Principles of Computer Graphics

Theory and Practice Using OpenGL and Maya®

Principles of Computer Graphics

Theory and Practice Using OpenGL and Maya®

Shalini Govil-Pai
Sunnyvale, CA, U.S.A.

Springer

Shalini Govil-Pai
896 Savory Drive,
Sunnyvale, CA 94087

Email: sgovil@gmail.com

Library of Congress Cataloging-in-Publication Data

Govil-Pai, Shalini
Principles of Computer Graphics: Theory and Practice Using OpenGL and Maya® / Shalini Govil-Pai
p.cm.
Includes bibliographical references and index.

ISBN: 0-387-95504-6 (HC) e-ISBN 0-387-25479-X Printed on acid-free paper
ISBN-13: 978-0387-95504-9 e-ISBN-13: 978-0387-25479-1

9 8 7 6 5 4 3 2 1 SPIN 10879906 (HC) / 11412601 (eBK)

springeronline.com

Contents

Preface

Computer Graphics: the term has become so widespread now, that we rarely stop to think about what it means. What is Computer Graphics? Simply defined, Computer Graphics (or CG) is the images generated or modified on a computer. These images may be visualizations of real data or imaginary depictions of a fantasy world.

The use of Computer Graphics effects in movies such as *The Incredibles* and games such as *Myst* have dazzled millions of viewers worldwide. The success of such endeavors is prompting more and more people to use the medium of Computer Graphics to entertain, to educate, and to explore.

For doctors, CG provides a noninvasive way to probe the human body and to research and discover new medications. For teachers, CG is an excellent tool to visually depict concepts to their students. For business people, CG has come to signify images of charts and graphs used for analysis of data. But for most of us, CG translates into exciting video games, special effects and entire films-what are often referred to as CG productions. This entertainment aspect of CG is what has made it such a glamorous and sought-after field.

Ten years ago, CG was limited to high-end workstations, available only to an elite few. Now, with the advances in PC processing power and the availability of 3D graphics cards, even high school students can work on their home PC to create professional quality productions

The goal of this book is to expose you to the fundamental principles behind modern computer graphics. We present these principles in a fun and simple manner. We firmly believe that you don't have to be a math whiz or a high tech computer programmer to understand CG. A basic knowledge of trigonometry, algebra, and computer programming is more than sufficient.

As you read this book, you will learn the bits and bytes of how to transform your ideas into stunning visual imagery. We will walk you through the processes that professionals employ to create their productions, Based on the principles that we discuss, you will follow these processes step and step, to design and create your own games and animated movies.

We will introduce you to the OpenGL API—a graphics library that has become the de facto standard on all desktops. We will also introduce you to the workings of Maya, a 3D software package. We will demonstrate the workings of the Maya Personal Learning Edition—a (free) download is required.

Organization of the Book

The book is organized into three sections. Every section has detailed OpenGL code and examples. Appendix B details how to install these examples on your desktop.

Section 1: The Basics

The first section introduces the most basic graphics principles. In Chapter 1, we discuss how the computer represents color and images. We discuss how to describe a two-dimensional (2D) world, and the objects that reside in this world. Moving objects in a 2D world involves 2D transformations. Chapter 2 describes the principles behind transformations and how they are used within the CG world. Chapter 3 discusses how the computer saves images and the algorithms used to manipulate these images. Finally, in Chapter 4, we combine all the knowledge from the previous chapters to create our very own version of an arcade game.

Section 2: It's 3D time

Section 2 will expand your horizon from the 2D world to the 3D world. The 3D world can be described very simply as an extension of the 2D world. In Chapter 5, we will introduce you to 3D modeling. Chapter 6 will discuss rendering: you will have the opportunity to render your models from Chapter 5 to create stunning visual effects. Chapter 7 is an advanced chapter for those interested in more advance concepts of CG. We will introduce the concept of Nurbs as used in modeling surfaces. We will also introduce you to advanced shading concepts such as ray tracing. Chapter 8 focuses on teaching the basics of Maya and the Maya Personal Learning Edition of Maya (Maya PLE). Maya is the most popular software in the CG industry and is extensively used in every aspect of production. Learning the basics of this package will be an invaluable tool for those interested in pursuing this area further.

Section 3: Making Them Move

Section 3 discusses the principles of animation and how to deploy them on the computer. In Chapter 9, we discuss the basic animation techniques. Chapter 10 discusses a mode of animation commonly deployed in games, namely, viewpoint animation. In Chapter 11, you will have the opportunity to combine the working knowledge from the previous chapters to create your own movie using Maya.

Appendices

In Appendix A, you will find detailed instructions on how to install the OpenGL and GLUT libraries. Appendix B describes how to download, install the sample code that is detailed in this book. You will also find details on how to compile and link your code using the OpenGL libraries. Appendix C describes the Maya PLE and how to download it.

OpenGL and Maya

Every concept discussed in the book is followed by examples and exercises using C and the OpenGL API. We also make heavy use of the GLUT library, which is a cross-platform OpenGL utility toolkit. The examples will enable you to visually see and appreciate the theory explained. We do not expect you to know OpenGL, but we do expect basic knowledge in C and C++ and knowledge of compiling and running these programs. Some chapters detail the workings of Maya, a popular 3D software package. Understanding Maya will enable you to appreciate the power of the CG concepts that we learn in the book.

Why are we using OpenGL and GLUT?

OpenGL is now a widely accepted industry standard and is used by many (if not all) professional production houses. It is not a programming language but an API. That is, it provides a library of graphics functions for you to use within your programming environment. It provides all the necessary communication between your software and the graphics hardware on your system.

GLUT is a utility library for cross-platform programming. Although our code has been written for the Windows platform, GLUT makes it easier to compile the example code on other platforms such as Linux or Mac. GLUT also eliminates the need to understand basic Windows programming so that we can focus on graphics issues only.

Why are we using Maya?

Some concepts in the book will be further illustrated with the help of industry leading 3D software Maya. Academy-Award winning Maya 3D animation and effects software has been inspired by the film and video artists, computer game developers, and design professionals who use it daily to create engaging digital imagery, animation, and visual effects. Maya is used in almost every production house now, so learning the basics of it will prove to be extremely useful for any CG enthusiast. In addition, the good folks at Alias now let you download a free version of Maya (Maya PLE) to use for learning purposes.

The system requirements for running the examples in this book, as well as for running Maya PLE are as follows:

Software Requirements

- Windows 2000 or higher
- C/C++ compiler such as Microsoft Visual Studio on Windows or GCC (Gnu Compiler Collection) on Unix

Hardware Requirements

- Intel Pentium II or higher/AMD Athlon processor
- 512 MB RAM
- Hardware-accelerated graphics card (comes standard on most systems)

In addition, we expect some kind of Internet connectivity so that you can download required software.

Intended Audience

This book is aimed at undergraduate students who wish to gain an overview of Computer Graphics. The book can be used as a text or as a course supplement for a basic Computer Graphics course.

The book can also serve as an introductory book for hobbyists who would like to know more about the exciting field of Computer Graphics, and to help them decide if they would like to pursue a career in it.

Acknowledgments

The support needed to write and produce a book like this is immense. I would like to acknowledge several people who have helped turn this idea into a reality, and supported me through the making of it:

First, my husband, Rajesh Pai, who supported me through thick and thin. You have been simply awesome and I couldn't have done it without your constant encouragement.

A big thanks to my parents, Anuradha and Girjesh Govil, who taught me to believe in myself, and constantly egged me on to publish the book.

Thanks to Carmela Bourassa of Alias software, who helped provide everything I needed to make Maya come alive.

A very special thanks to my editor, Wayne Wheeler, who bore with me through the making of this book and to the entire Springer staff who helped to produce this book in its final form.

I would like to dedicate this book to my kids, Sonal and Ronak Pai, who constantly remind me that there is more to life than CG.

CG technology is emerging and changing every day. For example, these days, *sub-division surfaces*, *radiosity*, and *vertex shaders* are in vogue. We cannot hope to cover every technology in this book. The aim of the book is to empower you with the basics of CG-providing the stepping-stone to pick up on any CG concept that comes your way.

A key tenet of this book is that computer graphics is fun. Learning about it should be fun too. In the past 30 years, CG has become pervasive in every aspect of our lives. The time to get acquainted with it is now—so read on!

Section 1

The Basics

Imagine how the world would be if computers had no way of drawing pictures on the screen. The entire field of Computer Graphics—flight simulators, CAD systems, video games, 3D movies—would be unavailable. Computers would be pretty much what they were in the 1960s - just processing machines with monitors displaying text in their ghostly green displays.

Today, computers do draw pictures. It's important to understand how computers actually store and draw graphic images. The process is very different from the way people do it. First, there's the problem of getting the image on the screen. A computer screen contains thousands of little dots of light called pixels. To display a picture, the computer must be able to control the color of each pixel. Second, the computer needs to know how to organize the pixels into meaningful shapes and images. If we want to draw a line or circle on the screen, how do we get the computer to do this?

The answers to these questions form the basis for this section. You will learn how numbers written in the frame buffer control the colors of the pixels on the screen. We will expose you to the concept of two-dimensional coordinate systems and how 2D shapes and objects can be drawn and transformed in this 2D world.You will learn the popular algorithms used to draw basic shapes such as lines and circles on the computer.

These days, three-dimensional graphics is in vogue. As a reader, you too must be eager to get on to creating gee-whiz effects using these same principles. It is important, however, to realize that all 3D graphics principles are actually extensions of their 2D counterparts. Understanding concepts in a 2D world is much easier and is the best place to begin your learning. Once you have mastered 2D concepts, you will be able to move on to the 3D world easily. At every step, you will also have the opportunity to implement the theory discussed by using OpenGL.

At the end of the section, we shall put together everything we have learned to develop a computer game seen in many video arcades today.

Chapter 1

From Pixels to Shapes

The fundamental building block of all computer images is the picture element, or the *pixel*. A pixel is a dot of light on the computer screen that can be set to different colors. An image displayed on the computer, no matter how complex, is always composed of rows and columns of these pixels, each set to the appropriate color and intensity. The trick is to get the right colors in the right places.

Since computer graphics is all about creating images, it is fitting that we begin our journey into the computer graphics arena by first understanding the pixel. In this chapter, we will see how the computer represents and sets pixel colors and how this information is finally displayed onto the computer screen. Armed with this knowledge, we will explore the core graphics algorithms used to draw basic shapes such as lines and circles.

In this chapter, you will learn the following concepts:

- What pixels are
- How the computer represents color
- How the computer displays images
- The core algorithm used to draw lines and circles
- How to use OpenGL to draw shapes and objects

1.1 Computer Display Systems

The computer display, or the monitor, is the most important device on the computer. It provides visual output from the computer to the user. In the Computer Graphics context, the display is everything. Most current personal computers and workstations use *Cathode Ray Tube* (CRT) technology for their displays.

As shown in Fig.1.1, a CRT consists of

- An electron gun that emits a beam of electrons (cathode rays)
- A deflection and focusing system that directs a focused beam of electrons towards specified positions on a phosphorus-coated screen
- A phosphor-coated screen that emits a small spot of light proportional to the intensity of the beam that hits it

The light emitted from the screen is what you see on your monitor.

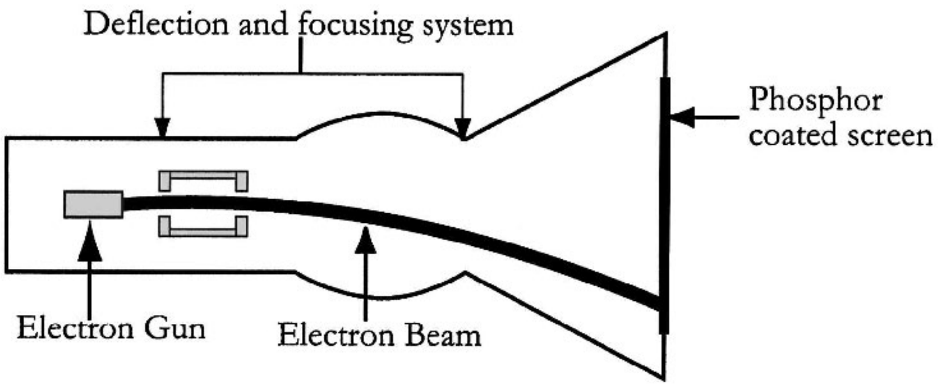

Fig.1.1: A cathode ray tube

The point that can be lit up by the electron beam is called a pixel. The intensity of light emitted at each pixel can be changed by varying the number of electrons hitting the screen. A higher number of electrons hitting the screen will result in a brighter color at the specified pixel. A grayscale monitor has just one phosphor for every pixel. The color of the pixel can be set to black (no electrons hitting the phosphor), to white (a maximum number of electrons hitting the phosphor), or to any gray range in between. A higher number of electrons hitting the phosphor results in a whiter-colored pixel.

A color CRT monitor has three different colored phosphors for each pixel. Each pixel has red, green, and blue-colored phosphors arranged in a triangular group. There are three electron guns, each of which generates an electron beam to excite one of the phosphor dots, as shown in Fig.1.2. Depending on the monitor manufacturer, the pixels themselves may be round dots or small squares, as shown in Fig.1.3.

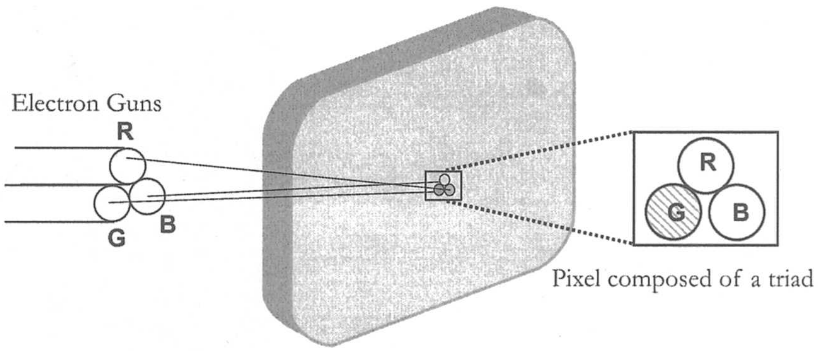

Fig.1.2: Color CRT uses red green and blue triads

Because the dots are close together, the human eye fuses the three red, green, and blue dots of varying brightness into a single dot/square that appears to be the color combination of the three colors. (For those of us who missed art class in school, all colors perceived by humans can be formed by the right brightness combination of red, green, and blue color.)

Conceptually, we can think of the screen as a discrete two-dimensional array (a matrix) of pixels representing the actual layout of the screen, as shown in Fig.1.3.

The number of rows and columns of pixels that can be shown on the screen is called the *screen resolution*. On a display device with a resolution of 1024 x 768, there are 768 rows (scan lines), and in each scan line there are 1024 pixels. That means the display has 768 x 1024=786,432 pixels! That is a lot of pixels packed together on your 14-inch monitor. Higher-end workstations can achieve even higher resolutions.

Fig.1.4 shows two images displayed in different resolutions. At lower resolutions, where pixels are big and not so closely packed, you can start to notice the "pixelated" quality of the image as in the image shown on the right. At higher resolutions, where pixels are packed close together, your eye perceives a smooth image. This is why the resolution of the display (and correspondingly that of the image) is such a big deal.

You may have heard the term *dpi*, which stands for *dots per inch*. The word *dot* is really referring to a pixel. The higher the number of dots per inch of the

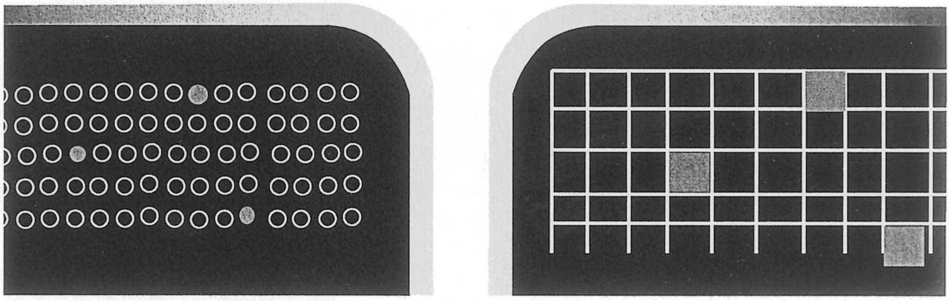

Fig.1.3: Computer display: rows and columns of pixels

screen/image, the higher the resolution and hence the crisper the image.

We have seen we can represent a computer display as a matrix of pixels. But how can we identify an individual pixel and set its color? And how can we then organize the pixels to form meaningful images? In the next section, we explore how pixel colors are set and manipulated.

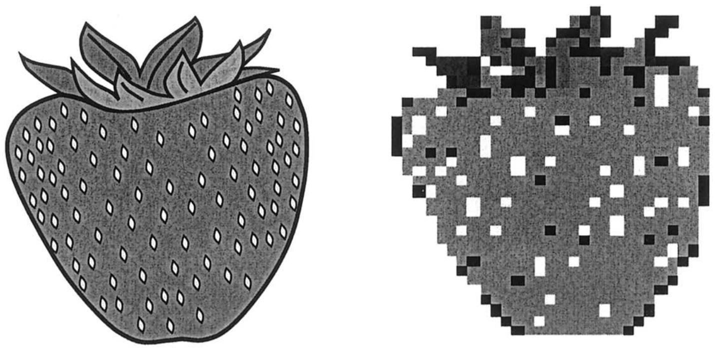

Fig.1.4: The same image at different reolsutions

1.2 Frame Buffers

The light on the screen generated by the beam of electrons in our CRT fades quickly—in 10 to 60 microseconds. In order to keep a picture on the screen for a while, the picture needs be redrawn before it disappears off the screen. This is called *refreshing* the screen. Most display systems use raster scan technology to perform the refresh process. In this technology, the electron beam is directed discretely across the screen, one row at a time from left to right, starting at the upper left corner of the screen. When the beam reaches the bottommost row, the process is repeated, effectively refreshing the screen.

Raster scan systems use a memory buffer called frame buffer (or refresh buffer) in which the intensities of the pixels are stored. Refreshing the screen is performed using the information stored in the frame buffer. You can think of frame buffer as a two dimensional array. Each element of the array keeps the

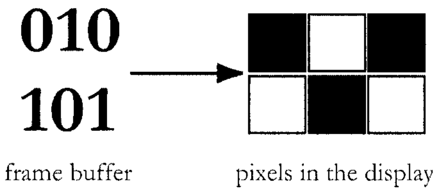

Fig.1.5: Monochrome display: frame buffer for turning pixels on and off

intensity of the pixel on the screen corresponding to that element.

For a monochrome display, the frame buffer has one bit for each pixel. The display controller keeps reading from the frame buffer and turns on the electron gun only if the bit in the buffer is as shown in Fig.1.5.

Systems can have multiple buffers. Foreground buffers draw directly into the window specified. Sometimes a background buffer is also used. The background buffer is not displayed on the screen immediately. We shall talk about buffering modes in more detail when we study animation.

How about color?

You may recall from school physics that all colors in the world can be represented by mixing differing amounts of the three primary colors, namely, red, green, and blue. In CG, we represent color as a triplet of the **R**ed, **G**reen, and **B**lue components. The triplet defines the final color and intensity. This is called the *RGB color model.* Color Plate 1 shows an image of Red, Green and Blue circles and the resultant colors when they intersect.

Some people use a minimum of 0 and a maximum of 255 to represent the intensities of the three primaries, and some people use a floating-point number between 0 and 1. In this book (as is the case in OpenGL), we shall use 0 to represent no color and 1.0 to represent the color set to its maximum intensity. Varying the values in the RGB triplet yields a new color. Table 1.1 lists the RGB components of common colors.

On color systems, each pixel element in the frame buffer is represented by an RGB triplet. This triplet controls the intensity of the electron gun for each of the red, green, and blue phosphors, respectively of the actual pixel on the screen. Our eye perceives the final pixel color to be the color combination of the three colors.

Each pixel color can be set independent of the other pixels. The total number of colors that can be displayed on the screen at one time, however, is limited by the number of bits used to represent color. The number of bits used is called the *color resolution* of the monitor.

For lower resolution systems like VGA monitors, the color resolution is

Color	Red	Green	Blue
Yellow	1	1	0
White	1	1	1
Black	0	0	0
Cyan	0	1	1
Magenta	1	0	1

Table 1.1: The RGB components of common colors

usually 8 bits. Eight-bit systems can represent up to 256 colors at any given time. These kinds of systems maintain a color table. Applications use an index (from 1 to 256) into this color table to define the color of the screen pixel. This mode of setting colors is called *color index mode* and is shown in Fig.1.6.

Of course, if we change a color in this table, any application that indexes into this table will have its color changed automatically. Not always a desirable effect!

Most modern systems have a 24-bit color resolution or higher. A 24-bit system (8 bits for the red channel, 8 for the green channel, and 8 for the blue channel) can display 16 million colors at once. Sometimes an additional 8 bits is added, called the *alpha channel.* We shall look into this alpha channel and its uses later when we learn about fog and blending.

With so many colors available at any given time, there is no need for a color table. The colors can be referred to directly by their RGB components. This way

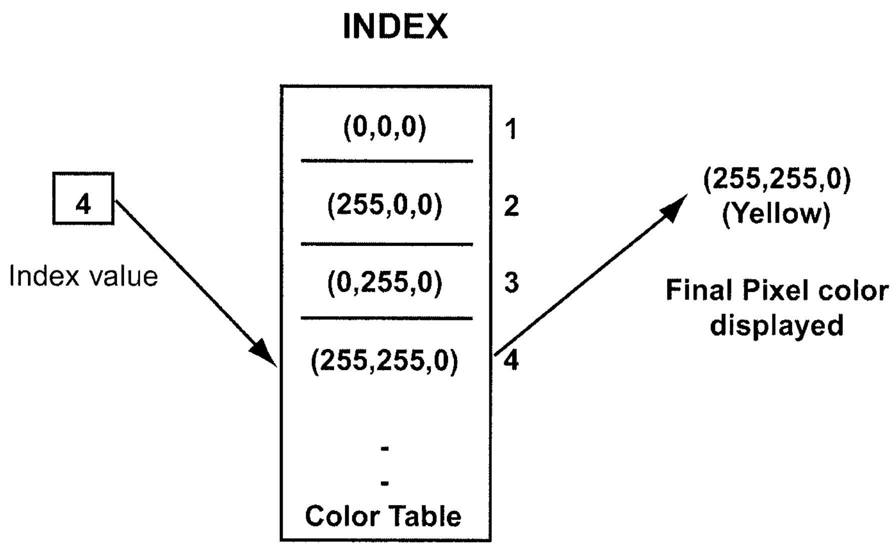

Fig. 1.6:Color index mode

of referring to colors is called *RGB mode*. We shall employ the RGB mode to refer to color throughout the rest of this book.

We have seen how pixel colors are stored and displayed on the screen. But we still need to be able to identify each pixel in order to to set its color. In the next section, we shall see how to identify individual pixel points that we want to paint.

1.3 Coordinate Systems: How to Identify Pixel Points

Coordinates are sets of numbers that describe position—position along a line, a surface of a sphere, etc. The most common coordinate system for plotting both two dimensional and three-dimensional data is the Cartesian coordinate system. Let us see how to use this system to identify point positions in 2D space.

The Cartesian coordinate system is based on a set of two straight lines called the *axes*. The axes are perpendicular to each other and meet at the origin. Each axis is marked with the distances from the origin. Usually an arrow on the axis indicates positive direction. Most commonly, the horizontal axis is called the *x*-axis, and the vertical axis is called the *y*-axis.

Fig.1.7 shows a Cartesian coordinate system with an *x*- and a *y*-axis. To define any point *P* in this system, we draw two lines parallel to the *xy*-axes. The

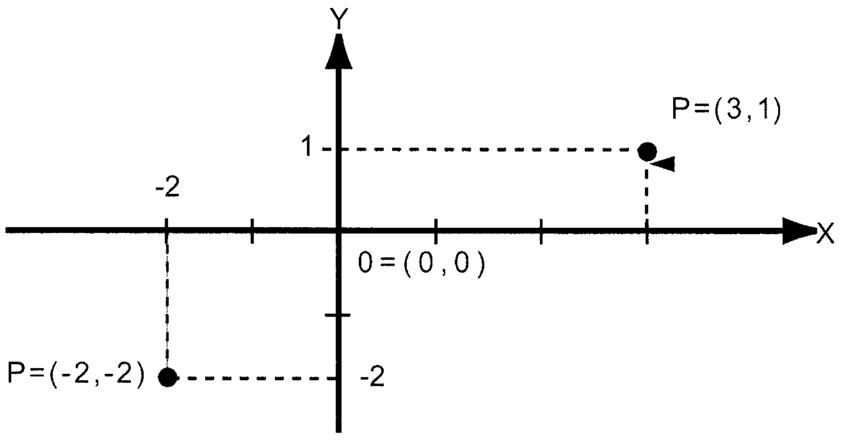

Fig.1.7: Cartesian coordinate system

values of *x* and *y* at the intersections completely define the position of this point. In the Cartesian coordinate system, we label this point as (*x*,*y*). x and y are called the *coordinates* of the point (and could have a negative value). In this system, the origin is represented as (0,0), since it is at 0 distance from itself. A coordinate system can be attached to any space within which points need to be located.

Coming back to the world of computers, recall that our computer display is represented physically in terms of a grid of pixels. This grid of pixels can be defined within its own Cartesian coordinate system. Typically, it is defined with an origin at the upper left corner of the screen. We refer to this Cartesian space as the *physical coordinate system*. Within this system, each pixel can then be uniquely identified by its (*x*,*y*) coordinates, as shown in Fig.1.8.

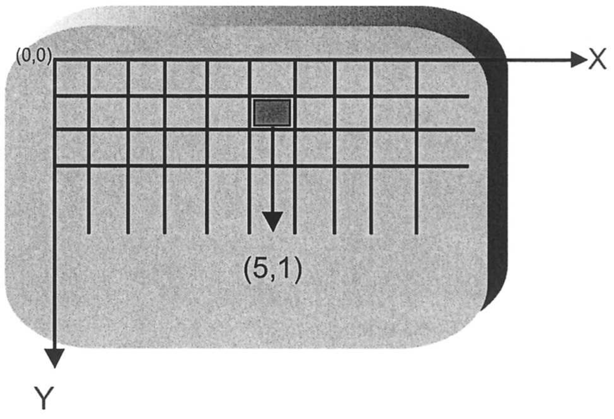

Fig.1.8: Identifying pixels on the screen

Now, consider an application window being shown on this display. We can specify a Cartesian space within which the application resides within. In Fig.1.9, the *x*- coordinates of the window define a boundary ranging from -80 to 80 and the *y*-coordinates from -60 to 60. This region is called the *clipping area*, and is also referred to as the logical or *world coordinate system* for our application. This is the coordinate system used by the application to plot points, draw lines

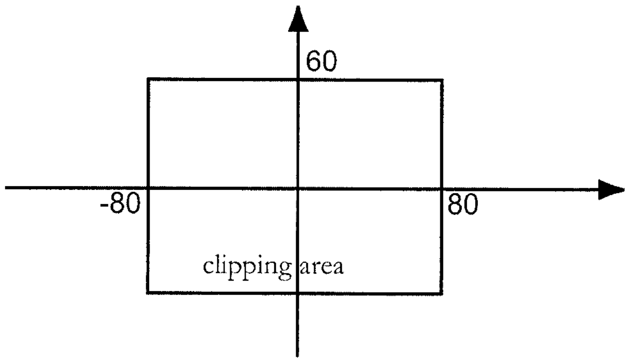

Fig.1.9: Application (clipping) area

and shapes, etc. Objects within the clipping area are drawn, and those outside this area are removed or *clipped* from the scene. The clipping area is mapped onto a physical region in the computer display by mapping the application boundaries to the physical pixel boundaries of the window.

If the clipping area defined matches the (physical) resolution of the window, then each call to draw an (*x*,*y*) point (with integer values) in the world coordinate system will have a one-to-one mapping with a corresponding pixel in the physical coordinate system.

For most applications, the clipping area does not match the physical size of the window. In this case, the graphics package needs to perform a transformation

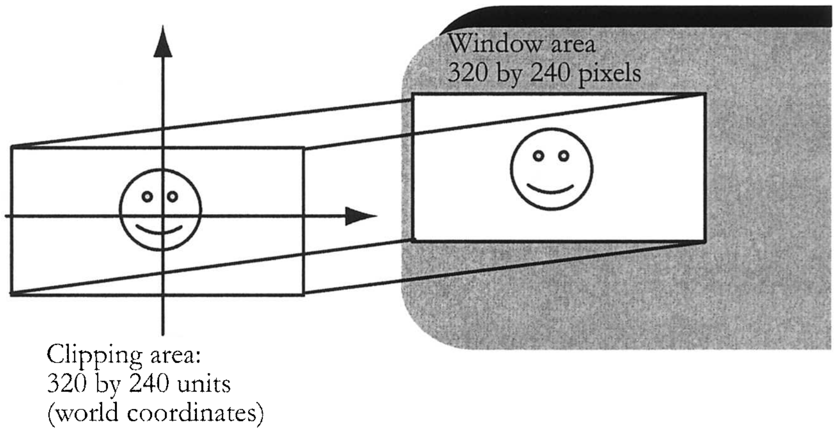

Fig.1.10: Mapping Clipping Area onto the window

from the world coordinate system being used by the application to the physical window coordinates. This transformation is determined by the clipping area, the physical size of the window, and another setting known as the *viewport.* The viewport defines the area of the window that is actually being used by the application.

For now, we will assume that the viewport is defined as the entire window (but this is not always necessary, as shown in Fig.1.11).

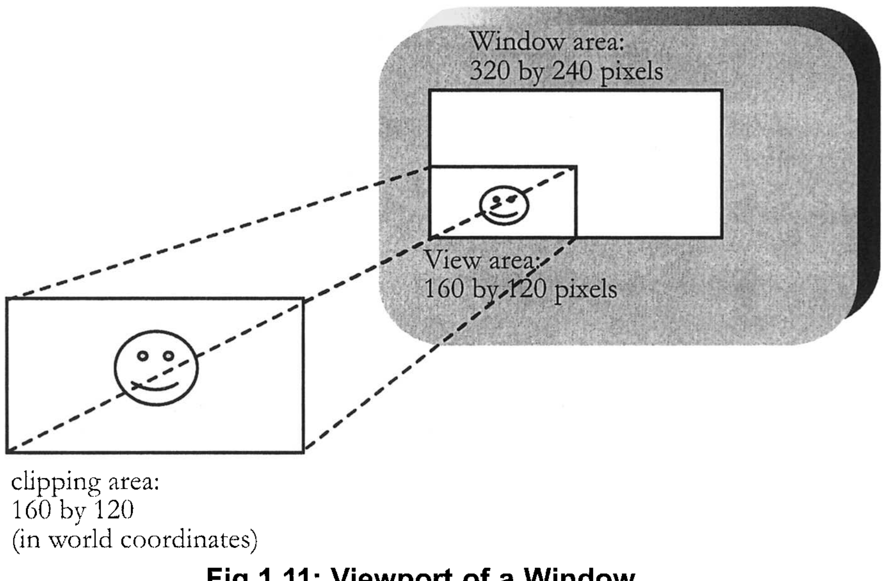

Fig.1.11: Viewport of a Window

Example Time

Let us run our first OpenGL program to get a handle on some of the concepts we have just learned. For information on OpenGL and GLUT and how to install it on your system refer to Appendix A on details. For information on how to download the sample example code from the Internet and how to compile and link your programs, refer to Appendix B.

The following example displays a window with physical dimensions of 320 by 240 pixels and a background color of red. We set the clipping area (or the world coordinates) to start from (0,0) and extend to (160,120). The viewport is set to occupy the entire window, or 320 by 240 pixels. This setting means that every increment of one coordinate in our application will be mapped to two pixel increments in the physical coordinate system (application window boundaries (0,0) to (160,120) vs. physical window boundaries (0,0) to (320,240)). If the viewport had been defined to be only 160 by 120 pixels, then there would have been a one-to-one mapping from points in the world coordinate space to the physical pixels.

However, only one fourth of the window would have been occupied by the application! Depending on where you install the example files, you can find the source code for this example in: *Example1_1/Example1_1.cpp.*

```
//Example1_1.cpp: A simple example to open a window
// the windows include file, required by all windows apps
#include <windows.h>

// the glut file for windows operations
// it also includes gl.h and glu.h for the OpenGL library  calls
#include <gl\glut.h>

void Display(void)
{
            //clear all pixels with the specified clear color
            glClear(GL_COLOR_BUFFER_BIT);
//don't wait, start flushing OpenGL calls to display buffer
            glFlush();
}
void init(void){
            //set the clear color to be red
            glClearColor(1.0,0.0,0.0,1.0);
            //set the viewport to be 320 by 240, the initial size of the window
            glViewport(0,0,320,240);
// set the 2D clipping area
            gluOrtho2D(0.0, 160.0, 0.0, 120.0);
}

void main(int argc, char* argv[])
{
            glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
```

```
        glutInitWindowSize (320, 240);
        glutCreateWindow("My First OpenGL Window");
        init();
        glutDisplayFunc(Display);
        glutMainLoop();
}
```

Since this is our first OpenGL program, Let us understand the example line by line.

The Include Files

There are only two include files:

```
#include <windows.h>
#include <gl\glut.h>
```

The windows.h is required by all windows applications. The header file glut.h includes the GLUT library functions as well as gl.h and glu.h, the header files for the OpenGL library functions. All calls to the glut library functions are prefixed with glut. Similarly, all calls to the OpenGL library functions start with the prefix gl or glu.

The Body

Let us look at the main program first. The line

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
```

tells the GLUT library what type of display mode to use when creating the application window. In this case, we specify that we will be using only a single frame buffer (GLUT_SINGLE) and we want to specify colors in the RGB color mode (GLUT_RGB). We will discuss more about frame buffers in a later chapter.
The next call

```
glutInitWindowSize (320, 240);
```

initializes the window to have an initial size (physical resolution) of 320 by 240 pixels.
The call

```
glutCreateWindow("My First OpenGL Window");
```

actually creates the window with the caption "My First OpenGL Window".

The next function

```
init();
```

initializes some of the OpenGL parameters before we actually display the rendered window.

OpenGL and glut work with the help of callback functions. Events that occur on the computer (such as mouse clicks, keyboard clicks, moving the window etc.) that you wish your program to react to need to be registered with OpenGL as callback functions. When the event occurs, OpenGL automatically calls the function registered to react to the event appropriately.
The function

```
glutDisplayFunc(Display);
```

registers the callback function for display redrawing to be the function *Display*. The Display callback is triggered whenever the window needs to be redrawn, and GLUT will automatically call the Display function for you. When does the window need to be redrawn? When you first display the window, when you resize the window, or even when you move the window around. We shall see what the init function and the Display function actually do in a bit.
Finally, we make a call to the glut library function

```
glutMainLoop();
```

This function simply loops, monitoring user actions and making the necessary calls to the specified callback functions (in this case, the '*Display*' function) until the program is terminated.

The init() function

The init function itself is defined to initialize the GL environment. It does this by making three calls:

```
glClearColor(1.0, 0.0, 0.0, 1.0);
```

This gl library command sets the color for clearing out the contents in the frame buffer (which then get drawn into the window). It expects the RGB values, in that order, as parameter,s as well as the alpha component of the color. For now, we set the alpha to always be 1. The above command will set the clear color to be pure red. Try experimenting with different clear colors and see what effect this has on the window display.

Next, we define the viewport to be equal to the initial size of the window by calling the function

```
glViewport(0, 0, 320, 240)
```

And we set the clipping area, or our world coordinate system, to be (0,0) to (160,120) with the glu library command

```
gluOrtho2D(0.0, 160.0, 0.0, 120.0);
```

The Display() function

The Display function simply makes two OpenGL calls:

```
glClear(GL_COLOR_BUFFER_BIT);
```

On a computer, the memory (frame buffer) holding the picture is usually filled with the last picture you drew, so you typically need to clear it with some background color before you start to draw the new scene.

OpenGL provides glClear as a special command to clear a window. This command can be much more efficient than a general-purpose drawing command since it clears the entire frame buffer to the current clearing color. In the present example, we have set the clear color earlier to be red.

In OpenGL, the frame buffer can be further broken down into buffers that hold specialized information.

The color buffer (defined as GL_COLOR_BUFFER_BIT) holds the color information for the pixel. Later on, we shall see how the depth buffer holds depth information for each pixel.

The single parameter to glClear() indicates which buffers are to be cleared. In this case, the program clears only the color buffer.
Finally, the function

```
glFlush();
```

forces all previously issued OpenGL commands to begin execution. If you are writing your program to execute within a single machine, and all commands are truly executed immediately on the server, glFlush() might have no effect. However, if you're writing a program that you want to work properly both with and without a network, include a call to glFlush() at the end of each frame or scene. Note that glFlush() doesn't wait for the drawing to complete —it just forces the drawing to begin execution.

Voila: when you run the program you will see a red window with the caption "My First OpenGL window". The program may not seem very interesting, but it demonstrates the basics of getting a window up and running using OpenGL. Now that we know how to open a window, we are ready to start drawing into it.

Plotting Points

Objects and scenes that you create in computer graphics usually consist of a combination of shapes arranged in unique combinations. Basic shapes, such as points, lines, circles, etc., are known as graphics primitives. The most basic primitive shape is a point.

In the previous section, we considered Cartesian coordinates and how we can map the world coordinate system to actual physical screen coordinates. Any point that we define in our world has to be mapped onto the actual physical screen coordinates in order for the correct pixel to light up. Luckily, for us, OpenGL handles all this mapping for us; we just have to work within the extent of our defined world coordinate system. A point is represented in OpenGL by a set of floating-point numbers and is called a *vertex.*

We can draw a point in our window by making a call to the gl library function

```
glVertex{2,3,4}{s,I,d,f}
```

The {2,3,4} option indicates how many coordinates define the vertex and the {s,i,d,f} option defines whether the arguments are short, integers, double precision, or floating-point values. By default, all integer values are internally converted to floating-point values.
For example, a call to

```
glVertext2f(1.0,2.0)
```

refers to a vertex point (in world coordinate space) at coordinates (1.0,2.0). Almost all library functions in OpenGL use this format. Unless otherwise stated, we will always use the floating-point version of all functions. To tell OpenGL what set of primitives you want to define with the vertices, you bracket each set of vertices between a call to

glBegin() and glEnd().

The argument passed to glBegin() determines what sort of geometric primitive it is. To draw vertex points, the primitive used is GL_POINTS. We modify *Example1_1* to draw four points. Each point is at a distance of (10,10) coordinates away from the corners of the window and is drawn with a different color. Compile and execute the code shown below. You can also find the code for this example under *Example1_2/Example1_2.cpp..*

```
// Example1_2.cpp: let the drawing begin

#include <windows.h>
#include <gl\glut.h>

void Display(void)

{
```

```
          //clear all pixels with the specified clear color
          glClear(GL_COLOR_BUFFER_BIT);
          //draw the four points in four colors
          glBegin(GL_POINTS);
                    glColor3f(0.0, 1.0, 0.0);               // green
                    glVertex2f(10.,10.);
                    glColor3f(1.0, 1.0, 0.0);               // yellow
                    glVertex2f(10.,110.);
                    glColor3f(0.0, 0.0, 1.0);               // blue
                    glVertex2f(150.,110.);
                    glColor3f(1.0, 1.0, 1.0);               // white
                    glVertex2f(150.,10.);
          glEnd();

          //dont wait, start flushing OpenGL calls to display buffer
          glFlush();
}

void reshape (int w, int h)
{
           // on reshape and on startup, keep the viewport to be the entire size of the window
          glViewport (0, 0, (GLsizei) w, (GLsizei) h);
          glMatrixMode (GL_PROJECTION);
          glLoadIdentity ();
          // keep our world coordinate system constant
          gluOrtho2D(0.0, 160.0, 0.0, 120.0);
}

void init(void){
          glClearColor(1.0,0.0,0.0,1.0);
          // set the point size to be 5.0 pixels
          glPointSize(5.0);
}

void main(int argc, char* argv[])
{
          glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
          glutInitWindowSize (320, 240);
          glutCreateWindow("My First OpenGL Window");
          init();
          glutDisplayFunc(Display);
          glutReshapeFunc(reshape);
          glutMainLoop();
}
```

Most of the code should be self-explanatory.
The main function sets up the initial OpenGL environment.
In the init() function, we set the point size of each vertex drawn to be 5 pixels, by calling the function

```
glPointSize(3.0);
```

The parameter defines the size in pixels of the points being drawn. A 5 pixel point is large enough for us to see it without squinting our eyes too much!
In this function, we also we set the clear color to be red. The Display function defines all the drawing routines needed
The function call

```
glColor3f(0.0, 1.0, 0.0);                    // green
```

sets the color for the next openGL call. The parameters are, in order, the red, green, and blue components of the color. In this case, we redefine the color before plotting every vertex point. We define the actual vertex points by calling the function

```
glVertex2f(10.,10.);
```

with the appropriate *(x,y)* coordinates of the point. In this example, we define two callback functions. We saw how to define the callback function for redrawing the window. For the rest of the book, we will stick with the convention of this function being called "Display".

In this example we define the viewport and clipping area settings in a new callback function called *reshape*. We register this callback function with OpenGL with the command

```
glutReshapeFunc(reshape);
```

This means that the reshape function will be called whenever the window resizes itself (which includes the first time it is drawn on the screen!) The function receives the width and height of the newly shaped window as its arguments. Every time the window is resized, we reset the viewport so as to always cover the entire window. We always define the world coordinate system to remain constant at ((0,0),(160,120)). As you resize the window, you will see that the points retain their distance from the corners. What mapping is being defined? If we change the clipping area to be defined as

```
gluOrtho2D(0.0, (Gldouble)w, 0.0, (Gldouble)h);
```

you would see that the points maintain their distance from each other and not from the corners of the window. Why?

1.4 Shapes and Scan Converting

We are all familiar with basic shapes such as lines and polygons. They are easy enough to visualize and represent on paper. But how do we draw them on the computer? The trick is in finding the right pixels to turn on!

The process by which an idealized shape, such as a line or a circle, is transformed into the correct "on" values for a group of pixels on the computer is called scan conversion and is also referred to as *rasterizing*.

Over the years, several algorithms have been devised to make the process of scan converting basic geometric entities such as lines and circles simple and fast.

The most popular line-drawing algorithm is the midpoint-line algorithm. This algorithm takes the *x*- and *y*- coordinates of a line's endpoints as input and then calculates the *x,y*-coordinate pairs of all the pixels in between. The algorithm begins by first calculating the physical pixels for each endpoint. An ideal line is then drawn connecting the end pixels and is used as a reference to determine which pixels to light up along the way. Pixels that lie less than 0.5 units from the line are turned on, resulting in the pixel illumination as shown in Fig.1.12.

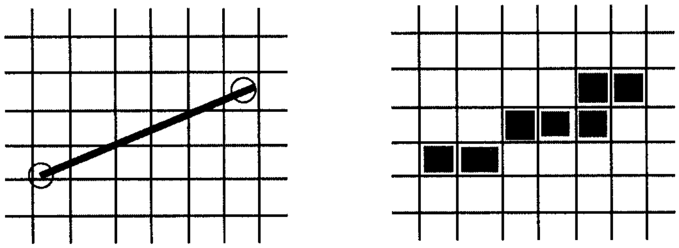

Fig.1.12: Midpoint algorithm for line drawing

All graphic packages (OpenGL included) incorporate predefined algorithms to calculate the pixel illuminations for drawing lines.

Basic linear shapes such as triangles and polygons can be defined by a series of lines. A polygon is defined by *n* number of vertices connected by lines, where n is the number of sides in the polygon. A quadrilateral, which is a special case of a polygon is defined by four vertices, and a triangle is a polygon with three vertices as shown in Fig.1.13.

To specify the vertices of these shapes in OpenGL, we use the function that we saw earlier:

glVertex.

To tell OpenGL what shape you want to create with the specified vertices, you bracket each set of vertices between a call to glBegin() and a call to glEnd(). The

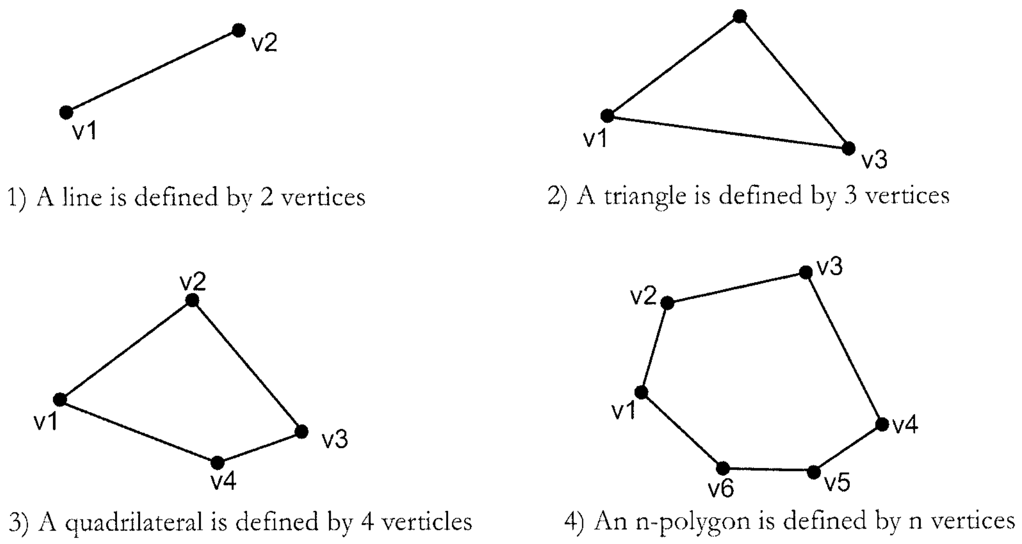
v2
v1
1) A line is defined by 2 vertices
v1
v3
2) A triangle is defined by 3 vertices
v2
v1
v3
v4
3) A quadrilateral is defined by 4 verticles
v3
v2
v1
v4
v6
v5
4) An n-polygon is defined by n vertices

Fig.1.13: Vertices needed to define different kinds of basic shapes

argument passed to glBegin() determines what sort of geometric primitive; some of the commonly used ones are described in Table 1.2.

Primitive definition	Meaning
GL_POINTS	individual points
GL_LINES	pair of vertices defining a line
GL_LINE_STRIP	series of connected lines
GL_TRIANGLES	strip of linked triangles
GL_POLYGON	vertices define a simple convex polygon

Table 1.2: OpenGL geometric primitive types

Note that primitives are all straight-line primitives. There are algorithms like the midpoint algorithm that can scan convert shapes like circles and other hyperbolic fig.s. The basic mechanics for these algorithms are the same as for lines: figure out the pixels along the path of the shape, and turn the appropriate pixels on.

Interestingly, we can also draw a curved segment by approximating its shape using line segments. The smaller the segments, the closer the approximation.

For example, consider a circle. Recall from trigonometry that any point on a circle of radius r (and centered at the origin) has an x,y-coordinate pair that can be represented as a function of the angle theta the point makes with the axes, as shown in Fig.1.14.

$$P(\theta)=((r\cos\theta),(r\sin\theta))$$

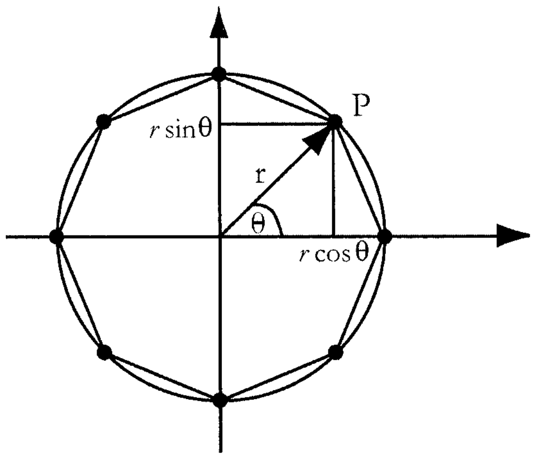

Fig.1.14: Points along a circle

As we vary theta from 0 to 360 degrees (one whole circle), we can get the (x,y) coordinates of points along the circle. So if we can just plot "enough" of these points and draw line segments between them, we should get a fig. that looks close enough to a circle. A sample code snippet to draw a circle with approximately 100 points is shown below. Note that we need to add the center of the circle to our equations to position our circle appropriately.

```
#define PI 3.1415926535898
// cos and sin functions require angles in radians
// recall that 2PI radians - 360 degrees, a full circle

GLint circle_points - 100;
void MyCircle2f(GLfloat centerx, GLfloat centery, GLfloat radius){
        GLint i;
        GLdouble theta;
        glBegin(GL_POLYGON);
        for (i - 0; i < circle_points; i++) {
                theta- 2*PI*i/circle_points; // angle in radians
                glVertex2f(centerx+radius*cos(theta),
                        centery +radius*sin(theta));
        }
        glEnd();
}
```

Remember that the math functions *cos* and *sin* require angles in radians and that 2PI radians make 360 degrees—hence our conversions in the code. We can construct more complex shapes by putting these basic primitives together. Shown below is a snippet of code to draw a stick figure of a person as shown in