Z axis, respectively. Then, the magnitude of the vector can be computed as,

$$|\vec{V}| = \sqrt{V_x^2 + V_y^2 + V_z^2}$$

The vector direction can be given in terms of the *direction angles*, that is, the angles α, β, and γ the vector makes with each of the three axes (see Fig. A.5). More precisely, direction

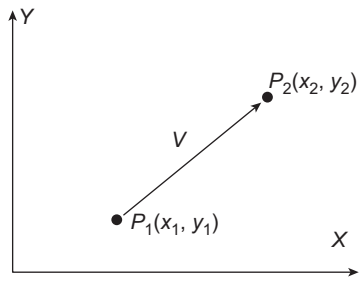


Fig. A.4 A 2D vector **V** defined in a Cartesian frame as the difference of two points

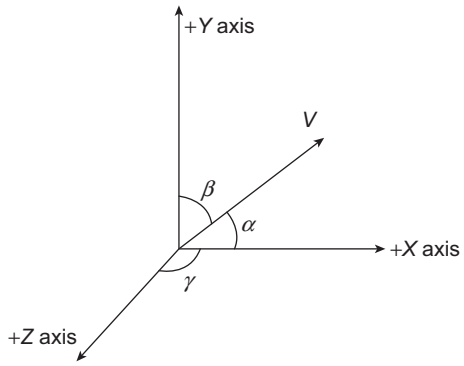


Fig. A.5 Three direction angles for a 3D vector

angles are the positive angles the vector makes with each of the positive coordinate axes. The three direction angles can be computed as,

$$\cos\alpha = \frac{V_x}{|V|}$$
$$\cos\beta = \frac{V_y}{|V|}$$
$$\cos\gamma = \frac{V_z}{|V|}$$

The values $\cos\alpha$, $\cos\beta$, and $\cos\gamma$ are known as the *direction cosines* of the vector. In fact, we need to specify any two of the three cosines to find the direction of the vector. The third cosine can be determined from the two since,

$$\cos^2\alpha + \cos^2\beta + \cos^2\gamma = 1$$

In many situations, we deal with *unit* vectors. It is a vector with magnitude 1. While we usually denote a vector with an arrow on top, such as $\vec{V}$, a unit vector is denoted by putting a *hat* on top of the vector symbol, such as $\hat{V}$. However, the most common notation for unit vector is $\hat{u}$. Calculation of the unit vector along the direction of a vector is easy. Suppose $\vec{V} = (V_x, V_y, V_z)$ be the given vector. Then the unit vector $\hat{V}$ along the direction of $\vec{V}$ is given by Eq. (A.2).

$$\hat{V} = \left(\frac{V_x}{|V|}, \frac{V_y}{|V|}, \frac{V_z}{|V|}\right) \tag{A.2}$$

where $|V| = \sqrt{V_x^2 + V_y^2 + V_z^2}$ is the magnitude of the vector.

A.2.1 Vector Addition and Scalar Multiplication

The addition of two vectors is defined as the addition of corresponding components. Thus, we can represent vector addition as follows.

$$\vec{V}_1 + \vec{V}_2 = (V_{1x} + V_{2x}, V_{1y} + V_{2y}, V_{1z} + V_{2z})$$

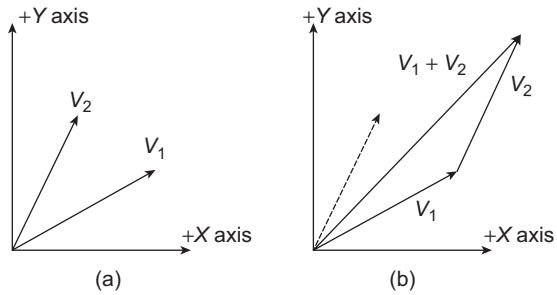


Fig. A.6 Illustration of vector addition (a) Original vectors. (b) $\vec{V}_2$ repositioned to start where $\vec{V}_1$ ends

The direction and magnitude of the new vector is determined from its components as before. The idea is illustrated in Fig. A.6 for 2D vector addition. Note that the second vector starts at the tip of the first vector. The resulting vector starts at the start of the first vector and ends at the tip of the second vector.

Addition of a vector with a scalar quantity is not defined, since a scalar quantity has only magnitude without any direction. However, we can multiply a vector with a scalar value. We do this by simply multiplying the scalar value to each of the components as follows:

$$s\vec{V} = (sV_x, sV_y, sV_z)$$

A.2.2 Multiplication of Two Vectors

We can multiply two vectors in two different ways—*scalar* or *dot* product and *vector* or *cross* product. In the case of a dot product of two vectors, we obtain a scalar value, whereas we get a vector from the cross product of two vectors.

The dot product of two vectors $\vec{V}_1$ and $\vec{V}_2$ is calculated as,

$$\vec{V}_1.\vec{V}_2 = |\vec{V}_1||\vec{V}_2|\cos\theta \qquad 0 \geq \theta \geq \pi$$

where θ is the smaller of the two angles between the vector directions. We can also determine the vector dot product in terms of their Cartesian components as,

$$\vec{V}_1.\vec{V}_2 = V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z}$$

Note that the dot product of a vector with itself produces the square of the vector magnitude. There are two important properties satisfied by the dot product. They are *commutative*. In other words, $\vec{V}_1.\vec{V}_2 = \vec{V}_2.\vec{V}_1$. Also, dot products are *distributive with respect to vector addition*. Thus, $\vec{V}_1.(\vec{V}_2 + \vec{V}_3) = \vec{V}_1.\vec{V}_2 + \vec{V}_1.\vec{V}_3$.

The cross product of two vectors is defined as,

$$\vec{V}_1 \times \vec{V}_2 = \hat{u}|\vec{V}_1||\vec{V}_2|\sin\theta \qquad 0 \geq \theta \geq \pi$$

In this expression, $\hat{u}$ is a *unit* vector (magnitude 1), which is perpendicular to both $\vec{V}_1$ and $\vec{V}_2$ (Fig. A.7). The direction for $\hat{u}$ is determined by the *right-hand rule*: we *grasp* with our

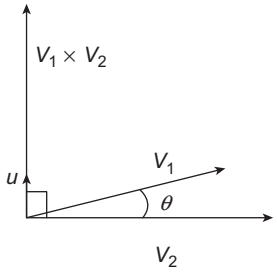


Fig. A.7 Illustration of the cross product of two vectors

right hand an axis that is perpendicular to the plane containing $\vec{V}_1$ and $\vec{V}_2$ such that the fingers *curl* from $\vec{V}_1$ to $\vec{V}_2$. Thus, the right thumb denotes the direction of $\hat{u}$. The cross product can also be expressed in terms of the Cartesian components of the constituent vectors as,

$$\vec{V}_1 \times \vec{V}_2 = (V_{1y}V_{2z} - V_{1z}V_{2y}, V_{1z}V_{2x} - V_{1x}V_{2z}, V_{1x}V_{2y} - V_{1y}V_{2x})$$

The cross product of two parallel vectors is zero. Therefore, the cross product of a vector with itself is zero. It is also not commutative since $\vec{V}_1 \times \vec{V}_2 = -(\vec{V}_2 \times \vec{V}_1)$. However, cross product of two vectors is distributive with respect to vector addition similar to the dot product, that is, $\vec{V}_1 \times (\vec{V}_2 + \vec{V}_3) = \vec{V}_1 \times \vec{V}_2 + \vec{V}_1 \times \vec{V}_3$.

A.3 MATRICES AND MATRIX OPERATIONS

A matrix is a rectangular array of *elements*, which can be numerical values, expressions, or even functions. We have already encountered several examples of matrices in the book. In general, we can represent a matrix $\mathbf{M}$ with r rows and c columns as,

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1c} \\ m_{21} & m_{22} & \dots & m_{2c} \\ \vdots & \vdots & & \vdots \\ m_{r1} & m_{r2} & \dots & m_{rc} \end{bmatrix}$$

where m_{ij} represents the elements of $\mathbf{M}$. As per the convention, the first subscript of any element denotes the row number and the column number is given by the second subscript.

A matrix with a single row or single column represents a vector (the elements represent the coordinate components of the vector). When a vector is represented as a single-row matrix, it is called a *row vector*. A single column matrix is called a *column vector*. Thus, a matrix can also be considered as a collection of row or column vectors.

A.3.1 Scaler Multiplication and Matrix Addition

In order to multiply a matrix $\mathbf{M}$ with a scalar value s, we multiply each element m_{ij} with s. For example, if

$$\mathbf{M} = \begin{bmatrix} 3 & 2 & 1 \\ 2 & 1 & 3 \\ 1 & 3 & 2 \end{bmatrix}$$

then

$$2\mathbf{M} = \begin{bmatrix} 6 & 4 & 2 \\ 4 & 2 & 6 \\ 2 & 6 & 4 \end{bmatrix}$$

Two matrices can be added only if they both have the same number of rows and columns. In order to add two matrices, we simply add their corresponding elements. Thus,

$$\begin{bmatrix} 3 & 2 & 1 \\ 2 & 1 & 3 \\ 1 & 3 & 2 \end{bmatrix} + \begin{bmatrix} 6 & 4 & 2 \\ 4 & 2 & 6 \\ 2 & 6 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 6 & 3 \\ 6 & 3 & 9 \\ 3 & 9 & 6 \end{bmatrix}$$

A.3.2 Matrix Multiplication

Given two matrices **A** (dimension $m \times n$) and **B** (dimension $p \times q$), we can multiply them if and only if $n = p$. In other words, the number of columns in **A** should be the same as the number of rows in **B**. The resultant matrix $\mathbf{C} = \mathbf{AB}$ will have the dimension $m \times q$. We can obtain the elements of **C** (c_{ij}) from the elements of **A** (a_{ij}) and **B** (b_{ij}) as,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

In the following example, we multiplied a 2×3 matrix with a 3×2 matrix to obtain the 2×2 matrix.

$$\begin{bmatrix} 3 & 2 & 1 \\ 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} 6 & 4 \\ 4 & 2 \\ 2 & 6 \end{bmatrix} = \begin{bmatrix} 28 & 22 \\ 22 & 22 \end{bmatrix}$$

Note that matrix multiplication is not commutative: $\mathbf{AB} \neq \mathbf{BA}$. However, it is distributive with respect to matrix addition. That is, $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$.

A.3.3 Matrix Transpose

Given a matrix **M**, its transpose $\mathbf{M}^T$ is obtained by interchanging rows and columns. For example,

$$\begin{bmatrix} 3 & 2 & 1 \\ 1 & 3 & 2 \end{bmatrix}^T = \begin{bmatrix} 3 & 1 \\ 2 & 3 \\ 1 & 2 \end{bmatrix}$$

We can also define transpose of a matrix product as,

$$(M_1 M_2)^T = M_2^T M_1^T$$

A.3.4 Determinant of a Matrix

A useful concept in operations involving matrices is the *determinant* of a matrix. Determinant is defined only for square matrices (i.e., those matrices having the same number of rows and columns). The *second-order determinant* for a 2×2 square matrix $\mathbf{M}$ is defined as,

$$\mathbf{det}\ \mathbf{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

Higher-order determinants are obtained recursively from the lower-order determinant values. In order to calculate a determinant of order 2 or greater of an $n \times n$ matrix $\mathbf{M}$, we select any column k and compute the determinant as,

$$\mathbf{det}\ \mathbf{M} = \sum_{j=1}^{n} (-1)^{j+k} m_{jk} \mathbf{det}\ \mathbf{M}_{jk}$$

where *det* $\mathbf{M}_{jk}$ is the $(n-1)$ by $(n-1)$ determinant of the submatrix obtained from $\mathbf{M}$ after removing the jth row and kth column. Alternatively, we can select any row j and calculate the determinant as,

$$\mathbf{det}\ \mathbf{M} = \sum_{j=1}^{n} (-1)^{j+k} m_{jk} \mathbf{det}\ \mathbf{M}_{jk}$$

Efficient numerical methods exist to compute determinants for large matrices ($n > 4$).

A.3.5 Matrix Inverse

Another useful matrix operation is the *inverse* of a matrix. This is again defined for only square matrices. Moreover, a square matrix can have an inverse if and only if the determinant of that matrix is *non-zero*. If an inverse exists, we call the matrix *non-singular*. Otherwise, it is a *singular* matrix.

For an $n \times n$ matrix $\mathbf{M}$, its inverse is usually denoted by $\mathbf{M}^{-1}$. The matrix and its inverse also satisfy the relation,

$$\mathbf{M}\mathbf{M}^{-1} = \mathbf{M}^{-1}\mathbf{M} = I$$

where $\mathbf{I}$ is the *identity* matrix. Only the diagonal elements of $\mathbf{I}$ are 1 and all other elements are zero.

We can calculate the elements of $\mathbf{M}^{-1}$ from the elements of $\mathbf{M}$ as,

$$m_{jk}^{-1} = \frac{(-1)^{j+k} \mathbf{det}\ \mathbf{M}_{kj}}{\mathbf{det}\ \mathbf{M}}$$

where m_{jk}^{-1} is the element of the jth row, kth column of the inverse matrix. $\mathbf{M}_{kj}$ is the $(n-1)$ by $(n-1)$ submtarix obtained by deleting the kth row, jth column of $\mathbf{M}$. Usually, more efficient numerical methods are employed to compute the inverse of large matrices.

A.4 LINE EQUATION AND INTERSECTION CALCULATION

In clipping algorithms (Chapter 7), we saw the need for determining line-boundary intersection points. How do we do that? Suppose we are given a line segment specified by two end points **P** (x_1, y_1) and **Q** (x_2, y_2). We can use the *point-slope* form (Eq. A.3) to derive the line equation.

$$y - y_1 = m(x - x_1) \tag{A.3}$$

In Eq. A.3, m is the *slope* (or *gradient*) of the line, it indicates how *steep* the line is. We can derive m in terms of the two end points as in Eq. A.4.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{A.4}$$

For example, suppose **P** (2,3) and **Q** (4,5). Then $m = \frac{5-3}{4-2} = 1$. Therefore, we can have the line equation as: $y - 3 = 1(x - 2)$.

Another way to represent a line is the *slope-intercept* form: $y = mx + c$. In the equation, c is known as the *intercept* (of the line with the Y-axis). We can recast the *point-slope* form to the *standard* form by rearranging the terms in Eq. A.3, as follows:

$$y = y_1 + mx - mx_1 = mx + (y_1 - mx_1) = mx + c$$

Let us illustrate this with the previous example. We have the *point-slope* form $y - 3 = 1(x-2)$. After expnading, we get $y-3 = x-2$. We rearrange the terms to get $y = x-2+3$ or $y = x + 1$. This is the standard form.

Now suppose we are given two line segments: L_1 $(y = m_1x + c_1)$ and L_2 $(y = m_2x + c_2)$. If the lines intersect, there must be a common point (x, y), which lies on both the lines. Therefore, the following relation must hold.

$$m_1x + c_1 = m_2x + c_2$$

From this, we derive the common x-coordinate as,

$$x = \frac{c_2 - c_1}{m_1 - m_2}$$

We can substitute this value of x in any of the line equations to get the common y-coordinate as,

$$y = m_1\frac{c_2 - c_1}{m_1 - m_2} + c_1 \text{ using } L_1 \text{ or}$$
$$y = m_2\frac{c_2 - c_1}{m_1 - m_2} + c_2 \text{ using } L_2$$

When we are considering a vertical line, the line equation is given as $x = \text{b}$ where b is some constant. Now, suppose we are trying to calculate the intersection point of a line $y = mx + \text{c}$ with a vertical line $x = \text{b}$. We substitute b for x in the line equation to get the

y-coordinate as $y = m\mathrm{b} + \mathrm{c}$. Thus, the intersection point is (b, mb + c). In the case of a horizontal line, we have its equation as $y = \mathrm{b}$, where b is a constant. We substitute this value for y in the line equation to compute the x-coordinate as: $b = m\mathrm{x} + \mathrm{c}$ or $x = \frac{\mathrm{b}-\mathrm{c}}{m}$. Hence, the intersection point is ($\frac{\mathrm{b}-\mathrm{c}}{m}$, b).

A.5 PLANE EQUATION AND PLANE NORMAL

At many places of the 3D graphics pipeline (such as lighting and hidden surface removal), we have to deal with 3D surfaces and surface normals. The implicit form of a surface in general is given by Eq. A.5.

$$f(x, y, z) = 0 \tag{A.5}$$

For any point $\mathbf{P}(x, y, z)$ on the surface, Eq. A.5 should evaluate to zero, that is, $f(\mathbf{p}) = 0$. For points that are not on the surface, Eq. A.5 returns some non-zero value. However, for the purpose of this book, we shall restrict ourselves to the discussion of plane surfaces only, rather than any arbitrary curved surface. This is so since we mostly considered objects with polygonal surfaces. We also know that any surface can be represented as a mesh of polygonal surfaces.

The most familiar way to represent a planar surface if the *point-normal* form, shown in Eq. A.6.

$$\mathrm{A}x + \mathrm{B}y + \mathrm{C}z + \mathrm{D} = 0 \tag{A.6}$$

Equation A.6 is also known as the *general form* of a plane equation. Let $\vec{n}$ be the *normal* to the plane, that is, a vector perpendicular to the planar surface. Then, the constants A, B, and C in the plane equation (Eq. A.6) denote the corresponding Cartesian components $\vec{n}$. In other words, $\vec{n} = (a, b, c)$.

Sometimes, we want to derive a plane equation given three points on the plane (say **a**, **b**, and **c**). Each of these points can be represented as a point vector, that is, a vector from the origin to the point. Thus, we can form three point vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$. From the three point vectors, we can derive two vectors that *lie* on the plane. For example, the vectors $(\vec{b} - \vec{a})$ and $(\vec{c} - \vec{a})$ are two vectors on the plane.

Since the two vectors are on the plane, we can take their cross-product to obtain a vector that is perpendicular to the two vectors, that is, the plane itself. Since the vector is perpendicular to the plane, it is the normal vector. Thus, $\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$. Once we know $\vec{n}$, we have the three constants A, B, and C. We then use Eq. A.6 by putting any one of the three points in the equation to obtain the value of D.

Let us illustrate the idea with an example. Suppose we are given the three points **a**(1, 1, 1), **b**(3, 4, 5), and **c**(7, 7, 7). Therefore, the three point vectors are,

$$\vec{a} = (1, 1, 1)$$
$$\vec{b} = (3, 4, 5)$$
$$\vec{c} = (7, 7, 7)$$

From these three point vectors, we form two vectors that lie *on* the plane, as follows.

$$\vec{b} - \vec{a} = (3, 4, 5) - (1, 1, 1) = (2, 3, 4)$$
$$\vec{c} - \vec{a} = (7, 7, 7) - (1, 1, 1) = (6, 6, 6)$$

The cross-product of these two vectors yield (see Section A.2 for details),

$$(\vec{b} - \vec{a}) \times (\vec{c} - \vec{a}) = (2, 3, 4) \times (6, 6, 6) = (-6, 12, -6)$$

Therefore, the normal to the plane is $\vec{n} = (-6, 12, -6)$. Thus, the three plane constants are $A = -6$, $B = 12$, $C = -6$. Replacing these values in Eq. A.6, we get,

$$-6x + 12y - 6z + D = 0$$

Now let us take any one of the three points, say **b**(3, 4, 5). Since the point is on the plane, we should have

$$-6.3 + 12.4 - 6.5 + D = 0$$

From this equation, we get $D = 0$
Hence, the plane equation is $-6x + 12y - 6z = 0$ or $-x + 2y - z = 0$.

APPENDIX

B Fractals

In computer graphics, many a times we need to generate shapes that occur in nature. For example, mountains, trees, tree leaves, shorelines and so on. Of course, we can use simple primitives such as lines or triangles to approximate these shapes. For instance, simple triangles may be used to generate a mountain. However, such crude approximations affect the quality of the image. In fact, we can do much better, by applying a more sophisticated modelling technique for such shapes. The technique is known as the *fractal method*, which is a *procedural* technique. In a procedural technique, we do not use equations to represent objects unlike in techniques such as boundary representation (Chapter 2). Instead, we use a procedure following a particular set of rules. As the name suggests, the fractal method is based on the notion of *fractal objects* or simply *fractals*. Let us first try to understand the concept of a fractal.

B.1 CONCEPT OF DIMENSION AND FRACTAL

There are two defining characteristics of a fractal:

1. There are *infinite* details at every point on the object.
2. There is a certain *self-similarity* between object parts and the overall object features.

The first point means that the closer you look at a fractal, you will get to see more and more details. The second implies that the details will look similar to the original. However, the self-similarity property is not unique to the fractals. Many other non-fractal objects also exhibit the same. For example, consider a straight line segment with length l. Enlarge the line segment by a scaling factor $k = 2$. You get a new line segment of length $2l$, which is the same as having two line segments of length l. In other words, when we use a scaling factor 2 on a line segment, we get *two* line segments having the same length as that of the original. They are also *similar* to the original shape. Thus, we can say that, as a result of the enlargement, we get an object having *two* subparts, each of which is similar to the original (Fig. B.1). Clearly, this is an example of an object, which, when looked at closely, reveals subparts that are similar to the original. Let us denote by N, the number of subparts we get, that are similar to the original.

Let us consider another example, a square. We enlarge all its sides by a factor of two (i.e., $k = 2$). We get four new squares, each of which is similar to the original (Fig. B.2). Thus, in this case we have $N = 4$. In a likewise manner, we can show that for a cube, we get $N = 8$ for $k = 2$ (Fig. B.3).

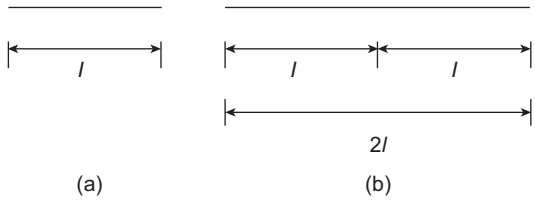


Fig. B.1 Example of line self-similarity (a) Original line (b) Enlarged line ($k = 2$) Note that the enlarged line consists of two self-similar line segments

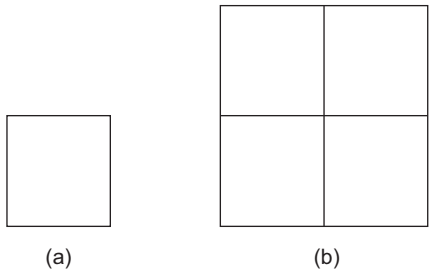


Fig. B.2 Example of square self-similarity (a) Original square (b) Enlarged square ($k = 2$)

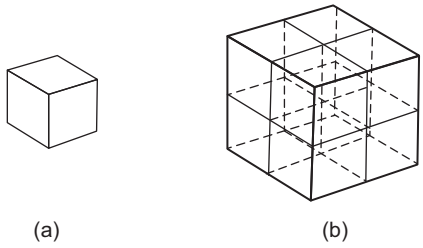


Fig. B.3 Example of cube self-similarity (a) Original cube (b) Enlarged cube ($k = 2$)

Thus, all these well-known objects satisfy the self-similarity property. But we know for sure that they are not fractals. Clearly, we require some other property to distinguish between fractals and non-fractals. This is provided by the notion of *dimension*. We all are familiar with the term and have some understanding of what it means. However, we require a more formal definition of it.

Consider the aforementioned case of the square. We saw that $N = 4$ when $k = 2$ for the square. We can therefore relate the two as follows.

$$k^2 = N \tag{B.1}$$

In other words,

$$\log_k N = 2 \tag{B.2}$$

The term $\log_k N$ is defined as the *dimension*. Therefore, we can define the following.

1. For a line, the dimension is $= \log_2 2 = 1$.
2. For a square, the dimension is $= \log_2 4 = 2$.
3. For a cube, the dimension is $= \log_2 8 = 3$.

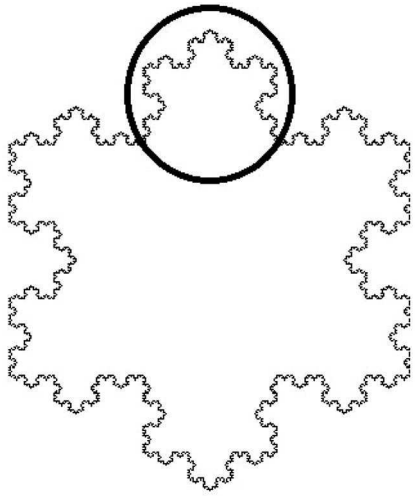


Fig. B.4 The Koch snowflake, made up of *six* sides. One of the sides is shown inside the circle.

Note that for all these objects, we have *integer* dimensions. A fractal, on the other hand, has *fractional dimension* (hence the name). Let us consider an example fractal for illustration: the Koch snowflake. The object is shown in Fig. B.4. As the name suggests, it can be used to model snowflakes in a computer-generated scene. So, how can we generate it?

As Fig. B.4 shows, the fractal is made up of six sides. One of the sides is shown in the figure as an example. Each of these six sides is a fractal in itself, known as the *Koch curve*. So, if we can generate the Koch curves and put six of them together as in Fig. B.4, we will get the *snowflake*. How can we generate the Koch curve?

We can actually write an iterative procedure to generate a Koch curve. The procedure starts with an initial shape (called an *initiator*). The initiator for a Koch curve is simply a straight line (Fig. B.5a), having unit length. After the first iteration, the initiator is transformed to the *hat* shape shown in Fig. B.5(b). The shape is obtained with the following two steps.

1. Put an eqilateral triangle whose base is the *middle* one-third of the line.
2. Remove the triangle base.

Clearly, the object we get after the first iteration has a length $= 4 \times \frac{1}{3} = \frac{4}{3}$. In the second iteration, we perform the same operation on *each* of the four line segments of the *hat*. Thus, we get the shape shown in Fig. B.5(c). So, what is the length of the new shape?

Each line segment of the *hat* shape after the first iteration has a length of $\frac{1}{3}$. In the second iteration, each of these line segments is replaced with the same *hat* shape. Thus, each line segment of the *hats*, after the second iteration, has length $= \frac{\frac{1}{3}}{3} = \frac{1}{9}$. There are 16 such line segments in the overall shape (comprising four *hats*). Therefore, total length of the object is $= 16 \times \frac{1}{9} = \frac{16}{9} = (\frac{4}{3})^2$.

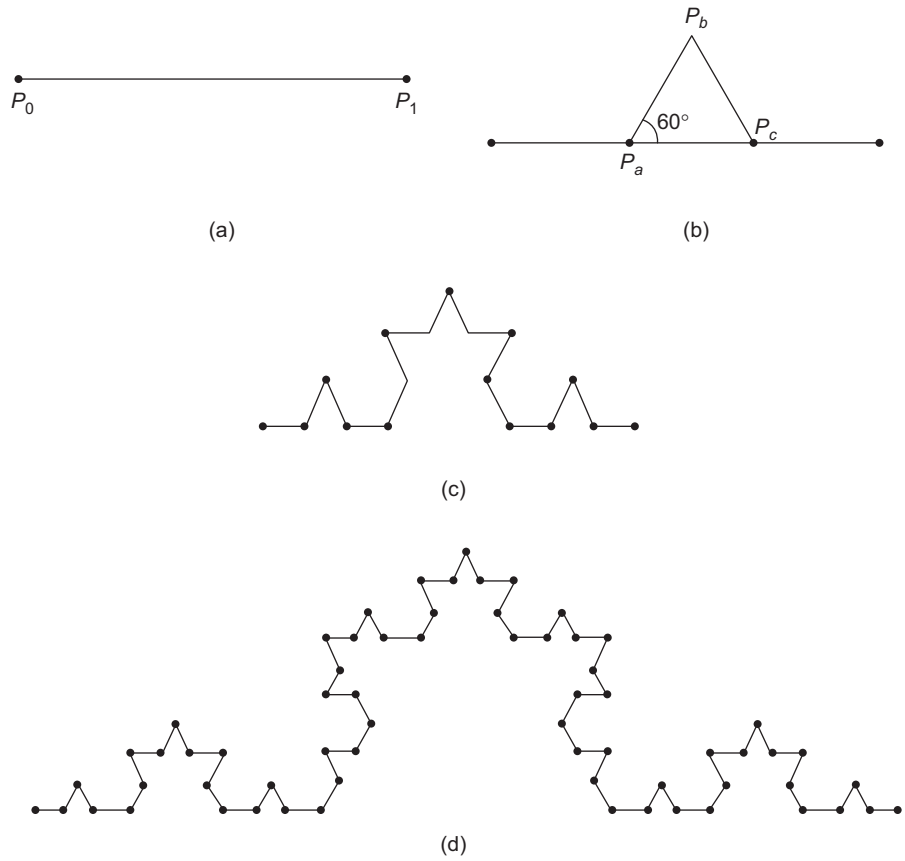


Fig. B.5 The first three iterations for the generating the Koch curve

In the third iteration, we again replace each of the 16 line segments with a new *hat* shape to get the overall shape shown in Fig. B.5(d). Using a similar argument as before, we can show that the length of this new object is $= (\frac{4}{3})^3$. In general, after the nth iteration, we get an object with length n. Thus, if we go for a very large number of iterations (∞), the length of the object will also be very large ($= (\frac{4}{3})^\infty = \infty$).

As you can see, the length of the Koch curve is actually very very large (∞). When we put six of them together to create the Koch snowflake (Fig. B.4), the perimeter of the snowflake is also very very large. However, the area it bounds is finite. Now, let us try to understand why the Koch curve is a fractal.

Consider Fig. B.5(d) again. It shows the curve after three iterations. Note that the curve after the third iteration consists of four components, each of which is similar to the original. In other words, we can say that there are four sub-parts that are similar to the original, if we enlarge it by a factor of three (i.e., $N = 4$ when $k = 3$). Therefore, the dimention of a Koch curve is,

$$\log_3 4 = 1.261 \tag{B.3}$$

For a Koch curve, the dimension is fractional. Hence, it is a fractal. In general, it is not very easy to determine fractal dimensions. Sophisticated methods are used for the purpose. We will not discuss those here as they are out of scope of this book. However, interested readers may refer to the references mentioned at the end of Chapter 2 (Bibliographic Note) for more information.

Depending on the dimension value, we can categorize fractals into the following classes:

Planar fractal curve If a fractal lies completely within a two-dimensional plane, it has dimension $D > 1$. The closer D is to 1, the smoother the fractal curve is. When $D = 2$, we get a *Peano curve*, which completely *fills* a finite region of 2D space. Sometimes, we can have a fractal lying in a 2D plane with $2 < D < 3$. In such cases, the curve self-intersects. Planar fractal curves are good for modelling natural boundaries such as shorelines.

Fractal surface For a fractal surface, we have (typically) $2 < D \leq 3$. If $D = 3$, the surface *fills* a volume of space. We get overlapping coverage of the volume enclosed by the surface if $D > 3$. We can use fractal surfaces to model objects such as terrain, clouds, and water.

Fractal solid The fractal solids usually have $3 < D \leq 4$. If $D > 4$, we have self-overlapping objects. With fractal solids, we can model sophisticated object properties such as water vapor density or temperature within a cloud.

B.2 GENERATION OF FRACTALS

As we mentioned at the begining of this chapter, fractals are generated through some *iterative procedures*. What these procedures do in general is to repeatedly apply a *specified* transformation function ($F()$) to points within a region of space. So, if $P_0 = (x_0, y_0, z_0)$ is a selected initial position, each iteration of the procedure applies F to generate successive levels of detail as follows:

$$P_1 = F(P_0), P_2 = F(P_1), P_3 = F(P_2), \ldots$$

The way this general idea is implemented depends on the type of fractal we are dealing with. Broadly, there are the following three types.

Self-similar fractals When we *zoom-in* on a fractal, subparts become visible (e.g., the koch snowflake discussed in the preceding section). If these subparts are scaled-down version of the original (as in the Koch curve), with the scaling factor same along all directions (x, y, z), the fractal is called self-similar. These fractals are good for modelling objects such as trees, tree leaves, shrubs, etc.

Self-affine fractals If the fractal subparts are scaled-down versions of the original and the scaling factor is different in each coordinate (x, y, or z) direction, the fractal is called self-affine. These are useful to model terrain, water, and clouds.

Invariant fractal sets These are fractal objects formed with *non-linear transformations* (e.g., self-squaring, self-inverse). Examples of such fractals include the Mandelbrot set and the Julia set. Usually, they are not preferred for generating useful shapes.

Among these three, self-similar fractals are generated in the simplest way. The generation starts with an initial shape (called the *initiator*) and a geometric pattern (called the *generator*). In each iteration of the fractal generation procedure, the subparts of the initiator is replaced (transformed) with the generator. We have already seen the working of this approach in the preceding section, in the generation of the Koch curve. In that case, the initiator was the straight line whereas the generator was the triangle without base in the *hat*. Generation of the other two fractal types are more complex. Interested readers may use the references cited at the end of Chapter 2 (Bibliographic Note) for more information.

APPENDIX C

Ray-tracing Method for Surface Rendering

In Chapter 4, we learnt the simple lighting model for computing color at surface points. As we discussed, the model is based on many simplistic assumptions. For example, the assumption that all surfaces are ideal reflectors, light path does not shift during refraction, or the ambient lighting effect can be modeled by a single number. Consequently, the model is incapable of producing realistic effects. What we require is a *global illumination model*, that is, a model of illumination that takes into account *all* the reflections and refractions that affect the color at any particular surface point. In this chapter, we shall learn about one such model known as *ray tracing*. Obviously, the computational cost for implementing this model is much higher.

C.1 RAY-TRACING: BASIC IDEA

Generating or synthesizing an image on a computer screen is equivalent to computing pixel color values. We can view this process in a slightly different way. In a scene, there are light rays from the surfaces emanating in all directions. Some of these rays are passing through the screen pixels towards the viewer. Thus, a pixel color is determined by the color of the light ray passing through it. In the ray-tracing method, we trace these rays *backward*—from the viewer towards the source. The idea is illustrated in Fig. C.1.

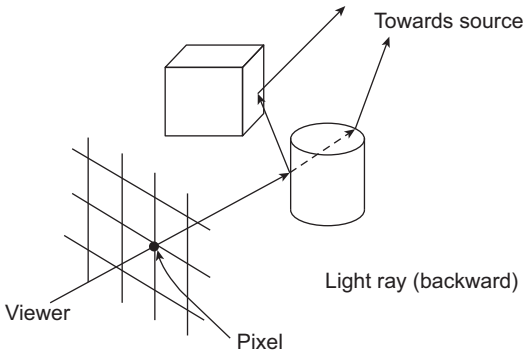


Fig. C.1 The basic idea of ray tracing. Each original ray passing through a pixel towards the viewer is traced backward (from viewer to source).

C.1.1 Steps Involved

The first step in the ray-tracing method is the generation of rays (from the viewer to the light source) passing through the pixels. These are called *pixel rays*. There is one pixel ray for each pixel. Depending on the type of projection we want, the rays differ. If we are interested in parallel projection, the rays are generated perpendicular to the view plane (screen). If perspective projection is required, we create rays starting from a common *projection point* (i.e., center of projection, see Chapter 6). In both the cases, rays are assumed to pass through the pixel centers.

Next, for each pixel ray, we check if it intersects any of the surfaces present in the scene. For this purpose, we maintain a list of all surfaces. For each member of the list, we check for ray-surface intersection. If for a surface, an intersection is found, we calculate the distance of the surface from the pixel. Among all those intersecting surfaces, we choose the one having the least distance. This is the *visible surface*. The pixel ray used for detecting visible surface is called the *primary ray*.

Once we detect the visible surface with the primary ray, we perform reflection and refraction of the primary ray. We treat the primary ray as the incident ray. At the (primary ray–visible surface) intersection point, the primary ray is reflected along the *specular reflection direction* (or the ideal reflection direction; see Chapter 4). In addition, if the surface is transparent, the ray is transmitted through the surface along the refraction direction. As we can see, two more rays are generated off the intersection point due to the reflection and refraction. These are called the *secondary rays*. We then treat each secondary ray as primary ray and repeat the procedure *recursively* (i.e., determine the visible surface and generate new secondary rays from the intersection point).

We maintain a binary *ray-tracing tree* to keep track of the primary and secondary rays. Surfaces serve as nodes of the tree. The left edges denote a reflection path between the two surfaces. Right edges denote the refraction path. The idea is illustrated in Fig. C.2. As the

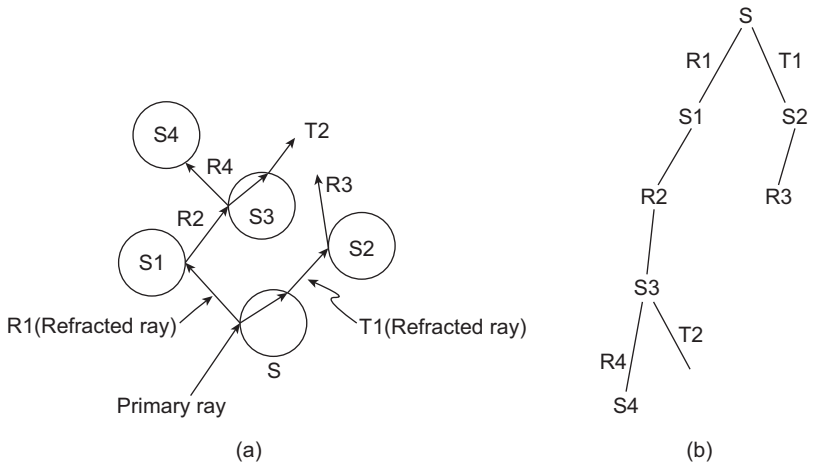


Fig. C.2 The construction of a binary ray-tracing tree. In (a), we show the backward tracing of the ray path through a scene having five surfaces. The corresponding ray-tracing tree is shown in (b).

figure shows, at each step of the recursive process, at most two new nodes and edges are added to the tree. We can set the maximum depth of the tree as a user option, depending on the available storage. The recursive process stops if any of the following conditions is satisfied:

1. The current primary ray intersects no surface in the list.
2. The current primary ray intersects a light source, which is *not* a reflecting surface.
3. The ray-tracing tree has reached its maximum depth.

At each ray–surface intersection point, we compute the intensity using a lighting model. There are the following *three* components in the intensity.

Local contribution The intensity contribution due to the light source

Reflected contribution The intensity contribution due to the light that comes after reflection from other surfaces

Transmitted contribution The intensity contribution due to the light that comes after transmitting through the surface from the background

Thus, the total light intensity at any surface point I can be computed as,

$$I = I_l + I_r + I_t \tag{C.1}$$

In the equation, I_l is the local contribution. We can use the simple lighting model discussed in Chapter 4 to compute it. For this calculation, we require the three vectors $\vec{N}$ (the surface normal at the point), $\vec{V}$ (the vector from the surface point to the viewer, i.e., along the opposite direction of the primary ray), and $\vec{L}$ (the vector from the point to the light source). In order to determine $\vec{L}$, we send a ray from the intersection point towards the light source. This ray is called the *shadow ray*. We check if the shadow ray intersects any surface in its path. If it does, the intersection point is in shadow with respect to the light source. Hence, we do not require calculation of actual intensity due to the light source. Instead, we can apply some technique (e.g., ambient light/texture pattern, see Chapter 4) to create the shadow effect. The other two components in the Eq. C.1 (I_r and I_t) are calculated recursively using the steps mentioned before. The intensity value at each intersection point is stored at the corresponding surface node position of the ray-tracing tree.

Once the tree is complete for a pixel, we accumulate all the intensity contributions starting at the leaf nodes. Surface intensity from each node is *attenuated* (see Chapter 4) by the distance from the parent node and then added to the intensity of the parent surface. This bottom-up procedure continues till we reach the root node. The root node intensity is set as the pixel intensity. For some pixel rays, it is possible that the ray does not intersect any surface. In that case, we assign background color to the pixel. It is also possible that, instead of any surface, the ray intersects a (non-reflecting) light source. The light source intensity is then assigned to the pixel.

C.1.2 Ray–Surface Intersection Calculation

The prerequisite for the ray–surface intersection calculation is the *representation* of pixel rays. How we can represent a ray? Note that a ray is *not* a vector. This is because a vector is defined having a direction and a magnitude. However, in the case of a ray, it has a *starting point* and a *direction*. Thus, we cannot use a single vector to represent it. Instead, we can use a sum of the following two vectors to represent a ray.

1. The ray origin vector $\vec{s}$, which is a vector from the coordinate origin to the ray origin. The ray origin point can be chosen as either the pixel position or the point of projection (for perspective projection).
2. The ray direction vector $\vec{d}$, which is along the ray direction. Usually, we use the unit direction vector $\hat{d}$.

In terms of these two vectors, we can represent a ray in parametric form as shown in Eq. C.2.

$$\vec{r}(t) = \vec{s} + t\hat{d}, t \geq 0 \tag{C.2}$$

How to determine $\hat{d}$? In the case of parallel projection, it is simply the unit normal vector of the XY plane (view plane). For perspective projection, we require little more computation. With respect to the coordinate origin, let us denote the *point of projection* by the vector $\vec{Pr}$ and the pixel point by the vector $\vec{Px}$. Then, the unit vector $\hat{d}$ should be along the direction from $\vec{Pr}$ to $\vec{Px}$. Hence, we can compute $\hat{d}$ as,

$$\hat{d} = \frac{\vec{Px} - \vec{Pr}}{|\vec{Px} - \vec{Pr}|} \tag{C.3}$$

The various vectors for perspective projection is shown in Fig. C.3 for illustration. In order to determine the ray–surface intersection point, we simultaneously solve the ray equation and the surface equation. This gives us a value for the parameter t from which the intersection coordinates are determined. At each intersection point, we update $\vec{s}$ and $\hat{d}$ for each of the secondary rays. The new $\vec{s}$ is the vector from origin to the intersection point. For the reflected ray, the unit vector $\hat{d}$ is calculated along the specular reflection direction. For the ray due to refraction, $\hat{d}$ is determined along the refraction path.

Let us illustrate the ray–surface intersection calculation in terms of an example—intersection of a ray with a sperical surface. Assume that the sphere center is at the point p_c

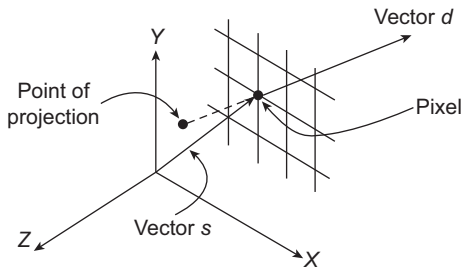


Fig. C.3 Representing a ray with the two vectors—perspective projection

and the length of its radius is r. Then, for any point p on the surface, the surface can be represented with the equation,

$$|\vec{p} - \vec{p_c}| = r \tag{C.4}$$

If the ray intersects the surface, there should be a common surface point. Thus, we can replace the point in the surface equation with the corresponding ray equation.

$$|\vec{s} + t\vec{d} - \vec{p_c}| = r \tag{C.5}$$

Next, we square both the sides in Eq. (C.5) to get,

$$|\vec{s} + t\hat{d} - \vec{p_c}|^2 = r^2 \tag{C.6}$$

After expanding and rearranging Eq. (C.6), we get

$$t = \frac{-B \pm \sqrt{B^2 - A.C}}{A} \tag{C.7}$$

where, $A = |\hat{d}|^2 = 1$, $B = (\vec{s} - \vec{p_c}).\hat{d}$, and $C = |\vec{s} - \vec{p_c}|^2 - r^2$. Depending on the values of A, B and C, we have the following scenarios:

1. If $B^2 - A.C < 0$, there is no intersection between the surface and the ray.
2. If $B^2 - A.C = 0$, the ray *touches* the surface. Clearly, there will be no secondary rays generated in this case.
3. If $B^2 - A.C > 0$, the ray intersects the surface. There are two possible parameter values as per Eq. C.7.
   (a) If both values are negative, there is no ray–surface intersection.
   (b) If one of the values is zero and the other positive, the ray originates *on* the sphere and intersects it. Usually in graphics, we are not interested in modelling such cases.
   (c) If the two values differ in sign, the ray originates *inside* the sphere and intersects it. This is again a mathematical possibility, which is usually not considered in graphics.
   (d) If both are positive, the ray intersects the sphere twice (enter and exit). The smaller value corresponds to the intersection point that is closer to the starting point of the ray. Thus, we take the smaller value to determine the intersection coordinates.

C.2 REDUCING INTERSECTION CALCULATION

For commonly occurring shapes such as spheres, cubes, and splines, efficient ray–surface intersection algorithms have been developed. However, as we can see, it takes a lot of computation to check for intersections of all the pixel rays with all the surfaces present in the scene. In fact, intersection calculations take up about 95% of the time it requires to render a scene using the ray-tracing method. Therefore, it is obvious that we try to reduce the intersection calculations as much as possible.

Various techniques are used in the ray-tracing method to speed up the surface rendering process. We can broadly divide these techniques into the following two groups:

1. Bounding volume techniques
2. Spatial subdivision methods

C.2.1 Bounding Volume Techniques

In the bounding volume technique, a group of closely placed objects in the scene are assumed to be *enclosed* by a *bounding volume*. Usually regular shapes such as spheres or cubes are used for bounding volume. Before checking for ray–surface intersection, we first check if the ray intersects the bounding volume. Only if the bounding volume is intersected by the ray, do we go for ray–surface intersection checks for all the surfaces in the bounding volume. Otherwise, we remove all the enclosed surfaces from further intersection checks.

The basic bounding volume approach can be extended to the *hierarchical* bounding volume. In that case, we create a hierarchy of bounding volumes. The ray first checks for intersection with the top-level bounding volume. If an intersection is found, the volumes in the next level are checked for intersection. The process goes on till the lowest level of the hierarchy.

C.2.2 Spatial Subdivision Methods

In the spatial subdivision approach, we enclose the entire scene within a cube. Then, we recursively divide the cube into cells. The subdivision can proceed in either of the two ways.

Uniform subdivision At each subdivision step, we divide the current cell into eight equal-sized octants.

Adaptive subdivision At each subdivision step, we divide the current cell only if it contains surfaces.

The recursion continues till each cell contains no more than a predefined number of surfaces. The process is similar to the space subdivision method we discussed in Chapter 2. We can use octrees to store the subdivision. Along with the subdivision information, information about surfaces stored in the cells are also maintained.

First, we check for intersection of the ray with the outermost cube. Once an intersection is detected, we check for intersection of the ray with the inner cubes (next level of subdivision) and continue in this way till we reach the final level of subdivision. We perform this check for only those cells that contain surfaces. In the final level of subdivision, we check for intersection of the ray with the surfaces. The *first* surface intersected by the ray is the *visible surface*.

C.3 ANTI-ALIASED RAY TRACING

In the ray-tracing method, we take discrete samples (pixels) to depict a continuous scene. Clearly, the phenomenon of *aliasing* (see Chapter 9) is an important issue in ray tracing also.

We need some means (anti-aliasing techniques) to eliminate or reduce the effect of aliasing. There are broadly two ways of antialiasing in ray tracing.

1. Supersampling
2. Adaptive sampling

In supersampling, each pixel is assumed to represent a finite region. The pixel region is divided into subregions (subpixels). Instead of a single pixel ray, we now generate pixel rays for each of these subregions and perform ray tracing. The pixel color is computed as the *average* of the color values returned by all the subpixel rays.

The basic idea behind the adaptive sampling is as follows: we start with *five* pixel rays for each pixel instead of one as before. Among these five, one ray is sent through the pixel center and the remaining four rays are sent through the four *corners* (assuming each pixel is represented by a square/rectangular region) of the pixel. Then we perform color computation using the ray-tracing method for each of these five rays. If the color values returned by them are *similar*, we do not divide the pixel further. However, in case the five rays return *dissimilar* color values, we divide the pixel into a 2×2 subpixel grid. We then repeat the process for each subpixel in the grid. The process terminates when a preset level of subdivision is reached.

Bibliography

T. Akenine-Moller and E. Haines, *Real-time Rendering*, A.K. Peters, Natick, MA, Second edition, 2002.

J. Arvo, (ed.), *Graphics Gems II*, Academic Press, San Diego, CA, 1991.

R.H. Bartels, J.C. Beatty, and B.A. Barsky, *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann, San Francisco, CA, 1987.

R. Barzel, *Physically-based Modeling for Computer Graphics*, Academic Press, San Diego, CA, 1992.

V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers, Boston, Second edition, 1997.

J. Birn, *[Digital] Lighting and Rendering*, New Riders Publishing, Indianapolis, IN, 2000.

G. Bishop and D.M. Wiemer, 'Fast phong shading' in *Computer Graphics (Proceedings of SIGGRAPH 1986)*, Volume 20, pp. 103–106, 1986.

J.F. Blinn, 'A trip down the graphics pipeline: The homogeneous perspective transform', *IEEE Computer Graphics and Applications*, 13(3):75–80, 1993.

J.F. Blinn and M.E. Newell, 'Clipping using homogeneous coordinates', in *Computer Graphics (Proceedings of SIGGRAPH 1978)*, Volume 12, pp. 245–251, 1978.

D.L. Bouquet, 'An interactive graphics application to advanced aircraft design', in *Computer Graphics (Proceedings of SIGGRAPH 1978)*, Volume 12, pp. 330–335, 1978.

J.E. Bresenham, 'Algorithm for computer control of a digital plotter', *IBM Systems Journal*, 4(1):25–30, 1965.

J.E. Bresenham, 'A linear algorithm for incremental digital display of circular arcs', *Communications of the ACM*, 20(2):100–106, 1977.

P. Brunet and I. Navazo, 'Solid representation and operation using extended octrees', *ACM Transactions on Graphics*, 9(2):170–197, 1990.

N. Burtnyk and M. Wein, 'Computer generated key frame animation', *Journal of the Society of Motion Picture and Television Engineers*, 8(3):149–153, 1971.

N. Burtnyk and M. Wein, 'Interactive skeleton techniques for enhancing motion dynamics in key frame animation', *Communications of the ACM*, 19(10):564–569, 1976.

E. Catmull, 'The problems of computer-assisted animation', in *Computer Graphics (Proceedings of SIGGRAPH 1978)*, Volume 12, pp. 348–353, 1978.

J.C. Chung, M.R. Harris, F.P. Brooks, H. Fuchs, M.T. Kelly, J. Hughes, M. Ouh-young, C. Cheung, R.L. Holloway, and M. Pique, Exploring virtual worlds with head-mounted displays', in *Proceedings of SPIE Conference on Three-dimensional Visualization and Display Technologies*, Volume 1083, pp. 15–20, 1989.

R.L. Cook, L. Carpenter, and E. Catmull, 'The reyes image rendering architecture', in *Computer Graphics (Proceedings of SIGGRAPH 1987)*, Volume 21, pp. 95–102, 1987.

F.C. Crow, 'A comparison of antialiasing techniques', *IEEE Computer Graphics and Applications*, 1(1):40–49, 1981.

M. Cyrus and J. Beck, 'Generalized two- and three-dimensional clipping', *Computers and Graphics*, 3(1):23–28, 1978.

C. De Boor, *A Practical Guide to Splines*, Springer-Verlag, Berlin, 2001.

O. Demers, *[Digital] Texturing & Painting*, New Riders Publishing, Indianapolis, IN, 2002.

H.J. Durrett, (ed.), *Color and the Computer*, Academic Press, Boston, 1987.

R.A. Earnshaw, (ed.), *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, Berlin, 1985.

P. Ekman and W. Friesen, *Facial Action Coding System: A Technique for the Measurement of Facial Movement*, Consulting Psychologists Press, Palo Alto, 1978.

G. Elber and E. Cohen, 'Hidden-curve removal for free form surfaces', in *Computer Graphics (Proceedings of SIGGRAPH 1990)*, Volume 24, pp. 95–104, 1990.

R. Fernando, (ed.), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics*, Addison-Wesley, Reading, MA, 2004.

K.P. Fishkin and B.A. Barsky, 'A family of new algorithms for soft filling', in *Computer Graphics (Proceedings of SIGGRAPH 1984)*, Volume 18, pp. 235–244, 1984.

J.D. Foley, A. Van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: Principles and Practice in C*. Addison-Wesley Professional, Second edition, 1995.

W.R. Franklin and M.S. Kankanhalli, 'Parallel object-space hidden surface removal', in *Computer Graphics (Proceedings of SIGGRAPH 1990)*, Volume 24, pp. 87–94, 1990.

A. Fujimoto and K. Iwata, 'Jag-free images on raster displays', *IEEE Computer Graphics and Applications*, 3(9):26–34, 1983.

T.N. Gardner and H.R. Nelson, 'Interactive graphics developments in energy exploration', *IEEE Computer Graphics and Applications*, 3(2):33–34, 1983.

A.S. Glassner, (ed.), *Graphics Gems*, Academic Press, San Diego, CA, 1990.

A.S. Glassner, *Principles of Digital Image Synthesis*, Volumes 1–2, Morgan Kaufmann, San Francisco, CA, 1995.

R.C. Gonzalez and R.E. Woods, *Digital Image Processing*, Prentice Hall, Upper Saddle River, NJ, Second edition, 2002.

D. Gordon and S. Chen, 'Front-to-back display of BSP trees', *IEEE Computer Graphics and Applications*, 11(5):79–85, 1991.

H. Gouraud, 'Continuous shading of curved surfaces', *IEEE Transactions on Computers,* Volume C-20(6):623–628, 1971.

S.L. Grotch, 'Three-dimensional and stereoscopic graphics for scientific data display and analysis', *IEEE Computer Graphics and Applications.*, 3(8):31–43, 1983.

P. Haeberli and K. Akeley, 'The accumulation buffer: Hardware support for high-quality rendering', in *Computer Graphics (Proceedings of SIGGRAPH 1990)*, Volume 24, pp. 309–318, 1990.

R. Hall, *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York, 1989.

D. Hearn and M.P. Baker, 'Scientific visualization: An introduction', Technical report, Eurographics '91 Technical Report Series, Tutorial Lecture 6, Vienna, Austria, 1991.

D. Hearn and M.P. Baker, *Computer Graphics with OpenGL*, Pearson, 2004.

P.S. Heckbert, (ed.), *Graphics Gems IV*, Academic Press Professional, Cambridge, MA, 1994.

M.R. Kappel, 'An ellipse-drawing algorithm for faster displays', in *Fundamental Algorithms for Computer Graphics*, pp. 257–280, Springer-Verlag, Berlin, 1985.

D. Kirk, (ed.), *Graphics Gems III*, Academic Pres, San Diego, CA, 1992.

D. Kirk and J. Arvo, 'Unbiased sampling techniques for image synthesis', in *Computer Graphics (Proceedings of SIGGRAPH 1991)*, Volume 25, pp. 153–156, 1991.

D.E. Knuth, 'Digital halftones by dot diffusions', *ACM Transactions on Graphics*, 6(4):245–273, 1987.

J.U. Korien and N.I. Badler, 'Temporal antialiasing in computer-generated animation', in *Computer Graphics (Proceedings of SIGGRAPH 1983)*, Volume 17, pp. 377–388, 1983.

J. Lasseter, 'Principles of traditional animation applied to 3D computer animation', in *Computer Graphics (Proceedings of SIGGRAPH 1987)*, Volume 21, pp. 35–44, 1987.

Z. Li and M.S. Drew, *Fundamentals of Multimedia*, Prentice Hall of India, 2004.

Y.D. Liang and B.A. Barsky, 'An analysis and algorithm for polygon clipping', *Communications of the ACM*, 26(11):868–877, 1983.

Y.D. Liang and B.A. Barsky, 'A new concept and method for line clipping', *ACM Transactions on Graphics*, 3(1):1–22, 1984.

B.B. Mandelbrot, *Fractals: Form, Chance and Dimension*, Freeman Press, San Francisco, 1977.

A. Menache, *Understanding Motion Capture for Computer Animation and Video Games*, Morgan Kaufmann, New York, 2000.

J.B. Mitroo, N. Herman, and N.I. Badler, 'Movies from music: Visualizing music compositions', in *Computer Graphics (Proceedings of SIGGRAPH 1979)*, Volume 13, pp. 218–225, 1979.

M. Mortenson, *Geometric Modeling*, John Wiley & Sons, New York, 1985.

B. Naylor, J. Amanatides, and W. Thibault, 'Merging BSP trees yield polyhedral set operations', in *Computer Graphics (Proceedings of SIGGRAPH 1990)*, Number 4, pp. 115–124, 1990.

T.M. Nicholl, D.T. Lee, and R.A. Nicholl, 'An efficient new algorithm for 2D line clipping: Its development and analysis', in *Computer Graphics (Proceedings of SIGGRAPH 1987)*, Volume 21, pp. 253–262, 1987.

J. Nielsen, *Multimedia and Hypertext: The Internet and Beyond*, Morgan Kaufmann, 1995.

A.W. Paeth, (ed.), *Graphics Gems V*, Morgan Kaufman, San Diego, CA, 1995.

R. Parent, *Computer Animation: Algorithms and Techniques*, Elsevier, Second edition, 2008.

W.B. Pennebaker and J.L. Mitchell, *The JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.

M. Pharr and R. Fernando, (ed.), *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, Addison-Wesley, Reading, MA, 2005.

B.T. Phong, 'Illumination for computer-generated images', *Communications of the ACM*, 18(6):311–317, 1975.

K.C. Pohlmann, *Principles of Digital Audio*, McGraw-Hill, New York, Fourth edition, 2000.

C.A. Poynton, *A Technical Introduction to Digital Video*, Wiley, New York, 1996.

W. Reeves, 'In-betweening for computer animation utilizing moving point constraints', in *Computer Graphics (Proceedings of SIGGRAPH 1981)*, Volume 15, pp. 263–270, 1981.

W.T. Reeves, 'Particle systems: A technique for modeling a class of fuzzy objects', *ACM Transactions on Graphics*, 2(2):91–108, 1983.

A.A.G. Requicha and J.R. Rossignac, 'Solid modeling and beyond', *IEEE Computer Graphics and Applications*, 12(5):31–44, 1992.

E. Ribble, 'Synthesis of human skeletal motion and the design of a special-purpose processor for real-time animation of human and animal figure motion', *Master's thesis*, Ohio State University, Columbia, Ohio, June 1982.

D.F. Rogers and J.A. Adams, *Mathematical Elements for Computer Graphics*, McGraw-Hill, New York, 1990.

R.J. Rost, *OpenGL Shading Language*, Addison-Wesley, Reading, MA, 2004.

K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann, San Francisco, 2000.

T. Sederberg, 'Free-form deformation of solid geometric models,' in *Computer Graphics (Proceedings of SIGGRAPH 1986)*, Volume 20, pp. 151–160, 1986.

M. Segal, 'Using tolerances to gurantee valid polyhedral modeling results', in *Computer Graphics (Proceedings of SIGGRAPH 1990)*, Volume 24, pp. 105–114, 1990.

S. Sherr, *Electronic Displays*, John Wiley & Sons, New York, 1993.

A. Shilling and W. Strasser, 'Exact: Algorithm and hardware architecture for an improved A-buffer', in *Computer Graphics (Proceedings of SIGGRAPH 1993)*, pp. 85–92, 1993.

D. Shreiner, J. Neider, M. Woo, and T. Davis, *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, Fourth edition, 2004.

R.F. Sproull and I.E. Sutherland, 'A clipping divider', in *AFIPS Fall Joint Computer Conference*, 1968.

H. Stark and J.W. Woods, *Probability and Random Processes with Application to Signal Processing*, Prentice Hall, Upper Sadle River, NJ, 2001.

I.E. Sutherland and G.W. Hodgman, 'Reentrant polygon clipping', *Communications of the ACM*, 17(1):32–42, 1974.

L.E. Tannas Jr, (ed.), *Flat-panel Displays and CRTs*, Van Nostrand Reinhold, New York, 1985.

D.S. Taubman and M.W. Marcellin, *JPEG2000: Image Compression Fumdamentals, Standards and Practice*, Kluwer Academic Publishers, Norwell, MA, 2002.

A.M. Tekalp, *Digital Video Processing*, Prentice Hall, Upper Saddle River, NJ, 1995.

D. Travis, *Effective Color Displays*, Academic Press, London, 1991.

K. Turkowski, 'Antialiasing through the use of coordinate transformations', *ACM Transactions on Graphics*, 1(3):215–234, 1982.

L. Velho and J.D.M. Gomes, Digital halftoning with space-filling curves', in *Computer Graphics (Proceedings of SIGGRAPH 1991)*, Volume 25, pp. 81–90, 1991.

K. Weiler, 'Polygon comparison using a graph representation', in *Computer Graphics (Proceedings of SIGGRAPH 1980)*, Volume 14, pp. 10–18, 1980.

K. Weiler and P. Atherton, 'Hidden-surface removal using polygon area sorting', in *Computer Graphics (Proceedings of SIGGRAPH 1977)*, Volume 11, pp. 214–222, 1977.

C. Welman, 'Inverse kinematics and geometric constraints for articulated figure manipulation', *Master's thesis*, Simon Frasier University, 1993.

X. Wu, 'An efficient antialiasing technique', *Computer Graphics (Proceedings of SIGGRAPH 1991)*, 25(4):143–152, 1991.

G. Wyszecki and W.S. Stiles, *Color Science*, John Wiley & Sons, New York, 1982.

K. Yamaguchi, T.L. Kunii, and F. Fujimura, 'Octree-related data structures and algorithms,' *IEEE Computer Graphics and Applications*, 4(1):53–59, 1984.

Index

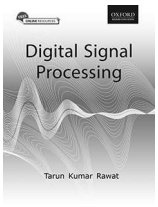
About the Author

Dr Samit Bhattacharya is Assistant Professor at the Department of Computer Science and Engineering, IIT Guwahati. He did his MS (by research) and Ph D, both in the area of Computer Science and Engineering, from IIT Kharagpur. Dr Bhattacharya has more than 6 years of teaching experience at both the undergraduate and postgraduate levels.

An active researcher in the field of human computer interaction (HCI), Dr Bhattacharya has more than three dozen publications in reputed international journals, book volumes, and conferences under his credit. He has also received several awards and honors from both the government and the industry for his research and teaching.

Dr Bhattacharya jointly holds an Indian patent right for *Sanyog*—an augmentative and alternative communication (AAC) system for Indians afflicted with neuromotor impairments. More details about him can be found at his website: http://www.iitg.ernet.in/samit/.

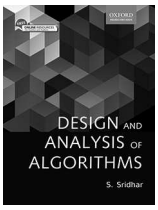
Related Titles



Digital Signal Processing (9780198081937)
Tarun Kumar Rawat, Department of Electronics and Communication Engineering, Netaji Subhas Institute of Technology (NSIT), Delhi

Digital Signal Processing is a comprehensive textbook designed for undergraduate and postgraduate students of engineering. Following a step-by-step approach, the book will help students master the fundamental concepts and applications of digital signal processing (DSP). Each topic is explained lucidly through abstract mathematical reasoning, illustrations, and solved examples.

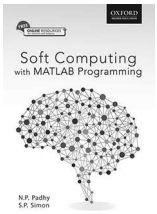
- Provides an understanding of the fundamentals, implementation, and applications of DSP from a practical point of view
- Contains mathematical derivations presented in the simplest possible way
- Contains more than 600 solved examples with step-by-step solutions interspersed within the text
- Includes a section on MATLAB programs at the end of all relevant chapters
- Provides numerous end-of-chapter problems and multiple-choice questions (with answers)



Design and Analysis of Algorithms (9780198093695)
Sridhar, Associate Professor, Department of Information Science and Technology, College of Engineering, Guindy Campus, Anna University, Chennai

Design and Analysis of Algorithms is designed to serve as a textbook for the undergraduate students of computer science engineering and information technology as well as postgraduate students of computer applications. The book aims to empower students with in-depth knowledge of the fundamental concepts and the design, analysis, and implementation aspects of algorithms.

- In-depth treatment for topics such as greedy approach, dynamic programming, transform-and-conquer technique, decrease-and-conquer technique, linear programming, and randomized and approximation algorithms
- Dedicated chapters on backtracking and branch-and-bound techniques, string matching algorithms, and parallel algorithms
- Extensive discussion on the developing and designing aspects of algorithms using minimal mathematics and numerous examples
- Judicious presentation of algorithms using step-wise approach and pseudocodes throughout the text
- Historical notes on various topics and chapter-end crossword puzzles provided to engage readers and enhance their interest in the subject



Soft Computing with MATLAB Programming (9780199455423)
N.P. Padhy, Professor and Chair Professor of NEEPCO (North East Electric Power Company), Department of Electrical Engineering, IIT Roorkee
S.P. Simon, Assistant Professor, Department of Electrical and Electronics Engineering, National Institute of Technology, Tiruchirappalli

Soft Computing with MATLAB Programming is a textbook designed for undergraduate students of computer science, information technology, electrical and electronics, and electronics and communication engineering, as well as those pursuing an MCA degree.

It provides a comprehensive coverage of all the major constituents of soft computing, viz., expert systems, neural networks, fuzzy logic, and evolutionary computation including evolutionary algorithms, genetic algorithms, and swarm intelligence techniques. Most of the swarm-based algorithms have been implemented and explained in an easy step-by-step approach using MATLAB programming.

- Offers exhaustive coverage on artificial neural networks (ANNs) and fuzzy logic
- Discusses swarm intelligence systems with their fundamental concepts of exploration and exploitation
- Elaborates on artificial bee colony algorithm and cuckoo search algorithm with suitable applications
- Provides more than 25 MATLAB programs with step-by-step comments and 75 solved problems

Other Related Titles

9780198068914 Raj Kamal: *Mobile Computing 2e*

9780198082873 Naresh Chauhan: *Operating Systems*

9780198061847 Naresh Chauhan: *Software Testing*

9780198070788 S. Sridhar: *Digital Image Processing*

9780198079064 Senthil Kumar, et al: *Microprocessors and Interfacing*

9780198066644 Muneeswaran: *Compiler Design*

9780195671544 Padhy: *Artificial Intelligence and Intelligent Systems*