Lab on Computer Graphics

DCAP313

Edited by:
Avinash Bhagat

LOVELY PROFESSIONAL UNIVERSITY

LAB ON COMPUTER GRAPHICS

Edited By

Avinash Bhagat

Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

Lab on Computer Graphics

Objectives:

- To enable the student to understand various technicalities of hardware devices used in computer graphics.
- To enable the student to learn various techniques used to draw images.
- To enable the student to understand two dimensional and three dimensional operations on objects.
- To enable the student to understand technicalities to draw polygons.
- To enable the student to learn graphics programming.

COURSE CONTENTS:

S. No.	Topics
1.	**Fundamentals Of Computer Graphics:** Applications of computer Graphics in various fields, Evolution of computer Graphics.
2.	**Graphics Systems:** Video Display Unit, Random scan displays, raster scan displays, Displaying Colours, Frame Buffer, Digitization, Persistence, Resolution
3.	Implementing Line Algorithm
4.	Implementing Circle Algorithm
5.	Implementing Ellipse Algorithm
6.	Implementing polygon filling algorithm
7.	Implementation of Hidden Surface in 2D.
8.	Implementing of Scaling in 2D Transformation.
9.	Translation
10.	Sharing, Rotation, Reflection.

CONTENT

Unit 1: Fundamentals of Computer Graphics

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the basic concept of the fundamental of computer graphics
- Define the application of computer graphics in various fields
- Understand the evolution of computer graphics

Introduction

Computer graphics commonly indicates creation, storage and manipulation of models and images. Such models come from diverse and expanding set of fields including physical, mathematical, artistic, biological, and even conceptual (abstract) structures. "Perhaps the best way to define computer graphics is to find out what it is not. It is not a machine. It is not a computer, nor a group of computer programs. It is not the know-how of a graphic designer, a programmer, a writer, a motion picture specialist, or a reproduction specialist. Computer graphics is all these –a consciously managed and documented technology directed toward communicating information accurately and descriptively.

Fundamentals of computer graphics, it gives both the theoretical and practical skills to begin developing Computer Graphics software. These notes are a supplementary resource to the lectures and lab sessions. All Computer Graphics demands a working knowledge of linear algebra (matrix manipulation, linear systems, etc.). This course also requires a working knowledge of the C language.

Notes

Image Representation

How digital images are characterized in a computer. This 'mini'-topic explores different forms of frame-buffer for storing images, and also different ways of representing colour and key issues that arise in colour.

Geometric Transformation

How to use linear algebra, e.g. matrix transformations, to manipulate points in space. This work focuses heavily on the concept of reference frames and their central role in Computer Graphics. Also on this theme, Eigen value decomposition is discussed and a number of applications relating to visual computing are explored.

OpenGL Programming

Discusses how the mathematics on this course can be implemented directly in the C programming language using the OpenGL library. Note much of this content is covered in PowerPoint handouts sooner than these notes.

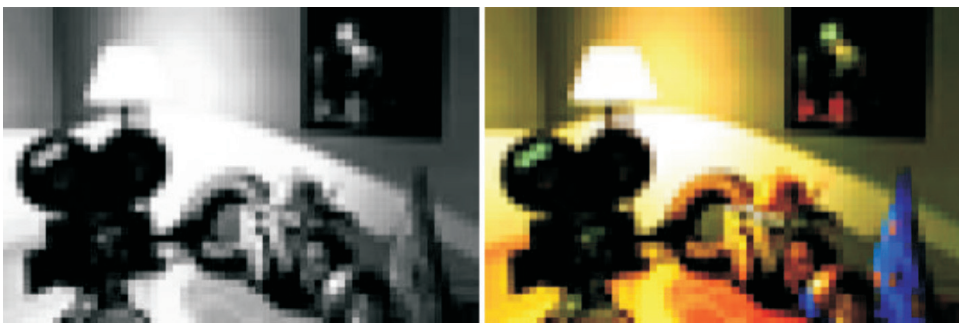
Geometric Modeling

Whereas center on manipulating/positioning of points in 3D space, how these points can be "joined up" to form curves and surfaces. This allows the modelling of objects and their trajectories. The computer graphics is a huge field that encompasses almost any graphical facet; we are mainly interested in the generation of images of 3-dimensional scenes. Computer imagery has applications for film special effects, simulation and training, games, medical imagery, flying logos; etc. Graphics relies on an internal model of the view, that is, a mathematical representation suitable for graphical computations. The model describes the 3D shapes, layout and of the scene. This 3D representation then has to be projected to compute a 2D image from a given viewpoint. the involves projecting the objects (perspective), handling (which parts of objects are hidden) and computing their appearance and lighting interactions, for animated sequence, the motion of objects has to be specified.

Pixel

A computer image is usually represented as an isolated grid of picture elements a.k.a. pixels. The number pixel determines the resolution of the image (See Figure 1.1). Typical resolutions range from 320*200 to 2000*1500. For a black and white image, a number explains the intensity of each pixel. It can be expressed between 0.0 (black) and 1.0 (white). However, for internal binary representation reasons, it is usually stored as an integer between 0 (black) and 255 (white).

Figure 1.1: The Low Resolution Digital Image. Left: Black and White. Right: Colour



Notes

For a colour image, each pixel is described by a triple of numbers characterizing the intensity of red, green and blue. For example, pure red is (255, 0, 0) and purple is (255, 0, 255). The image is represented by a isolated array of pixels, aliasing problems may occur. The most classical form of aliasing is the jaggy aspect of lines. Ant aliasing techniques are thus required. In the case of the line (See Figure 1.2), it consists in using intermediate gray levels to "smooth" the appearance of the line. Another form of aliasing can be observed on television when people wear shirts with a fine stripped texture. A flickering prototype is observed because the size of the pattern is on the same order of magnitude as the pixel size.

Figure 1.2: A Line Without and with Anti-aliasing

1.1 Concept of Computer Graphics

The creation of, manipulation of, study of, and interaction with graphic representations of objects and data using computers graphics is also called image synthesis.

Computer graphics comprises the creation and representation of simple graphical elements and images, as well as modern techniques for rendering a virtual reality. To apply these techniques correctly, one requires a basic understanding of the fundamental concepts in graphics.

The most important basic concept of computer graphics is pairing the technical background and theory with practical examples and applications throughout. Its user-friendly approach enables the reader to gain understanding through the theory at work, via the many example programs provided.

One of the most common uses for computer graphics is the creation of an image that looks like a photograph from real life, but portrays something we could not really take a picture of. For example, we might want to create an animation for a movie that portrays an alien creature or location, or we might want to create an architectural rendering to get an idea of what the final building will look like. We will call this type of computer graphics photo-realistic. Normally there is a significant amount of time available for creating these images, and so the techniques often attempt to capture the properties of lights and surfaces as accurately as possible, at the expense of time needed to compute the image.

Another common use for computer graphics is the creation of images very swiftly so that they can be used in an interactive animation; the application of these algorithms that most people are well-known with is computer games. As a user controls their nature, we want to create a series of images very quickly (hopefully 30 per second or more) that respond to their inputs. We will refer to this type of computer graphics as real-time, or interactive. We might want these images to look as sensible as possible, but the fact of the situation is that if we have more time to devote to computation, we can create better results. Thus, real-time computer graphics is usually concerned with techniques for creating the best-looking picture possible with the very short amount of time available between frames.

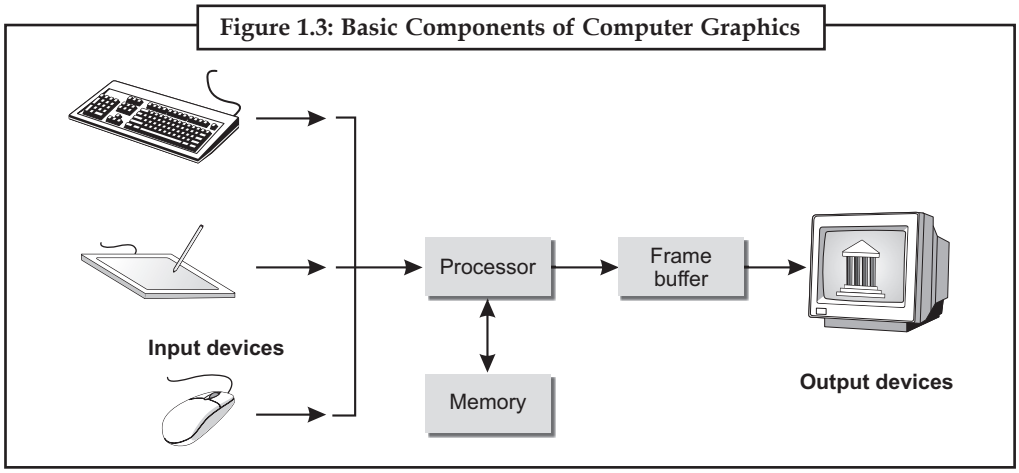
There are many other applications for computer graphics as well. We might be interested in automating a complex picture that we could make by hand for convenience, or we might be interested in creating an image that is abstract or random in nature. In fact, there are many applications of computer graphics that fall somewhere in-between the two extremes presented

Notes

above, such as medical imaging and visualization systems. However, for an introductory text, it will suffice to present the two main techniques that are mixed and matched for the desired results in practice.

In the text that follows, we will begin by attempting to understand the concepts of pixels and how computer represent colour, which is the format for what computer graphics algorithms output. After that, we will consider how we represent the input data that programs in computer graphics use, the 3D objects that get rendered. With these basics out of the way, we will then delve into the techniques of photo-realistic and real-time rendering. Figure 1.3 shows the basic components of computer graphics.

Figure 1.3: Basic Components of Computer Graphics



- Computer graphics generally means creation, storage and manipulation of models and images
- Computer graphics deals with all aspects of creating images with a computer
  1. Hardware
  2. Software
  3. Applications

We use images/pictures?

- "One picture is better than a thousand words"
- Vision is the most important sense of humans
- Humans communicate well with images

Image Processing

- Digital Image Processing
  - Any form of signal processing for which the input is an image, the output of image processing can be either an image or a set of characteristics or parameters related to the image.

The Study Computer Graphics

The Computer Graphics program has been designed in reaction to demand from local industry for graduates with technical abilities in programming in general and graphics in particular. Feedback from prospective students has also been taken into account to produce a professional qualification that covers a diverse and interesting range of subject material.

Notes

Computer graphics is a fun and rewarding area to work in, and has now become a well established field of Computer Science.

The phrase "Computer Graphics" was coined in 1960 by William Fetter, a graphic designer for Boeing. The field of computer graphics developed with the emergence of computer graphics hardware.

1.2 Application of Computer Graphics

Computer-aided Design for engineering and architectural systems etc. Objects maybe presented in a wireframe outline form. Multi-window environment is also privileged for producing various zooming scales and views. Animations are useful for testing performance. Presentation Graphics to produce illustrations which summarize various kinds of data. Except 2D, 3D graphics are good tools for reporting more complex data.

Computer Art Painting packages are available. With cordless, pressure-sensitive stylus, artists can produce electronic paintings which replicate different brush strokes, brush widths, and colors. Photorealistic techniques, morphing and animations are very useful in commercial art. For films, 24 frames per second are required. For video monitor, 30 frames per second are required. Entertainment Motion pictures, Music videos, and TV shows, Computer games Education and Training with computer-generated models of specialized systems such as the training of ship captains and aircraft pilots.

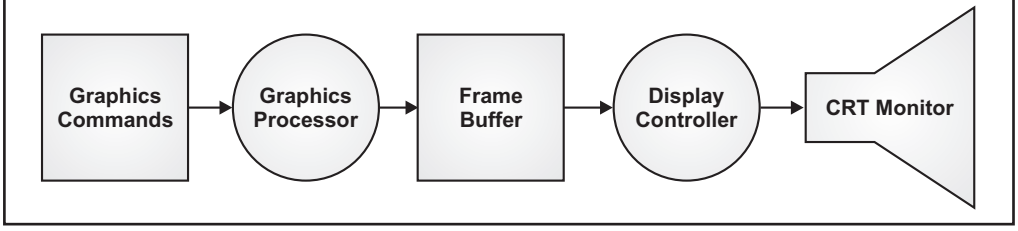
Visualization for analyzing scientific, engineering, medical and business data or behaviour. Converting data to visual form can help to understand mass volume of data very efficiently. Image Processing Image processing is to apply techniques to modify or interpret existing pictures. It is widely used in medical applications. Graphical User Interface multiple window, icons, menus allow a computer setup to be utilized more efficiently.

The use of Real Time graphic applications as educational tools specifically oriented to working with people who have certain learning difficulties. We first focus on identifying the most related traits (from a psychological point of view) of those disorders, then we continue by analyzing the advantages of graphics in Real Time in this context, and how they can be used to complement the conventional teaching methods. Finally, we review the main characteristics of two applications belonging to this category, which serve as a practical example of this encounter between education and technology.

1.2.1 Overview of Graphics Systems

In this context we argue the graphics systems of raster-scan devices. A graphics processor accepts graphics commands from the CPU and performs the graphics commands which may involve drawing into the frame buffer. The frame buffer acts as a temporary store of the image and also as a de coupler to let the graphics processor and the display controller to operate at different speeds. The display controller reads the frame buffer line by line and generates the control signals for the screen. Figure 1.4 shows the block diagram of a CRT graphics system.

Figure 1.4: Block Diagram of a CRT Graphics System



Notes

Graphics Commands

- Draw point
- Draw polygon
- Draw text
- Clear frame buffer
- Change drawing colour

Graphics Processor

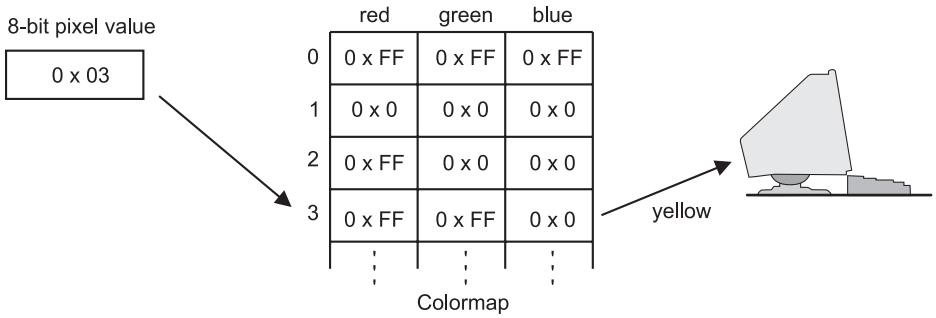
Two kinds of graphics processors:

2D Graphics Processors: Execute commands in 2D coordinates, when objects not be separate, the one being drawn will obscure objects drawn formerly in the region. Bitable operations (Bit Block Transfer) are usually provided for moving/copying one rectangular region of frame buffer contents to another region.

3D Graphics Processors: Execute commands in 3D coordinates, When objects overlap, it is required to determine the visibility of the objects according to the z values.

Display Controller for a raster display device reads the frame buffer and generates the control signals for the screen, i.e. the signals for horizontal scanning and vertical scanning. Most display controllers include a colour map (or video look-up table). The major purpose of a colour map is to provide a mapping between the input pixel values to the output colour.

Figure 1.5: Display Controller



1.2.2 Input Devices

Common Devices: keyboard, mouse, trackball and joystick Specialized devices.

Data Gloves : are electronic gloves for detecting fingers' movement. In some applications, a sensor is also close to the glove to notice the hand movement as a whole in 3D space. A tablet contains a stylus and a drawing surface and it is mainly used for the input of drawings. A tablet is usually more accurate than a mouse, and is commonly used for large drawings.

Scanners : are used to convert drawings or pictures in hardcopy format into digital signal for computer processing.

Touch Panel : allow displayed objects or screen positions to be selected with the touch of a finger. In these devices a touch-sensing mechanism is fitted over the video monitor screen. Touch input can be recorded using optical, electrical, or acoustical methods.

Notes

1.2.3 Hard-copy Devices

Pictures are directing to a printer or plotter to produce hard-copy output on 35-mm slides, overhead transparencies or plain paper. The quality of the pictures depends on dot size and number of dots per inch (DPI).

Types of printers: line printers, LaserJet, ink-jet, and dot-matrix.

LaserJet: printers use a laser beam to create a charge allotment on a turning drum coated with a photoelectric material. Toner is applied to the drum and then transferred to the paper. To produce colour outputs, the 3 colour pigments (cyan, magenta, and yellow) are deposited on separate passes.

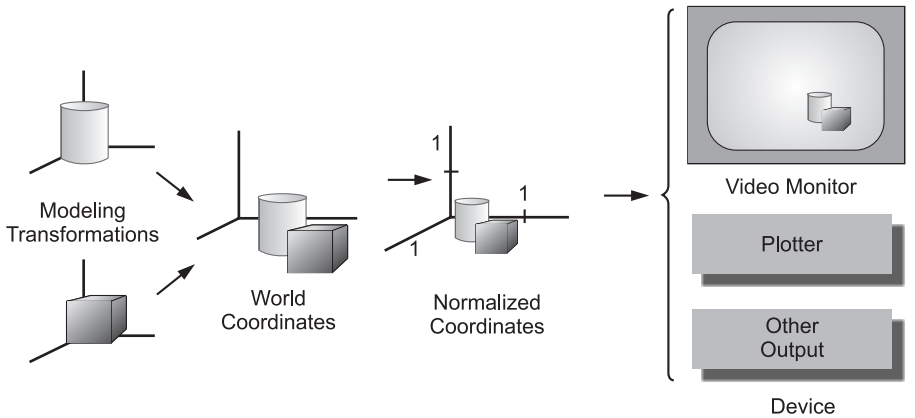
Inkjet: printers produce output by squirting ink in horizontal rows across a roll of paper wrapped on a drum. To produce colour outputs, the 3 colour pigments are shot simultaneously on a single pass along each print line on the paper.

Inkjet or Pen Plotters: These are used to generate drafting layouts and other drawings of normally larger Sizes. A pen plotter has one or more pens of different colours and widths mounted on a carriage which spans a sheet of paper.

1.2.4 Coordinate Representations in Graphics

General graphics packages are designed to be used with Cartesian coordinate representations (x, y, and z). Usually several different Cartesian reference frames are used to construct and display a scene:

Figure 1.6: Coordinate Representation in Graphics



Modelling coordinates: are used to construct individual object shapes.

World coordinates: are computed for specifying the placement of individual objects in appropriate positions.

Normalized coordinates: are converted from world coordinates, such that x, y values are ranged from 0 to 1.

Device coordinates: are the final locations on the output devices.

1.2.5 Output Primitives

Shapes and colours of objects can be illustrated internally with pixel arrays or sets of basic geometric arrangements such as straight line segments and polygon colour areas. The functions

Notes

provided by graphics programming packages to deal with these basic geometric structures are called output primitives.

For example:

Drawing a point: SetPixel(100,200,RGB(255,255,0));

Drawing a line: MoveTo(100,100); LineTo(100,200);

Drawing some text: SetText(100,200,"Hello");

Drawing an ellipse: Ellipse(100,100,200,200);

Painting a picture: BitBlt(100,100,50,50,srcImage,0,0,SRCCOPY);

1.2.6 Properties

- In graphical output primitives, objects are normally connected with properties. E.g.

  Point: color

  Line: width, style, color

  Polygon: edge color, filling color

  Text: font size, color, bold or not bold, italic or not, underlined or not, etc.

In graphical packages, we can specify such properties, e.g. In PowerPoint, we can modify the properties of objects by a format command.

In programming tools, we may pass the properties as arguments when we call the functions of these primitives, or we may pre-select the properties before calling the functions.

Self Assessment Questions

1. Image …… is to apply techniques to modify or interpret existing pictures.

   (*a*) applications (*b*) processing

   (*c*) graphics (*d*) technology

2. …….. are electronic gloves for detecting fingers' movement.

   (*a*) Scanners (*b*) Touch panel

   (*c*) Data gloves (*d*) Mouse

3. …… are used to generate drafting layouts and other drawings of normally larger Sizes.

   (*a*) LaserJet (*b*) Inkjet

   (*c*) Dot-matrix (*d*) Pen Plotters

4. …….. are the final locations on the output devices.

   (*a*) Device coordinates (*b*) World coordinates

   (*c*) Normalized coordinates (*d*) Modeling coordinates

5. Modeling coordinates are used to construct individual object shapes.

   (*a*) True (*b*) False

1.3 The Evolution of Computer Graphics

The evolution of computer graphics is intertwined with textual display, and it is difficult to consider the two unconnectedly. An old saying has it that a picture is worth a thousand words.

Notes

The exact quantification of the value of imagery versus text appears to vary somewhat with subject matter, and is probably better left to psychologists and social scientists. But there is little question of the kernel of truth in the saying, and it has been a driver of computer architecture for many years.

Computer graphics are taken for granted today. But it has been a long and painful struggle, with hardware rarely keeping up with the demand for better images. In English, there are a relatively small number of characters which comprise text. The same is not true of images: graphics are computationally intensive. They always seem to take as much speed and memory as there are available. But the demand was high enough that early computer graphics could be fairly crude and still be in demand.

The advance in computer graphics was to come from one MIT student, Ivan Sutherland. In 1961 Sutherland created another computer drawing program called Sketchpad. Using a light pen, Sketchpad allowed one to draw simple shapes on the computer screen, save them and even recall them later. The light pen itself had a small photoelectric cell in its tip. This cell emitted an electronic pulse whenever it was placed in front of a computer screen and the screen's electron gun fired directly at it. By simply timing the electronic pulse with the current location of the electron gun, it was easy to pinpoint exactly where the pen was on the screen at any given moment. Once that was determined, the computer could then draw a cursor at that location.

Sutherland seemed to find the perfect solution for many of the graphics troubles he faced. Even today, many standards of computer graphics interfaces got their start with this early Sketchpad program. One example of this is in drawing constraints. If one wants to draw a square for example, it does not have to worry about drawing four lines perfectly to form the edges of the box. One can simply specify that she/he wants to draw a box, and then specify the location and size of the box. The software will then construct a perfect box, with the right dimensions and at the right location. Another example is that Sutherland's software modelled objects - not just a picture of objects. In other words, with a model of a car, one could change the size of the tires without affecting the rest of the car. It could stretch the body of the car without deforming the tires.

These early computer graphics were Vector graphics, composed of thin lines whereas modern day graphics are Raster based using pixels. The difference between vector graphics and raster graphics can be illustrated with a shipwrecked sailor. He creates an SOS sign in the sand by arranging rocks in the shape of the letters "SOS." He also has some brightly coloured rope, with which he makes a second "SOS" sign by arranging the rope in the shapes of the letters. The rock SOS sign is similar to raster graphics. Every pixel has to be individually accounted for. The rope SOS sign is equivalent to vector graphics. The computer simply sets the starting point and ending point for the line and perhaps bends it a little between the two end points. The disadvantages to vector files are that they cannot represent continuous tone images and they are limited in the number of colours available. Raster formats on the other hand work well for continuous tone images and can reproduce as many colours as needed.

Also in 1961 another student at MIT, Steve Russell, created the first video game, Spacewar. Written for the DEC PDP-1, Spacewar was an instant success and copies started flowing to other PDP-1 owners and eventually even DEC got a copy. The engineers at DEC used it as a diagnostic program on every new PDP-1 before shipping it. The sales force picked up on this quickly enough and when installing new units, would run the world's first video game for their new customers.

E. E. Zajac, a scientist at Bell Telephone Laboratory (BTL), created a film called "Simulation of a two-giro gravity attitude control system" in 1963. In this computer produced film, Zajac showed how the attitude of a satellite could be altered as it orbits the Earth. He created the animation on an IBM 7090 mainframe computer. Also at BTL, Ken Knowlton, Frank Sindon and Michael Noll started working in the computer graphics field. Sindon created a film called Force, Mass and Motion illustrating Newton's laws of motion in operation. Around the same time, other

Notes

scientists were creating computer graphics to illustrate their research. At Lawrence Radiation Laboratory, Nelson Max created the films, "Flow of a Viscous Fluid" and "Propagation of Shock Waves in a Solid Form." Boeing Aircraft created a film called "Vibration of an Aircraft."

It was not long before main corporations started taking an interest in computer graphics. TRW, Lockheed-Georgia, General Electric and Sperry Rand are among the many companies that were getting started in computer graphics by the mid 1960's. IBM was quick to respond to this interest by releasing the IBM 2250 graphics terminal, the first commercially available graphics computer.

Also in 1966, Sutherland at MIT invented the first computer controlled head-mounted display (HMD). Called the Sword of Damocles because of the hardware required for support, it displayed two separate wireframe images, one for each eye. This allowed the viewer to see the computer scene in stereoscopic 3D. After receiving his Ph.D. from MIT, Sutherland became Director of Information Processing at ARPA (Advanced Research Projects Agency), and later became a professor at Harvard.

Dave Evans was director of engineering at Bendix Corporation's computer division from 1953 to 1962. After which he worked for the next five years as a visiting lecturer at Berkeley. There he continued his interest in computers and how they interfaced with people. In 1968 the University of Utah recruited Evans to form a computer science program, and computer graphics quickly became his primary interest. This new department would become the world's primary research centre for computer graphics.

In 1967 Sutherland was recruited by Evans to join the computer science program at the University of Utah. There he perfected his HMD. Twenty years later, NASA would find again his techniques in their virtual reality research. At Utah, Sutherland and Evans were highly sought after consultants by large companies but they were irritated at the lack of graphics hardware available at the time so they started formulating a plan to start their own company.

A student by the name of Ed Catmull got started at the University of Utah in 1970 and signed up for Sutherland's computer graphics class. Catmull had just come from The Boeing Company and had been working on his degree in physics. Growing up on Disney, Catmull loved animation yet quickly discovered that he did not have the talent for drawing. Now Catmull (along with many others) saw computers as the natural progression of animation and they wanted to be part of the revolution. The first animation that Catmull saw was his own. He created an animation of his hand opening and closing. It became one of his goals to produce a feature length motion picture using computer graphics. In the same class, Fred Parkes created an animation of his wife's face. Because of Evan's and Sutherland's presence, UU was gaining quite a reputation as the place to be for computer graphics research so Catmull went there to learn 3D animation.

As the UU computer graphics laboratory was attracting people from all over, John Warnock was one of those early pioneers; he would later found Adobe Systems and create a revolution in the publishing world with his PostScript page description language. Tom Stockham led the image processing group at UU which worked closely with the computer graphics lab. Jim Clark was also there; he would later found Silicon Graphics, Inc.

The first major advance in 3D computer graphics was created at UU by these early pioneers, the hidden-surface algorithm. In order to draw a representation of a 3D object on the screen, the computer must determine which surfaces are "behind" the object from the viewer's perspective, and thus should be "hidden" when the computer creates (or renders) the image.

1.3.1 Blinking Lights to Plotters

Getting computers to type text was, in comparison, an easy process. Still in the early days of computing, there were existing devices which could translate a simple binary prototype into text. The military, for example, had used teletype machines for many years. Programming a computer to output the pattern that outputs the code for a textual character on a teletype machine

Notes

is relatively simple. Early computers used mostly flashing lights, with punched cards or paper tape for input and output. When there is only room for a few hundred instructions, you take input and output in its simplest form. But sometimes the available technology drives applications, and sometimes, the need to do something becomes a driver of looking for new technology.

The latent for getting a computer to produce a picture of the data was not missed. It would be more valuable if the picture were produced rapidly enough for the user to interact, but even producing an image of some sort that represented the computer contents or calculations in the recent past had its merits.

IBM was offering an output printer on its 701 model in 1952. It also offered a primitive graphics solution (the model 740 "Cathode Ray Tube Output Recorder") in 1954. The 740 express just how big the demand for graphics was, and how minimal a capability was considered meaningful. The 740 was a cathode ray tube to which a camera could be attached. Digital-to-analog converters drove the cathode ray tube, slowly drawing lines, based on the digital outputs of the computer. This method gradually came to be known as "vector graphics," to distinguish it from other technologies.

Figure 1.7: An IBM Model 701 with Model 740 Cathode Ray Tube Output Recorder

Lines were plotted one point at a time. IBM justifiably boasted (at the time) that points were plotted at a rate of 8,000 per second, with a display accuracy of a given point of only 3%, but with good repeatability. You could not constantly scale the resulting image, but the image would have at least conceptual value.

Typically, the camera shutter was opened when the drawing started, and closed when it finished. At that time, the film could be developed, and the image could be viewed later the same day. Needless to say, this tech was not suitable for playing video games. Of course, one could maintain a simpler image on the display simply by repeating the drawing instructions at a fairly high rate. But this used most or all of the CPU time, and limited the detail which could be drawn.

The 740 had a sister display, the 780, which had a long-persistence phosphor (20 seconds). While not as precise, when paralleled with the 740, it allowed the operator to verify that the image

Notes

being produced was indeed the one desired. When you have to wait several hours for the film image to be developed, that is a good idea.

But there is another way to get an image with a slow computer and one that will yield an image to the users much sooner: a plotter. Gene Said and Robert Morton, two of the founders of Cal Comp, developed the idea in 1953, but lack of funding kept the device off the market until 1959.

The idea is simple: drive a pen on two axes. That takes a pair of stepping motors, and something to put the pen down at the start of a line, and lift it again at the end. Software can calculate when the pen should be stepped in either axis to draw straight lines between two points, curved lines, or whatever.

In time, fairly sophisticated software packages were developed which isolated the users from the plotter, and allowed them to describe the image in more human-friendly terms such as "m=1, plot y=mx+b for x=1 to 6," and so on. If the plotter drew a shape, the shape could be filled with solid, dashed, or dotted lines to simulate black or greyscale. Calculating the bounds of the lines within the shape became a part of early plotting packages. Unless one is doing ray tracing, or something similar, the bounds for shading or colour within a surface must still be calculated today, and the techniques still draw on these early methods.

In time, plotters had additional pens added to do drawings in colour. But the number of colours was limited, and the drawings were still vectors.

The Odyssey, a home video game, was the first consumer computer graphics product, developed by Ralph Baer in 1966.

Self Assessment Questions

6. Computer graphics generally means creation …….. and manipulation of models and images.
   (*a*) storage (*b*) graphics
   (*c*) algorithm (*d*) None of these.
7. Computer graphics was coined in …….. by William Fetter.
   (*a*) 1970 (*b*) 1960
   (*c*) 1956 (*d*) 1958
8. The refreshing rate, called the frame rate, is normally…….. frames per second.
   (*a*) 50 to 80 (*b*) 60 to 90
   (*c*) 60 to 80 (*d*) 45 to 65
9. Inkjet printers produce output by squirting ink in …….. rows across a roll of paper wrapped on a drum.
   (*a*) horizontal (*b*) grid
   (*c*) vertical (*d*) None of these.
10. Modelling coordinates are used to construct …….. object shapes.
    (*a*) horizontal (*b*) grid
    (*c*) individual (*d*) visual
11. A graphics processor accepts graphics commands from the …….. and executes the graphics commands.
    (*a*) CPU (*b*) RAM
    (*c*) monitor (*d*) All of these.

Notes

12. IBM was offering an output printer on it is …….. model in 1952.

(*a*) 706 (*b*) 703

(*c*) 702 (*d*) 701

13. In graphical …….. primitives, objects are normally associated with properties.

(*a*) output (*b*) roll over

(*c*) input (*d*) All of these.

14. Directing pictures to a printer or plotter to produce hard-copy output on …….. slides overhead.

(*a*) 40-mm (*b*) 35-mm

(*c*) 38-mm (*d*) All of these.

15. The evolution of computer graphics is intertwined with …… display.

(*a*) textual (*b*) output

(*c*) input (*d*) All of these.

1.4 Summary

- Computer graphics generally means creation, storage and manipulation of models and images. Such models come from diverse and expanding set of fields including physical, mathematical, artistic, biological, and even conceptual (abstract) structures.
- A computer image is usually represented as a discrete grid of picture elements a.k.a. pixels. The number pixel determines the resolution of the image.
- Computer graphics comprises the creation and representation of simple graphical elements and images, as well as modern techniques for rendering a virtual reality.
- Fundamentals of computer graphics provide both the theoretical and practical skills to begin developing computer graphics software.
- The digital image processing for which the input is an image, the output of image processing can be either an image or a set of characteristics or parameters related to the image.
- Image processing is to apply techniques to modify or interpret existing pictures. It is widely used in medical applications. Graphical User Interface multiple window, icons, menus allow a computer setup to be utilized more efficiently.
- A graphics processor accepts graphics commands from the CPU and executes the graphics commands which may involve drawing into the frame buffer.

1.5 Keywords

Eigen value: Eigen values are a special set of scalars associated with a linear system of equations (i.e., a matrix equation) that are sometimes also known as characteristic roots, characteristic values.

Frame-buffer: Frame-buffers differ significantly from the vector displays that were common prior to the advent of the frame-buffer. With a vector display, only the vertices of the graphics primitives are stored.

Image Synthesis: A synthesized human image is constructed by first sampling the target by means of 3D scanning and photography and creating a 3D model based on the samples, applying information and approximations.

Notes

Segments: A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Shaded Scenes: Shading refers to depicting depth perception in 3D models or illustrations by varying levels of darkness.

Shadow Mask: A shadow mask is a metal plate punched with minute holes that separate the coloured phosphors in the layer behind the front glass of the screen. Three electron guns at the back of the screen sweep across the mask, with the beams only reaching the screen when they pass over the holes.

Transformations: A function or operator that takes an image as its input and produces an image as its output. Depending on the transform chosen, the input and output images may appear entirely different and have different interpretations.

1.6 Review Questions

1. What is the pixel and also explain?
2. What is the concept of computer graphics?
3. What is the need of computer graphics?
4. Explain the application of computer graphics?
5. Explain the CRT graphics diagram.
6. Discus the input device and output primitives of computer graphics.
7. Discuss the evolution of computer graphics?
8. What are the properties of computer graphics?
9. Define the coordinate representation in graphics.

Answers for Self Assessment Questions

1.	(b)	2.	(c)	3.	(d)	4.	(a)	5.	(a)
6.	(*a*)	7.	(*b*)	8.	(*c*)	9.	(*a*)	10.	(*c*)
11.	(*a*)	12.	(*d*)	13.	(*a*)	14.	(*b*)	15.	(*a*)

1.7 Further Readings

Books

Fundamentals of computer graphics, by Steve Maschner

Computer Graphics Techniques, by David F. Rogers

Online link

http://books.google.co.in/books/about/Fundamentals_of_Computer_Graphics.html?id=WY5Urwcqsa4C&redir_esc=y

Unit 2: Graphics Systems

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the computer graphics system
- Explain the video display devices
- Discuss vector scan/random scan display
- Define raster scan systems
- Explain colour displays
- Understand frame buffer

Introduction

The term computer graphics explains any use of computers to create and manipulate images. Graphics can be two- or three-dimensional; images can be fully synthetic or can be produced by manipulating snaps. Many powerful tools have been developed to imagine data. Computer

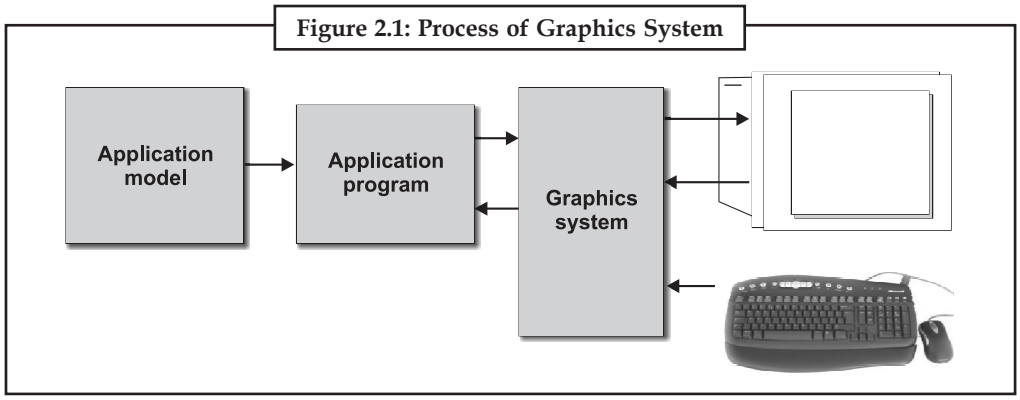
Notes

generated imagery can be categorized into several different types: 2D, 3D, and animated graphics. As technology has improved, 3D computer graphics have become more common, but 2D computer graphics are still broadly used. Computer graphics has emerged as a sub-field of computer science which studies methods for digitally synthesizing and manipulating optical content. Specialized fields have been developed like information visualization, and scientific visualization more concerned with "the visualization of three dimensional phenomena where the emphasis is on realistic renderings of volumes and surfaces".

A picture may take the form of a static image or that of an animated form. Whatever the outcome, a system will need to incorporate mechanisms to promote interaction; between the user and the system itself. For interaction to take place, the system needs to employ tools to facilitate input variation and output devices which work on a transitory basis. Computer graphics involves display, manipulation and storage of pictures and experimental data for suitable visualization using a computer. Typical graphics system comprises of a host computer with support of fast processor, large memory, frame buffer.

- Display devices (colour monitors),
- Input devices (mouse, keyboard, joystick, touch screen, trackball)
- Output devices (LCD panels, laser printers, colour printers. Plotters etc.)
- Interfacing devices such as, video I/O, TV interfaces etc.

Figure 2.1: Process of Graphics System



Computer Graphics systems could be active or passive. In both cases, the input to the system is the scene description and output is a static or animated scene to be displayed. In case of active systems, the user controls the display with the help of a GUI, using an input device. Computer Graphics is at the present time, a significant Computer of approximately all systems and applications of computers in every field of life. Various fundamental concepts and principles in Computer Graphics are:

Display Systems: Storage displays, Random scan, Raster refresh displays, CRT basics, video basics, Flat panel displays.

Transformations: Affine (2D and 3D): Rotation, Translation, Scale, Reflection and Shear. Viewing: The Camera Transformations -perspective, orthographic, isometric and stereographic views, Quaternion.

Scan Conversion and Clipping: Drawing of Points, Lines, Markers, Curves, Circles, Ellipse, Plotline, and Polygon. Area filling, fill-style, fill pattern, clipping algorithms, anti-aliasing etc.

Hidden Surface Removal: Back face culling, Painter's algorithm, scan-line algorithm, BSP-trees, Z-buffer/sorting, Ray tracing etc.

Notes

Shading and Illumination: Phuong's shading model, texture mapping, bump mapping, Gouraud shading, shades and background, Colour models etc.

Solid Modelling: Wire-frame, Octrees, Sweep, Boundary demonstrations. Regularized Boolean set operations, Constructive Solid Geometry.

Curves and Surfaces: Bezier (Bernstein Polynomials) Curves, BSplines, Cubic-Splines, Quadratic surfaces, parametric and non-parametric forms, Hermite Curves etc.

Miscellaneous: Animation, Fractals, Projection and Viewing, Geometry, Modelling, Image File formats, Image Morphing, Interaction (sample and event-driven) etc.

Advanced Raster Graphics Architecture: Display Processors, Pipeline and parallel architectures, multi-processor systems, hybrid architectures.

2.1 Computer Graphics System

The term computer graphics has been used in a broad intellect to explain "approximately everything on computers that is not text or sound". Classically, the word computer graphics refers to several different things: the representation and manipulation of image data by a computer the diverse technologies used to create and manipulate images the images so produced, and the sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Today, computers and computer-generated images touch many pieces of daily life. Computer imagery is found on television, in daily, for example in weather reports, or in all kinds of medical investigation and surgical procedures. A well-constructed graph can present complex statistics in a form that is easier to understand and interpret. In the media "such graphs are used to illustrate papers, reports, thesis", and other presentation material. There are many other areas that involve computer graphics, and whether they are core graphics areas is a matter of opinion. These will all be at least touched on in the text. Such related areas include the following:

User Interaction: It deals with the interface between input devices such as mice and tablets, the application, feedback to the user in imagery, and other sensory feedback. Historically, this area is connected with graphics mainly because graphics researchers had some of the earliest access to the input/output devices that are now everywhere.

Virtual reality: It attempts to immerse the user into a 3D virtual world. This typically requires at least stereo graphics and response to head motion. For true virtual reality, sound and force feedback should be provided as well. Because this area requires advanced 3D graphics and advanced display technology, it is often closely linked with graphics.

Visualization: this attempt to provide users nearby into complex information by visual display. Often there are graphic issues to be addressed in a visualization problem.

Image processing: It deals with the manipulation of 2D images and is used in both the fields of graphics and vision.

3D scanning: It uses range-finding technology to create measured 3D models. Such models are useful for creating rich visual imagery, and the processing of such models often requires graphics algorithms.

Computational photography: This deals with the use of computer graphics, computer vision, and image processing methods to enable new ways of photographically capturing objects, scenes, and environments.

Visualization is a technique for creating images, diagrams, or animations to communicate a message. Visualization through visual imagery has been an efficient way to commune both

Notes

abstract and tangible ideas. Visualization today has ever-expanding applications in science, education, engineering (e.g., product visualization), interactive multimedia, medicine, etc. Typical visualization application is the field of computer graphics. The invention of computer graphics may be the most important development in visualization since the invention of central perspective. The use of visualization to present information is not a new phenomenon. It has been used in maps, scientific drawings, and data plots for over a thousand years. Computer graphics has from its beginning been used to study scientific problems.

Most people are familiar with the digital animations produced to present meteorological data during weather reports on television, though few can distinguish between those models of certainty and the satellite photos that are also shown on such programs. TV also proposes scientific visualizations when it shows computer drawn and animated restorations of road or airplane accidents. Some of the most popular examples of scientific visualizations are: computer-generated images that show real spacecraft in action, out in the void far beyond Earth, or on other planets. Dynamic forms of visualization, such as educational animation or timelines, have the potential to enhance learning about systems that change over time.

Apart from the distinction between interactive visualizations and animation, the most useful categorization is probably between abstract and model-based scientific visualizations. Data visualization is a related subcategory of visualization dealing with statistical graphics and geographic or spatial data (as in thematic cartography) that is abstracted in schematic form.

2.1.1 Scientific Visualization

Scientific visualization is the transformation, selection, or representation of data from simulations or experiments, with an implicit or explicit geometric arrangement, to allow the examination, analysis, and understanding of the data. Traditional areas of scientific visualization are flow visualization, medical visualization, astrophysical visualization, and chemical visualization. There are several different techniques to visualize scientific data, with surface reconstruction and direct volume rendering being the more common.

2.2 Video Display Devices

The display devices are known as output devices. The most commonly used output device in a graphics video monitor. The operations of most video monitors are based on the standard cathode-ray-tube (CRT) design. How the Interactive Graphics display works the modern graphics display is really easy in creation.

It consists of three components:

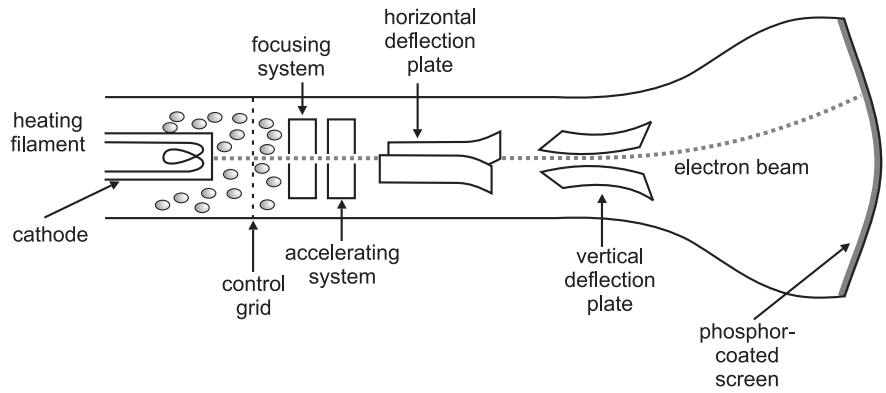
- A digital memory, or frame buffer, in which the displayed Image is stored as a matrix of intensity values.
- A monitor.
- A display regulator which is a simple interface that passes the contents of the frame buffer to the monitor. Inside the frame buffer the image is stored as a pattern of binary digital numbers, which represent a rectangular array of picture components, or pixel. The pixel is the smallest addressable screen element. In the Simplest case where we wish to store only black and white images, we can characterize black pixels by 0's in the frame buffer and white Pixels by 1's. The display regulator simply reads each successive byte of data from the frame buffer and converts each 0 and 1 to the corresponding video signal. This signal is then fed to the monitor. If we wish to change the displayed picture all we need to do is to change of modify the frame buffer contents to represent the new pattern of pixels.

Notes

2.2.1 Cathode-ray-tubes

A CRT is a displaced glass tube. An electron gun at the rear of the tube produces a beam of electrons which is directed towards the front of the tube (screen). The internal side of the screen is coated with phosphor material which gives off light when it is stroked by electrons. It is possible to control the point at which the electron beam hits the screen, and thus the position of the dot upon the screen, by deflecting the electron beam. The fig below shows the electrostatic deflection of the electron beam in a CRT. The deflection system of the cathode-ray-tube consists of two pairs of parallel plates, referred to as the vertical and horizontal deflection plates. The voltage applied to vertical plates controls the vertical deflection of the electron beam and voltage applied to the horizontal deflection plates controls the horizontal deflection of the electron beam. There are two techniques used for producing image on the CRT screen.

Figure 2.2: Cathode-ray-tube (CRT)



A cathode ray tube (CRT) is a dedicated vacuum tube in which images are produced when an electron beam strikes a phosphorescent surface. Most desktop computer displays make use of CRTs. The CRT in a computer display is similar to the "picture tube" in a television receiver.

All CRT's have three main elements: an electron gun, a deflection system, and a screen. The electron gun provides an electron beam, which is a highly concentrated stream of electrons. The deflection system positions the electron beam on the screen, and the screen displays a small spot of light at the point where the electron beam strikes it.

Basic Operation of a CRT

The basic operation of CRT is as follows:

Electron Gun: The primary parts of an electron gun in a CRT are the heated metal cathode and a control grid. The cathode is heated by an electric current passed through a coil of wire called the filament. This causes electrons to be boiled off the hot cathode surface. In the vacuum inside the CRT envelope, negatively charged electrons are then accelerated toward the phosphor coating by a high positive voltage. The accelerating voltage can be generated with a positively charged metal coating on the inside of the CRT envelope near the phosphor screen, or an accelerating anode can be used. Sometimes the electron gun is built to hold the accelerating anode and focusing system within the same unit.

Focusing System: The focusing system is used to produce a clear picture by focusing the electrons into a narrow beam. Otherwise, electrons would repel each other and beam would spread out as it reaches the screen. Focusing is accomplished with either electric or magnetic fields.

Deflection System: Deflection of the electron beam can be controlled by either electric fields or magnetic fields. In case of magnetic field, two pairs of coils are used, one for horizontal deflection

Notes

and other for vertical deflection. In case of electric field, two pairs of parallel plates are used, one for horizontal deflection and next for vertical.

The CRT Screen: The inside of the large end of a CRT is covered with a fluorescent material that gives off light when struck by electrons. When the electrons in the beam is collides with phosphor coating screen, they stopped and their kinetic energy is absorbed by the phosphor. Then a part of beam energy is changed into heat energy and the remainder part causes the electrons in the phosphor atom to move up to higher energy levels. After a short time the excited electrons come back to their ground state. During this period, we see a glowing spot that quickly fades after all excited electrons are returned to their ground state.

Persistence: It is defined as the time they continue to produce light after the CRT beam is removed. Persistence is defined as the time it takes the emitted light from the screen to decay to one-tenth of its original intensity. Lower-persistence phosphors require higher refresh rates to maintain a picture on the screen without flicker. A phosphor with low persistence is useful for animation; a high-persistence phosphor is useful for displaying highly complex, static pictures. Although some phosphor have persistence greater than 1 second, graphics monitor are usually constructed with persistence in the range from 10 to 60 microseconds.

Resolution: The number of points per centimetre that can be used be plotted horizontally and vertically or total number of points in each direction.

The resolutions of a CRT is depend on

- type of phosphor
- intensity to be displayed
- focusing and deflection system

Aspect Ratio: It is ratio of horizontal to vertical points. Example: An aspect ratio of 3/4 means that a vertical line plotted with three points has equal length as horizontal line plotted with four points.

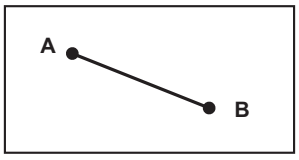
Did u know?

The first cathode ray tube to use a hot cathode was developed by John B. Johnson (who gave his name to the term Johnson noise) and Harry Weiner Weinhart of Western Electric, and became a commercial product in 1922.

2.3 Vector Scan/Random Scan Display

A vector scan CRT display directly mark outs out only the desired lines on CRT i.e. if we want a line connecting point A with point B on the vector graphics display; we simply drive the beam deflection circuitry, which will cause beam to go directly from point A to B. If we want to move the beam from point A to point B without showing a line between points, we can vacant the beam as we move it. To move the beam across CRT, the information about, magnitude and direction is required. This information is generated with the help of vector graphics generator.

Figure 2.3: Vectors Scan Display

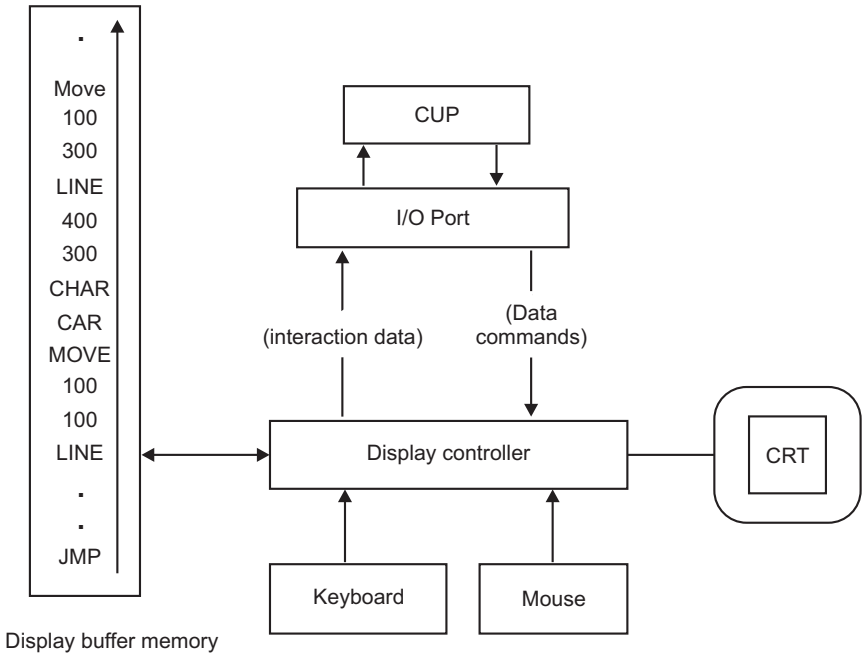


The typical vector display architecture. It consists of display controller, central processing unit (CPU), display buffer memory and a CRT. A display controller is connected as an I/O peripheral

Notes

to the central processing unit. The display buffer memory stores the computer produced display list or display program. The program holds point and line plotting commands with (x, y) or (x, y, z) end point co-ordinates, as well as character plotting commands. The display controller interprets commands for plotting points, lines and characters and sends digital and point coordinates to a vector generator. The vector generator then changes the digital coordinate values to analogue voltages for beam-deflection circuits that relocate an electron beam writing on CRT's phosphor coating.

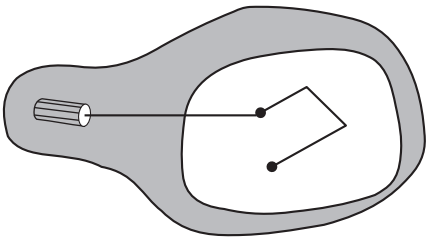
Figure 2.4: Vector Display Architecture



In vector display beam is deflected from end point to end point, therefore this technique is also called random scan. We know as beam, strikes phosphor it produces light. But a phosphor light decay after few milliseconds and therefore it is necessary to repeat through the display list to refresh the phosphor at least 30 times per second to avoid flicker. As display buffer is used to store display list and it is used for refreshing, the display buffer memory is also called refresh buffer.

In Random Scan System, an electron beam is directed to only those parts of the screen where a picture is to be drawn. The picture is drawn one line at a time, so also called vector displays or stroke writing displays. After drawing the picture the system cycles back to the first line and design all the lines of the picture 30 to 60 time each second.

Figure 2.5: Random Scan System



Notes

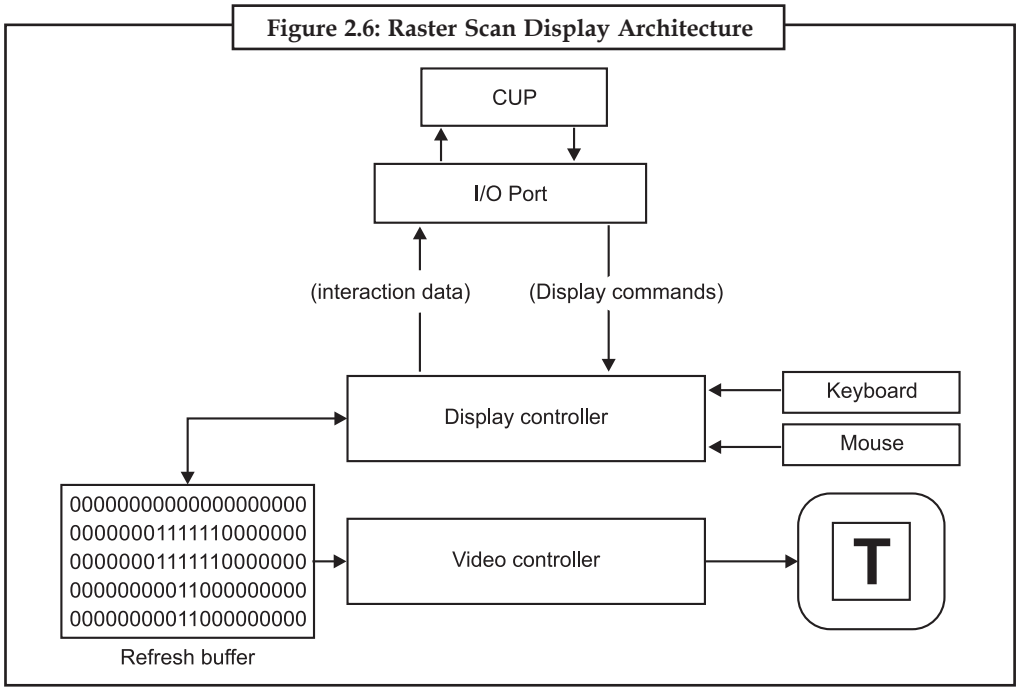
Self Assessment Questions

1. …… attempts to give users insight into complex information via visual display.
   (*a*) 3D scanning (*b*) Image processing
   (*c*) Virtual reality (*d*) Visualization
2. …….. is a related subcategory of visualization dealing with statistical graphics and geographic that is abstracted in schematic form.
   (*a*) Data visualization (*b*) Astrophysical visualization
   (*c*) Medical visualization (*d*) Chemical visualization
3. …….. are several different techniques to visualize scientific data, with surface reconstruction and direct volume rendering being the more common.
   (*a*) Flow visualization (*b*) Astrophysical visualization
   (*c*) Medical visualization (*d*) All of these
4. The primary components of an …… in a CRT are the heated metal cathode and a control grid.
   (*a*) focusing system (*b*) electron gun
   (*c*) deflection system (*d*) CRT screen
5. Does not display controller interprets commands for plotting points, lines and characters and sends digital and point coordinates to a vector generator.
   (*a*) True (*b*) False

2.4 Raster Scan Systems

It consists of display controller, central processing unit (CPU), video controller, refresh buffer, keyboard, mouse and the CRT.As shown in the fig the display image is stored in the form of 1s and 0s in the refresh buffer. The video controller reads this refresh buffer and creates the actual image on the screen. It does this by scanning one scan line at a time, from top to bottom and then back to the top.

Figure 2.6: Raster Scan Display Architecture

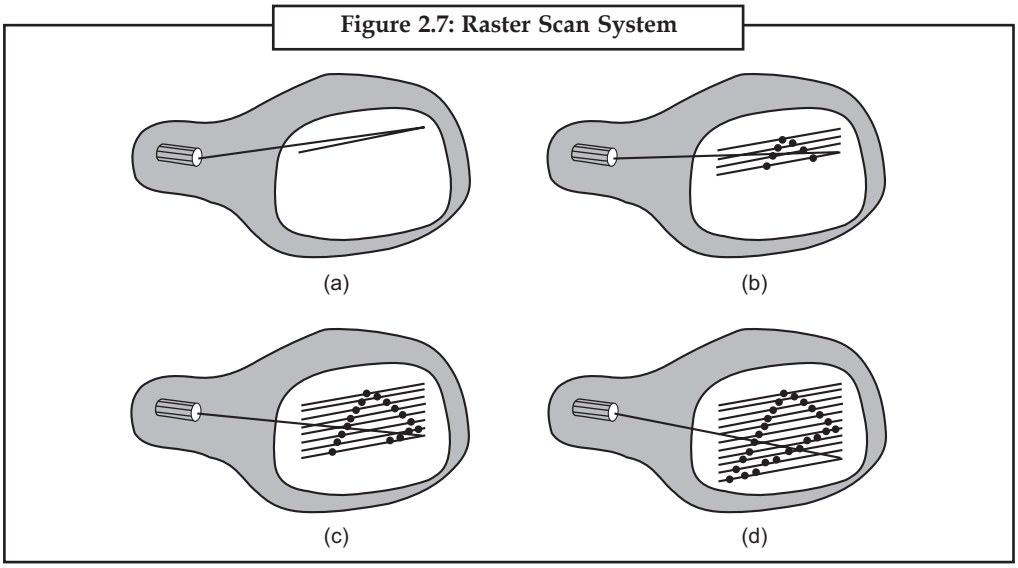


Notes

Raster scan is the most ordinary method of displaying images on the CRT screen. In this method, the horizontal and vertical deflection signals are generated to shift the beam all over the screen in a pattern. Here, the beam is swept back and forth from the left to the right across the screen. When the beam is shifted from the left to the right, it is ON. The beam is OFF, when it is shifted from the right to the left. When the beam reaches the bottom of the screen, it is made OFF and rapidly retraced back to the top left to start once more. A display produced in this way is called raster scan display. In raster scan displays a special area of memory is dedicated to graphics only. This memory area is called frame buffer. It holds the set of intensity values for all the screen points. The stored intensity values are retrieved from frame buffer and displayed on the screen one row (scan line) at a time. Each screen point is referred to as a pixel. Each pixel on the screen can be specified by its row and column number. Thus by specifying row and column number we can specify the pixel position on the screen. Intensity range for pixel positions depends on the capability of the raster system. It can be a simple black and white system or colour system. In a simple black and white system, each pixel position is either on or off, so only one bit per pixel is needed to control the intensity of the pixel positions. Additional bits are required when colour and intensity variations can be displayed. Up to 24 bits per pixel are included in high quality display systems, which can require several megabytes of storage space for the frame buffer. On a black and white system with one bit per pixel, the frame buffer is called a bitmap. F or systems with multiple bits per pixel, the frame buffer is often referred to as a pixmap.

It is the most common type of graphics monitor based on television technology. In a raster scan system, the electron beam is swept across the screen, one row at a time from top to bottom. When electron beam moves across each row the beam intensity is turned ON and OFF to generate a pattern of illuminated spots. Picture meaning is stored in a memory called frame buffer which holds the set of intensity values, which are then retrieved from the frame buffer and indicated on the screen one row at a time as shown.

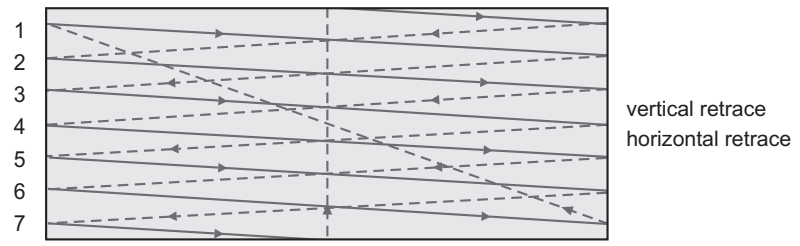
Figure 2.7: Raster Scan System



At the end of each line the beam should be turned off and redirect to the left hand side of the CRT, this is called Horizontal Retrace. At the end of each frame, the electron beam return to top left corner of the screen to begin the next frame called Vertical Retrace.

Notes

Figure 2.8: Graph Representation



Advantages

- Produce realistic images
- Also produced different colours
- And shadows scenes

Disadvantages

- Low resolution
- Expensive
- Electron beam directed to whole screen

2.4.1 Difference between Raster Scan and Random Scan Technique

Raster-scan System

- Raster displays have less resolution.
- The lines produced are *ziz-zag* as the plotted values are discrete.
- High degree realism is achieved in picture with the aid of advanced shading and hidden surface technique.
- Decreasing memory costs have made raster systems popular.

Random Scan System

- Random displays have high resolutions since the picture definition is stored as a set of line drawing commands and not as a set of intensity values.
- Smooth lines are produced as the electron beam directly follows the line path.
- Realism is difficult to get.
- Random-scan system's are generally costlier

2.4.2 Important Characteristics of Video Display Devices

Following are the characteristics of video display devices:

Persistence: The major difference between phosphors is their persistence. It decides how long they continue to emit light later than the electron beam is removed. Persistence is defined as the time it takes the emitted light from the screen to decay to one-tenth of its original intensity. Lower persistence phosphors require higher refreshing rates to maintain a picture on the screen without flicker. However it is useful for displaying animations. On the other hand higher persistence phosphors are valuable for displaying static and highly complex pictures.

Resolution: Resolution indicates the maximum number of points that can be displayed without overlap on the CRT. It is defined as the number of points per centimetre that can be plotted horizontally and vertically. Resolution depends on the type of phosphor, the intensity to be displayed and the focusing and deflection systems used in the CRT.

Aspect Ratio: It is the ratio of vertical points to horizontal points to produce equal length lines in both directions on the screen. An aspect ratio of 4/5 means that a vertical line plotted with four points has the same length as a horizontal line plotted with five points.

Notes

In 1991, S3 Graphics introduced the S3 86C911, which its designers named after the Porsche 911 as a suggestion of the performance increase it promised. The 86C911 spawned a host of imitators: by 1995, all major PC graphics chip makers had added 2D acceleration support to their chips.

2.5 Resolution

It refers to the sharpness and clearness of a picture. The word is most often used to explain monitors, printers, and bit-mapped graphic images. In the case of dot-matrix and laser printers, the resolution shows the number of dots per inch. For example, a 300-dpi (dots per inch) printer is one that is able of printing 300 distinct dots in a line 1 inch long. This means it can print 90,000 dots per square inch.

For graphics monitors, the screen resolution signifies the number of dots (pixels) on the whole screen. For example, a 640-by-480 pixel screen is capable of displaying 640 distinct dots on each of 480 lines, or about 300,000 pixels. This translates into different dpi measurements depending on the size of the screen. For example, a 15-inch VGA monitor (640×480) displays about 50 dots per inch.

Printers, monitors, scanners, and other I/O devices are frequently classified as high resolution, medium resolution, or low resolution. The real resolution ranges for each of these grades is continually shifting as the technology improves.

The basic building block of any graphic image is a pixel, a reduction of "Picture Element", arranged in precise rows and columns. The number of rows and columns of pixels is referred to as the "resolution" of the image and is typically expressed by the number of horizontal pixels (rows) multiplied by the number of vertical pixels (columns), for example: 800*600, 1024*768, 1152*864. Note that these resolutions are sized at a 4:3 ratio. The motive for this is that monitors are manufactured with a 4:3 aspect ratio, so for a pixel to show square it should appear at the same 4:3 width to height aspect ratio.

However, the resolution alone defines nothing except the physical size of the image. Each of these resolutions also has an associated "bit depth", which defines the number of colours that can be presented. The bit depth refers to how many bits of data are associated with each pixel and are available to store a colour value. A 24 bit image, commonly referred to as "true colour", allocates for the storage of 24 bits of data per pixel. To clarify the measurement, there are of course 8 bits in every byte. This raises the question of how colours are defined.

2.6 Colour Displays

Accepting how colours are defined in graphics data is important to understanding graphics file formats. In this, we touch on a few of the many factors governing how colours are perceived. We just want to make certain that you have an approval of some of the problems that come up when people start to deal with colour.

2.6.1 How we See Colour

The eye has a finite number of colour receptors that, taken together, respond to the full range of light frequencies (about 380 to 770 nanometres). As a result, the eye hypothetically supports only the perception of about 10,000 different colours simultaneously (although, as we have mentioned, many more colours than this can be perceived, though not resolved simultaneously).

The eye is also biased to the kind of light it detects. It is most sensitive to green light, followed by red, and then blue. It is also the case that the visual perception system can sense contrasts between adjacent colours more simply than it can sense complete colour differences, particularly if those colours are physically divided in the object being viewed. In addition, the ability to discern colours varies from person to person; it is been estimated that one out of every twelve people has some form of colour blindness.

In addition, the eye is limited in its ability to resolve the colour of little objects. The size of a pixel on a typical CRT display screen, for example, is less than a third of a millimetre in diameter.

Notes

When a large number of pixels are packed together, each one a different colour, the eye is unable to resolve where one pixel ends and the next one begins from a normal viewing distance. The brain, however, must do something to bridge the gap between two adjacent differently coloured pixels and will integrate average, ignore the blur, or otherwise adapt to the situation. For these reasons and others, the eye typically perceives many fewer colours than are physically displayed on the output device.

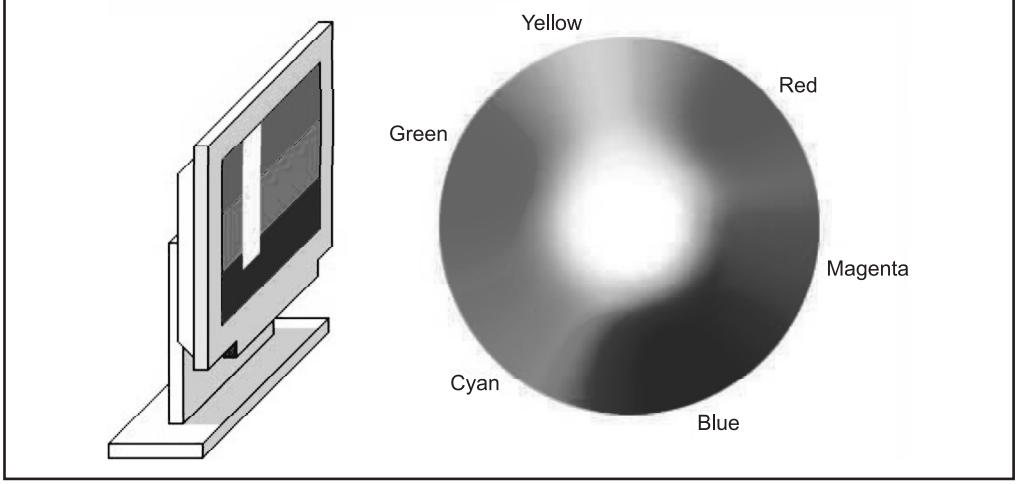
How a colour is created also plays an important role in how it is perceived. Normally, we think of colours as being associated with a single wavelength of light. We know, however, that two or more colours can be mixed together to produce a new colour. An example of this is mixing green and red light to produce yellow light, or mixing yellow and blue pigments to produce green pigment. This mixing of colours can also occur when an object is illuminated by light. The colour of the object will always mix with the colour of the light to produce a third colour. A blue object illuminated by white light appears blue, while the same object illuminated by red light will appear violet in colour.

One implication of this is that the same image rendered to two different devices will look different. Two different colour monitors, for example, seldom produce identically perceived images, even if the monitors are the same make and model. Another implication is that images rendered to different types of devices will look different. An example is the difference between an image rendered to a monitor and one rendered to a colour hardcopy device. Although there are numerous schemes designed to minimize colour-matching problems, none is wholly satisfactory.

For these and other reasons, the accurate rendition of colour is full of difficulties, and much work continues to be done. Although a number of mechanical devices have recently appeared on the market, they are for the most part designed to work with one type of output device. The ultimate arbiter of colour quality will always be the person who views the image on the output device.

Colour monitors for desktop microcomputers are based on cathode ray tubes (CRTs) or back-lighted flat-screen technologies. Because monitors transmit light, displays use the red-green-blue (RGB) additive colour model. The RGB model is called "additive" because a combination of the three pure colours red, green, and blue "adds up" to white light:

Figure 2.9: The RGB Model



The computer's operating system organizes the display screen into a grid of *x* and *y* coordinates, like a checkerboard. Each little box on the screen is called a "pixel" (short for "picture element"). Current Macintosh and Windows displays are composed of these grids of pixels.

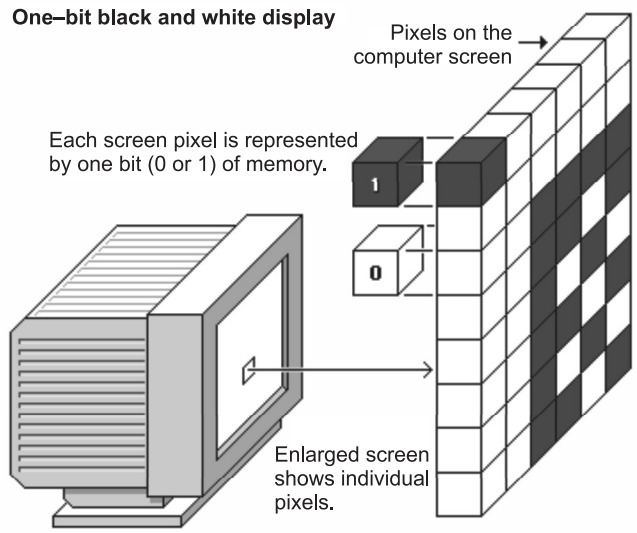
2.6.2 Pixels and Colour Depth

To control the colour of each pixel on the screen, the operating system must grant a small amount of memory to each pixel. In collective this memory dedicated to the display screen is often

referred to as "video RAM" or "VRAM" (Video Random Access Memory). In the simplest form of black-and-white computer displays, a single bit of memory is allocated to each pixel. Because each memory bit is either positive or negative (0 or 1), a 1-bit display system can manage only two colours (black or white) for each pixel on the screen:

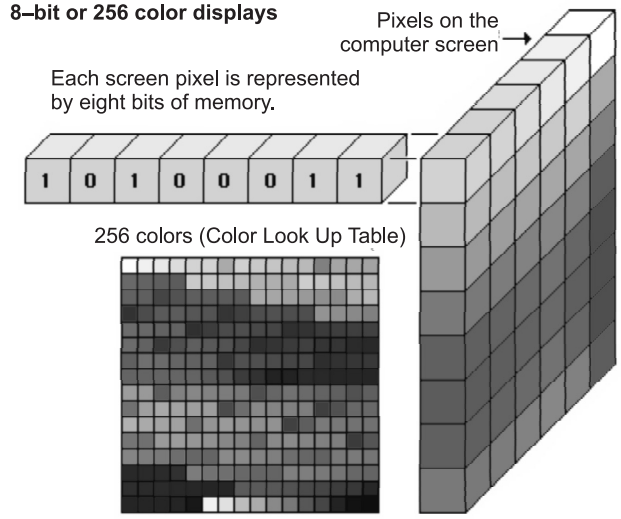
Notes

Figure 2.10: One-bit Black and White Display



If more bits of memory are dedicated to each pixel in the display, more colours can be directed. When 8 bits of memory are dedicated to each pixel, each pixel could be one of 256 colours. (256 = 2 to the eighth power; in other words, 256 is the maximum number of unique combinations of zeros and ones you can make with 8 bits.) This kind of computer display is called an "8-bit" or "256-colour" display, and is common on older laptop computers and desktop machines. Although the exact colours that an 8-bit screen can display are not fixed, there can never be more than 256 unique colours on the screen at once:

Figure 2.11: 8-bit or Colour Displays

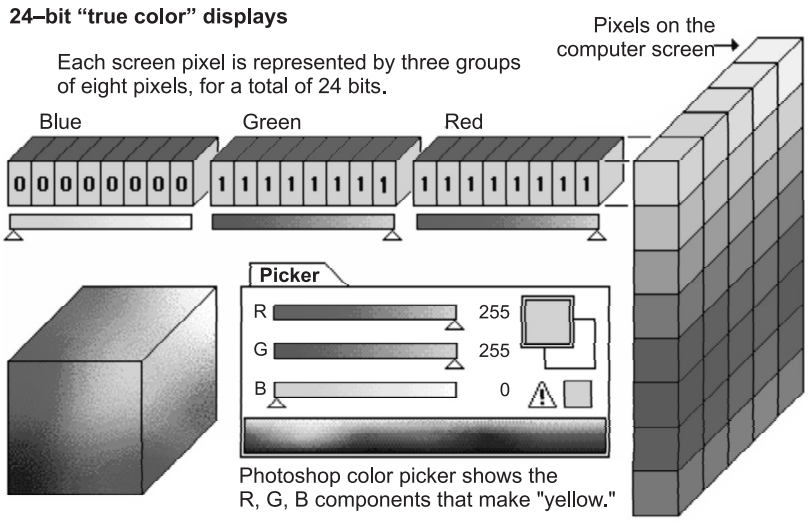


If still more memory is dedicated to each pixel, nearly photographic colour is realizable on the computer screen. "True-colour" or "24-bit" colour displays can show millions of unique colours

Notes

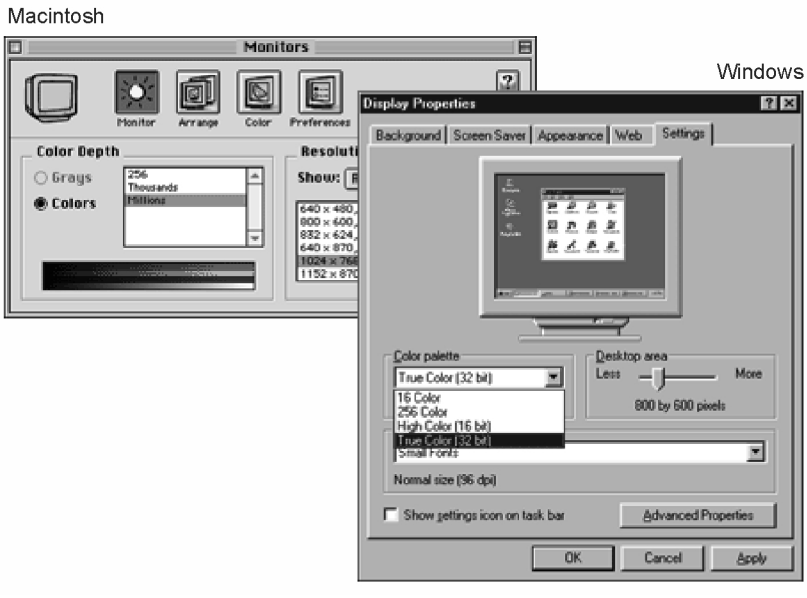
simultaneously on the computer screen. True-colour images are created by dedicating 24 bits of memory to each pixel; 8 each for the red, green, and blue components (8 + 8 + 8 = 24):

Figure 2.12: 24-bit True Colour Displays



The amount of VRAM dedicated to each screen pixel in the display is commonly referred to as the "colour depth" of the monitor. Most Macintosh and Windows microcomputers sold in recent years can simply display colour deepness in thousands (16-bit) or millions (24-bit) of simultaneous colours. To check your computer system for the variety of colour depths available to you, use the "Display" control panel (Windows) or the "Monitors" control panel (Macintosh):

Figure 2.13: Range of Colour Depths Available



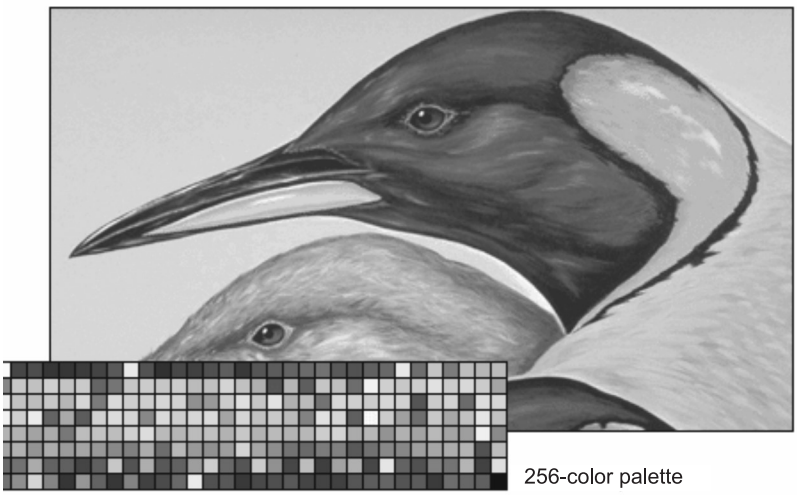
2.6.3 Colour Depth and Graphics Files

The terminology and memory systems used in colour displays are directly analogous to those used to describe colour depth in graphics files. In their uncompressed states, 8-bit, or 256-colour,

Notes

image files dedicate 8 bits to each colour pixel in the image. In 8-bit images the 256 colours that make up the image are stored in an array called a "palette" or an "index." The colour palette may also be referred to as a "colour lookup table" (CLUT). As mentioned above, 8-bit images can never contain more than 256 exclusive colours:

Figure 2.14: 256 Colour Palette

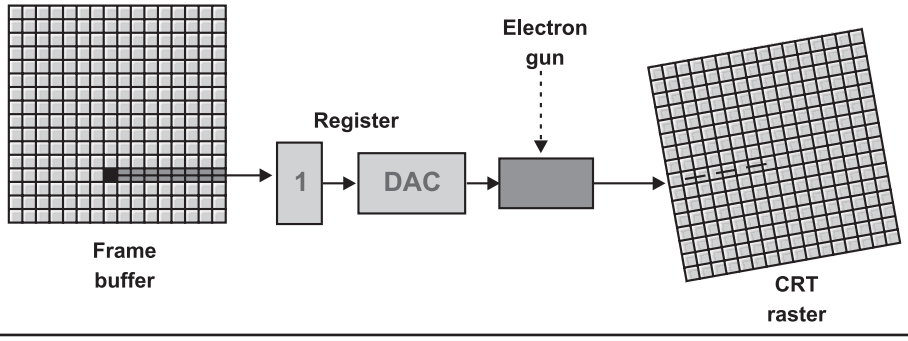


True-colour, or 24-bit, images are typically much larger than 8-bit images in their uncompressed state, because 24-bits of memory are dedicated to each pixel, typically arranged in three monochrome layers red, green, and blue:

2.7 Frame Buffer

A frame buffer is a huge, close part of computer memory. At a least there is one memory bit for each pixel in the rater; this amount of memory is called a bit plane. The picture is built up in the frame buffer one bit at a time. You know that a memory bit has only two states, and then a single bit plane yields a black-and white display. You know that a frame buffer is a digital device and the CRT is an analogue device. Therefore, a change from a digital representation to an analogue signal must take place when information is read from the frame buffer and presented on the raster CRT graphics device. For this you can use a digital to analogue converter (DAC). Each pixel in the frame buffer must be accessed and converted before it is visible on the raster CRT.

Figure 2.15: A Single Bit-pack White Frame Buffer Raster Graphic Device

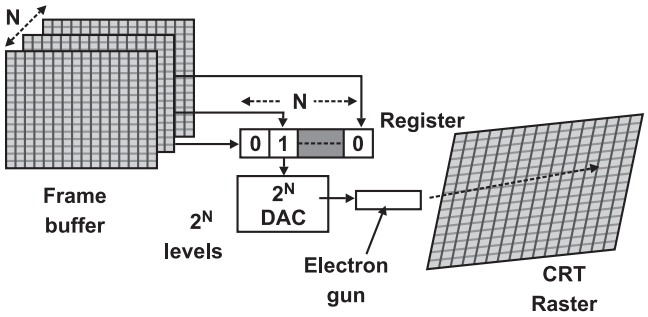


Notes

2.7.1 N-bit Colour Frame Buffer

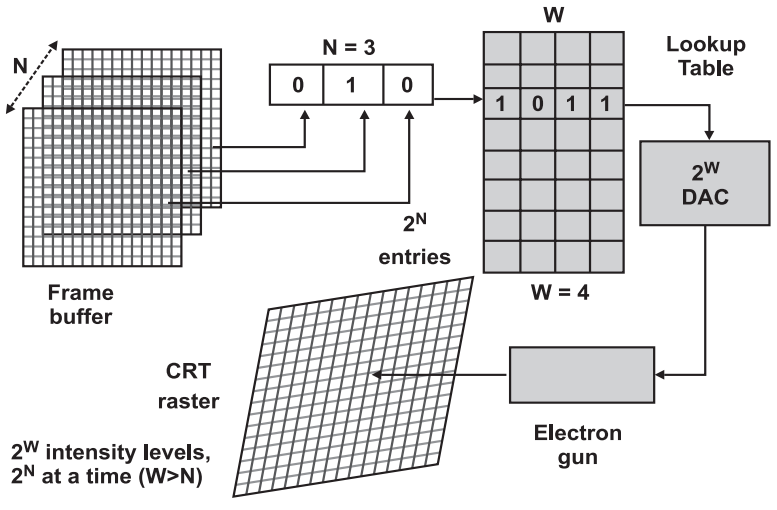
Colour or gray scales are included into a frame buffer raster graphics device by using additional bit planes. The intensity of each pixel on the CRT is controlled by a corresponding pixel location in each of the N bit planes. The binary value from each of the N bit planes is loaded into corresponding locations in a register. The resulting binary number is interpreted as an intensity level between 0 (dark) and 2^n -1 (full intensity). This is exchanged into an analogue voltage between 0 and the maximum voltage of the electron gun by the DAC. A total of 2^N intensity levels are possible. Show illustrates a system with 3 bit planes for a total of 8 (2^3) intensity levels. Each bit plane requires the full complement of memory for a given raster resolution; e.g., a 3-bit plane frame buffer for a 1024 X 1024 raster requires 3,145,728 (3 X 1024 X 1024) memory bits.

Figure 2.16: N-bit Plane Gray Level Frame Buffer



An increase in the number of available intensity levels is achieved for a reserved increase in required memory by using a lookup table. Upon reading the bit planes in the frame buffer, the resulting number is used as an index into the lookup table. The look up table must contain 2^N entries. Each entry in the lookup table is W bit wise. W may be greater than N. When this occurs, 2^W intensities are available; but only 2^N different intensities are available at one time. To get additional intensities, the lookup table must be changed.

Figure 2.17: 2^n-bit Plane Gray Level Frame Buffer

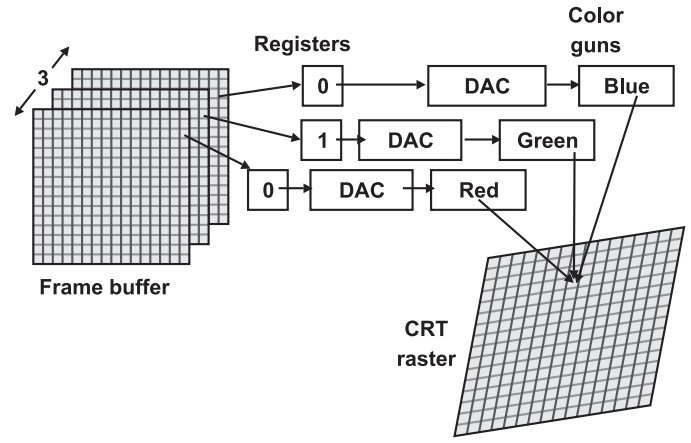


Because there are three primary colours, a simple colour frame buffer is implemented with three bit planes, one for each primary colour. Each bit plane drives an individual colour gun for each

of the three primary colours used in colour video. These three primaries (red, green, and blue) are combined at the CRT to yield eight colours.

Notes

Figure 2.18: Simple Colour Buffer



Self Assessment Questions

6. The display devices are known as …….. devices.

   (*a*) input (*b*) output

   (*c*) storage (*d*) None of these

7. The operations of most video monitors are based on the standard …… design.

   (*a*) raster-scan-system (*b*) Random Scan Display

   (*c*) cathode-ray-tube (*d*) None of these

8. Deflection System of the electron beam can be controlled by either ……..

   (*a*) electric or magnetic fields (*b*) magnetic or electric field

   (*c*) magnetic field (*d*) None of these

9. Raster-scan system displays have …… resolution.

   (*a*) high (*b*) less

   (*c*) high to less (*d*) None of these

10. It is the ratio of …… to produce equal length lines in both directions on the screen.

    (*a*) vertical to horizontal (*b*) horizontal to vertical

    (*c*) only horizontal (*d*) only vertical

11. A raster scan system, the electron beam is swept across the screen ……..…. at a time from top to bottom.

    (*a*) two row (*b*) three row

    (*c*) one row (*d*) one too many

12. The function of a plotter is like a…….

    (*a*) monitor (*b*) projector

    (*c*) printer (*d*) None of these

Notes

13. The term computer graphics has been used in a broad sense to describe "almost everything on computers that is not text or sound".

    (*a*) True (*b*) False

14. A cathode ray tube (CRT) is not a specialized vacuum tube in which images are produced when an electron beam strikes a phosphorescent surface.

    (*a*) True (*b*) False

15. Resolution indicates the maximum number of points that cannot be display without overlap on the CRT.

    (*a*) True (*b*) False

2.8 Summary

- The term computer graphics describes any use of computers to create and manipulate images. Graphics can be two- or three-dimensional; images can be completely synthetic or can be produced by manipulating photographs.
- Computer Graphics systems could be active or passive. In both cases, the input to the system is the scene description and output is a static or animated scene to be displayed.
- The term computer graphics has been used in a broad sense to describe "almost everything on computers that is not text or sound".
- Visualization is a technique for creating images, diagrams, or animations to communicate a message. Visualization through visual imagery has been an effective way to communicate both abstract and concrete ideas.
- Data visualization is a related subcategory of visualization dealing with statistical graphics and geographic or spatial data (as in thematic cartography) that is abstracted in schematic form.
- A CRT is an evacuated glass tube. An electron gun at the rear of the tube produces a beam of electrons which is directed towards the front of the tube (screen).
- Colour monitors for desktop microcomputers are based on cathode ray tubes (CRTs) or back-lighted flat-screen technologies.

2.9 Keywords

Boolean Set: Boolean algebra also deals with other values on which Boolean operations can be defined, such as sets or sequences of bits. However, Boolean algebra is unlike many other systems of algebra in that it obeys exactly the same laws (educational properties), neither more nor fewer, no matter which of these other values are employed. Much of the subject can therefore be introduced without reference to any values besides 0 and 1.

Bump Mapping: Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object. This is achieved by perturbing the surface normals of the object and using the perturbed normal during lighting calculations.

Pixel: In digital imaging, a pixel, or palette, (picture element) is a physical point in a raster image, or the smallest, addressable element in a display device; so it is the smallest, controllable element of a picture represented on the screen.

Texture Mapping: A texture map is applied (mapped) to the surface of a shape or polygon. This process is akin to applying patterned paper to a plain white box. Every vertex in a polygon is

Notes

assigned a texture coordinate (which in the 2d case is also known as a UV coordinate) either via explicit assignment or by procedural definition.

Vacuum Tube: Vacuum tubes are thus used for rectification, amplification, switching, or similar processing or creation of electrical signals. All modern day tubes consist of a sealed container with a vacuum inside, and essentially rely on thermionic emission of electrons from a hot filament or hot cathode.

Visualization: Visualization is any technique for creating images, diagrams, or animations to communicate a message. Visualization through visual imagery has been an effective way to communicate both abstract and concrete ideas since the dawn of man.

2.10 Review Questions

1. Explain the various fundamental concepts and principles in computer graphics.
2. What is the computer graphics system and also explain the some component?
3. What is the visualization?
4. Explain the video display devices.
5. What are the cathode-ray-tubes? And basic operation of a cathode-ray-tubes.
6. Explain the vector scan/random scan display. And write the advantages and disadvantages raster scan systems.
7. What is the difference between raster scan and random scan technique?
8. What is the resolution?
9. Explain the colour displays? How do we see colour display.
10. What is the frame buffer?

Answers for Self Assessment Questions

1.	(*b*)	2.	(*a*)	3.	(*d*)	4.	(*b*)	5.	(*b*)
6.	(*b*)	7.	(*c*)	8.	(*a*)	9.	(*b*)	10.	(*a*)
11.	(*c*)	12.	(*c*)	13.	(*a*)	14.	(*a*)	15.	(*b*)

2.11 Further Readings

Books

Advances in Computer Graphics Hardware II, by Alphonsus A. M. Kuijk

Interactive Computer Graphics, by Angel

Online link

http://books.google.co.in/books?id=btLtyXQsJIC&pg=PA65&dq=graphics+system+in+computer+science&hl

Notes

Unit 3: Implementing Line Algorithm

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of implementing line algorithm
- Explain the concept of Bresenham's algorithm
- Define the DDA line algorithm

Introduction

Line drawing is our first endeavor into the area of scan conversion. The need for scan conversion, techniques is a direct result of scanning nature of raster displays (thus the names). Vector displays are mainly well suited for the display of lines. All that is needed on a vector display to generate a line is to supply the appropriate control voltages to the x and y deflection circuitry, and the electron beam would traverse the line illuminating the desired segment. The only inaccuracies in the lines drawn a vector display resulted from various non-line arties, such as quantization and amplifier saturation, and the various noise sources in the display circuitry. When raster displays came along the process of drawing lines became more difficult. Luckily, raster display pioneers could benefit from previous work done in the area of digital plotter algorithms. A pen-plotter is a hardcopy device used primarily to display engineering line drawings. Digital plotters, like raster displays, are discretely addressable devices, where position of the pen on a plotter is controlled by special motors called stepper motors that are connected to mechanical linkages that translate the motor's rotation into a linear translation. Stepper motors can precisely turn a fraction of a rotation (for example 2 degrees) when the proper controlling voltages are

Notes

applied. A typical flat-bed plotter uses two of these motors, one for the *x*-axis and a second for the ***y***-axis, to control the position of a pen over a sheet of paper. A solenoid is used to raise and lower the actual pen when drawing and positioning.

3.1 Concept of Implementing Line Algorithm

The essential "line drawing" algorithm used in computer graphics is Bresenham's Algorithm. This algorithm was developed to draw lines on digital plotters, but has found broad-spread usage in computer graphics. The algorithm is fast it can be applied with integer computations only and very easy to explain.

The following types of implementing line algorithm are:

1. Bresenham's Line Algorithm
2. DDA Line Algorithm

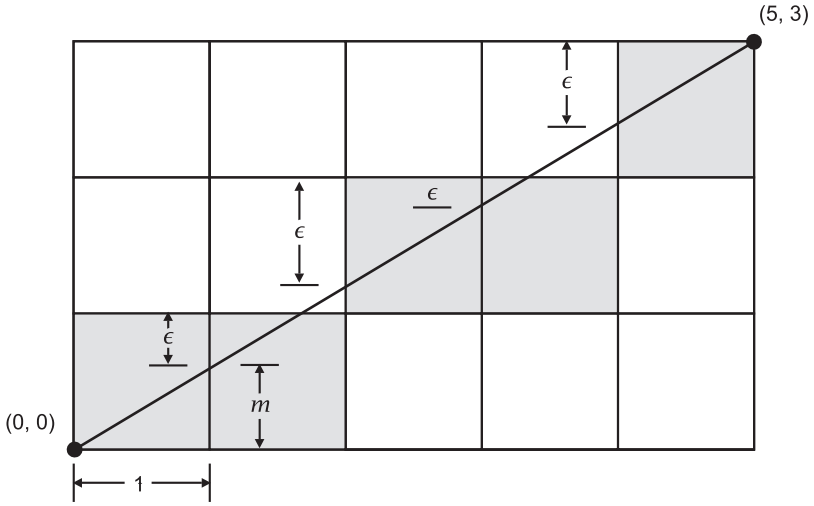
3.2 Concept of Bresenham's Algorithm

The Bresenham Algorithm for drawing lines on the discrete plane, for instance computer monitor is one of the primary algorithms in computer graphics.

3.2.1 Bresenham's Algorithm

The Bresenham line algorithm is an algorithm which decides which points in an n-dimensional raster should be plotted so as to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures. It is one of the initial algorithms developed in the field of computer graphics. A minor extension to the original algorithm also deals with drawing circles.

Consider a line with initial point (x_1, y_1) and terminal point (x_2, y_2) in device space. If $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$, we define the driving axis (DA) to be the x-axis if $|\Delta x| \geq |\Delta y|$, and the y-axis if $|\Delta y| > |\Delta x|$. The DA is used as the "axis of control" for the algorithm and is the axis of maximum movement. Within the main loop of the algorithm, the coordinate corresponding to the DA is incremented by one unit. The coordinate corresponding to the other axis (usually denoted the passive axis or PA) is only incremented as needed. The best way to describe Bresenham's algorithm is to work through an example. Consider the following example, in which we wish to draw a line from (0, 0) to (5, 3) in device space.



Notes

Bresenham's algorithm begins with the point (0, 0) and "illuminates" that pixel. Since x is the DA in this example, it then increments the x coordinates by one. Rather than keeping track of the y coordinate (which increases by m = Δy/Δx, each time the x increases by one), the algorithm keeps an error bound at each stage, which represents the negative of the distance from the point where the line exits the pixel to the top edge of the pixel. This value is first set to m − 1, and is incremented by m each time the x-coordinate is incremented by one. If becomes greater than zero, we know that the line has moved upwards one pixel, and that we must increment our y coordinate and readjust the error to represent the distance from the top of the new pixel – which is done by subtracting one from The reader can examine the above illustration and the table to see the complete operation of the algorithm on this example:

(x, y)	$\in$	description
(0, 0)	–0.4	illuminate pixel (0, 0)
	0.2	increment $\in$ by 0.6
(1, 0)		increment x by 1
(1, 0)	0.2	illuminate pixel (1, 0)
		since $\in$ > 0
(1, 1)		increment y by 1
	–0.8	decrement $\in$ by 1
	–0.2	increment $\in$ by 0.6
(2, 1)		increment x by 1
(2, 1)	–0.2	illuminate pixel (2, 1)
	0.4	increment $\in$ by 0.6
(3, 1)		increment x by 1
(3, 1)	0.4	illuminate pixel (3, 1)
		since $\in$ > 0
(3, 2)		increment y by 1
	–0.6	decrement $\in$ by 1
	0.0	increment $\in$ by 0.6
(4, 2)		increment x by 1
(4, 2)	0, 0	illuminate pixel (4, 2)

Assuming that the DA is the x-axis, an algorithmic description of Bresenham's algorithm is as follows:

Bresenham's Algorithm

The points (x_1, y_1) and (x_2, y_2) are assumed not equal and integer valued. The ε is supposed to be real.

Let $\Delta x = x_2 - x_1$

Let $\Delta y = y_2 - y_1$

Let $m = \frac{\Delta y}{\Delta x}$

Let $j = y_1$

Let $\in = m - 1$

for $i = x_1$ to $x_2 - 1$

illuminate (i, j)

if $(\in \geq 0)$

j + = 1

$\in$ + = 1.0

Notes

end if

i + = 1

$\in$ + = m

next i

finish

All algorithms presented in these notes suppose that Δx and Δy are positive. If this is not the case, the algorithm is basically the equal with the exception of the following:

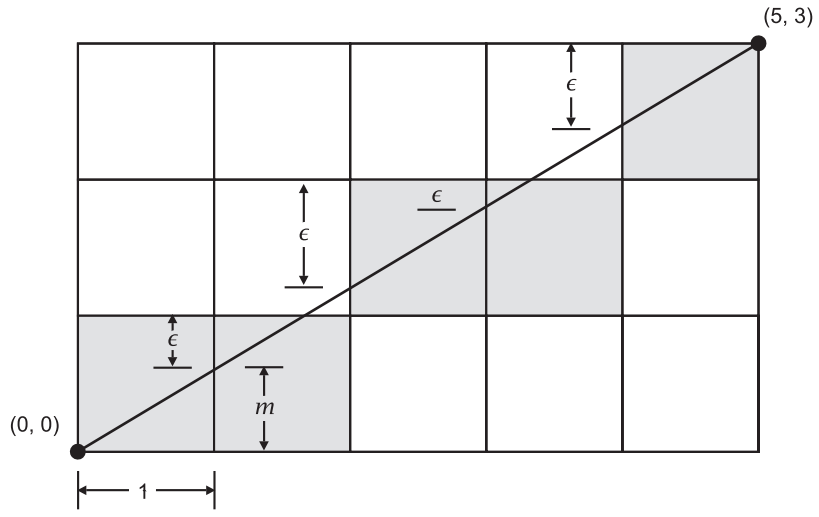
- ε is calculated using |Δy/Δx|.
- x and y are decremented (instead of incremented) by one if the sign of Δx or Δy is less than zero, respectively.

3.2.2 The Integer Bresenham's Algorithm

Bresenham's Algorithm, as given, in the previous section, requires the use of floating point arithmetic to determine the slope of the line and to evaluate the error term. We note that is initialized to ε = (Δy/Δx) − 1 and is incremented by Δy/Δx at each step. Since both Δy and Δx are integer quantities, we can convert to an all integer format by multiplying the operations through by Δx. That is, we will consider the integer quantity $\bar{\in}$, where $\bar{\in}$ is initialized to:

$$\bar{\in} = \Delta x\, \varepsilon = \Delta y - \Delta x$$

And we will increment $\bar{\in}$ by Δy at each step, and decrement it by Δx when becomes positive. Our example from the section above, which attempts to draw a line from (0, 0) to (5, 3) in screen space, can now be converted to an integer algorithm. Consider the figure and table, where Δx = 5, Δy = 3 and $\bar{\in}$ = Δx − Δy = −2.



Thus the integer version of Bresenham's algorithm is constructed as follows:

Bresenham's Algorithm Using Integer Arithmetic

The points (x_1, y_1) and (x_2, y_2) are assumed not equal and integer valued. $\bar{\in}$ is assumed to be integer valued.

Let $\Delta x = x_2 - x_1$

Let $\Delta y = y_2 - y_1$

Let $j = y_1$

Let $\bar{\in} = \Delta y - \Delta x$

Notes

(x, y)	$\bar{\in}$	description
(0, 0)	–2	illuminate pixel (0, 0)
	1	increment ε by Δy
(1, 0)		increment x by 1
(1, 0)	1	illuminate pixel (1, 0)
		since $\bar{\in} > 0$
(1, 1)		increment y by 1
	–4	decrement $\bar{\in}$ by 5
	–1	increment ε by Δy
(2, 1)		increment x by 1
(2, 1)	–1	illuminate pixel (2, 1)
	2	increment ε by Δy
(3, 1)		increment x by 1
(3, 1)	2	illuminate pixel (3, 1)
		since $\bar{\in} > 0$
(3, 2)		increment y by 1
	–3	decrement $\bar{\in}$ by 5
	0	increment ε by Δy
(4, 2)		increment x by 1
(4, 2)	0	illuminate pixel (4, 2)

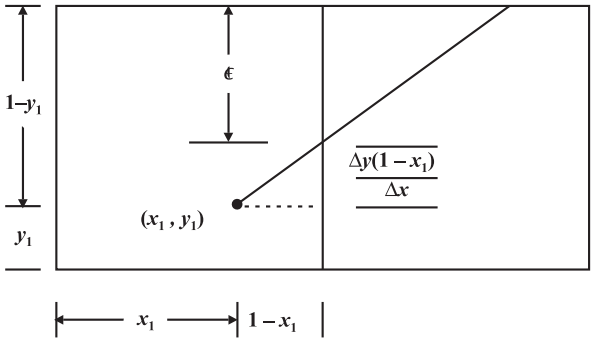
for i = x_1 to x_2 – 1
  illuminate (i, j)
  if ($\in \geq 0$)
    j += 1
    $\bar{\in}$ – = Δx
    end if
  i += 1
  $\bar{\in}$ + = Δy
  next i
finish

3.2.3 Bresenham's Algorithm for Lines with Arbitrary Endpoints

Bresenham's algorithm is limited by the detail that the lines to be drawn have endpoints with integer coordinates. Now we consider a version of Bresenham's algorithm for lines that have endpoints with real coordinates. The only problem to conquer is the initial setting of the error. Once this is done, the algorithm proceeds as before.

Consider a line with initial point (x_1, y_1) and terminal point (x_2, y_2) in device space, where we assume the points are not the same. To calculate the correct in this case, we refer to the following figure.

Notes

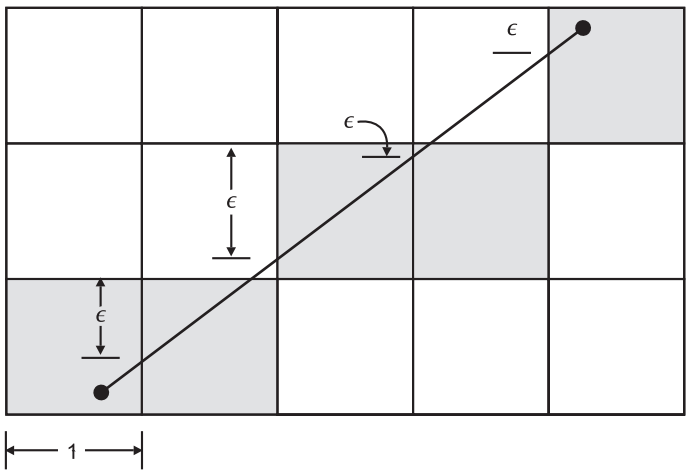


If we consider the lower-left-hand corner of the grid to be (0, 0), then it is simply seen that the early is:

$$\epsilon = -\left(1 - y_1 - \frac{\Delta y(1 - x_1)}{\Delta x}\right)$$

We now utilize this to modify Bresenham's algorithm accordingly.

Consider the following example, which attempts to draw a line from (0.75, 0.125) to (4.3, 2.8) in screen space,



Bresenham's algorithm calculates the new as:

$$\epsilon = -\left(1 - 0.125 - \frac{2.55(1 - .75)}{3.55}\right)$$

$$= -\left(0.75 - \frac{.6375}{3.55}\right)$$

$$= -(0.75 - 0.17958)$$

$$= -0.57042$$

The algorithm starts with the point (0, 0) = $(\lfloor 0.75 \rfloor, \lfloor 0.125 \rfloor)$ and then proceeds in closely the same way as Bresenham's algorithm for lines having endpoints with integer coordinates. The reader can examine the above illustration and the following table to see the complete operation of the algorithm on this example. In this case m = 2.55/3.55 = .71831, and the lower left corner of the grid is $(\lfloor z_1 \rfloor, (\lfloor z_2 \rfloor))$.

Notes

(x, y)	$\in$	description
(0, 0)	–0.57042	illuminate pixel (0, 0)
	0.14789	increment $\in$ by 0.71831
(1, 0)		increment x by 1
(1, 0)	0.14789	illuminate pixel (1, 0)
		since ε > 0
(1, 1)		increment y by 1
	–0.85211	decrement ε by 1
	–0.1338	increment ε by 0.71831
(2, 1)		increment x by 1
(2, 1)	–0.1338	illuminate pixel (2, 1)
	0.58451	increment ε by 0.71831
(3, 1)		increment x by 1
(3, 1)	0.58451	illuminate pixel (3, 1)
		since ε > 0
(3, 2)		increment y by 1
	–0.41549	decrement ε by 1
	0.30282	increment ε by 0.71831
(4, 2)		increment x by 1
(4, 2)	0.30282	illuminate pixel (4, 2)

Assuming that the DA is the x-axis, the algorithmic description of Bresenham's algorithm for lines with arbitrary endpoints is as follows:

Bresenham's Algorithm

The points (x_1, y_1) and (x_2, y_2) are assumed not equal

And have arbitrary real coordinates is assumed to be real.

Let $\Delta x = x_2 - x_1$

Let $\Delta y = y_2 - y_1$

Let $m = \dfrac{\Delta y}{\Delta x}$

Let $i_1 = \lfloor y_1 \rfloor$

Let $j = \lfloor y_1 \rfloor$

Let $i_2 = \lfloor x_2 \rfloor$

Let $\in = -\left(1 - (y_1 - j) - \dfrac{\Delta y(1 - (x_1 - i_1))}{\Delta x}\right)$

for $i = i_1$ to i_2

illuminate (i, j)

if $(\in \geq 0)$

j + = 1

$\in$ – = 1.0

Notes

end if

i += 1

$\in$ + = m

next i

finish

The Integer Bresenham's Algorithm for Lines with Arbitrary Endpoints

Bresenham's Algorithm was adapted to lines that have endpoints with arbitrary genuine coordinates. This algorithm again requires the use of floating point arithmetic to calculate the slope of the line and to evaluate the error term. We note that is initialized to:

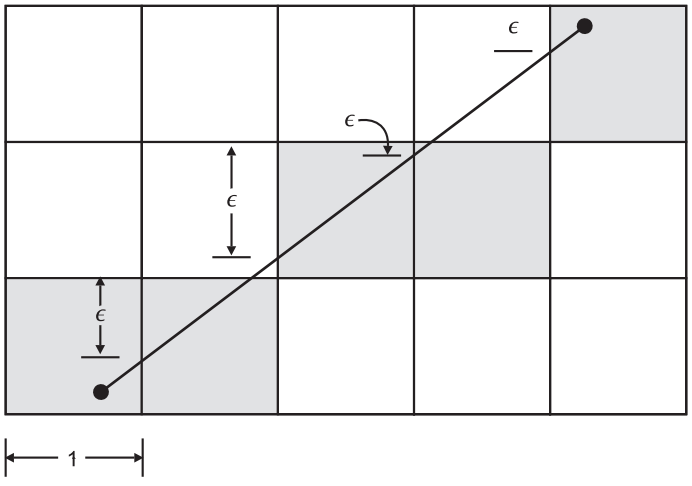
$$\in = -\left(1 - (y_1 - \lfloor y_1 \rfloor) - \frac{\Delta y(1 - (x_1 - \lfloor x_1 \rfloor)}{\Delta x}\right)$$

And is incremented by $\Delta y/\Delta x$ at each step. However, we cannot do the same implications with these algorithms we did with the integer algorithm above, since in this case both Δy and Δx are real ε not integer.

If we multiply through by Δx, we again obtain an approximation for $\bar{\in}$ that, at least, does not require division.

$$\bar{\in} = \Delta x\varepsilon = -\Delta x(1 - (y_1 - \lfloor y_1 \rfloor)) + \Delta y(1 - (x_1 - \lfloor x_1 \rfloor))$$

However, to bring this algorithm back to integer form we assume that our basic pixel element is no longer 1×1 in size, but now it is $m \times m$ in size, and utilize increments and decrements in the algorithm of $[m\Delta x]$ and $[m\Delta y]$, respectively. The error term is first set to $[m\bar{\varepsilon}]$ and the algorithm then proceeds as does the version with endpoints that have integer coordinates. If we address the example, which attempts to draw a line from (0.75, 0.125) to (4.3, 2.8) in screen space.



And if we let m = 1024, then we have that:

$$\lfloor m\Delta x \rfloor = \lfloor 1024(3.55) \rfloor$$

$$= \lfloor 3635.2 \rfloor$$

$$= 3635$$

$$\lfloor m\Delta y \rfloor = \lfloor 1024(2.55) \rfloor$$

$$= \lfloor 2611.2 \rfloor$$

$$= 2611$$

Notes and

$$m = -1024\ (\Delta x\ (1 - 0.125) - \Delta y\ (1 - 0.75))$$
$$= -1024\ (3.55(1 - 0.125) - 2.55(1 - 0.75)$$
$$= -1024\ (2.6625 - .6375)$$
$$= -2073.6$$

And so we initialize the integer quantity $\bar{\in} = -2073$ and the algorithm proceeds as follows:

Thus the integer version for Bresenham's algorithm with arbitrary endpoints is constructed as follows:

Integer Bresenham's Algorithm

The points (x_1, y_1) and (x_2, y_2) are assumed not equal and have arbitrary real coordinates $\bar{\in}$ is

(x, y)	$\bar{\in}$	description
(0, 0)	–2073	illuminate pixel (0, 0)
	538	increment ε by 2611
(1, 0)		increment x by 1
(1, 0)	538	illuminate pixel (1, 0)
		since $\bar{\in} > 0$
(1, 1)		increment y by 1
	–3097	decrement $\bar{\in}$ by 3635
	–486	increment ε by 2611
(2, 1)		increment x by 1
(2, 1)	–486	illuminate pixel (2, 1)
	2125	increment ε by 2611
(3, 1)		increment x by 1
(3, 1)	2125	illuminate pixel (3, 1)
		since $\bar{\in} > 0$
(3, 2)		increment y by 1
	–1510	decrement $\bar{\in}$ by 3635
	1101	increment ε by 2611
(4, 2)		increment x by 1
(4, 2)	1101	illuminate pixel (4, 2)

assumed to be integer valued.

Let $\Delta x = m\ (x_2 - x_1)$

Let $\Delta y = m\ (y_2 - y_1)$

Let $i_1 = \lfloor x_1 \rfloor$

Let $i_2 = \lfloor x_2 \rfloor$

Let $j = \lfloor y_1 \rfloor$

Let $\bar{\in} = \lfloor \Delta y\ (1 - (x_1 - i) - \Delta x(1 - y_1 - j) \rfloor$

for $i = i_1$ to i_2

illuminate (x, y)

if $(\bar{\in} \geq 0)$

$y +$ $= 1$

Notes

$\bar{\in}$ – = Δx

end if

x + = 1

$\bar{\in}$ + = Δy

next i

finish

3.2.4 Specifying the Driving Axis

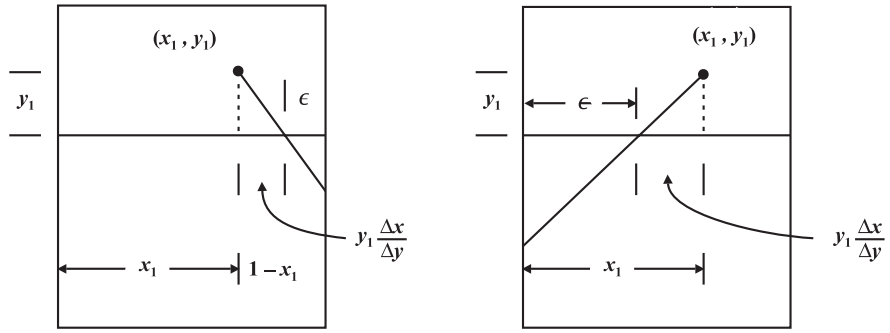
If x is the driving axis, Bresenham's algorithm creates only one illuminated cell per column in the matrix of pixels. This feature permits the algorithm to be helpful in the rasterization of polygons in image space. In this algorithm, we need only one illuminated pixel per row be produced which is likely with Bresenham's algorithm by fixing the driving axis as the y axis. This can be done by a simple change to the main loop of the algorithm. Normally, within the main loop of the algorithm, the coordinate corresponding to the DA is incremented by one unit and the coordinate corresponding to the other axis need only be incremented occasionally. When we fix the driving axis, the coordinate on the DA is still incremented by one unit; however the coordinate on the other axis may be incremented several times. This is actually a simple change to the algorithm, and can be implemented by replacing the statement:

if ($\bar{\in} \geq 0$)

with

while ($\bar{\in} \geq 0$)

The algorithm now appears as follows, we also note we have changed the algorithm to reflect that the y axis is the driving axis. We note that m =Δx/Δy, and that according to the following picture:



We must consider two possible initial values for, one if Δx > 0 (the left-hand illustration) and one if Δx < 0 (the right-hand illustration). Referring to the illustration, we have either

$$\varepsilon = -(1 - x_1 - y_1 \Delta x/\Delta y)$$

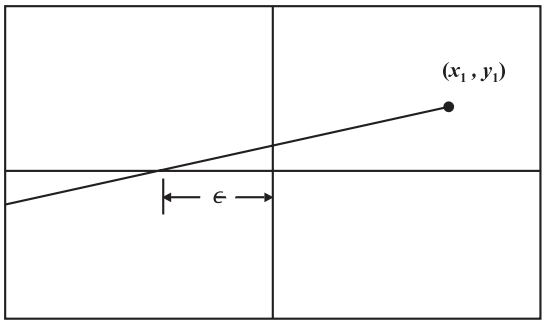
in the first case, or

$$\varepsilon = -(x_1 - y_1 \Delta x/\Delta y)$$

In general x_1 must be replaced by $x - 1\lfloor x_1 \rfloor$ and y_1 by $y - 1\lfloor y_1 \rfloor$ as the figure is drawn as if the lower-left-hand corner of the pixel is (0, 0). We also need to recognize that may be greater than zero immediately, as in the following figure. Therefore, we must move the "illuminate" step in our algorithm below the "while" statement. This will insure that we are at the correct boundary pixel for the trapezoid.

Notes

We are using the non-integer form of the algorithm, which is clearer in its presentation. The conversion to the integer algorithm is straightforward.



The algorithm now appears as follows:

Bresenham's Algorithm with y as the Driving Axis

The points (x_1, y_1) and (x_2, y_2) are assumed not equal is assumed to be real.

Let $\Delta x = x_2 - x_1$

Let $\Delta y = y_2 - y_1$

Let $m = \left|\frac{\Delta x}{\Delta y}\right|$

Let $i = \lfloor x_1 \rfloor$

Let $j_1 = \lfloor y_1 \rfloor$

Let $j_2 = \lfloor y_2 \rfloor$

if $\Delta x > 0$

$i_{inc} = 1$

$\epsilon = -\left(1 - (x_1 - i) - (y_1 - j_1)\frac{\Delta x}{\Delta y}\right)$

else

$i_{inc} = -1$

$\varepsilon = -\left((x_1 - i) - (y_1 - j_1)\frac{\Delta x}{\Delta y}\right)$

end if

for $j = j_1$ to j_2

while $(\epsilon \geq 0)$

$i \mathrel{+}= 1$

$\epsilon \mathrel{-}= 1.0$

end while

illuminate (i, j)

$j \mathrel{+}= 1$

$\epsilon \mathrel{+}= m$

next j

finish

Notes

This algorithm will continue to increase the x value (and decrement) as long as is greater than zero. It illuminates only one pixel per row on the line. We are assuming that Δy = !0 which is valid since we are only allowing for the non-horizontal edges of the trapezoids that we want to rasterizing. It should also be mentioned that the above algorithm is written for the case where m is positive. If m = 0, then $\epsilon = -(1 - (x_1 - i))$ and is never incremented. In this case, we have a vertical line, and the x value will never need to be incremented. If m < 0, then we must decrement i in the algorithm, and add |m| to ε.

3.2.5 C Code Line Drawing Algorithm

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
   int gd = DETECT, gm;
   int dx, dy, p, end;
   float x1, x2, y1, y2, x, y;
   initgraph(&gd, &gm, "c:\tc\bgi");
   printf("Enter Value of X1:");
   scanf("%f", &x1);
   printf("Enter Value of Y1:");
   scanf("%f", &y1);
   printf("Enter Value of X2:");
   scanf("%f", &x2);
   printf("Enter Value of Y2:");
   scanf("%f", &y2);
   dx = abs(x1 – x2);
   dy = abs(y1 – y2);
   p = 2 * dy – dx;
   if(x1 > x2)
   {
      x = x2;
      y = y2;
      end = x1;
   }
   else
   {
      x = x1;
      y = y1;
      end = x2;
   }
   putpixel(x, y, 10);
   while(x < end)
   {
      x = x + 1;
      if(p < 0)
```

Notes

```
        {
            p = p + 2 * dy;
        }
        else
        {
            y = y + 1;
            p = p + 2 * (dy – dx);
        }
            putpixel(x, y, 10);
    }
    getch();
    closegraph();
}
```

Did u know?

The line algorithm was developed by Jack E. Bresenham in 1962 at IBM.

Self Assessment Questions

1. …….. is one of the earliest algorithms developed in the field of computer graphics.

   (*a*) Bresenham line algorithm (*b*) DDA Line Algorithm

   (*c*) Both of these (*d*) None of these

2. The DA is not used as the axis of control for the algorithm and is the axis of maximum movement.

   (*a*) True (*b*) False

3. In Bresenham's algorithm using integer arithmetic:

   (*a*) The ε is assumed to be real

   (*b*) Arbitrary real coordinates is assumed to be real

   (*c*) $\bar{\varepsilon}$ is assumed to be integer valued

   (*d*) Arbitrary real coordinates $\bar{\varepsilon}$ is assumed to be integer valued

4. Bresenham's algorithm with y as the driving axis:

   (*a*) The points (x_1, y_1) and (x_2, y_2) are assumed not equal and have arbitrary real coordinates $\bar{\varepsilon}$ is assumed to be integer valued.

   (*b*) The points (x_1, y_1) and (x_2, y_2) are assumed not equal and integer valued. $\bar{\varepsilon}$ is assumed to be integer valued.

   (*c*) The points (x_1, y_1) and (x_2, y_2) are assumed not equal and have arbitrary real coordinates is assumed to be real.

   (*d*) The points (x_1, y_1) and (x_2, y_2) are assumed not equal is assumed to be real.

5. For integer Bresenham's algorithm is the points (x_1, y_1) and (x_2, y_2) are assumed not equal is assumed to be real.

   (*a*) True (*b*) False

3.3 DDA Line Algorithm

Notes

Digital Differential Analyzer (DDA) is a scan alteration line algorithm stand on calculating either dy or dx. We sample the line at unit gaps in one coordinate and decide corresponding integer values neighboring to the line path for the other coordinate. The DDA method is not very proficient by reason of the need for division and rounding. Bresenham's line algorithm is a more efficient method to draw lines because it uses only addition, subtraction and bit shifting.

The algorithm accepts as input the two endpoint pixel positions. Horizontal and vertical differences between the endpoint positions are allocated to parameters dx and dy. The difference with the greater magnitude concludes the increase of the parameter steps. Starting with the pixel position (x_a, y_a), we determine the offset needed at each step to generate the next pixel position along the line path. We loop through this process steps? Times. If the magnitude of dx is greater that the magnitude of d_y and x_a is less that x_b, the values of the increments in the x and y directions are i and m. It is a faster method of calculating pixel positions. However, the accumulation of round off error in successive additions can cause the calculated pixel positions to draft away from the true line path for long line segments.

This algorithm is also a scan conversion algorithm now as Bresenham's algorithm. This works as well by selecting which pixels to put on between two points, but instead of calculating the next point by looking at the errors between the possible choices, this algorithm always increments the value along one axis and calculates the value of the other co-ordinate. The decision to which coordinate to move at a constant step is by finding which of the two have a greater gradient, i.e., if dy< dx, then co-ordinate x will move in steps, and co-ordinate y will be calculated.

Current = (x, y)

calculate dy and dx

if (dy > dx) then

steps = dx

else

steps = dy

x_{inc} = dx /steps

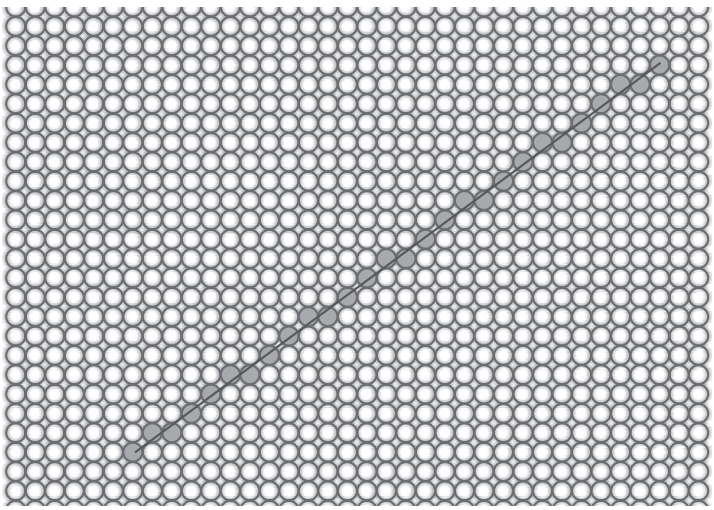
y_{inc} = dy /steps

repeat until current = (x_{end}, y_{end})

Plot (Current)

xCurrent = xCurrent + xinc

yCurrent = (xCurrent + l, yCurrent + y_{inc}



Notes

The line drawing algorithm using the Digital Differential Analyses (DDA) method. Select any two points on the pseudo screen with mouse clicks. The 'ideal line' is first drawn as thin blue straight line between the two selected points. Determining suitable intermediate pixels using the DDA algorithm enables the line to be drawn. Click 'clear' to refresh the screen, and to select a new pair of points.

Line Equation

The Cartesiand slop-intercept equation for a straight line is:

$$y = mx + b \quad (1)$$

with

m-> slope

b-> y intercept

The 2 end points of a line segment are specified at a position(x_1, y_1)

Determine the values for the slope m and y intercept b with the following calculation.

Here, slope m:

$$m = (y_2 - y_1) / (x_2 - x_1)$$

$$m = Dy / Dx \quad (2)$$

y intercept b

$$b = y_1 - mx_1 \quad (3)$$

- Algorithms for displaying straight line based on this equation
- y interval Dy from the equation

$$m = Dy / Dx$$

$$Dy = m\, Dx \quad (4)$$

Similarly x interval Dx from the equation

$$m = Dy / Dx$$

$$Dx = Dy / m \quad (5)$$

3.3.1 Line DDA Algorithm

- The digital differential analyzer (DDA) is a scan conversion line algorithm based on calculation either Dy or Dx.
- The line at unit intervals is one coordinate and determine corresponding integer values nearest line for the other coordinate.
- Consider first a line with positive slope.

Step 1

If the slope is less than or equal to 1, the unit x intervals Dx = 1 and compute each successive y values.

$$Dx = 1$$

$$m = Dy / Dx$$

$$m = (y_2 - y_1) / 1$$

$$m = (y_k + 1 - y_k) /1$$

$$y_k + 1 = y_k + m \quad (6)$$

Notes

- Subscript k takes integer values starting from 1, for the first point and increment by 1 until the final end point is reached.
- m->any real numbers between 0 and 1
- Calculate y values must be rounded to the nearest integer

Step 2

If the slope is greater than 1, the roles of x any y at the unit y intervals Dy = 1 and compute each successive y values.

$$Dy = 1$$

$$m = Dy / Dx$$

$$m = 1/ (x_2 - x_1)$$

$$m = 1 / (x_k + 1 - x_k)$$

$$x_k + 1 = x_k + (1/m) \quad (7)$$

- Equation 6 and Equation 7 that the lines are to be processed from left end point to the right end point.

Step 3

If the processing is reversed, the starting point at the right

$$Dx = - 1$$

$$m = Dy / Dx$$

$$m = (y_2 - y_1) / - 1$$

$$y_k + 1 = y_k - m \quad (8)$$

Intervals Dy = 1 and compute each successive y values.

Step 4

Here,

$$Dy = - 1$$

$$m = Dy / Dx$$

$$m = - 1 / (x_2 - x_1)$$

$$m = -1 / (x_k + 1 - x_k)$$

$$x_k + 1 = x_k + (1 / m) \quad (9)$$

Equation 6 and Equation 9 used to calculate pixel position along a line with –ve slope.

Advantage

Faster method for calculating pixel position then the equation of a pixel position.

$$Y = mx + b$$

Disadvantage

The accumulation of round of error is successive addition of the floating point increments is used to find the pixel position but it take lot of time to compute the pixel position.

Algorithm: A naïve line-drawing algorithm

$dx = x_2 - x_1$

$dy = y_2 - y_1$

for x from x_1 to x_2

{

Notes

```
        y = y1 + (dy) * (x – x1)/(dx)
        pixel(x, y)
    }
```

3.3.2 C Code DDA Line Algorithm

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
    int gd=DETECT,gm=DETECT,s,dx,dy,m,x1,y1,x2,y2;
    float xi,yi,x,y;
    clrscr();
    printf("Enter the sarting point x1 & y1n");
    scanf("%d%d",&x1,&y1);
    printf("Enter the end point x2 & y2n");
    scanf("%d%d",&x2,&y2);
    initgraph(&gd,&gm, "");
    cleardevice();
    dx=x2–x1;
    dy=y2–y1;
    if(abs(dx)>abs(dy))
        s=abs(dx);
    else
        s=abs(dy);
    xi=dx/(float)s;
    yi=dy/(float)s;
    x=x1;
    y=y1;
    putpixel(x1,y1,4);
    for(m=0; m < s; m++)
    {
        x+=xi;
        y+=yi;
        putpixel(x,y,4);
    }
        getch();
}
```

Self Assessment Questions

6. Which one is a line algorithm for …….. line conversion?

(*a*) Bresenham's (*b*) Beizer

(*c*) Both a and b (*d*) Midpoint

Notes

7. LCD stands for:

   (*a*) Liquid Crystal Display (*b*) Liquid Carbon Display

   (*c*) Liquefied Crystal Display (*d*) Liquid Crystal Design

8. There are two types of computer graphics:

   (*a*) Raster and vector (*b*) Raster and faster graphics

   (*c*) Random and vector graphics (*d*) Vector and scalar graphics

9. The DDA stands for:

   (*a*) Digital Differential algorithm (*b*) Differential Digital Analyses

   (*c*) Digital Differential Analyses (*d*) Raster and faster graphics

10. The equation $\bar{\in}$ is initialized to:

    (*a*) $\bar{\in} = \varepsilon = \Delta y - \Delta x$ (*b*) $\bar{\in} = \Delta x\ \varepsilon = \Delta y - \Delta x.\ \Delta y$

    (*c*) $\bar{\in} = \Delta x\ \varepsilon = \Delta y - \Delta x$ (*d*) $\bar{\in} = \Delta x\ \varepsilon = \Delta x - \Delta y$

11. Line of equation represents…….

    (*a*) Y=mx+c (*b*) X=my+c

    (*c*) $Y^2=mx^2+c$ (*d*) $X^2=my^2+c$

True or False:

12. Digital plotters, like raster displays are discretely addressable devices.

    (*a*) True (*b*) False

13. The Bresenham's Algorithm for drawing circle on the discrete plane.

    (*a*) True (*b*) False

14. Digital Differential Analyzer is a scan conversion line algorithm.

    (*a*) True (*b*) False

15. The differential equation for a line is: m = dx / dy.

    (*a*) True (*b*) False

3.4 Summary

- Bresenham's Algorithm provides the means for the fast and efficient way to represent continuous abstract lines onto discrete plane of computer display.
- Bresenham's Algorithm requires the use of floating point arithmetic to calculate the slope of the line and to evaluate the error term.
- Bresenham's line algorithm is a more efficient method to draw lines because it uses only addition, subtraction and bit shifting.
- The DDA algorithm is also a scan conversion algorithm, just as Bresenham's algorithm. This works as well by selecting which pixels to put on between two points.
- The DDA algorithm faster method for calculating pixel position then the equation of a pixel position.

Notes

3.5 Keywords

Magnitude: Magnitude is the "size" of a mathematical object, a property by which the object can be compared as larger as or smaller than other objects of the same kind.

Pen-plotter: The plotter is a computer printer for printing vector graphics. In the past, plotters were used in applications such as computer-aided design, though they have generally been replaced with wide-format conventional printers.

Pixels: A pixel (picture element) is a physical point in a raster image, or the smallest, addressable element in display; so it is the smallest, controllable element of a picture represented on the screen.

Pseudo: A pseudo-event is an event or activity that exists for the sole purpose of the media publicity and serves little to no other function in real life.

Quantization: Quantization is the procedure of constraining something from a relatively large or continuous set of values (such as the real numbers) to a relatively small discrete set.

Translation: A translation moves all points of an object a fixed distance in a specified direction. It can also be expressed in terms of two frames by expressing the coordinate system of object in terms of translated frames.

3.6 Review Questions

1. What is the concept of implementing line algorithm?
2. Explain the types of implementing line algorithm.
3. Discuses the Bresenham's Algorithm.
4. What is the DDA Line Algorithm?
5. Write a program in C code the DDA line algorithm.
6. Write a C program Bresenham's Algorithm.
7. Explain the integer Bresenham's algorithm.
8. Explain the all steps of line DDA algorithm.
9. Write advantages and disadvantages of DDA algorithm.
10. Explain the integer Bresenham's algorithm for lines with arbitrary endpoints.

Answers for Self Assessment Questions

1.	(*a*)	2.	(*b*)	3.	(*c*)	4.	(*d*)	5.	(*b*)
6.	(*a*)	7.	(*a*)	8.	(*a*)	9.	(*c*)	10.	(*c*)
11.	(*a*)	12.	(*a*)	13.	(*b*)	14.	(*a*)	15.	(*b*)

3.7 Further Readings


Books

Introduction to computer graphics, by Krishnamurthy

Comprehensive Computer Graphics, by V.K. Pachghare


Online link

http://www.cglabprograms.com/2008/10/dda-line-drawing-algorithm.html

Unit 4: Implementing Circle Algorithm

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand the concept of circle algorithm
- Define the midpoint circle drawing algorithm
- Discuss the Bresenham's circle drawing algorithm

Introduction

Critical to this definition is the detail that every simple closed curve declares a well-defined interior; that follows from the Jordan Curve Theorem. All simple closed curves can be classified as negatively oriented (clockwise), absolutely oriented (counterclockwise), or non-orient able. A circle oriented counterclockwise is an example of a positively oriented curve. The same circle oriented clockwise would be a negatively oriented curve.

The concept of orientation of a curve is just a particular case of the notion of orientation of a manifold (that is, besides orientation of a curve one may also speak of orientation of a surface, hyper surface, etc.). Here, the interior and the exterior of a curve both inherit the usual orientation of the plane. The positive orientation on the curve is then the orientation it inherits as the boundary of its interior; the negative orientation is inherited from the exterior. A line forming a closed loop, every point on which is a fixed distance from a center point.

A circle is a type of line. Imagine a straight line segment that is bent around until its ends join. Then arrange that loop until it is exactly circular - that is, all points along that line are the same distance from a centre point. There is a difference between a circle and a disk. A circle is a line, and so, for example, has no area - just as a line has no area. A disk however is a round portion of a plane which has a circular outline. If you draw a circle on paper and cut it out, the round piece is a disk.

Notes

Properties of a Circle

Centre	A point inside the circle. All points on the circle are equidistant (same distance) from the centre point.
Radius	The radius is the distance from the centre to any point on the circle. It is half the diameter.
Diameter	The distance across the circle. The length of any chord passing through the centre. It is twice the radius.
Circumference	The circumference is the distance around the circle.
Area	Strictly speaking a circle is a line, and so has no area. What is usually meant is the area of the region enclosed by the circle.
Chord	A line segment linking any two points on a circle.
Tangent	A line passing a circle and touching it at just one point.
Secant	A line that intersects a circle at two points.

4.1 Concept of Circle Algorithm

A circle is a geometrical structure, and is not of much use in algebra, because the equation of a circle is not a function. But you may require working with circle equations. In "primitive" terms, a circle is the shape formed in the sand by driving a stick (the "center") into the sand, putting a loop of string around the center, pulling that loop taut with another stick, and dragging that second stick through the sand at the further extent of the loop of string. The resulting figure drawn in the sand is a circle.

It is known that a circle can be divided into 360 degrees. When looking at the circle it is obvious that the circle is symmetric with regard to the co-ordinate axis x and y and the axes of quadrants as well (function x = y and x = – y). It holds for the circle with a centre in (0, 0), that if a point (x, y) lies on the circle, then the points (y, x), (y, –x), (x, –y), (–x, –y), (–y, –x), (–y, x) and (–x, y) lie also on the circle. For defining all the point on the circle, it is sufficient to compute 1/8th of the arc of the circle. In algebraic terms, a circle is the set of points (x, y) at some fixed distance r from some fixed point (h, k). The value of r is called the "radius" of the circle, and the point (h, k) is called the "centre" of the circle.

The "general" equation of a circle is:

$$x^2 + y^2 + Dx + Ey + F = 0$$

The "centre-radius" form of the equation is:

$$(x - h)^2 + (y - k)^2 = r^2$$

where the h and the k come from the centre point (h, k) and the r^2 comes from the radius value r. If the centre is at the origin, so (h, k) = (0, 0), then the equation simplifies to $x^2 + y^2 = r^2$.

You can convert the "centre-radius" form of the circle equation into the "general" form by multiplying things out and simplifying; you can convert in the other direction by completing the square.

The centre-radius form of the circle equation comes directly from the Distance Formula and the definition of a circle. If the centre of a circle is the point (h, k) and the radius is length *r*, then every point (*x*, *y*) on the circle is distance *r* from the point (h, k). Plugging this information into the Distance Formula (using *r* for the distance between the points and the centre), we get:

Notes

$$r = \sqrt{(x-h)^2 + (y-k)^2}$$

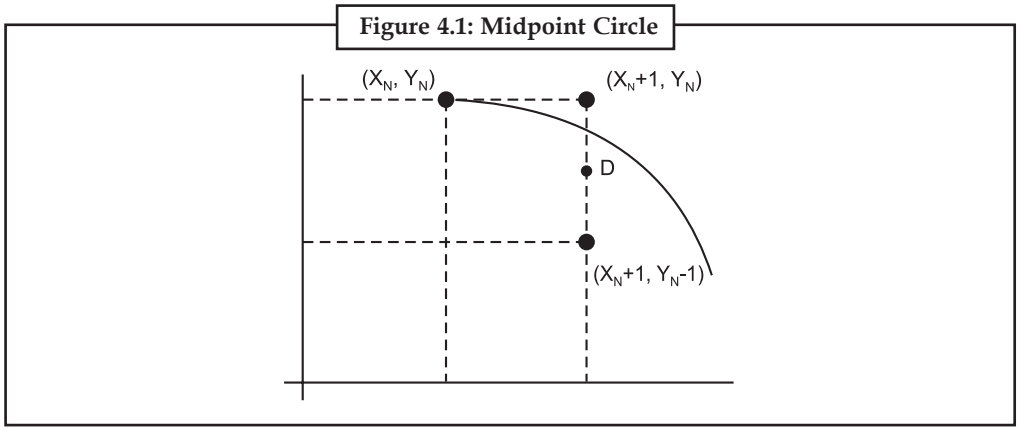
$$(r)^2 = \left(\sqrt{(x-h)^2 + (y-k)^2}\right)^2$$

$$r^2 = (x-h)^2 + (y-k)^2$$

4.2 Midpoint Circle Drawing Algorithm

The midpoint circle drawing algorithm works on the same midpoint idea as the Bresenham's line algorithm. In the midpoint circle drawing algorithm, you decide the next pixel to be plotted based on the position of the midpoint between the current and next successive pixel. The following shows the current pixel (x_n, y_n) and the possible next pixels, $(x_n + 1, y_n)$ and $(x_n + 1, y_n - 1)$ to be determined using the Midpoint circle drawing algorithm:

Figure 4.1: Midpoint Circle



You use the Midpoint circle drawing algorithm to determine the next pixel to be plotted depending on the position of the midpoint D. If D is inside the circle, choose $(x_n + 1, y_n)$ as the next pixel. If D is outside the circle, choose $(x_n + 1, y_n - 1)$ as the next pixel. If D lies on the circumference of the circle, choose either pixel.

The equation of the circle is given by g(x, y) as:

$$g(x, y) = x^2 + y^2 - R^2 = 0$$

where R is the radius of the circle.

The g(x, y) is positive for a point outside the circle, negative for a point inside the circle, and zero for a point on the circumference of the circle. These relations help confirm the position of the midpoint D to determine the next pixel to be plotted on the circle.

4.2.1 Implement Midpoint Algorithm

The following steps of implementing midpoint algorithm:

1. Enter the centre of the circle, (x_c, y_c), and its radius R.
2. Plot the pixel (x_c, R) as the initial pixel.
3. To determine the next pixel, ensure the sign of g(D). You can determine this from the value of $g(x_n + 1, y_n - 1/2)$ denoted by p_n:

$$P_n = g(x_n + 1, y_n - 1/2) = (x_n + 1)^2 + (y_n - \tfrac{1}{2})^2 - R^2.$$

Notes

4. Choose the next pixel depending on the value of p_n. If $p_n < 0$, choose pixel $(x_n + 1, y_n)$, if $p_n > 0$, choose pixel $(x_n + 1, y_n - 1)$, and if $p_n = 0$, choose either pixel.
5. If you chose pixel $(x_n + 1, y_n)$, determine the next pixel from the sign of $(x_n + 2, y_n - ½)$ denoted by p_n+1:

$$P_n+1 = g(x_n+2, y_n-1/2)$$
$$= (x_n+2)^2 + (y_n-1/2)^2 - R^2$$

=> The increment in p is $\Delta p_1 = p_n+1 - p_n = 2x + 3$.

If you choose pixel (x_n+1, y_n-1), determine the next pixel from the sign of $(x_n+2, -3/2)$ denoted by p_n+1:

$$P_n+1 = g(x_n+2, y_n-3/2)$$
$$= (x_n+2)^2 + (y_n-3/2)^2 - R^2$$

=> The increment in p is $\Delta p_2 = p_n+1 - p_n = 2x_n - 2y_n + 5$.

6. Calculate the initial condition, p_0. The first point on the circle is (0, R) and the first midpoint D is at $(1, R - ½)$. Calculate p_0 as:

$$=> p_0 = g(1, R - ½)$$
$$= 12 + (R - ½)^2 - R^2$$
$$= 5/4 - R.$$

If all the attributes of the circle, such as the centre coordinates and the radius, are given as integers, you can round p_0 to $p_0 = 1 - R$.

7. To draw the circle, start with the pixel (0, R) and initialize p to 0.

After plotting the pixel (0, R), increment the value of p by p_0. For subsequent pixels, determine the next pixel depending upon the value of p. If $p < 0$, choose (x_n+1, y_n) as the next pixel and increment p by Δp_1. If $p > 0$, choose pixel (x_n+1, y_n+1) and increment p by Δp_1. If $p = 0$, choose either. Plot the pixels on the circumference of the circle up to $x = R/\sqrt{2}$ or $y < x$. For each pixel calculated, use the symmetry () function to obtain all the symmetric pixels.

4.2.2 Implement Midpoint Circle Drawing Algorithm in C

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void draw_circle(int, int, int);
void symmetry(int, int, int, int);
main()
{
    int xc, yc, R;
    int driver, mode;
    clrscr();
    printf("Enter the center of the circle:\n");
    printf("Xc =");
    scanf("%d", &xc);
    printf("Yc =");
    scanf("%d", &yc);
```

Notes

```
    printf("Enter the radius of the circle :");
    scanf("%d", &R);
    clrscr();
    driver = DETECT;
    initgraph(&driver, &mode, "\\tc\\bgi");  //the path may be different in your case.
    draw_circle(xc, yc, R);
    getch();
    closegraph();
}
void draw_circle(int xc, int yc, int rad)
{
    int x = 0;
    int y = rad;
    int p = 1–rad;
    symmetry(x, y, xc, yc);
    for(x= 0; y>x; x++)
    {
    if(p<0)
    p += 2*x + 3;
    else
    {
    p += 2*(x–y) + 5;
    y– –;
    }
    symmetry(x, y, xc, yc);
    }
}
void symmetry(int x, int y, int xc, int yc)
{
    putpixel(xc+x, yc – y, EGA_WHITE);//For pixel (x, y)
    putpixel(xc+y, yc – x, EGA_WHITE);//For pixel (y, x)
    putpixel(xc+y, yc+x, EGA_WHITE);//For pixel (y, – x)
    putpixel(xc+x, yc+y, EGA_WHITE);//For pixel (x, – y)
    putpixel(xc – x, yc+y, EGA_WHITE);//For pixel (–x,–y)
    putpixel(xc–y, yc+x, EGA_WHITE);//For pixel (–y,–x)
    putpixel(xc–y, yc–x, EGA_WHITE);//For pixel (–y, x)
    putpixel(xc–x, yc–y, EGA_WHITE);//For pixel (–x, y)
}
```

Output

Example Result Output Computer Graphics

Enter the centre point:

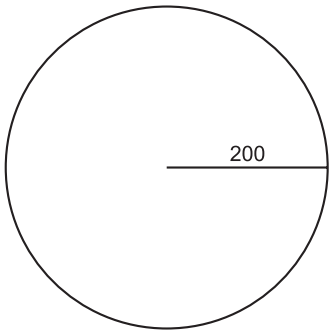
400

200

Notes

Enter the radius:

100



4.2.3 Advantage and Disadvantage of Midpoints Circle Drawing Algorithm

Following are the advantages and disadvantages of midpoint circle drawing algorithm:

Disadvantage

- Time consumption is high
- The distance between the pixels is not equal so we would not get smooth circle

Advantage

The midpoint method for deriving efficient scan-conversion algorithms to draw geometric curves on raster displays in described. The method is general and is used to transform the nonparametric equation f(x, y) = 0, which describes the curve, into an algorithms that draws the curve. Floating point arithmetic and time-consuming


Did u know?

The midpoint circle algorithm sometimes known as Bresenham's circle algorithm but it was not actually invented by Jack E. Bresenham. This algorithm is related to work by Pitteway and Van Aken.

Self Assessment Questions

1. The general form of the circle equation comes directly from the distance formula and the definition of a circle.

   (*a*) True (*b*) False

2. In this $(x - h)^2 + (y - k)^2 = r^2$ the value of r is called the ……..

   (*a*) radius (*b*) center point

   (*c*) circle (*d*) fixed point

3. The equation of the circle is given by g(x, y) as:

   (*a*) $x^2 + y^2 = R$ (*b*) $x^2 + y^2 + R^2 = 0$

   (*c*) $x^2 + y^2 - R^2 = 0$ (*d*) None of these

4. Which steps of implementing midpoint algorithm?

   (*a*) Enter the center of the circle, (x, y), and its radius R

   (*b*) Plot the pixel (xc, R) as the initial pixel

   (*c*) Choose the next pixel depending on the value of pn. If pn < 1

   (*d*) To draw the circle, start with the pixel (0, R) and initialize p to 1

5. If D is outside the circle, choose (xn+1, yn–1) as the next pixel.

Notes

(*a*) True (*b*) False

4.3 Bresenham's Circle Drawing Algorithm

The following steps of Bresenham's Circle Drawing Algorithm:

1. Start
2. Initialize the graphics mode
3. Read the radius of the circle
4. Initialize the graphics mode
5. Initialize starting points x=0,y=r
6. Initialize decision variable d=3 – 2 * r
7. Plot (x,y)

   a. plot eight points (+x,+y),(+x,–y),(–x,+y),(–x,–y),(+y,+x) ,(+y,–x),

   (–y,+x),(–y,–x)
8. If d< =0, d = d + 4 * x + 6 else {d = d + 4 (x–y) + 10, y = y–1}
9. x=x +1
10. Insert delay to observe the drawing process
11. Repeat 6- 10 until x<y

4.3.1 Bresenham's Circle Drawing Algorithm Using C Programming

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
    int gd=DETECT, gm;
    int x,y,r;
    void cir(int, int, int);
    printf("Enter the Mid points and Radious:");
    scanf("%d%d%d", &x, &y, &r);
    initgraph(&gd, &gm, "");
    cir(x, y, r);
    getch();
    closegraph();
}
void cir(int x1, int y1, int r)
{
    int x=0,y=r,p=1–r;
    void cliplot(int, int, int, int);
    cliplot(x1, y1, x, y);
    while(x<y)
```

Notes

```
    {p+=2*x+1;
    else
    {
    Y– –;
    p+=2*(x–y)+1;
    }
    cliplot(x1, y1, x, y);
    }
    }
void cliplot(int xctr, int yctr, int x, int y)
{
    putpixel(xctr +x, yctr +y, 1);
    putpixel(xctr –x, yctr +y, 1);
    putpixel(xctr +x, yctr –y, 1);
    putpixel(xctr –x, yctr –y, 1);
    putpixel(xctr +y, yctr +x, 1);
    putpixel(xctr –y, yctr +x, 1);
    putpixel(xctr +y, yctr –x, 1);
    putpixel(xctr –y, yctr –x, 1);
    getch();
}
```

Example Result Output Computer Graphics

Enter the centre point:

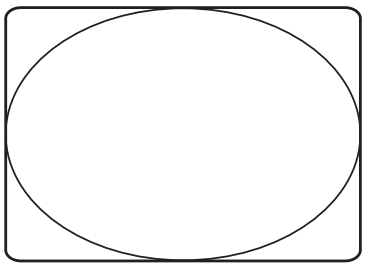
400

200

Enter the radius:

100

Output



Self Assessment Questions

6. Crucial to this definition is the fact that every simple closed curve admits a well-defined interior Closed curves can be classified as negatively oriented …….

   (*a*) counter clockwise (*b*) clockwise

   (*c*) Both (*a*) and (*b*) (*d*) None of these

Notes

7. The "general" equation of a circle is:

   (*a*) $x^2 + y^2 + Dx + Ey + F = 0$ (*b*) $x^3 + y^3 + Dx + Ey + F = 0$

   (*c*) $x + y + Dx + Ey + F = 0$ (*d*) $x + y^2 + Dx + F = 0$

8. The "centre-radius" form of the equation is:

   (*a*) $x^2+y^2=r^2$ (*b*) $(x - h)^2 + (y - k)^2 = r^2$

   (*c*) Both (*a*) and (*b*) (*d*) None of these

9. The midpoint circle drawing algorithm works on the same midpoint concept as the …….. line algorithm.

   (*a*) DDA (*b*) Bresenham's

   (*c*) Both (*a*) and (*b*) (*d*) None of these

10. A line that intersects a circle at two points, is called:

    (*a*) Circumference (*b*) Radius

    (*c*) Secant (*d*) None of these

11. A line segment linking any two points on a circle is called:

    (*a*) Circumference (*b*) Chord

    (*c*) Secant (*d*) None of these

12. A line passing a circle and touching it at just one point is called:

    (*a*) Tangent (*b*) Chord

    (*c*) Secant (*d*) None of these

True of False

13. The positive orientation on the curve is then the orientation it inherits as the boundary of its interior; the negative orientation is inherited from the exterior

    (*a*) True (*b*) False

14. The circumference is the distance around the circle.

    (*a*) True (*b*) False

15. The circumference is the distance around the circle.

    (*a*) True (*b*) False

4.4 Summary

- All simple closed curves can be classified as negatively oriented (clockwise), positively oriented (counterclockwise).
- The positive orientation on the curve is then the orientation it inherits as the boundary of its interior; the negative orientation is inherited from the exterior.
- A line forming a closed loop, every point on which is a fixed distance from a center point.
- The circumference is the distance around the circle.
- The midpoint circle drawing algorithm works on the same midpoint concept as the Bresenham's line algorithm.

Notes

4.5 Keywords

Circumference: The circumference is the distance around the circle.

Counter Clockwise: Circular motion can occur in two possible directions. A clockwise (typically abbreviated as CW) motion is one that proceeds in the same direction as clock's hands: from the top to the right, then down and then to the left, and back to the top. The opposite sense of rotation or revolution is counterclockwise (CCW).

Curve: A curve (also called a curved line in older texts) is, generally speaking, an object similar to a line but which is not required to be straight. This entails that a line is a special case of curve, namely a curve with null curvature.

Dragging Death: A dragging death is a death caused by someone being dragged behind or underneath a moving vehicle, whether accidental or as a deliberate act of murder. If it is homicide, then it is also known as a dragging murder.

Pixel: A pixel is a physical point in a raster image, or the smallest, addressable element in a display device; so it is the smallest, controllable element of a picture represented on the screen.

4.6 Review Questions

1. What is the circle algorithm?
2. Explain the properties of a circle.
3. What is the midpoint circle drawing algorithm?
4. Write an algorithm implement midpoint circle drawing.
5. Write midpoint circle drawing algorithm in C.
6. What is the advantage and disadvantage of midpoints circle drawing algorithm?
7. What is the Bresenham's circle drawing algorithm?
8. Write a Bresenham's circle drawing algorithm using C programming.
9. Differentiate between midpoint circle drawing and Bresenham's circle drawing.
10. Explain the advantages of Bresenham's circle drawing.

Answers for Self Assessment Questions

1. (*b*)	2. (*a*)	3. (*c*)	4. (*b*)	5. (*a*)
6. (*b*)	7. (*a*)	8. (*c*)	9. (*b*)	10. (*c*)
11. (*b*)	12. (*a*)	13. (*a*)	14. (*b*)	15. (*a*)

4.7 Further Readings

Books

Computer Graphics Techniques, by David F. Rogers

Computer Graphics, *C Version*, by Hearn

Lab Exercise

http://www.google.co.in/#q=dda+circulars+computer+graphics&html

Unit 5: Implementing Ellipse Algorithm

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Define Bresenham's ellipse drawing algorithm
- Explain midpoint ellipse algorithm
- Define ellipse area, sector area and segment area
- Extending the core segment algorithm

Introduction

Ellipses are helpful in many applied circumstances, and in widely unlike fields. In our research, we have encountered a common need for proficiently calculating the overlap area between two ellipses. In one case, the design for a solar calibrator onboard an orbiting satellite required an efficient algorithm for finding ellipse overlap areas. In a more down-to-earth setting, calculating ellipse overlap areas is useful for modeling pedestrian dynamics. The approach described in surrounds each pedestrian by an elliptical footprint area that the model uses to anticipate obstacles and other pedestrians in or near the intended path. A force-based model produces a repulsive force between overlapping exclusion areas, causing the pedestrians to slow down or change course when the exclusion force becomes large. Implementing the force-based model with elliptical exclusion areas in a simulation requires calculating the overlap area between many different ellipses in the most general orientations. The ellipse area overlap algorithm must also be efficient, so as not to bog down the simulation.

Notes

We provide an algorithm that determines the overlap area between two general ellipses. We first find the area of an ellipse segment, which is the area between a secant line and the ellipse boundary. The segment algorithm then forms the basis of an application for calculating the overlap area between two general ellipses, once the points of intersection between the two ellipses have been found. The algorithm is implemented in c-code, and algebraic details are provided in the text for all numerical operations in the code.

We present a general algorithm for finding the overlap area between two ellipses. The algorithm is based on finding a segment area (the area between an ellipse and a secant line) given two points on the ellipse. The Gauss-Green formula is used to determine the ellipse sector area between the two points, and a triangular area is added or subtracted to give the segment area. For two ellipses, overlap area is calculated by adding the areas of appropriate segments and polygons. Intersection points for two general ellipses are found using Ferrari's quadratic formula to solve the polynomial that results from combining the two ellipse equations. All cases for the number of intersection points (0, 1, 2, 3, 4) are handled. The algorithm is implemented in c-code, and has been tested with a range of input ellipses. The code is efficient enough for use in simulations that require many overlap area calculations.

5.1 Bresenham's Ellipse Drawing Algorithm

There is a well-known algorithm for plotting straight lines on a display machine or a plotter where the grid in excess of which the line is drawn consists of distinct points or pixels. In working with a lattice of points it is useful to avoid floating point arithmetic. Integer arithmetic has the advantages of speed and precision; working with floating point values requires more time and memory and such values would need to be rounded to integers anyway.

```
#include "stdio.h"
#include "conio.h"
#include "math.h"
#include "graphics.h"
main()
{
int gd=DETECT,gm;
int xcenter,ycenter,rx,ry;
int p,x,y,px,py,rx1,ry1,rx2,ry2;
initgraph(&gd,&gm, "c:\\tc\\bgi");
printf("Enter The Radius Value:\n");
scanf("%d%d",&rx,&ry);
printf("Enter The xcenter and ycenter Values:\n");
scanf("%d%d",&xcenter,&ycenter);
ry1=ry*ry;
rx1=rx*rx;
ry2=2*ry1;
rx2=2*rx1;
/* REGION 1 */
x=0;
y=ry;
plotpoints(xcenter, ycenter, x, y);
```

Notes

```
p=(ry1–rx1*ry+(0.25*rx1));
px=0;
py=rx2*y;
while(px<py)
{
x=x+1;
px=px+ry2;
if(p>=0)
y=y–1;
py=py–rx2;
if(p<0)
p=p+ry1+px;
else
p=p+ry1+px–py;
plotpoints(xcenter,ycenter,x,y);
/* REGION 2*/
p=(ry1*(x+0.5)*(x+0.5)+rx1*(y–1)*(y–1)–rx1*ry1);
while(y>0)
{
y=y–1;
 py=py–rx2;
if(p<=0)
{
x=x+1;
px=px+ry2;
}
if(p>0)
p=p+rx1–py;
else
p=p+rx1–py+px;
plotpoints(xcenter,ycenter,x,y);
}
}
getch();
return(0);
}
int plotpoints(int xcenter,int ycenter,int x,int y)
{
putpixel(xcenter+x,ycenter+y,6);
putpixel(xcenter–x,ycenter+y,6);
putpixel(xcenter+x,ycenter–y,6);
putpixel(xcenter–x,ycenter–y,6);
}
```

Notes

Output

Enter the Radius Value: (Rx, Ry) =10, 30

Enter the xcenter and ycenter Values: =300,150

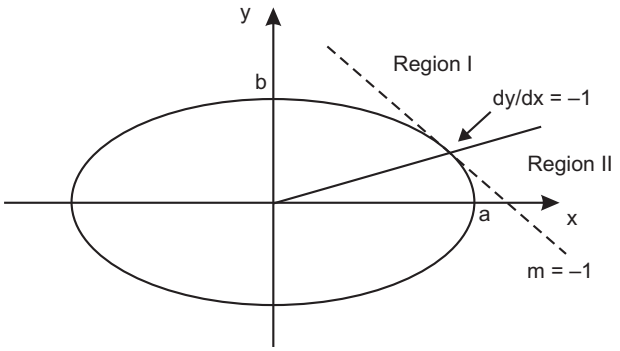
Did u know?

In 1970 Danny Cohen presented at the "Computer Graphics 1970" conference in England a linear algorithm for drawing ellipses and circles.

5.2 Midpoint Ellipse Algorithm

Midpoint ellipse algorithm is a technique for drawing ellipses in computer graphics. This method is customized from Bresenham's algorithm. The advantage of this modified method is that single calculation operations are required in the program loops. This leads to simple and fast execution in all processors.

Let us consider one quarter of an ellipse. The curve is divided into two regions. In region I, the slope on the curve is greater than –1 while in region II less than –1.



Consider the general equation of an ellipse,

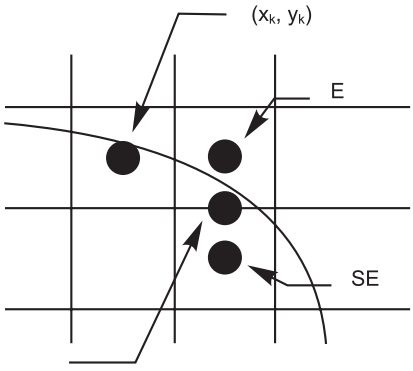
$$b^2x^2 + a^2y^2 - a^2b^2 = 0$$

In region I ($dy/dx > -1$),

x is always incremented in each step, i.e. $x_k + 1 = x_k + 1$.

$y_{k+1} = y_k$ if E is selected, or $y_{k+1} = y_k - 1$ if SE is selected.

In order to make decision between S and SE, a prediction $(x_k + 1, y_k - ½)$ is set at the middle between the two candidate pixels. A prediction function Pk can be defined as follows:



Region I

$$P_k = f(x_k + 1, y_k - ½)$$

$$= b^2(x_k + 1)^2 + a^2(y_k - ½)^2 - a^2b^2$$

Notes

$$= b^2(x_k^2 + 2x_k + 1) + a^2(y_k^2 - y_k + ¼) - a^2b^2$$

If $P_k < 0$, select E:

$$P_{k+1}^E = f(x_k + 2, y_k - ½)$$
$$= b^2(x_k + 2)^2 + a^2(y_k - ½)^2 - a^2b^2$$
$$= b^2(x_k^2 + 4x_k + 4) + a^2(y_k^2 - y_k + ¼) - a^2b^2$$

Change of P_k^E is : $\Delta P_k^E = P_{k+1}^E - P_k = b^2(2x_k + 3)$

If $P_k > 0$, select SE:

$$P_{k+1}^{SE} = f(x_k + 2, y_k - 3/2)$$
$$= b^2(x_k + 2)^2 + a^2(y_k - 3/2)^2 - a^2b^2$$
$$= b^2(x_k^2 + 4x_k + 4)\, a^2(y_k^2 - 3y_k + 9/4) - a^2b^2$$

Change of P_k^{SE} = is $\Delta P_k^{SE} = P_{k+1}^{SE} - P_k = b^2(2x_k + 3) - 2a^2(y_k - 1)$

Calculate the changes of ΔPk:

If E is selected,

$$\Delta P_{k+1}^E = b^2(2x_k + 5)$$
$$\Delta^2 P_k^E = \Delta P_{k+1}^E - \Delta P_k^E = 2b^2$$
$$\Delta P_{k+1}^{SE} = b^2(2x_k + 5) - 2a^2(y_k - 1)$$
$$\Delta^2 P_k^{SE} = \Delta P_{k+1}^{SE} - \Delta_k^{SE} = 2b^2$$

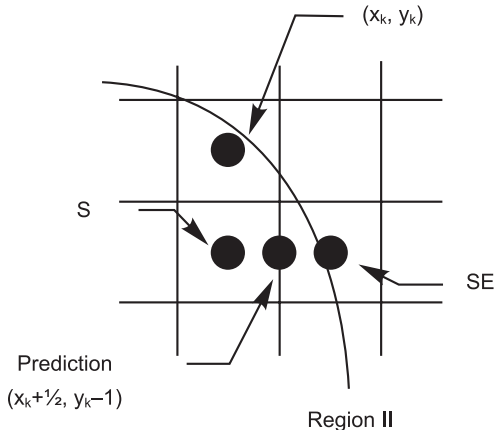
If SE is selected,

$$\Delta P_{k+1}^E = b^2(2x_k + 5)$$
$$\Delta^2 P_k^E = \Delta P_{k+1}^E - \Delta P_k^E = 2b^2$$
$$\Delta P_{k+1}^{SE} = b^2(2x_k + 5) - 2a^2(y_k - 2)$$
$$\Delta^2 P_k^{SE} = \Delta P_{k+1}^{SE} - \Delta P_k^{SE} = 2(a^2 + b^2)$$

Initial values:

$$x_0 = 0,\ y_0 = b,\ P_0 = b^2 + ¼a^2\,(1 - 4b)$$
$$\Delta P_0^E = 3b^2,\ \Delta P_0^{SE} = 3b^2 - 2a^2\,(b - 1)$$

In region II (dy/dx < –1), all calculations are similar to that in region we except that y is decremented in each step.



Notes

y is always decremented in each step, i.e. $y_{k+1} = y_k - 1$.

$x_{k+1} = x_k$ if S is selected, or $x_{k+1} = x_k + 1$ if SE is selected.

$$\begin{aligned} P_k &= f(x_k + ½, y_k - 1) \\ &= b^2(x_k + ½)^2 + a^2(y_k - 1)^2 - a^2b^2 \\ &= b^2(x_k^2 + x_k + ¼) + a^2(y_k^2 - 2y_k + 1) - a^2b^2 \end{aligned}$$

If $P_k > 0$, select S:

$$\begin{aligned} P_{k+1}^S &= f(x_k + ½, y_k - 2) \\ &= b^2(x_k + ½)^2 + a^2(y_k - 2)^2 - a^2b^2 \\ &= b^2(x_x^2 + x_k + ¼) + a^2(y_k^2 - 4y_k + 4) - a^2b^2 \end{aligned}$$

Change of P_k^S is: $\Delta P_k^S = P_{k+1}^S - P_k = a^2(3 - 2y_k)$

If $P_k < 0$, select SE:

$$\begin{aligned} P_{k+1}^{SE} &= f(x_k + 3/2, y_k - 2) \\ &= b^2(x_k + 3/2)^2 + a^2(y_k - 2)^2 - a^2b^2 \\ &= b^2(x_k^2 + 3x_k + 9/4) + a^2(y_k^2 - 4y_k + 4) - a^2b^2 \end{aligned}$$

Change of P_k^{SE} is $\Delta P_k^{SE} = P_{k+1}^{SE} - P_k = 2b^2(x_k + 1) + a^2(3 - 2y_k)$

Calculate the changes of ΔP_k :

If S is selected,

$$\begin{aligned} \Delta p_{k+1}^S &= a^2(5 - 2y_k) \\ \Delta^2 P_k^S &= \Delta P_{k+1}^S - \Delta P_k^S = 2a^2 \\ \Delta P_{k+1}^{SE} &= 2b^2(x_k + 1) + a^2(5 - 2y_k) \\ \Delta^2 P_k^{SE} &= \Delta P_{k+1}^{SE} - \Delta P_k^{SE} = 2a^2 \end{aligned}$$

If SE is selected,

$$\begin{aligned} \Delta P_{k+1}^S &= a^2(5 - 2y_k) \\ \Delta^2 P_k^S &= \Delta P_{k+1}^S - \Delta P_k^S = 2a^2 \\ \Delta P_{k+1}^{SE} &= 2b^2(2x_k + 2) - a^2(5 - 2y_k) \\ \Delta^2 P_k^{SE} &= \Delta P_{k+1}^{SE} - \Delta P_k^{SE} = 2(a^2 + b^2) \end{aligned}$$

Determine the boundary between region I and II:

$$\text{Set } f(x, y) = 0, \quad \frac{dy}{dx} = \frac{-bx}{a^2\sqrt{1 - x^2/a^2}}$$

When $dy/dx = -1$, $x = \dfrac{a^2}{\sqrt{a^2 + b^2}}$ and $y = \dfrac{b^2}{\sqrt{a^2 + b^2}}$.

At region I, $dy/dx > -1$, $x < \dfrac{a^2}{\sqrt{a^2 + b^2}}$ and $y > \dfrac{b^2}{\sqrt{a^2 + b^2}}$, therefore

$$\Delta P_k^{SE} < b^2\left(\frac{2a^2}{\sqrt{a^2 + b^2}} + 3\right) - 2a^2\left(\frac{b^2}{\sqrt{a^2 + b^2}} - 1\right) = 2a^2 + 3b^2.$$

Initial values at region II:

$$x_0 = \frac{a^2}{\sqrt{a^2 + b^2}} \quad \text{and} \quad y_0 = \frac{b^2}{\sqrt{a^2 + b^2}}$$

Notes

x_0 and y_0 will be the accumulative results from region I at the boundary. It is not necessary to calculate them from values of a and b.

$P_0 = P_k^I - ¼[a^2(4y_0 - 3) + b^2(4x_0 + 3)]$ where P_k^I is the accumulative result from region I at the boundary.

$$\Delta P_0^E = b^2(2x_0 + 3)$$

$$\Delta P_0^{SE} = 2a^2 + 3b^2$$

Implementation of the Algorithm

The algorithm explained shows how to get the pixel coordinates in the first quarter only. The ellipse centre is assumed to be at the origin. In actual implementation, the pixel coordinates in other quarters can be simply obtained by use of the symmetric characteristics of an ellipse. For a pixel (x, y) in the first quarter, the corresponding pixels in other three quarters are (x, –y), (–x, y) and (–x, –y) respectively. If the centre is at (xc, yc), all calculated coordinate (x, y) should be adjusted by adding the offset (xc, yc). For easy implementation, a function

PlotEllipse() is defined as follows:

```
PlotEllipse (xC, yC, x, y)
putpixel(xC+x, yC+y)
putpixel(xC+x, yC–y)
putpixel(xC–x, yC+y)
putpixel(xC–x, yC–y)
end PlotEllipse
```

The function to draw an ellipse is described in the following pseudo-codes:

```
DrawEllipse (xC, yC, a, b)
Declare integers x, y, P, ΔPE, ΔPS, ΔPSE, Δ2PE, Δ2PS and Δ2PSE
// Set initial values in region I
        Set x = 0 and y = b
            P = b^2 + (a^2(1 – 4b) – 2)/4   //Intentionally – 2 to round off the value
          ΔP^E = 3b^2
         Δ^2P^E = 2b^2
         ΔP^SE = ΔP^E – 2a^2(b – 1)
        Δ^2P^SE = Δ^2P^E + 2a^2
//Plot the pixel in region I
PlotEllipse(x_c, y_c, x, y)
While ΔP^SE < 2a^2 + 3b^2
if P < 0 then   // Select E
            P = P + ΔP^E
          ΔP^E = ΔP^E + Δ^2P^E
         ΔP^SE = ΔP^SE + Δ2P^E
else   // Select SE
            P = P + ΔP^SE
          ΔP^E = ΔP^E + Δ^2P^E
         ΔP^SE = ΔP^SE + Δ^2P^SE
decrement by
end if
```

Notes

increment x

PlotEllipse(x_c, y_c, x, y)

end while

//Set initial values in region II

$$P = P - (a^2(4y - 3) + b^2(4x + 3) + 2)/4$$

// Intentionally + 2 to round off the value

$$\Delta P^S = a^2(3 - 2y)$$

$$\Delta P^{SE} = 2b^2 + 3a^2$$

$$\Delta^2 P^S = 2a^2$$

//Plot the pixels in region II

While y > 0

If P > 0 then //Select S

$$P = P + \Delta P^E$$

$$\Delta P^E = \Delta P^E + \Delta^2 P^S$$

$$\Delta P^{SE} = \Delta P^{SE} + \Delta^2 P^S$$

else // Select SE

$$P = P + \Delta P^{SE}$$

$$\Delta P^E = \Delta P^E + \Delta 2P^S$$

$$\Delta P^{SE} = \Delta P^{SE} + \Delta^2 P^{SE}$$

Increment x

end if

decrement by

PlotEllipse (x_c, y_c, x, y)

end while

end DrawEllipse

Self Assessment Questions

1. In working with a lattice of points it is useful to avoid floating point arithmetic.
   (*a*) True (*b*) False
2. …….. method is modified from Bresenham's algorithm.
   (*a*) Bresenham's ellipse drawing algorithm
   (*b*) Bresenham's algorithm
   (*c*) Midpoint ellipse algorithm
   (*d*) None of these
3. The curve is divided into two regions. In region I, the slope on the curve is…….
   (*a*) less than –1 (*b*) equal 1
   (*c*) less than –2 (*d*) greater than –1
4. Consider the general equation of an ellipse:
   (*a*) $b_2x_2 - a_2y_2 - a_2b_2 = 0$
   (*b*) $b_2x_2 + a_2y_2 - a_2b_2 = 0$
   (*c*) $b_2x_2 + a_2y_2 + a_2b_2 = 0$
   (*d*) $b_2x_2 + a_2y_2 - x_2y_2 = 0$

5. A prediction (xk+1, yk–½) is set at the middle between the four candidate pixels. **Notes**

(*a*) True (*b*) False

5.3 Ellipse Area, Sector Area and Segment Area

The ellipse area, sector area and segment area are defined as follows:

5.3.1 Ellipse Area

Consider an ellipse that is cantered at the origin, with its axes aligned to the coordinate axes. If the semi-axis length along the x-axis is a and the semi-axis length along the y-axis is B, then the ellipse is defined by a locus of points that reassure the implicit polynomial Equation (1a), or alternatively defined parametrically as in Equation (1b):

$$\frac{x^2}{A^2} + \frac{y^2}{B^2} = 1 \tag{1a}$$

$$\left.\begin{aligned} x &= A \cdot \cos(t) \\ y &= B \cdot \cos(t) \end{aligned}\right\} 0 \le t \le 2\pi \tag{1b}$$

The area of such an ellipse can be found using the parameterized form given in Equation (1b) with the Gauss-Green formula:

$$\text{Area} = \frac{1}{2}\int_{t_1}^{t_2} [x(t) \cdot y'(t) - x(t) \cdot y'(t)]dt \tag{2a}$$

$$= \frac{1}{2}\int_{0}^{2\pi} [A \cdot \text{cost}(t) \cdot B \cdot \cos(t) - B \cdot \sin(t) \cdot (-A) \cdot \sin(t)]dt$$

$$= \frac{A \cdot B}{2}\int_{0}^{2\pi} [\cos^2(t) + \sin^2(t)]dt = \frac{A \cdot B}{2}\int_{0}^{2\pi} dt \tag{2b}$$

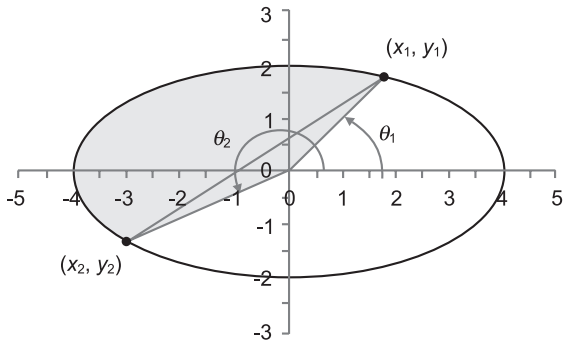
$$= \pi \cdot A \cdot B \tag{3}$$

For more general ellipses, e.g., ellipses that have been rotated and/or translated, the area determined by Equation (3) is still applicable, provided the semi-axis lengths are known. Other methods are also available for calculating ellipse areas.

5.3.2 Ellipse Sector Area

The ellipse sector between two points (x_1, y_1) and (x_2, y_2) on the ellipse is the area that is swept out by a vector from the origin to the ellipse, beginning at the first point (x_1, y_1), as the vector pass through the ellipse in a counter-clockwise direction from (x_1, y_1) to (x_2, y_2). An example is shown in Figure. (See Figure 5.1)

Figure 5.1: Ellipse Sector Area



Notes

The area of an ellipse sector between two points on the ellipse is the area swept out by a vector from the origin to the first point as the vector travels along the ellipse in a counter-clockwise direction to the second point. The area of an ellipse sector can be determined with the Gauss-Green formula, using the parametric angles θ_1 and θ_2.

The Gauss-Green formula applied to an ellipse (Equation (2b)) can be used to determine the area of an ellipse sector as shown in Figure 5.1:

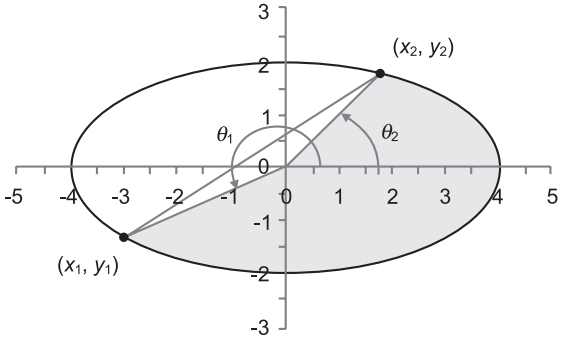
$$\text{Sector Area} = \frac{A \cdot B}{2} \int_{\theta_1}^{\theta_2} dt = \frac{(\theta_2 - \theta_1)\, A \cdot B}{2} \tag{4}$$

The parametric angle θ corresponding to a point (x, y) on the ellipse is formed between the positive x-axis, with positive θ in the counter-clockwise direction. Using the ellipse parameterizations, and the principal-valued opposite trigonometric functions that return angles in the in the range $0 \le \theta \le \pi$ for θ = across (z), and in the range $-\pi/2 \le \theta \le \pi/2$ for θ = arcsin (z), ellipse parametric angles can be found with the relations.

Quadrant II ($x \le 0$ and $y \ge 0$)	**Quadrant I** ($x \ge 0$ and $y \ge 0$)
$\theta = \arccos(x/A)$	$\theta = \arccos(x/A)$
$= \pi - \arcsin(y/B)$	$= \arcsin(y/B)$
Quadrant III ($x < 0$ and $y < 0$)	**Quadrant IV** ($x \ge 0$ and $y < 0$)
$\theta = 2\pi - \arccos(x/A)$	$\theta = 2\pi - \arccos(x/\mathrm{A})$
$= \pi - \arcsin(y/B)$	$= 2\pi + \arcsin(y/B)$

For calculating a sector area, the Gauss-Green formula is sensitive to the direction of integration. For the larger goal of determining ellipse overlap areas, we follow the convention that the sector area is calculated from the first point (x_1, y_1) to the second point (x_2, y_2) in a counter-clockwise direction. For example, if the points (x_1, y_1) and(x_2, y_2) of Figure 5.1 were to have their labels switched, then the ellipse sector defined by the new points will have an area that is complementary to that of the sector in Figure 5.1, as shown in Figure(See Figure 5.2)

Figure 5 2: The Ellipse Sector Area is Calculated From the First Point (x_1, y_1) to the Second Point (x_2, y_2) in a Counter-clockwise Direction



The definitions will always produce an angle in the range $0 \le \theta < 2\pi$ for any point on the ellipse; as such, with the point orientations shown in Figure 5.2, the corresponding angles will be ordered as $\theta_1 > \theta_2$. The first angle can be transformed by subtracting 2π to restore the condition that $\theta_1 < \theta_2$ and the sector area formula (Equation (4) can then be used, with the integration angle from $(\theta_1 - 2\pi)$ through θ_2.

Notes

5.3.3 Ellipse Segment

Area For the overall goal of determining overlap areas between ellipses and other curves, a useful measure is the area of an ellipse segment. A secant line drawn between two points on an ellipse partitions the ellipse area into two fractions, as shown in Figure 5.1 and Figure 5.2. An ellipse segment is the area confined by the secant line and the portion of the ellipse from the first point (x_1, y_1) to the second point (x_2, y_2) traversed in a counter-clockwise direction. The segment's complement is the second of the two areas demarcated by the secant line. For the ellipse shown in Figure 5.1, the segment area defined by the secant line through the points (x_1, y_1) and(x_2, y_2) is the area of the sector minus the area of a triangle defined by the two points and the ellipse centre (at the origin). The coordinates for the triangle's vertices are known, i.e., as (x_1, y_1), (x_2, y_2) and (0, 0), and the triangle area can be found by:

$$\text{Triangle Area} = \frac{1}{2} \cdot |x_1 \cdot y_2 - x_2 \cdot y_1| \quad (5)$$

For the case depicted in Figure 5.1, subtracting the triangle area (Equation (5)) from the area of the ellipse sector (Equation (4)) gives the area between the secant line and the ellipse, i.e., the area of the ellipse segment counter-clockwise from (x_1, y_1) to (x_2, y_2).

For the ellipse of Figure 5.2, the area of the segment shown is the sector area plus the area of the triangle. The key difference between the cases in Figure 5.1 and Figure 5.2 that requires the area of the triangle to be either subtracted from, or added to, the sector area is the size of the integration angle. If the integration angle is less than π, then the triangle area must be subtracted from the sector area to give the segment area. If the integration angle is greater than π, the triangle area must be added to the sector area.

$$\text{Segment Area} = \frac{(\theta_2 - \theta_1)A \cdot B}{2} \pm \frac{1}{2} \cdot |x_1 \cdot y_2 - x_2 \cdot y_1| \quad (6)$$

A robust algorithm for determining the area of an ellipse segment is achieved through a generalization of the cases illustrated in Figure 5.1, Figure 5.2 and Equation (6). For an ellipse that is centred at the origin with its axes parallel to the coordinate axes, the segment area is demarcated by a secant line and the ellipse proceeding counter-clockwise from a first given point (x_1, y_1) to a second given point (x_2, y_2). The ellipse is defined by specifying its semi-axes lengths, A > 0 and B > 0. The points are passed to the algorithm as (x_1, y_1) and (x_2, y_2), both of which must be on the ellipse. The ELLIPSE_SEGMENT algorithm is outlined in Equation (7).

1. $$\theta_1 = \begin{cases} \arccos(x_1 / A) & , y_1 \geq 0 \\ 2\pi - \arccos(x_1 / A), & y_1 < 0 \end{cases} \quad (7a)$$

$$\theta_2 = \begin{cases} \arccos(x_2 / A) & , y_2 \geq 0 \\ 2\pi - \arccos(x_2 / A), & y_2 < 0 \end{cases}$$

2. $$\theta_1 = \begin{cases} \theta_1, & \theta_1 < \theta_2 \\ \theta_1 - 2\pi, & \theta_1 > \theta_2 \end{cases} \quad (7b)$$

3. $$\text{Area} = \frac{(\theta_2 - \hat{\theta}_1)A \cdot B}{2} + \frac{\text{sign}(\theta_2 - \hat{\theta}_1 - \pi)}{2} \cdot |x_1 \cdot y_2 - x_2 \cdot y_1| \quad (7c)$$

Where the ellipse implicit polynomial equation is $\frac{x^2}{A^2} + \frac{y^2}{B^2} = 1$

A > 0 is the semi-axis length along the x-axis

B > 0 is the semi-axis length along the y-axis

(x_1, y_1) is the first given point on the ellipse

(x_2, y_2) is the second given point on the ellipse

θ_1 and θ_2 are the parametric angles corresponding to the points (x_1, y_1) and (x_2, y_2)

Notes

Pseudo-code and an implementation in c-code for ELLIPSE_SEGMENT are given in the along with examples.

Algorithm 1: ELLIPSE_SEGMENT: Calculate the segment area of a centred and coordinate-aligned ellipse given two points on the ellipse.

Inputs: Ellipse parameters semi-axis lengths A, B; coordinates of two points (x_1, y_1) and (x_2, y_2) where the secant intersects the ellipse.

Output: The area between the secant and the ellipse travelling counter-clockwise from (x_1, y_1) to (x_2, y_2), and a diagnostic message indicating either normal termination or an error condition

do if (A ≤ 0 or B ≤ 0)

then return (–1, ERROR_ELLIPSE_PARAMETERS): DATA CHECK

do if (|X12/A2 + Y12/B2 – 1| > ε or |X12/A2 + Y12/B2 – 1| > ε)

then return (–1, ERROR_POINTS_NOT_ON_ELLIPSE) :DATA CHECK

do if (|X1|/A > ε)

do if |X1| – A > ε

then return (–1, ERROR_INVERSE_TRIG)

else do if X1 < 0

then X1 ← – A

else X1 ← A

do if (|X2|/A > ε)

do if |X2| – A > ε

then return (–1, ERROR_INVERSE_TRIG)

else do if X2 < 0

then X2 ← –A

else X2 ← A

do if (Y1 < 0)

then θ1 ← 2π – acos (X1/A)

else θ1 ← acos (X1/A)

do if (Y2 < 0)

then θ2 ← 2π – acos (X2/A)

else θ2 ← acos (X2/A)

do if (θ1 > θ2)

then θ1 ←θ1 – 2π

do if ((θ2 – θ1) > π)

then trsgn ← +1.0

else trsgn ← +1.0

call_es.c

```
#include <stdio.h>
#include <math.h>
#include "program_constants.h"
double ellipse_segment (double A, double B, double X1, double Y1,
double X2, double Y2, int *MessageCode);
```

Notes

```
int main (int argc, char ** argv)
{
double A, B;double X1, Y1;double X2, Y2;
double area1, area2;int rtn;char msg[1024];
printf ("Calling ellipse_segment.c\n"); //-- case shown in Fig. 1A = 4.;
B = 2;
X1 = 4./sqrt (5.);
Y1 = 4./sqrt (5.);
X2 = –3;
Y2 = –sqrt (7.)/2;
area1 = ellipse_segment (A, B, X1, Y1, X2, Y2, &rtn);
sprintf (msg, "Figure 5.1: segment area = %15.8f, return_value = %d\n",
area1, rtn);
printf (msg);
//-- case shown in Figure 5.2
A = 4;
B = 2;
X1 = –3;
Y1 = –sqrt (7.)/2;
X2 = 4./sqrt (5.);
Y2 = 4./sqrt (5.);
area2 = ellipse_segment (A, B, X1, Y1, X2, Y2, &rtn);
sprintf (msg, "Fig 5.2: segment area = %15.8f, return_value = %d\n",
area2, rtn);
printf (msg);
sprintf (msg, "sum of ellipse segments = %15.8f\n", area1 + area2);
printf (msg);
sprintf (msg, "total ellipse area by pi*a*b = %15.8f\n", pi*A*B);
printf (msg);
return rtn;
}
```

5.4 Extending the Core Segment Algorithm to More General Cases

Core segment algorithm is given below:

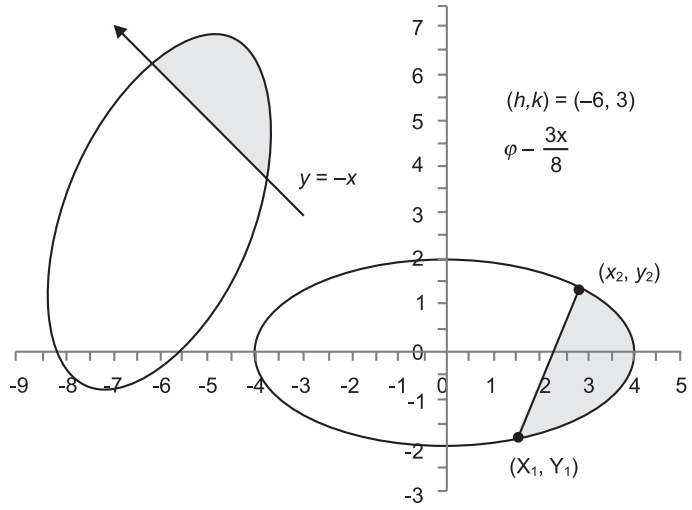
5.4.1 Segment Area for a (Directional) Line through a General Ellipse

The ELLIPSE_SEGMENT algorithm is based on an ellipse that is centred at the origin with its axes aligned to the coordinate axes. The core algorithm can be extended to rotated and/or translated ellipse forms. Consider an ellipse described by Equation (1), with semi-major axis lengths of A and B that is centred at the origin with its axes aligned with the coordinate axes. Suppose that the ellipse is rotated counter-clockwise through an angle φ, and that the ellipse is then translated so that its centre is at the point (h, k). The rotated + translated ellipse could be defined by the set of parameters (A, B, h, k, φ), with the understanding that the rotation through φ is performed before the translation through (h, k). Figure 3 illustrates this idea, showing the ellipse of Figure 1 rotated counter-clockwise through an angle φ = +3π/8, then translated by (h, k) = (–6, +3).

Notes

Suppose we seek to find the segment area within the rotated + translated ellipse that is demarcated by a line, say y = m·x + b. If (h, k) and φ are known for the rotated + translated ellipse, then any two points on the line y = m·x + b can be translated by an amount (–h, –k), and then rotated through –φ, to define two points on a new line defined by y = m′·x + b′. The segment area in the general ellipse with parameters (A, B, h, k, φ) that is demarcated by y = m·x + b will be the same as the segment area in the centred, un-rotated ellipse that is demarcated by y = m′·x + b′. Rotation and translation are affine transformations that also preserve lengths and areas. In particular, the semi-axis lengths in the general rotated ellipse are preserved by both transformations, and corresponding points on the two ellipses will demarcate equal segment areas. To find the segment area, first determine any intersection points between the line y = m′·x + b′ and the centred, un-rotated ellipse. Then, the core segment area algorithm can be called with the intersection points and the semi-axis lengths of the original ellipse, shown in Figure (See Figure 5.3)

Figure 5.3: Translation and Rotation are Affine Transformations that are also Length-and Area-preserving. Corresponding Points on the Two Ellipses will Demarcate Equal Partition Areas



Several concepts should be considered when using the core segment area algorithm. First, the order of points passed to ELLIPSE_SEGMENT will determine which segment area is returned. If the line has an inherent direction, as depicted in Figure 5.3, then the points can be passed to ELLIPSE_SEGMENT in the order that the line contacts the ellipse, and the area returned will be to the right of the line. A second point to consider is that the algorithms presented here assume that the semi-axis lengths A and B are in the direction of the x- and y-axes, respectively, in the un-rotated ellipse. In its rotated orientation, the semi-axis length A will rarely be oriented horizontally; e.g., for φ = π/4, the semi-axis length A will be oriented vertically.

For any point (x_i, y_i) on the line y = m·x + b, the transformations required to move the points into an orientation with respect to a standard ellipse that is analogous to their orientation to the given ellipse are the inverse of what it took to rotate and translate a standard ellipse to its desired position. The point is first translated, and then rotated by:

$$\begin{bmatrix} x_{1_{TR}} \\ y_{1_{TR}} \end{bmatrix} = \begin{bmatrix} \cos(-\varphi) & -\sin(-\varphi) \\ \sin(-\varphi) & \cos(-\varphi) \end{bmatrix} \cdot \begin{bmatrix} x_1 - h \\ y_1 - k \end{bmatrix}$$

Notes

Giving the following expressions for the translated + rotated point coordinates:

$$x_{1_{TR}} = \cos(\varphi) \cdot (x_i - h) + \sin(\varphi) \cdot (y_1 - k) \quad (8a)$$

$$y_{1_{TR}} = \sin(\varphi) \cdot (x_i - h) + \cos(\varphi) \cdot (y_1 - k) \quad (8b)$$

To accommodate cases where the translated + rotated points determined by Equation (8) result in a vertical line, points of intersection between the line and ellipse can be found using alternative substitutions. Define two alternative slopes and line equations as:

$$m_{yx} = \frac{y_{2_{TR}} - y_{1_{TR}}}{x_{2_{TR}} - x_{1_{TR}}}, y = y_{1_{TR}} + m_{yx} \cdot (x - x_{1_{TR}}) \quad (9a)$$

$$m_{xy} = \frac{x_{2_{TR}} - x_{2_{TR}}}{y_{2_{TR}} - y_{2_{TR}}}, x = x_{1_{TR}} + m_{xy} \cdot (y - y_{1_{TR}}) \quad (9b)$$

Points of intersection are found by substituting the appropriate line equation into the standard ellipse equation, and solving for the remaining variable:

$$\left[\frac{B^2 + A^2 \cdot (m_{yx})^2}{A^2}\right] \cdot x^2 + \left[2 \cdot \left(y_{1_{TR}} \cdot m_{yx} - (m_{yx})^2 \cdot x_{1_{TR}}\right)\right] \cdot x + \left[(y_{1_{TR}})^2 - 2 \cdot m_{yx} \cdot x_{1_{TR}} \cdot y_{1_{TR}} + (m_{yx} \cdot x_{1_{TR}})^2 - B^2\right] = 0 \quad (10a)$$

$$\left[\frac{A^2 + B^2 \cdot (m_{xy})^2}{B^2}\right] \cdot y^2 + \left[2 \cdot \left(x_{1_{TR}} \cdot m_{xy} - (m_{xy})^2 \cdot y_{1_{TR}}\right)\right] \cdot y + \left[(x_{1_{TR}})^2 - 2 \cdot m_{xy} \cdot x_{1_{TR}} \cdot y_{1_{TR}} + (m_{xy} \cdot y_{1_{TR}})^2 - A^2\right] = 0 \quad (10b)$$

If the translated + rotated line is not vertical, then use Equation (10a) to find x-values for any points of intersection. If the translated + rotated line is close to vertical, then Equation (10b) can be used to find y-values for any points of intersection. Since points of intersection between the line and the ellipse are determined by solving a quadratic equation, there are three cases to consider:

1. Complex Conjugate Roots (no points of intersection)
2. One Double Real Root (1 point of intersection; line tangent to ellipse)
3. Two Real Roots (2 points of intersection; line crosses ellipse)

For the first two cases, the segment area will be zero. For the third case, roots of the quadratic Equation (10a) represent x-values of the points of intersection between the line and the ellipse; roots of Equation (10b) represent y-values of intersection points. Determining intersection points from the quadratic Equation (10) roots is accomplished using the line Equation (10). The two intersection points are sent to ELLIPSE_SEGMENT to determine the segment area. To ensure consistency in area measures, the integration direction is specified to be from the first point to the second point. As such, the ellipse line overlap algorithm should be sensitive to the order that the points are passed to ELLIPSE_SEGMENT. We suggest giving the line a 'direction' from the first given point on the line to the second. The line 'direction' can then be used to determine which is to be the first point of intersection, i.e., the first intersection point is where the line enters the ellipse based on what 'direction' the line is pointing.

The approach outlined above for finding the overlap area between a line and a general ellipse is implemented in ELLIPSE_LINE_OVERLAP. Pseudo-code and an implementation in c-code are given along with examples. The ellipse is passed to the algorithm by specifying the counter clockwise rotation angle φ and the translation (h, k) that takes a standard ellipse and moves it to the desired orientation, along with the semi-axes lengths, $A > 0$ and $B > 0$. The line is passed to the algorithm as two points on the line, (x_1, y_1) and (x_2, y_2). The 'direction' of the line is taken to be from (x_1, y_1) toward (x_2, y_2). Then, the segment area returned from ELLIPSE_LINE_OVERLAP will be the area within the ellipse to the right of the line's path.

Notes

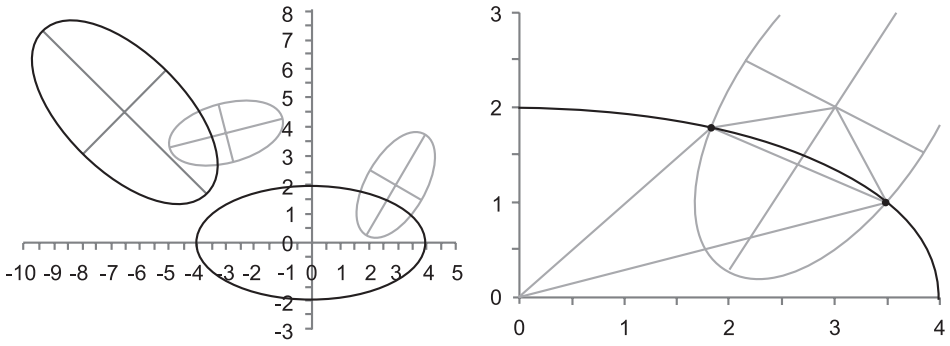
5.4.2 Ellipse-ellipse Overlap Area

The method described for determining the area between a line and an ellipse can be widened to the task of finding the overlap area between two general ellipses.

Suppose the two ellipses are defined by their semi-axis lengths, centre locations and axis rotation angles, e.g., as (A_1, B_1, h_1, k_1, φ_1) and (A_2, B_2, h_2, k_2, φ_2). The approach presented here will be to first translate both ellipses by an amount ($-h_1$, $-k_1$) that puts the centre of the first ellipse at the origin. Then, both translated ellipses are rotated about the origin by an angle $-\varphi_1$ that aligns the axes of the first ellipse with the coordinate axes, as illustrated in Figure 5.4. Intersection points are found for the two translated + rotated ellipses, by solving the two implicit polynomial equations simultaneously with Ferrari's quadratic formula. Finally, the segment algorithm described above is employed to find all the pieces of the overlap area.

For example, consider a case of two general ellipses with two (non-tangential) points of intersection, as shown in Figure 5.4. The translation and rotation transformations that put the first ellipse at the origin and aligned with the coordinate axes do not vary the overlap area. For the case illustrated in Figure 5.4, the overlap area consists of the union of two segments, one from each ellipse. The desired segment from the first ellipse can be found immediately with the segment algorithm, based on the points of intersection. To find the segment area of the second ellipse, the approach presented here further translates and rotates the second ellipses that the segment algorithm can also be used directly. All other possible cases, e.g. with 3 and 4 points of intersection, can also be handled using the segment algorithm. (See Figure 5.4)

Figure 5.4: Overlap Area of Ellipse



The overlap area algorithm presented here finds the area of appropriate segments of each ellipse that contribute to the overlap area. Contributing segments are demarcated by any points of intersection between the two ellipse curves.

Algorithm 2: ELLIPSE_LINE_OVERLAP: Calculate the Segment Area of a General Ellipse that is demarcated by a directional line.

Inputs

Ellipse parameters semi-axis lengths A, B (of the un-rotated ellipse); counter clockwise rotation angle φ; translation of the ellipse centre (H, K) away from the origin (where the translation is applied after the rotation angle); coordinates of two points (x_1, y_1) and (x_2, y_2) on the line which need not lie on the ellipse.

Outputs

The area between the line and the ellipse travelling counter-clockwise from the point on the ellipse where the line travelling from (x_1, y_1) to (x_2, y_2) first crosses the ellipse, and a diagnostic message indicating either normal termination or an error condition.

do if (A ≤ 0 or B ≤ 0)

Notes

then return (–1, ERROR_ELLIPSE_PARAMETERS) :DATA CHECK

do if ($|\varphi| > 2\pi$

then $\varphi \leftarrow (\varphi$ modulo $2\pi)$:BRING φ INTO $-2\pi \leq \varphi < 2\pi$ (?)

do if ($|X_1|/A > 2\pi$)

then X1 ← –A

$X10 \leftarrow \cos(\varphi)*(X_1 - H) + \sin(\varphi)*(Y_1 - K)$

$Y_{10} \leftarrow -\sin(\varphi)*(X_1 - H) + \cos(\varphi)*(Y_1 - K)$

$X_{20} \leftarrow \cos(\varphi)*(X_2 - H) + \sin(\varphi)*(Y_2 - K)$

$Y_{20} \leftarrow -\sin(\varphi)*(X_2 - H) + \cos(\varphi)*(Y_2 - K)$

do if ($|X_{20} - X_{10}| > \varepsilon$) :LINE IS NOT VERTICAL

then $m \leftarrow (Y_{20} - Y_{10})/(X_{20} - X_{10})$:STORE QUADRATIC COEFFICIENTS

$a \leftarrow (B_2 + (A*m)2)/A_2$

$b \leftarrow (2.0*(Y_{10}*m - m_2*X_{10}))$

$c \leftarrow (Y_{102} - 2.0*m*Y_{10}*X_{10} + (m*X_{10})2 - B_2)$

else if ($|Y_{20} - Y_{10}| > \varepsilon$) :LINE IS NOT HORIZONTAL

then $m \leftarrow (X_{20} - X_{10})/(Y_{20} - Y_{10})$:STORE QUADRATIC COEFFS

$a \leftarrow (A_2 + (B*m)2)/B_2$

$b \leftarrow (2.0*(X_{10}*m - m_2*Y_{10}))$

$c \leftarrow (X_{102} - 2.0*m*Y_{10}*X_{10} + (m*Y_{10})2 - A_2)$

else return (–1, ERROR_LINE_POINTS) :LINE POINTS TOO CLOSE

discrim ← b2 – 4.0*a*c

do if (discrim < 0.0):LINE DOES NOT CROSS ELLIPSE

then return (0, NO_INTERSECT)

else if (discrim > 0.0): TWO INTERSECTION POINTS

then root1 ← (–b – sqrt (discrim))/(2.0*a)

root2 ← (–b + sqrt (discrim))/ (2.0*a)

else return (0, TANGENT) :LINE TANGENT TO ELLIPSE

do if ($|X_{20} - X_{10}| > \varepsilon$) :ROOTS ARE X-VALUES

then do if ($X_{10} < X_{20}$) :ORDER PTS SAME AS LINE DIRECTION

then $x_1 \leftarrow root_1$

$x_2 \leftarrow root_2$

else $x_1 \leftarrow root_2$

$x_2 \leftarrow root_1$

else do if ($Y_{10} < Y_{20}$) :ROOTS ARE Y-VALUES

then $y_1 \leftarrow root_1$:ORDER PTS SAME AS LINE DIRECTION

$y_2 \leftarrow root_2$

else $y_1 \leftarrow root_2$

$y_2 \leftarrow root_1$

(Area,Code) ← ELLIPSE_SEGMENT (A,B,x_1,y_1,x_2,y_2)

Notes *Program*

```
#include <stdio.h>
#include <math.h>
#include "program_constants.h"
double ellipse_segment (double A, double B, double X1, double Y1,
double X2,
double Y2, int *MessageCode);
double ellipse_line_overlap (double PHI, double A, double B, double H,
double K, double X1, double Y1, double X2,
double Y2, int *MessageCode);0010 0011 int main (int argc, char ** argv)
{
double A, B;
double H, K, PHI;
double X1, Y1;
double X2, Y2;
double area1, area2;
int rtn;
char msg[1024];
printf ("Calling ellipse_line_overlap.c\n");
//-- case shown in Fig 5.3
A = 4;
B = 2;
H = –6;
K = 3;
PHI = 3.*pi/8.0;
X1 = –3;
Y1 = 3;
X2 = –7;
Y2 = 7;
area1 = ellipse_line_overlap (PHI, A, B, H, K, X1, Y1, X2, Y2, &rtn);
sprintf (msg, "Fig 5.4: area = %15.8f, return_value = %d\n", area1, rtn);
printf (msg);
//-- case shown in Fig.5 3, points reversed
A = 4;
B = 2;
H = –6;
K = 3;
PHI = 3.*pi/8.0;
X1 = –7;
Y1 = 7;
X2 = –3;
Y2 = 3;
```

Notes

```
area2 = ellipse_line_overlap (PHI, A, B, H, K, X1, Y1, X2, Y2, &rtn);
sprintf (msg, "Fig 5.3 reverse: area = %15.8f, return_value = %d\n",
area2, rtn);
printf (msg);
sprintf (msg, "sum of ellipse segments = %15.8f\n", area1 + area2);
printf (msg);
sprintf (msg, "total ellipse area by pi*a*b = %15.8f\n", pi*A*B);
printf (msg);
return rtn;
}
```

Determining Intersection Points

To find intersection points, the two ellipse equations are solved simultaneously, using the implicit polynomial forms. The first ellipse equation, in its translated + rotated position, is written using the appropriate semi-axis lengths:

$$\frac{x^2}{A_1^2} + \frac{y^2}{B_1^2} = 1 \tag{11}$$

In a general form of this problem, the translation + rotation that positions the first ellipse centered at the origin and oriented with the coordinate axes will typically leave the second ellipse displaced and rotated, as in Figure 5.4. The implicit polynomial form for a more general ellipse that is rotated and/or translated away from the origin is written in the conventional way as:

$$AA \cdot x^2 + BB \cdot x \cdot y + CC \cdot y^2 + DD \cdot x + EE \cdot y + FF = 0 \tag{12}$$

Any points of intersection for the two ellipses will satisfy Equation (11) and Equation (12) simultaneously. An intermediate goal is to find the implicit polynomial coefficients in Equation (12) that describes the second ellipse after the translation + rotation that positions the first ellipse centred at the origin and oriented with the coordinate axes. The second ellipse centre (h_2, k_2) is translated through an amount ($-h_1$, $-k_1$), then centre-point is rotated through $-\varphi_1$ to give a new centre point (h_2^{TR}, k_2^{TR}):

$$h_{2_{TR}} = \cos(-\varphi_1) \cdot (h_2 - h_1) - \sin(-\varphi_1) \cdot (k_2 - k_1) \tag{13a}$$

$$k_{2_{TR}} = \sin(-\varphi_1) \cdot (h_2 - h_1) + \cos(-\varphi_1) \cdot (k_2 - k_1) \tag{13b}$$

The coordinates for any point (x_{TR}, y_{TR}) from Equation (12) on the second ellipse in its new translated + rotated position can be found from the following parametric equations, based on an ellipse with semi-axis lengths A_2 and B_2 that is centred at the origin, then rotated and translated to the desired position:

$$\left.\begin{aligned} x_{TR} &= A_2 \cdot \cos(t) \cdot \cos(\varphi_2 - \varphi_1) - B_2 \cdot \sin(t) \cdot \sin(\varphi_2 - \varphi_1) + h_{2_{TR}} \\ y_{TR} &= A_2 \cdot \cos(t) \cdot \sin(\varphi_2 - \varphi_1) - B_2 \cdot \sin(t) \cdot \cos(\varphi_2 - \varphi_1) + k_{2_{TR}} \end{aligned}\right\} \; 0 \le t \le 2\pi \tag{14}$$

To find the implicit polynomial coefficients from the parametric form, transform the locus of points (x_{TR}, y_{TR}) so that they lie on the ellipse (A_2, B_2, 0, 0, 0), which is accomplished by first translating through ($-(h_1 - h_2)$,$(k_1 - k_2)$) and then rotating the point through the angle $-(\varphi_1 - \varphi_2)$:

$$x = \cos(\varphi_2 - \varphi_1) \cdot (x_{TR} - (h_1 - h_2)) - \sin(\varphi_2 - \varphi_1) \cdot (y_{TR} - (k_1 - k_2)) \tag{15a}$$

$$y = \sin(\varphi_2 - \varphi_1) \cdot (x_{TR} - (h_1 - h_2)) - \cos(\varphi_2 - \varphi_1) \cdot (y_{TR} - (k_1 - k_2)) \tag{15b}$$

The locus of points (x, y) should satisfy the standard ellipse equation with the appropriate semi-axis lengths, A_2 and B_2. Finally, the implicit polynomial coefficients for the second ellipse

Notes

Equation (12) are found by substituting the expressions for the point (x, y) into a standard ellipse equation, yielding the following ellipse equation:

$$\frac{[\cos(\varphi_2-\varphi_1)\cdot(x_{TR}-(h_1-h_2))-\sin(\varphi_2-\varphi_1)\cdot(y_{TR}-(k_1-k_2))]^2}{A_2^2}$$

$$+\frac{[\sin(\varphi_2-\varphi_1)\cdot(x_{TR}-(h_2-h_2))-\cos(\varphi_2-\varphi_1)\cdot(y_{TR}-(k_1-k_2))]^2}{B_2^2}=1 \quad (16)$$

Where (x_{TR}, y_{TR}) are defined in Equation (15). Expanding the terms of Equation (16), and then simplifying yields expressions for the implicit polynomial coefficients in Equation (12) for a general ellipse with parameters (A_2, B_2, h_2TR, k_2TR, $\varphi_2 - \varphi_1$):

$$AA=\frac{\cos^2(\varphi_2-\varphi_1)}{A_2^2}+\frac{\sin^2(\varphi_2-\varphi_1)}{B_2^2} \quad (17a)$$

$$BB=\frac{2\cdot\sin(\varphi_2-\varphi_1)\cdot\cos(\varphi_2-\varphi_1)}{A_2^2}-\frac{2\cdot\sin(\varphi_2-\varphi_1)\cdot\cos(\varphi_2-\varphi_1)}{B_2^2} \quad (17b)$$

$$CC=\frac{\sin^2(\varphi_2-\varphi_1)}{A_2^2}+\frac{\cos^2(\varphi_2-\varphi_1)}{B_2^2} \quad (17c)$$

$$DD=\frac{-2\cdot\cos(\varphi_2-\varphi_1)\cdot[h_{2_{TR}}\cdot\cos(\varphi_2-\varphi_1)+k_{2TR}\cdot\sin(\varphi_2-\varphi_1)]}{A_2^2}+$$

$$\frac{-2\cdot\sin(\varphi_2-\varphi_1)\cdot[k_{2_{TR}}\cdot\cos(\varphi_2-\varphi_1)-h_{2_{TR}}\cdot\sin(\varphi_2-\varphi_1)]}{B_2^2} \quad (17d)$$

$$EE=\frac{-2\cdot\sin(\varphi_2-\varphi_1)\cdot[h_{2_{TR}}\cdot\cos(\varphi_2-\varphi_1)-k_{2TR}\cdot\sin(\varphi_2-\varphi_1)]}{A_2^2}+$$

$$\frac{2\cdot\cos(\varphi_2-\varphi_1)\cdot[h_{2_{TR}}\cdot\sin(\varphi_2-\varphi_1)-k_{2_{TR}}\cdot\cos(\varphi_2-\varphi_1)]}{B_2^2} \quad (17e)$$

$$FF=\frac{[h_{2_{TR}}\cdot\cos(\varphi_2-\varphi_1)-k_{2_{TR}}\cdot\sin(\varphi_2-\varphi_1)]^2}{A_2^2}+\frac{[h_{2_{TR}}\cdot\sin(\varphi_2-\varphi_1)-k_{2_{TR}}\cdot\cos(\varphi_2-\varphi_1)]^2}{B_2^2}-1 \quad (17f)$$

Intersection points are found by solving simultaneously the two implicit polynomials denoted above as Equation (11) and Equation (12). Solving for x in Equation (11):

$$\frac{x^2}{A_1^2}+\frac{y^2}{B_1^2}=1 \quad \Rightarrow x=\pm\sqrt{A_1^2\cdot\left(1-\frac{y^2}{B_1^2}\right)} \quad (18)$$

Substituting the expressions for x of Equation (18) into Equation (12), where the coefficients are defined in Equation (17) yields a quadratic polynomial in y. substituting either the positive or the negative root gives the same quadratic polynomial coefficients, which are:

$$cy[4]\cdot y^4+cy[3]\cdot y^3+cy[2]\cdot y^2+cy[1]\cdot y^1+cy[0]=0 \quad (19)$$

where:

$$\frac{cy[4]}{B_1}=A_1^4\cdot AA^2+B_1^2\cdot\left[A_1^2\cdot(BB^2-2\cdot AA\cdot CC)+B_1^2\cdot CC^2\right]$$

$$\frac{cy[3]}{B_1}=2\cdot B_1\cdot\left[B_1^2\cdot CC\cdot EE+A_1^2\cdot(BB\cdot DD-AA\cdot EE)\right]$$

$$\frac{cy[2]}{B_1}=A_1^2\cdot\left\{\left[B_1^2\cdot(2\cdot AA\cdot CC-BB^2)+DD^2-2\cdot AA\cdot FF\right]-2\cdot A_1^2\cdot AA^2\right\}+B_1^2\cdot(2\cdot CC\cdot FF+EE^2)$$

$$\frac{cy[1]}{B_1}=2\cdot B_1\cdot\left[A_1^2\cdot(AA\cdot EE-BB\cdot DD)+EE\cdot FF\right]$$

$$\frac{cy[0]}{B_1}=\left[A_1\cdot(A_1\cdot AA-DD)+FF\right]\cdot[A_1\cdot AA+DD)+FF]$$

Notes

The quadratic polynomial of Equation (19) will have real roots if and only if the two curves intersect. If the ellipses do not intersect, then the quadratic will have only complex roots. Furthermore, any real roots of the quadratic polynomial will represent y-values of intersection points between the two ellipse curves. As with the quadratic equation that arises in the ellipse-line overlap calculation, the ellipse-ellipse overlap algorithm should handle all possible cases for the types of quadratic polynomial roots:

1. Four real roots (distinct or not); the ellipse curves intersect.
2. Two real roots (distinct or not) and one complex-conjugate pair; the ellipse curves intersect.
3. No real roots (two complex-conjugate pairs); the ellipse curves do not intersect.

For the method we present here, polynomial roots are found using Ferrari's quadratic formula. Four complex roots are returned, and any roots whose imaginary part is returned as zero is a real root. When polynomial coefficients are constructed as in Equation (19), the general case of two distinct ellipses typically results in a quadratic polynomial, i.e., the coefficient cy[4] of Equation (19) is non-zero. However, certain cases lead to polynomials of lesser degree. Fortunately, the solver is conveniently modular, providing separate functions BIQUADROOTS, CUBICROOTS and QUADROOTS to handle all the possible polynomial cases that arise when seeking points of intersection for two ellipses.

Algorithm 3: ELLIPSE_ELLIPSE_OVERLAP: Calculate the overlap area between two general ellipses.

Inputs

Parameters for both ellipses: semi-axis lengths A_1, B_1 (of the un-rotated first ellipse); counter-clockwise rotation angle φ_1; translation of the ellipse centre (H_1, K_1) away from the origin. Semi-axis lengths A_2, B_2 (of the un-rotated second ellipse); counter-clockwise rotation angle φ_2; translation of the ellipse centre (H_2, K_2) away from the origin.

Outputs

The overlap area between the two ellipses, and a diagnostic message indicating either ellipse configuration or an error condition.

do if $(A_1 \le 0$ or $B_1 \le 0)$ OR $(A_2 \le 0$ or $B_2 \le 0)$

then return (–1, ERROR_ELLIPSE_PARAMETERS): DATA CHECK

do if $(|\varphi_1| > 2\pi)$

then $\varphi_1 \leftarrow (\varphi 1$ modulo $2\pi)$

do if $(|\varphi_2| > 2\pi)$

then $\varphi_2 \leftarrow (\varphi_2$ modulo $2\pi)$

$H_2\_TR \leftarrow (H_2 - H_1)*\cos(\varphi_1) + (K_2 - K_1)*\sin(\varphi_1)$: TRANS+ROT ELL2

$K_2\_TR \leftarrow (H_1 - H_2)*\sin(\varphi_1) + (K_2 - K_1)*\cos(\varphi_1)$

$\varphi_2R \leftarrow \varphi_2 - \varphi_1$

do if $(|\varphi_2R| > 2\pi)$

then $\varphi_2R \leftarrow (\varphi_2R$ modulo $2\pi)$

AA ← $\cos(\varphi_2R)/A22 + \sin2(\varphi 2R)/B_{22}$:BUILD IMPLICIT COEFFS ELL2TR

BB ← $2*\cos(\varphi_2R)*\sin(\varphi_2R)/A_{22} - 2*\cos(\varphi_2R)*\sin(\varphi_2R)/B_{22}$

CC ← $\sin_2(\varphi_2R)/A22 + \cos_2(\varphi 2R)/B_{22}$

Notes

DD ← –2*cos (φ2R)*(cos (φ_2R)*H2_TR + sin (φ_2R)*K2_TR)/A_{22}

2*sin (φ_2R)*(sin (φ_2R)*H_2_TR – cos (φ_2R)*K_2_TR)/B2216

EE ← –2*sin (φ_2R)* (cos (φ_2R)*H_2_TR + sin (φ_2R)*K_2_TR)/A_{22} + 2*cos (φ_2R)* (sin (φ_2R)*H_2_TR – cos (φ_2R)*K_2_TR)/B2217

FF ← (–cos (φ_2R)*H2_TR – sin (φ_2R)*K_2_TR)2/A_{22}+ (sin (φ_2R)*H_2_TR – cos (φ_2R)*K_2_TR)2/B_{22} – 1

cy[4] ← A_{14}*AA_2 + B_{12}*(A_{12}*(BB_2 – 2*AA*CC)+ B_{12}*CC_2)

cy[3] ← 2*B_1*(B_{12}*CC*EE + A_{12}*(BB*DD – AA*EE))

cy[2] ← A_{12}*((B_{12}*(2*AA*CC – BB_2) + DD_2 – 2*AA*FF)–2*A_{12}*AA_2 + B_{12}*(2*CC*FF + EE2)

cy[1] ← 2*B_1*(A_{12}*(AA*EE – BB*DD) + EE*FF)

cy[0] ← (A_1*(A_1*AA – DD) + FF)*(A_1*(A_1*AA + DD) + FF)

py[0] ← 1

do if (|cy[4]| > 0)

then for i ← 0 to 3 by 1

py[4–i] ← cy[i]/cy[4]

r[][] ← BIQUADROOTS (py[])

nroots ← 4

else if (|cy[3]| > 0)

then for i ← 0 to 2 by 1

py[3–i] ← cy[i]/cy[3]

r[][] ← CUBICROOTS (py[])

nroots ← 3

else if (|cy[2]| > 0)

then for i ← 0 to 1 by 1

py[2–i] ← cy[i]/cy[2]

r[][] ← QUADROOTS (py[])

nroots ← 2

else if (|cy[1]| > 0)

then r[1][1] ← (–cy[0]/cy[1])

r[2][1] ← 0

nroots ← 1

else

nroots ← 0

nychk ← 0

for i ← 1 to nroots by 1

Notes

```
do if       (|r[2][i]| < EPS)48 then nychk ← nychk + 1
            ychk[nychk] ← r[1][i]*B1
for         j ← 2 to nychk by 1
            tmp0 ← ychk[j]
for         k ← (j – 1) to 1 by –1
do if       (ychk[k] ≤ tmp0)
then break
else        ychk[k+1] ← ychk[k]56 ychk[k+1] ← tmp0
nintpts
then        x1 ← 0
else        x1 ← A1*sqrt (1.0 – ychk[i]2/B12)
            x2 ← –x1
do if       (|ellipse2tr (x1,ychk[i],AA,BB,CC,DD,EE,FF)| < EPS/2)
then        nintpts ← nintpts + 1
do if       (nintpts > 4)
then return (–1, ERROR_INTERSECTION_PTS)
            xint[nintpts] ← x1
            yint[nintpts] ← ychk[i]
do if       ((|ellipse2tr (x2, ychk[i],AA,BB,CC,DD,EE,FF)| < EPS/2)
and         (|x2 – x1| > EPS/2))
then        nintpts ← nintpts + 1
do if       (nintpts > 4)
then return (–1, ERROR_INTERSECTION_PTS)76 xint[nintpts] ← x1
            yint[nintpts] ← ychk[i]
switch (nintpts) : HANDLE ALL CASES FOR # OF INTERSECTION PTS
case 0:
case 1:
(OverlapArea,Code) ← NOINTPTS (A1,B1,A2,B2,H1,K1,H2,K2,AA,BB,CC,DD,EE,FF)
return (OverlapArea,Code)
case 2:
Code ← istanpt (xint[1],yint[1],A1,B1,AA,BB,CC,DD,EE,FF)
do if (Code == TANGENT_POINT)
then (OverlapArea,Code) ← NOINTPTS (A1,B1,A2,B2,H1,K1,H2,K2,AA,BB,CC,DD,EE,FF)
else (OverlapArea,Code) ← TWOINTPTS (xint[],yint[],A1,
PHI_1,A2,B2,H2_TR,K2_TR,PHI_2,AA,BB,CC,DD,EE,FF)
```

Notes

return (OverlapArea, Code)

case 3:

(OverlapArea,Code) THREEINTPTS (xint,yint,A1,B1,PHI_1,A2,B2,H2_TR,K2_ TR,PHI_2,AA,BB,CC,DD,EE,FF)91

return (OverlapArea, Code)

case 4:

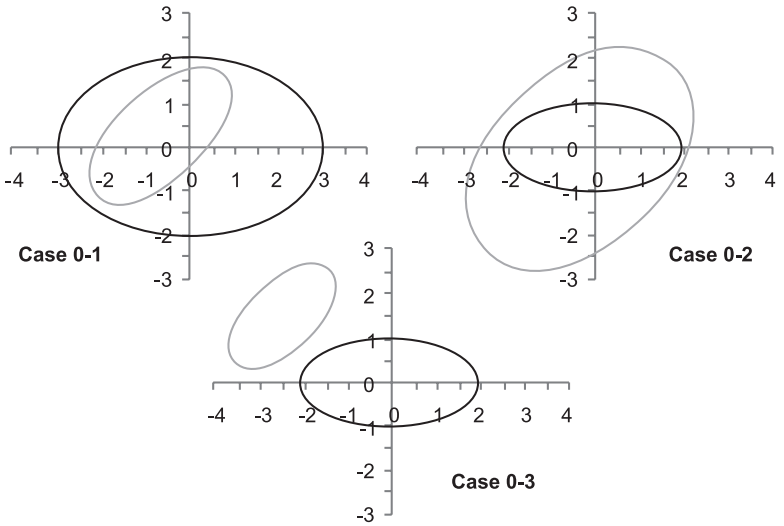
(OverlapArea,Code) ← FOURINTPTS (xint,yint,A1,B1,PHI_1,A2, B2,H2_TR,K2_ TR,PHI_2,AA,BB,CC,DD,EE,FF)

return (OverlapArea,Code)

Area Determination for Non-Intersecting Ellipses

If the polynomial solver returns no real roots of the polynomial, then the ellipse curves do not intersect. It follows that the two ellipse areas are either disjoint, or one ellipse area is fully contained within the other; all three possibilities are shown in Figure (See Figure 5.5)

Figure 5.5: When the Quadratic Polynomial has no Real Roots, the Ellipse Curves do not Intersect. It Follows that Either One Ellipse is Fully Contained within the Other, or the Ellipse Areas are Completely Disjoint, Resulting in three Distinct Cases for Overlap Area



Each sub-case in Figure 5.5 requires a different overlap-area calculation, i.e. either the overlap area is zero (Case 0-3), or the overlap is the area of the first ellipse (Case 0-2), or the overlap is the area of the second ellipse (Case 0-1). When the polynomial has no real roots, geometry can be used to determine which specific sub-case of Figure 5.5 is represented. An efficient logic starts by determining the relative size of the two ellipses, e.g., by comparing the product of semi-axis lengths for each ellipse. The area of an ellipse is proportional to the product of its two semi-axis lengths, so the relative size of two ellipses can be determined by comparing the product of semi-axis lengths:

$$A_1 \cdot B_1 \leq (\text{or} \geq)\ A_1 \cdot B_1 \Rightarrow \pi \cdot A_1 \cdot B_1 \leq (\text{or} \geq)\ \pi \cdot A_1 \cdot B_1 \tag{20}$$

Suppose the first ellipse is larger than the second ellipse, as determined by the relation in Equation (20). In this case, if the second ellipse center (h_2^{TR}, k_2^{TR}) is inside the first ellipse, then the second

Notes

ellipse is wholly contained within the first ellipse (Case 0-1); otherwise, the ellipses are disjoint (Case 0-3). The logic relies on the fact that there are no intersection points, which is indicated whenever there are no real solutions to the quadratic polynomial. To test whether the second ellipse centers inside the first ellipse, evaluate the first ellipse Equation (20) at the point $x = h_{2TR}$ and $y = k_2^{TR}$; if the result is less than one, then the point (h_2^{TR}, k_2^{TR}) is inside the first ellipse. A complete logic for differentiating between Case 0-1 and Case 0-3 is: If the polynomial has no real roots and $A_1 \cdot B_1 > A_2 \cdot B_2$, and Equation (11) evaluated at the point (h_{2TR}, k_{2TR}) is less than 1:

$$\frac{h_{2TR}^2}{A_1^2} + \frac{k_{2TR}^2}{B_1^2} < 1$$

Then the first ellipse wholly contains the second (Case 0-1), otherwise the two ellipses are disjoint (Case 0-3).

A similar logic can be applied to differentiate between Case 0-2 and Case 0-3. If the second ellipse is larger than the first ellipse (by Equation (20), $A_1 \cdot B_1 < A_2 \cdot B_2$), and the first ellipse center (0, 0) is inside the second ellipse (by Equation (12), FF < 0), then the first ellipse is wholly contained within the second ellipse (Case 0-2); otherwise the ellipses are disjoint (Case 0-3).Suppose that no intersection points exist, and that the two ellipses are the same size, i.e., by Equation (20), $A_1 \cdot B_1 = A_2 \cdot B_2$. These observations should indicate that the ellipses are disjoint (Case 0-3).

Area Determination for Intersecting Ellipses

If the polynomial solver returns either two or four real roots to the quadratic polynomial of Equation (19), then the ellipse curves intersect. All of the various possibilities for the number and type of real roots can be addressed by creating a list of distinct real roots. First, loop through the entire array of complex roots returned by the polynomial solver, and retrieve only real roots, i.e., only the roots whose imaginary component is zero. Then, sort the list of real roots, a step which supports an efficient check for multiple roots. As the sorted list of real roots is traversed, any root that is 'identical' to the previous root can be skipped.

Each distinct real root of the polynomial represents a y-value where the two ellipses intersect. Each y-value can represent either one or two potential points of intersection. Suppose that a root is $y = B_1$ (or $y = -B_1$), then the y-value produces a single intersection point, which is at $(0, B_1)$ (or $(0, -B_1)$). In the second case, if the y-value is in the open interval $(-B_1, B_1)$, then there are two potential intersection points where the y-value is on the first ellipse

$$\left(A_1 \cdot \sqrt{1 - \frac{y^2}{B_1^2}}, y\right), \text{ and } \left(-A_1 \cdot \sqrt{1 - \frac{y^2}{B_1^2}}, y\right) \tag{21}$$

Each potential intersection point (x_i, y_i) of Equation (21) is evaluated in the second ellipse Equation (12):

$$AA \cdot x_i^2 + BB \cdot x_1 \cdot y_1 + CC \cdot y_1^2 + DD \cdot x_1 + EE \cdot y_1 + FF, i = 1, 2 \tag{22}$$

If the expression of Equation (22) evaluates to zero, then the point (x, y) is on both ellipses, i.e., it is an intersection point. By checking all points (x, y) for each value of y that is a root of the polynomial, a list of distinct intersection points is generated. The number of distinct intersection points must be 0, 1, 2, 3 or 4. The case of zero intersection points is described above, with all possible sub-cases illustrated in Figure 5.5. If there is only one distinct intersection point, then the two ellipses must be tangent at that point. The three possibilities for a single tangent point are shown in Figure (See Figure 5.6)

For the purpose of determining overlap area, the cases of 0 or 1 intersection points can be handled in the same manner. When two intersection points exist, there are three possible sub-cases, shown in Figure 5.7. It is possible that both of the intersection points are tangents (Case 2-1 and Case 2-2). In both of these sub-cases, one ellipse must be fully contained within the other. The only other possibility for two intersection points is a partial overlap (Case 2-3). (See Figure 5.7)

Notes

Figure 5.6: When only one intersection point exists, the ellipses must be tangent at the intersection point. As with the case of zero intersection points, either one ellipse is fully contained within the other, or the ellipse areas are disjoint. The algorithm for finding overlap area in the case of zero intersection points can also be used when there is a single intersection point.

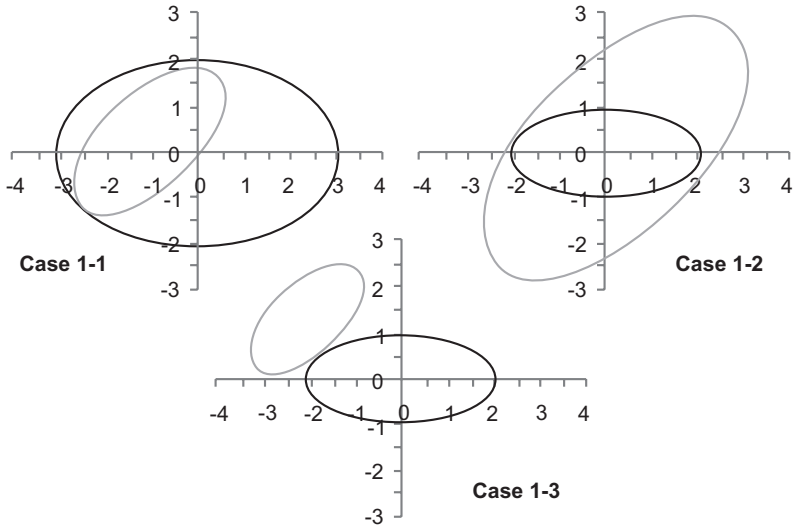
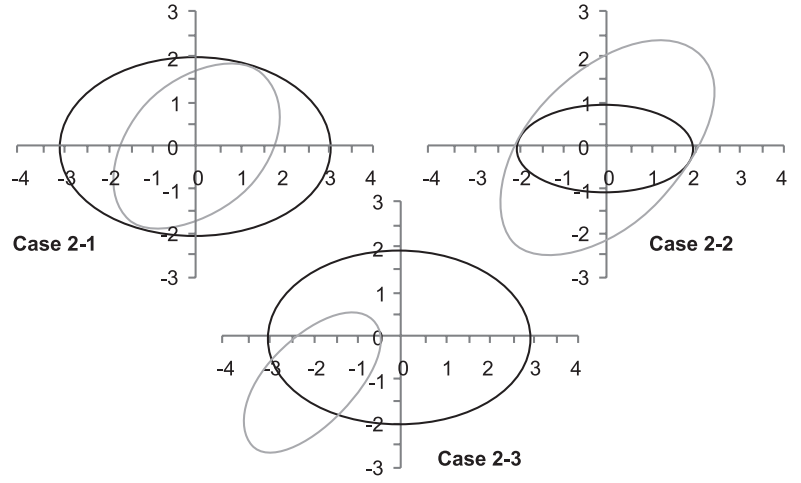


Figure 5.7: When two intersection points exist, either both of the points are tangents, or the ellipse curves cross at both points. For two tangent points, one ellipse must be fully contained within the other. For two crossing points, a partial overlap must exist.



Each sub-case for two intersection points requires a different overlap-area calculation. When there are two intersection points, if one point is a tangent, then both points must be tangents. And, if one point is not a tangent, then neither point is a tangent. It suffices to check one of the intersection points for tangency. Suppose the ellipses are tangent at an intersection point; then, points that lie along the first ellipse on either side of the intersection will lie in the same region of the second ellipse (inside or outside). That is, if two points are chosen that lie on the first ellipse, one on each side of the intersection, then both points will either be inside the second ellipse, or

Notes

they will both be outside the second ellipse. If the ellipse curves cross at the intersection point, then the two chosen points will be in different regions of the second ellipse.

A logic based on testing points that are adjacent to a tangent point can be implemented numerically to determine whether an intersection point is a tangent ora cross-point. Starting with an intersection point (x, y), calculate the parametric angle on the first ellipse, by the rules. A small perturbation angle can be calculated. For the method presented here, we seek to establish an angle that corresponds to a point on the first ellipse that is a given distance, approximately 2 · EPS, away from the intersection point:

$$EPS_{\text{Radian}} = \arcsin\left(\frac{2 \cdot EPS}{\sqrt{x^2 + y^2}}\right)$$

The angle EPS Radian is then used with the parametric form of the first ellipse (Equation (11)) to determine two adjacent points on each side of the intersection point (x, y):

$$x_1 = A_1 \cdot \cos(\theta + EPS_{\text{Radian}}),\ y_1 = B_1 \cdot \sin(\theta + EPS_{\text{Radian}})$$

$$x_2 = A_1 \cdot \cos(\theta + EPS_{\text{Radian}}),\ y_2 = B_1 \cdot \sin(\theta + EPS_{\text{Radian}})$$

Each adjacent point is then evaluated in the second ellipse equation (Equation (12)):

$$\text{test}_1 = AA \cdot x_i^2 + BB \cdot x_i \cdot y_i + CC \cdot y_i^2 + DD \cdot x_i + EE \cdot y_i + FF,\ i = 1, 2$$

If the ith test value is positive, then the adjacent point (x_i, y_i) is outside the second ellipse. It follows that the product of the two test-point evaluations test1·test2 will be positive if the intersection point is a tangent, since at a tangent point both test points will be on the same side of the ellipse. The product of the test-point evaluations will be negative if the two ellipse curves cross at the intersection point, since the test points will be on opposite sides of the ellipse. The function ISTANPT implements this logic to check whether an intersection point is a tangent or a cross-point; pseudo code is given.

When there are two intersection points, ISTANPT can be used to differentiate the case 2-3 (Figure 5.7) from the cases 2-1 and 2-2. Either of the two known intersection points can be checked with ISTANPT. If the intersection point is a tangent, then both of the intersection points must be tangents, so the case is either 2-1 or 2-2, anode ellipse must be fully contained within the other. For cases 2-1 and 2-2, the geometric logic used for 0 or 1 intersection points can also be used, i.e., NOINTPTS can be used to determine the overlap area. If the two ellipse curves cross at the tested intersection point, then the case must be 2-3, representing a partial overlap between the two ellipse areas.

For case 2-3, with partial overlie between the two ellipses, the approach for finding overlap area is based on using the two intersection points (x_1, y_1) and (x_2, y_2) with the segment area algorithm to determine the partial overlap area contributed by each ellipse. The total overlap area is the sum of two segment areas. The two intersection points divide each ellipse into two segment areas (See Figure 5.4). Only one segment area from each ellipse contributes to the overlap area. For the overlap area calculation, the two points must be passed to the segment algorithm in the order that will return the correct segment area. A check is made to determine whether the default order will return the desired segment area. First, the parametric angles θ_1 and θ_2 corresponding to (x_1, y_1) and (x_2, y_2) on the first ellipse are determined, by the rules Then, a point between (x_1, y_1) and (x_2, y_2) that is on the first ellipse is found:

$$x_{\text{mid}} = A_1 \cdot \cos\left(\frac{\theta_1 + \theta_2}{2}\right),\ y_{\text{mid}} = B_1 \cdot \sin\left(\frac{\theta_2 + \theta_2}{2}\right) \quad (23)$$

The point (x_{mid}, y_{mid}) is on the first ellipse between (x_1, y_1) and (x_2, y_2) when travelling counter-clockwise from (x_1, y_1) and (x_2, y_2). If (x_{mid}, y_{mid}) is inside the second ellipse, then the desired segment of the first ellipse contains the point (x_{mid}, y_{mid}). In this case, the segment algorithm

Notes

should integrate in the default order, counter clockwise from (x_1, y_1) to (x_2, y_2). Otherwise, the order of the points should be reversed before calling the segment algorithm. The area returned by the segment algorithm is the area contributed by the first ellipse to the partial overlap.

The desired segment area from the second ellipse is found in a way similar to the first ellipse segment. In the general case the second ellipse is in a displaced and rotated location. The approach presented here translates and rotates the second ellipse to the origin so that the segment algorithm can be used. It suffices to translate then rotate the two intersection points (x_1, y_1) and (x_2, y_2) by amounts that put the second ellipse centred at the origin and oriented with the coordinate axes:

$$x_{1_{TR}} = (x_1 - h_{2_{TR}}) \cdot \cos(\varphi_1 - \varphi_2) + (y_1 - k_{2_{TR}}) \cdot \sin(\varphi_2 - \varphi_1)$$

$$y_{1_{TR}} = (x_1 - h_{2_{TR}}) \cdot \sin(\varphi_1 - \varphi_2) + (y_1 - k_{2_{TR}}) \cdot \cos(\varphi_1 - \varphi_2)$$

$$x_{2_{TR}} = (x_2 - h_{2_{TR}}) \cdot \cos(\varphi_1 - \varphi_2) + (y_2 - k_{2_{TR}}) \cdot \sin(\varphi_2 - \varphi_1)$$

$$y_{2_{TR}} = (x_2 - h_{2_{TR}}) \cdot \sin(\varphi_1 - \varphi_2) + (y_2 - k_{2_{TR}}) \cdot \cos(\varphi_1 - \varphi_2)$$

The new points (x_1^{TR}, y_1^{TR}) and (x_2^{TR}, y_2^{TR}) lie on the second ellipse after it has been translated and rotated a second time to put its centre at the origin and its axes oriented with the coordinate axes. The new points can be used as inputs to the segment algorithm to determine the overlap area contributed by the second ellipse. The order of the points must be determined so that the segment algorithm returns the appropriate area. A check is made to determine whether this order will return the desired segment area. First, the parametric angles θ_1 and θ_2 corresponding to points (x_1^{TR}, y_1^{TR}) and (x_2^{TR}, y_2^{TR}) on the second ellipse are determined, by the rules. Then, a point (x_{mid}, y_{mid}) on the second ellipse between (x_1^{TR}, y_1^{TR}) and (x_2^{TR}, y_2^{TR}) is found, as in Equation (23). The point (x_{mid}, y_{mid}) is on the (twice translated + rotated) second ellipse between (x_1^{TR}, y_1^{TR}) and (x_{2TR}, y_{2TR}) when travelling counter- clockwise from (x_1^{TR}, y_1^{TR}) and $(x2^{TR}, y_2^{TR})$. To determine the desired segment of the second ellipse, the new point (x_{mid}, y_{mid}) must be rotated then translated back to a corresponding position on the once-translated + rotated second ellipse;

$$x_{\text{mid}_{RT}} = x_{\text{mid}} \cdot \cos(\varphi_2 - \varphi_1) + y_{\text{mid}} \cdot \sin(\varphi_1 - \varphi_2) + h_{2_{TR}}$$

$$y_{\text{mid}_{RT}} = x_{\text{mid}} \cdot \sin(\varphi_2 - \varphi_1) + y_{\text{mid}} \cdot \cos(\varphi_1 - \varphi_2) + k_{2_{TR}}$$

If $(x_{mid}^{RT}, y_{mid}^{RT})$ is inside the first ellipse, then the desired segment of the second ellipse contains the point (x_{mid}, y_{mid}), and the segment algorithm should integrate in the default order. Otherwise, the order of the points should be reversed before calling the segment algorithm, causing it to integrate counter clockwise from (x_2^{TR}, y_2^{TR}) to (x_1^{TR}, y_1^{TR}). The area returned by the segment algorithm is the area contributed by the second ellipse to the partial overlap. The sum of the segment areas from the two ellipses is then equal to the ellipse overlap area. The function TWOINTPTS calculates the overlap area for partial overlap with two intersection points (Case 2-3); pseudo-code is provided.

There are two possible sub-cases for three intersection points, shown in Figure 5.8. One of the three points must be a tangent point, and the ellipses must cross at the other two points. The cases are distinct only in the sense that the tangent point occurs with ellipse 2 on the interior side of ellipse 1 (Case 3-1), or with ellipse 2 on the exterior side of ellipse 1 (Case 3-2). The overlap area calculation is performed in the same manner for both cases, by calling TWOINTPTS with the two cross-point intersections. ISTANPT can be used to determine which point is a tangent; the remaining two intersection points are then passed to TWOINTPTS. This logic is implemented in the function THREEINTPTS, with pseudo-code given.

There is only one possible case for four intersection points, shown in Figure 5.8. The two ellipse curves must cross at all four of the intersection points, resulting in a partial overlap. The overlap

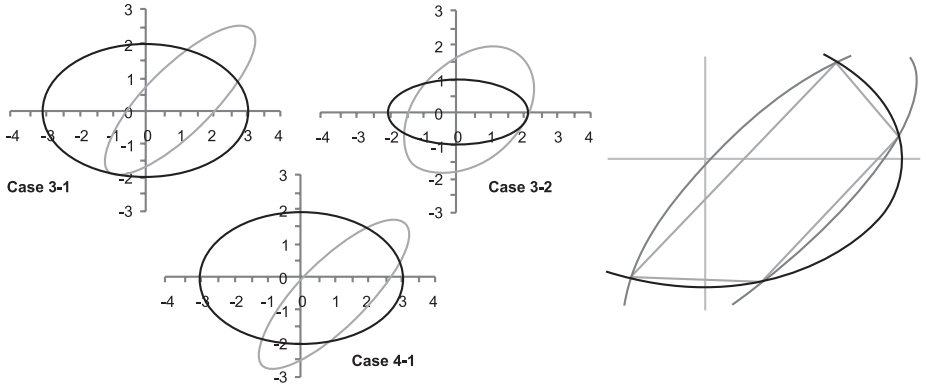
Notes

area consists of two segments from each ellipse, and a central convex quadrilateral. For the approach presented here, the four intersection points are sorted ascending in a counter-clockwise order around the first ellipse. The ordered set of intersection points is (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4). The ordering allows a direct calculation of the quadrilateral area. The standard formula uses the cross-product of the two diagonals:

$$\text{Area} = \frac{1}{2}\left|(x_3 - x_1, y_3 - y_1) \times (x_4 - x_2, y_4 - y_2)\right| = \frac{1}{2}\left|(x_3 - x_1) \cdot (y_4 - y_2) - (x_4 - x_2) - (x_3 - x_1)\right|$$

The point ordering also simplifies the search for the appropriate segments of each ellipse that contribute to the overlap area. Suppose that the first two sorted points (x_1, y_1) and (x_2, y_2) demarcate a segment of the first ellipse that contributes to the overlap area. It follows that contributing segments from the first ellipse are between(x_1, y_1) and (x_2, y_2), and also between (x_3, y_3) and (x_4, y_4). In this case, the contributing segments from the second ellipse are between (x_2, y_2) and (x_3, y_3), and between (x_4, y_4) and (x_1, y_1). To determine which segments contribute to the overlap area, it suffices to test whether a point midway between (x_1, y_1) and (x_2, y_2) is inside or outside the second ellipse. The segment algorithm is used for each of the four areas, and added to the quadrilateral to obtain the total overlap area. (See Figure 5.8)

Figure 5.8: When three intersection points exist, one must be a tangent, and the ellipse curves must cross at the other two points, always resulting in a partial overlap. When four intersection points exist, the ellipse curves must cross at all four points, resulting in a partial overlap consisting of two segments from each ellipse, and a central convex quadrilateral.



Self Assessment Questions

6. A (n) ……..is a circle whose centre is located on the circumference of another circle.

   (*a*) equant (*b*) deferent

   (*c*) ellipse (*d*) epicycle

7. The geometry of an ellipse is described by two numbers. The …….. which is half the longest diameter of the ellipse and the …….. which tells us the shape of the ellipse.

   (*a*) radius, eccentricity (*b*) radius, deferent

   (*c*) semi major axis, epicycle (*d*) semi major axis eccentricity

8. If the eccentricity is less than one then the conic is?

   (*a*) circle (*b*) parabola

   (*c*) ellipse (*d*) none of these

Notes

9. Engineering drawing commonly applies?

   (*a*) oblique projection (*b*) orthographic projection

   (*c*) perspective projection (*d*) None of above

10. …….. informal high-level description of the operating principle of a computer program or other algorithm.

    (*a*) Pseudo Code (*b*) binary code

    (*c*) ASSCII code (*d*) none of these

11. DDA stands for…

    (*a*) Discrete Differential Analyzer (*b*) Digital Differential Analyzer

    (*c*) Digital Data Analyzer (*d*) none of these

12. Midpoint ellipse algorithm is a method for drawing ellipses in computer graphics.

    (*a*) True (*b*) False

13. Ellipses are useful in many applied scenarios, and in widely disparate fields.

    (*a*) True (*b*) False

14. The advantage of midpoint modified method is that only addition operations are required in the program loops.

    (*a*) True (*b*) False

15. The Gauss-Green formula is used to determine the ellipse sector area between the two points, and a triangular area is added or subtracted to give the segment area.

    (*a*) True (*b*) False

5.5 Summary

- Ellipses are useful in many applied scenarios, and in widely disparate fields. In our research, we have encountered a common need for efficiently calculating the overlap area between two ellipses.
- Midpoint ellipse algorithm is a method for drawing ellipses in computer graphics. This method is modified from Bresenham's algorithm.
- The ellipse sector between two points (x_1, y_1) and (x_2, y_2) on the ellipse is the area that is swept out by a vector from the origin to the ellipse, beginning at the first point (x_1, y_1), as the vector travels along the ellipse in a counter-clockwise direction from (x_1, y_1) to (x_2, y_2).
- The ellipse is defined by specifying its semi-axes lengths, $A > 0$ and $B > 0$. The points are passed to the algorithm as (x_1, y_1) and (x_2, y_2), both of which must be on the ellipse.
- If the polynomial solver returns either two or four real roots to the quadratic polynomial of equation then the ellipse curves intersect.

5.6 Keywords

Algorithm: An algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

Notes

Ellipse: An ellipse is a plane curve that results from the intersection of a cone by a plane in a way that produces a closed curve.

Pseudo Code: It is an informal high-level description of the operating principle of a computer program or other algorithm.

Quadrilateral: A quadrilateral is a polygon with four sides (or edges) and four vertices or corners. Sometimes, the term quadrangle is used, by analogy with triangle.

Tangent: The tangent line to a plane curve at a given point is the straight line that "just touches" the curve at that point.

5.7 Review Questions

1. Define Bresenham's ellipse drawing algorithm.
2. Explain midpoint ellipse algorithm.
3. Define Ellipse area, sector area and segment area.
4. What is the core segment algorithm?
5. Define area for intersecting ellipses.
6. What is the ellipse segment?
7. Explain area for non-intersecting ellipses.
8. What is the segment area for a (directional) line through a general ellipse?
9. How can we do implementation of the algorithm?
10. Define ellipse-ellipse overlap area.

Answers for Self Assessment Questions

1.	(*a*)	2.	(*c*)	3.	(*d*)	4.	(*b*)	5.	(*b*)
6.	(*d*)	7.	(*d*)	8.	(*c*)	9.	(*b*)	10.	(*a*)
11.	(*b*)	12.	(*a*)	13.	(*a*)	14.	(*a*)	15.	(*a*)

5.8 Further Readings

Books *Implementation and Algorithms*, by Max K. Agoston

Ellipse Drawing Algorithms for VLSI Implementation, by Sifang Chu

Online link http://books.google.co.in/books?id=8OUzIgAACAAJ&dq=Implementing+Ellipse+Algorithm&source=bl&ots=hH-fhoSOpG&sig=C83UEQ0o77I9iQ0uxgYRXETy-AI&hl=en&sa=X&ei=5MQPUKrlAobUrQfe3oDIDw&ved=0CFEQ6AEwBg

Notes

Unit 6: Implementing Polygon Filling Algorithm

CONTENTS

Objectives

After studying this unit, you will be able to:

- Understand the concept of scan-line fill algorithm
- Explain boundary-fill algorithm
- Discuss about flood fill algorithm

Introduction

A polygon is a closed planar path composed of a finite number of chronological line segments. The straight line segments that make up the polygon are called its sides or edges and the points where the sides meet are the polygon's vertices. If a polygon is simple, then its sides (and vertices) constitute the boundary of a polygonal region, and the term polygon sometimes also describes the interior of the polygonal region (the open area that this path encloses) or the union of both the region and its boundary. An important issue that arises when filling polygons is that of deciding whether a particular point is interior or exterior to a polygon. A rule called the odd-parity (or the odd-even rule) is usually applied to test whether a point is interior or not. A half-line starting from the particular point and extending to infinity is drawn in any direction such that no polygon vertex intersects with the line. The point is considered to be interior if the number of intersections between the line and the polygon edges is odd.

Generation of solid areas from the simple edge of the vertex description is referred as solid scan conversion or polygon filling or contour filling. Given the edges defining a polygon, and a colour for the polygon, we need to fill all the pixels inside the polygon.

There are mainly three algorithms present for polygon filling:

1. Scan-line fill algorithm
2. Boundary-fill algorithm
3. Flood fill algorithm

6.1 Scan-line Fill Algorithm

Notes

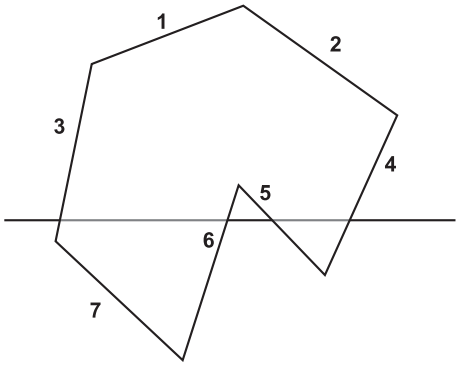
The scan line fill algorithm is an inventive way of satisfying in irregular polygons. The algorithm begins with a place of points. Each point is associated to the next, and the line between them is considered to be an edge of the polygon. The points of each edge are attuned to make sure that the point with the smaller y value appears first. Next, a data structure is created that contains a list of edges that begin on each scan line of the image. The program progresses from the first scan line upward. For each line, any pixels that contain an intersection between this scan line and an edge of the polygon are filled in. Then, the algorithm progresses along the scan line, turning on when it reaches a polygon pixel and turning off when it reaches another one, all the way across the scan line.

There are two special cases that are solved by this algorithm. First, a problem may happen if the polygon contains edges that are partially or completely out of the image. The algorithm solves this problem by moving pixel values that are outside the image to the boundaries of the image. This method is preferable to eliminating the pixel completely; because its deletion could result in a "backwards" colouring of the scan line i.e. pixels that should be on are off and vice versa.

The second case has to do with the concavity of the polygon. If the polygon has a concave portion, the algorithm will work correctly. The pixel on which the two edges meet will be marked twice, so that it is turned off and then on. If, however, the polygon is convex at the intersection of two edges, the colouring will turn on and then immediately off, resulting in "backwards" colouring for the rest of the scan line. The problem is solved by using the vertical location of the next point in the polygon to determine the concavity of the current portion. Overall, the algorithm is very robust. It turns out that the only difficulty comes with polygons that have large amounts of edges, like circles and ellipses. Filling in such a polygon would be very costly.

For example see Figure 6.1

Figure 6.1: Horizontal Scanning of the Polygon



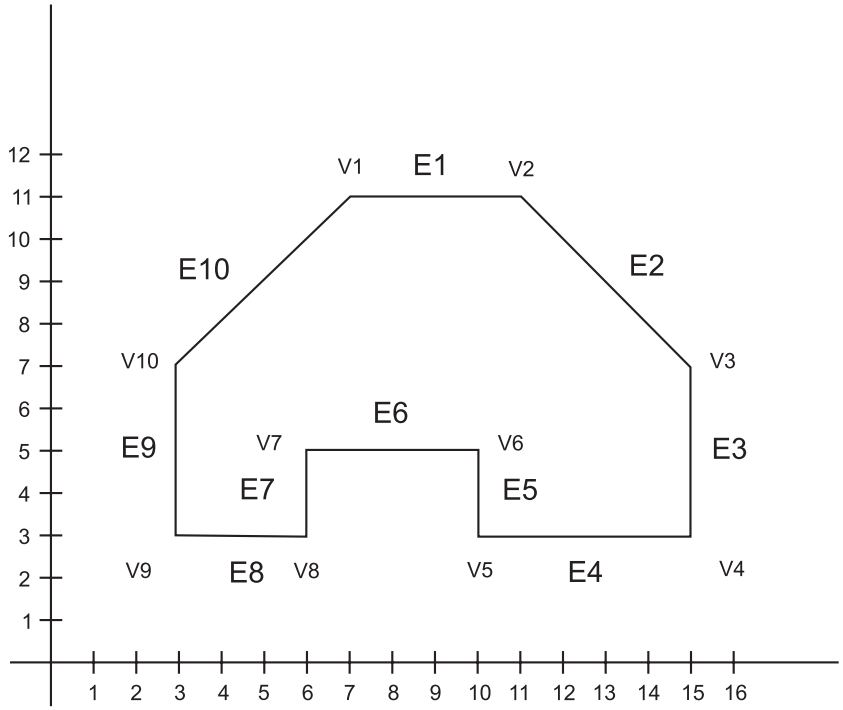
The sequence of edges sorted by their smallest y-coordinate (assume the y-axis goes from top to bottom) would be 1234567. The edges intersected by the scan line (the so-called active edges) are 3654. Calculating these four intersections and sorting them by their x-coordinates give two line segments 3-6 and 5-4 which can easily be drawn. Afterwards the scan line would be shifted down one pixel etc.

For each horizontal scan-line:

1. List all the points that intersect with the horizontal scan-line
2. Sort the intersection points in ascending order of the x coordinate
3. Fill in all the interior pixels between pairs of successive intersections

Notes

Figure 6.2: A Non-simple Polygon



The third step accepts a sorted list of points and connects them according to the odd-party rule. For example, given the list [p_1; p_2; p_3; p_4… p_{2n-1}; p_{2n}], it draws the lines $p_1 \rightarrow p_2$; $p_3 \rightarrow p_4$; …. ; $p_{2n-1} \rightarrow p_{2n}$. A decision must be taken as to whether the edges should be displayed or not: given that p_1= (x_1, y) and p_2 = (x_2, y), should we display the line (x_1, y, x_2, y) or just the interior points (x_1 + 1, y, x_2 – 1, y).

Example:

Consider the polygon in Figure 6.2. The edge Table 6.1 for such a polygon would be:

Table 6.1: Edge Table

Edge	Y_{min}	Y_{max}	X of Y_{min}	X of Y_{max}	$\frac{1}{m}$
E1	11	11	6	11	0
E2	7	11	15	11	–1
E3	3	7	15	15	0
E4	3	3	10	15	–
E5	3	5	10	10	0
E6	5	5	10	6	–
E7	3	5	6	6	0
E8	3	3	3	6	–
E9	3	7	3	3	0
E10	7	11	3	7	1

Notes

The edge list for such a polygon, for each of the scan-lines from 3 to 11 is:

Table 6.2: Edge List

Scan-line	Edge number
11	–
10	–
9	–
8	–
7	2, 10
6	–
5	–
4	–
3	3, 5, 7, 9

Note that in the Table 6.2 the horizontal lines are not added to the edge list. The active edges for scan-line 3 would be 3, 5, 7, 9; these are sorted in order of their x values, in this case 9, 7, 5, 3. The polygon fill routine would proceed to fill the intersections between (3, 3) (E9) and (6, 3) (E7) and (10, 3) (E5) to (15, 3) (E3). The next scan-line (4) is calculated in the same manner. In this the values of x do not change (since the line is vertical; it is incremented by 0). The active edge at scan-line 7 is 10 and 2 (correct order).

The sorting of the edges is not explicitly necessary: for each scan line position you could try to intersect it with all polygon edges which are not that effective. The sorting allows us to easily keep a list of active edges:

If the scan line moves over the endpoint of an edge with the smaller y-coordinate it is added to the list; if the scan line moves over the endpoint of an edge with the bigger y-coordinate it is deleted from the list.

6.1.1 Basic Scan-fill Algorithm

1. For each non-horizontal edge of the polygon boundary recognizes the upper and lower endpoints, (x_l , y_l) and (x_u , y_u), such that $y_u > y_l$, and creates a record for each that contains
   - Y_u, the y-coordinate at the upper endpoint.
   - $x = x_l$, the current x-intersection.
   - $w = 1/m = (x_u – x_l)/ (y_u – y_l)$, the reciprocal of the slope of the edge.
2. Set the AET (the active edge table) to be empty.
3. Apply a bucket sort algorithm to sort the edges using the y_l as the primary key, and x_l as the secondary, and w as the tertiary. N.B., Each bucket contains a list. The set of buckets is called the ET (edge table):
4. Set y equal to the smallest index in the ET that has a non empty bucket.
5. Repeat until the ET and the AET are empty:
   - Move any edges from bucket y in the ET to the AET.
   - Remove any edges from the AET that have a y_u equal to y.
   - Sort the AET according to x.

Notes

- Fill in the requisite pixels between the even and odd adjacent pairs of intersections in the AET: round up, $\lceil x \rceil$ the x-coordinate of "left" intersections, round down, $\lceil x - 1 \rceil$ that of the "right" intersections.
- Increment y by one.

6. Update $x \leftarrow x = w$ for every non vertical edge in the AET.

6.2 Boundary-fill Algorithm

This creates apply of coherence properties of the boundary of a primitive/figure: given a point inside the region the algorithm recursively plots the adjacent pixels until the primitive boundary is reached.

Given the FillColour, the BoundaryColour and a point inside the boundary, the following algorithm recursively sets the four adjacent pixels (2 horizontal and 2 vertical) to the FillColour.

```
Procedure BoundaryFill (x, y: Integer, FillColour, BoundaryColour:
ColourType);
Procedure BFill (x, y: Integer);
Var
CurrentColour: ColourType;
Begin
CurrentColour: = GetPixel (x, y);
If (CurrentColour < > FillColour) and
(CurrentColour < > BoundaryColour)
then
SetPixel (x, y, FillColour);
BFill (x + 1, y);
BFill (x – 1, y);
BFill (x, y + 1);
BFill (x, y – 1)
End;
Begin
BFill (x, y)
End;
```

Regions which can be totally filled with this algorithm are called 4-connected regions. Some regions cannot be filled by this algorithm. Such regions are called 8-connected and algorithms filling such areas consider the four diagonally adjacent pixels as well as the horizontal and vertical ones. The 8-connected algorithm is particularly susceptible.

```
Procedure BFill8 (x, y: Integer);
Var
CurrentColour: ColourType;
Begin
CurrentColour: = GetPixel (x, y);
If (CurrentColour < > FillColour) and
(CurrentColour < > BoundaryColour)
then
SetPixel (x, y, FillColour);
```

Notes

```
BFill8 (x + 1, y);
BFill8 (x – 1, y);
BFill8 (x, y + 1);
BFill8 (x, y – 1);
BFill8 (x + 1, y + 1);
BFill8 (x + 1, y – 1);
BFill8 (x – 1, y + 1);
BFill8 (x – 1, y – 1)
End;
```

Caution

Care must be taken to ensure that the boundary does not contain holes, which will cause the fill to 'leak'.

Self Assessment Questions

1. …….. is a not algorithm present for polygon filling.

   (*a*) Scan-line fill algorithm (*b*) Boundary-fill algorithm

   (*c*) Flood fill algorithm (*d*) Bucket sort algorithm

2. Scan line fill algorithm is two special cases that are solved by this algorithm.

   (*a*) two (*b*) three

   (*c*) four (*d*) five

3. The ……. step accepts a sorted list of points and connects them according to the odd-party rule.

   (*a*) first (*b*) second

   (*c*) third (*d*) forth

4. The ……..algorithm is particularly vulnerable.

   (*a*) 2-connected (*b*) 4-connected

   (*c*) 6-connected (*d*) 8-connected

5. Apply a bucket sort algorithm to sort the edges using the yl as the primary key, and xl as the secondary, and w as the tertiary.

   (*a*) True (*b*) False

6.3 Flood Fill Algorithm

Flood fill, also named seed fill, is an algorithm that decides the area attached to a given node in a multi-dimensional array. The flood-fill algorithm is used to fill a region which has the same color and whose boundary may have extra one color.

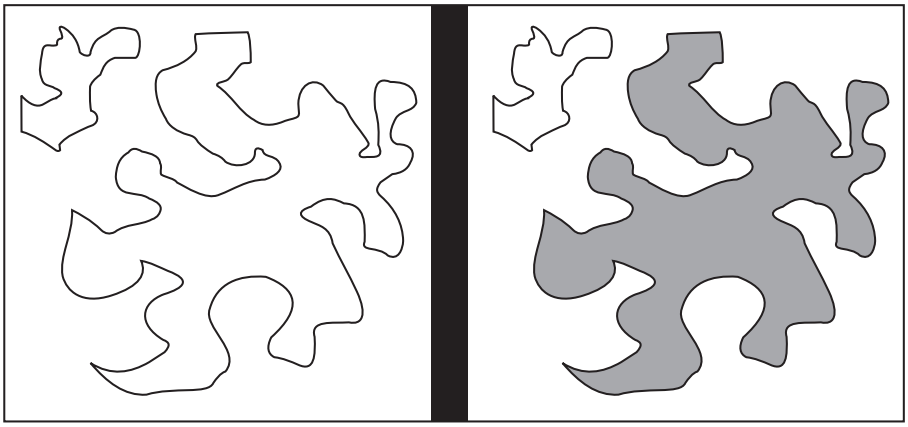
The seed fill or flood fill algorithm needs one of the pixel positions within the polygon or region to be filled. This pixel is inside the region and set to polygon value. This pixel is referred as seed pixel. The algorithm continues to search the pixels which are adjacent to the current seed pixel to see whether they are boundary pixels or the pixels inside the region. All the pixels interior to the region and are adjacent to the seed pixel are set to the polygon value.

The purpose of Flood Fill is to colour an entire area of connected pixels with the same colour. It is the bucket tool in many painting programs. Here is an example: the original image is on

Notes

the left. Then a flood fill was started inside the large shape, and the algorithm gave all pixels inside the shape the new colour, leaving its borders and the pixels outside intact.

Figure 6.3: Example of Flood Fill



Algorithm

Seed(x, y) is the seed pixel
Push is a function for placing a pixel on the stack
Pop is a function for removing a pixel from the stack
Pixel(x, y) = seed(x, y)
Initialize stack
Push pixel(x, y)
While (stack not empty)
get a pixel from the stack
Pop Pixel (x, y)
If Pixel(x, y) < > New value then
Pixel(x, y) = New value
Else
If (Pixel(x + 1, y) < > New value and
Pixel(x+1, y) < > Boundary value then
Push Pixel(x+1, y)
If Pixel(x, y+1) < > New value and
Pixel(x, y+1) < > Boundary value then
Push Pixel(x, y+1)
If Pixel(x – 1, y) < > New value and
Pixel(x+1, y) < > Boundary value then
Push Pixel(x-1,y)
If Pixel(x, y-1) < > New value and
Pixel(x, y-1) < > Boundary value then
Push pixel(x, y-1)
End if
End while

C Program to Fill Any Given Polygon Using Flood Filling Algorithm

Notes

```
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct Node
{
 int x;
 int y;
 struct Node* next;
};
void fill (int pt[][2], int clr);
void floodfill4 (int x, int y, int oldclr, int newclr);
void insert (int x, int y, struct Node** last);
void main()
{
 int i, j;
 int pt[3][2];
 int clr;
 printf ("This program demonstrates filling a polygon.\n");
 printf ("Enter the x- and y-coordinates for three points:\n");
 for (i = 0; i < 3; i++)
    for (j=0; j<2; j++)
     scanf ("%d", &pt[i][j]);
 printf ("Enter the fill-colour: (Any number from 1 to 14)");
 scanf ("%d", &clr);
 fill (pt, clr);
}
void fill (int pt[][2], int clr)
{
 int gd = DETECT, gm;
 int seedx, seedy;
 initgraph (&gd, &gm, "c\\tc\\bgi");
 setcolor (WHITE);
 line (pt[0][0], pt[0][1], pt[1][0], pt[1][1]);
 line (pt[1][0], pt[1][1], pt[2][0], pt[2][1]);
 line (pt[2][0], pt[2][1], pt[0][0], pt[0][1]);
 getch();
 seedx = (pt[0][0] + pt[1][0] + pt[2][0])/3;
 seedy = (pt[0][1] + pt[1][1] + pt[2][1])/3;
 floodfill4 (seedx, seedy, BLACK, clr);
 getch();
```

Notes

```
 closegraph();
 return;
}
void floodfill4 (int x, int y, int oldclr, int newclr)
{
 struct Node* first, *last, *tmp;
 first = (struct Node*) malloc (sizeof (struct Node));
 if (first == NULL)
 {
    closegraph();
    fprintf (stderr, "floodfill4: Out of memory.\n");
    exit (2);
 }
 if (oldclr == newclr)
 {
    free (first);
    return;
 }
 first->x = x;
 first->y = y;
 first->next = NULL;
 last = first;
 while (first != NULL)
 {
    putpixel (x, y, newclr);
    if (getpixel (x, y–1) == oldclr)
    {
     putpixel (x, y–1, newclr);
     insert (x, y–1, &last);
    }
    if (getpixel (x, y+1) == oldclr)
    {
     putpixel (x, y+1, newclr);
     insert (x, y+1, &last);
    }
    if (getpixel (x–1, y) == oldclr)
    {
     putpixel (x–1, y, newclr);
     insert (x–1, y, &last);
    }
    if (getpixel (x+1, y) == oldclr)
    {
```

Notes

```
        putpixel (x+1, y, newclr);
        insert (x+1, y, &last);
      }
      tmp = first;
      first = first->next;
      x = first->x;
      y = first->y;
      free (tmp);
   }
  }
  void insert (int x, int y, struct Node** last)
  {
   struct Node* p;
   p = (struct Node*) malloc (sizeof (struct Node));
   if (p == NULL)
   {
      closegraph();
      fprintf (stderr, "\n insert: Out of memory.\n");
      exit (2);
   }
   p->x = x;
   p->y = y;
   p->next = NULL;
   (*last)->next = p;
   *last = (*last)->next;
  }
```

Did u know?

Flood fill algorithm was first available commercially in 1981 on a Vicom Image Processing system manufactured by Vicom Systems.

Self Assessment Questions

6. A polygon consists of a finite number of …….. segments.
7. The line segment of the polygon is called ……...
8. The ……. is usually applied to test whether a point is interior or not.
9. The point is considered to be …….. if the number of intersections between the line and the polygon edges is odd.
10. Regions which can be completely filled with boundary-fill algorithm are called …… regions.
11. A polygon which has a concave portion, the ……. algorithm will work correctly.
12. The scan line fill algorithm is an ingenious way of filling …….. polygons.
13. The scan line algorithm solves the outer edge problem by …… from outside the image.

Notes

14. Some 4-connected regions cannot be filled using boundary-fill algorithm such regions are called ……..
15. Flood fill algorithm determines the area connected to a given node in a ……...

6.4 Summary

- A polygon is a closed planar path composed of a finite number of sequential line segments.
- An important issue that arises when filling polygons is that of deciding whether a particular point is interior or exterior to a polygon.
- The scan line fill algorithm is an ingenious way of filling in irregular polygons this algorithm begins with a set of points.
- Boundary-fill Algorithm makes use of coherence properties of the boundary of a point inside the region the algorithm recursively plots the surrounding pixels until the primitive boundary is reached.
- Flood fill, is an algorithm that determines the area connected to a given node in a multi-dimensional array.

6.5 Keywords

Array: It is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.

Data Structure: It is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Edge: The line between two points is considered to be an edge of the polygon.

Pixel: It is a physical point in a raster image or the smallest, addressable element in a display device; so it is the smallest, controllable element of a picture represented on the screen.

Polygon: It is traditionally a plane figure that is bounded by a closed path, composed of a finite sequence of straight line segments.

6.6 Review Questions

1. What is scan line algorithm?
2. How will you generate a polygon?
3. Which algorithms can be used to generate a polygon?
4. Explain the seed fill algorithm.
5. Which are the two conditions are required before setting the pixel?
6. Write down the method of implementing the scan line algorithm.
7. Write a program to implementing the flood fill algorithm.
8. Write a program to implementing the scan line algorithm.
9. What is boundary-fill algorithm? Explain in detail.
10. Write a program to implementing the boundary-fill algorithm.

Notes

Answers for Self Assessment Questions

1. (*d*) 2. (*a*) 3. (*c*) 4. (*d*) 5. (*a*)
6. Sequential line 7. Edges 8. Odd-parity
9. Interior 10. 4-connected 11. Scan-line fills
12. Irregular 13. Moving pixel values 14. 8-connected
15. Multi-dimensional array

6.7 Further Readings

Books *Computer Graphics: Principles and Practice*, by James D. Foley

Geometric Tools for Computer Graphics, by Philip J. Schneider, David H. Eberly

Online link http://www.dgp.toronto.edu/~hertzman/418notes.pdf

Notes

Unit 7: Implementation of Hidden Surface in 2D

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss visibility algorithm categorization
- Define the hidden surface removal algorithms

Introduction

The term rendering has many meanings in dissimilar perspectives. In computer graphics, rendering is the process of creating an image from a model. The model holds the essential information concerning the geometry and position of the objects in the scene, the visual attributes of the objects such as colour and reflection properties, and the needed lighting and screening information. In the early years of computer graphics, much attention was spent on how to perform rendering efficiently. The relatively slow processing speeds of early processors meant the amount of scene complexity that could be rendered in a reasonable amount of time was highly dependent on the speed of the rendering algorithm. When producing 30 frame-per-second animations, even moderate savings in individual frame rendering time was significant. In today's high-speed multi-core processor machines, speed and efficiency are not as critical, and complex models can be easily processed on desktop machines. With the advent of hundreds of specialized Graphics Processing Units on a display board, it is common to render high quality images of complex environments in real time for applications such as gaming and simulations.

Rendering can be divided into two basic processes:

1. Determining what surface, or more precisely point on a surface, is visible at each pixel.
2. Shading the point based on surface properties, lighting conditions, viewing parameters, and other elements in the scene.

Notes

One of the fundamental tasks of a renderer is to explain the visibility, or concealed surface, problem. That is, given a set of objects, say a list of polygons, objects that are in front of other objects from the point of the view of the spectator will obscure the scene behind them. While we normally think of this as a 3D problem, it applies to 2D animations as well which is sometimes known as 2.5D drawing. Picture a 2D cartoon in which one character is moving in front of another. If a computer is rendering the image, it needs to know which 2D object to draw in front of another. Objects can be assigned a priority to specify the ordering of drawing, which is essentially assigning a pseudo-depth to the objects. Different algorithms have been developed over the years to accomplish the hidden surface task for 3D scenes. .

In displaying objects, there regularly occurs overlapping, meaning that the objects which are closer are visible, and those behind them are non-visible, because they appear as hidden behind them. There have been planed many, rather complex algorithms for solving this issue. Their goal is to find out objects whose parts are visible only from the observer place. Sometimes it is necessary to draw also edges of overlapped objects, most often by dash lines. The visibility algorithms always depend on the fact of how the objects are represented in an area. The optimal variant is to have objects described using their limits, best by surfaces. Most of these algorithms too are prepared for this sort of representation of objects. The process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint. A hidden surface determination algorithm is a solution to the visibility problem, which was one of the first major problems in the field of 3D computer graphics. The process of hidden surface determination is sometimes called hiding, and such an algorithm is sometimes called a hider. The analogue for line rendering is hidden line removal. Hidden surface determination is necessary to render an image correctly, so that one cannot look through walls in virtual reality.

Some Basic Point of Hidden Surfaces

- Construct a 3D hierarchical geometric model
- Define a virtual camera
- Map points in 3D space to points in an image
- Produce a wireframe drawing in 2D from 3D object Of course, there's more work to be done
- Not every part of every 3D object is visible to a particular viewer. We need an algorithm to determine what parts of each object should get drawn
- Known as "hidden surface elimination" or "visible surface determination"
- Hidden surface elimination algorithms can be categorized in three major ways:
    - Object space vs. image space
    - Object order vs. image order
    - Sort first vs. sort last

Hidden surface (line) removal is the process of removing surfaces (lines) which are not visible from the chosen viewing position. Hidden surface/line algorithms are often classified as one of the following:

- Object space: algorithms which work on object definitions directly;
- Image space: algorithms which work on the projected image.

There is no one place in the viewing pipeline where these algorithms should be used the ideal position in the pipeline is often specific to the particular algorithm. Some algorithms do not perform hidden surface/line-removal completely, but are designed to do some of the work early on in the pipeline, thereby reducing the amount of surfaces to be processed.

Object Space Algorithms

Operate on geometric primitives

- For each object in the scene, compute the part of it which is not obscured by any other object, then draw

Notes

- Must perform tests at high precision
- Resulting information is resolution-independent

Complexity

- Must compare every pair of objects, so O(n2) for n objects
- For an mxm display, have to fill in colours for m2 pixels.
- Overall complexity can be O (k_{obj} n^2 + k_{disp} $dispm^2$).
- Best for scenes with few polygons or resolution-independent output

Implementation

- Difficult to implement!
- Must carefully control numerical error

Object Order vs. Image Order

Object order

- Consider each object only once - draw its pixels and move on to the next object
- Might draw the same pixel multiple times

Image order

- Consider each pixel only once - draw part of an object and move on to the next pixel
- Might compute relationships between objects multiple times

7.1 Visibility Algorithm Categorization

Sutherland, Sproull and Schumacker termed the visibility algorithms according to their approaches and principles used. They are either image area or object area types. According to the visibility algorithms can be split depending on the form of output data.

a. Line Algorithm *(Hidden line eliminator)*

We receive a set of abscissas representing visible edges. The advantage of this solution is the fact that we have a set of vectors, and we are not limited due to issues of resolution. In this case, we can scale the scene without any loss in quality.

b. Raster Algorithm *(Hidden surface eliminator)*

We receive a set of pixels representing the specific points of the scene in the fixed given resolution, without any option to scale the scene. Here it is, however, possible to draw also surfaces with shading and lighting.

Pre-processing of Data

It is time demanding to compute all algorithms; at the beginning it is necessary to examine the object in the view. This is called data pre-processing. First, all objects outside the visual angle, are excluded and all polygons of which the scalar vector product (of the normal vector of the polygon and the direction of view) is positive, and the given polygon is then reversed.

7.1.1 Line Visibility Algorithms

Robertson algorithm one of the first algorithms is the Robertson algorithm; originally aimed for the solution of visibility of a group of convex polyhedrons. This algorithm subsequently processes individual edges of the body. It tests every edge, whether it is not overlapped by another body. If it is hidden only partially, then this edge is split into two parts (one hidden, the other unhidden). Every unhidden part becomes a new tested edge.

7.1.2 Apple Algorithm

This algorithm works in the area of objects and is based on the fact that every edge is a conjunction of two polygons. The first step is to remove invisible polygons. First, we split polygons into

Notes

two groups: near (potentially visible) and reverse (invisible). The near polygons have an obtuse angle with the projection direction; and the reverse ones have an acute angle with the projection direction. In the following step, we split the edges into three types, according to which polygons they cross:

First type: A conjunction of two reverse polygons, called the invisible edge.

Second type: A conjunction of the near polygon and the reverse polygon, called the visible contour edge.

Third type: A conjunction between two near polygons, called the potential visible edge.

Did u know?

Newell's Algorithm is a 3D computer graphics procedure for elimination of polygon cycles in the depth sorting required in hidden surface removal. It was proposed in 1972 by brothers Martin Newell and Dick Newell, and Tom Sancha, while all three were working at CAD Centre.

Self Assessment Questions

1. The process of hidden surface determination is sometimes called hiding.
   (*a*) hider (*b*) hiding
   (*c*) hidden surface elimination (*d*) visible surface determination
2. Hidden surface removal is the process of removing surfaces (lines) which are not visible from the chosen viewing position.
   (*a*) True (*b*) False
3. According to the …… can be split depending on the form of output data.
   (*a*) line algorithm (*b*) raster algorithm
   (*c*) apple algorithm (*d*) visibility algorithms
4. ……. works in the area of objects and is based on the fact that every edge is a conjunction of two polygons.
   (*a*) line algorithm (*b*) raster algorithm
   (*c*) apple algorithm (*d*) visibility algorithms
5. Second type is a conjunction between two near polygons, called the potential visible edge.
   (*a*) True (*b*) False

7.2 Hidden Surface Removal Algorithms

Considering the rendering pipeline, the projection, the clipping, and the rasterization steps are handled differently by the following algorithms:

1. Z-buffering
2. Painter's algorithm
3. Binary space partitioning (BSP)
4. Ray tracing
5. The Warnock algorithm

7.2.1 Z-buffering

The Z-buffer or depth memory is one of the most common methods for solving the problem of an objects' visibility in a scene. It has relatively large memory demands because in displaying objects it will keep their z-coordinates in a special two-dimensional field with the dimensions of a table (mostly a display box), where there are stored the coordinates of calculated points. At the beginning, this field is filled in with a maximum value (e.g., maximum z-value of all objects in the display box).

Notes

In displaying in a display box we proceed so that we transfer every object into a raster form. In this way, we obtain points with the coordinates x, y a z. Because the display box is only two-dimensional, the objects which are overlapped should not be visible. The Z-buffer serves also to avoid the situation where more distant objects are drawn last and in this way overlap objects standing closer to the observer.

We put into the field all z-coordinates of the relevant points of the given objects. If during the test of a new point we found out that in the field there is a different value than the starting, then, providing the new value is lower.

A disadvantage of the Z-buffer is its memory demand. The line scanning algorithm reduces memory demands of storage as here it is necessary to store only a field equal to one horizontal line of the screen. This method is based on the decomposition of a polygon into raster form. Here, however, it is necessary to carry out the raster decomposition several times. For each row of the display box we test every surface in a scene, using the given field as a depth memory. An example of the algorithm as described.

Algorithm

- For every image point (x, y) we initialize the Z-buffer to the value of max as well as the bitmap of background.
- For every surface we assign in the project a polygon; and carry out its decomposition into a raster form, while for every image point of polygon (x, y) we assess a z-coordinate.
- If the distance of a given pixel (x, y) is lower than the Z-buffer of the relevant pixel, then, we put the value of the given point into the Z-buffer and the colour of the point into the bitmap

The order of processed polygons can be any, with every surface being processed only once. This method is hardware implemented in all 3D cards (depth of 16, 24, or 32 bits).

Visibility tricks for Z-buffers

- Z-buffering is the algorithm of choice for hardware rendering
- What is the complexity of the Z-buffer algorithm?
- What can we do to decrease the constants?

Z-buffer Analysis

Categorization:

- Easy to implement.
- Hardware implementation.
- Memory intensive.
- Incremental drawing calculations (uses coherence).
- Pre-processing required.
- On-line (does not need all objects before drawing begins).
- Handles transparency.
- Handles refraction.
- Polygon-based.
- Extra work for moving objects.

Notes

- Extra work for moving viewer.
- Efficient shading.
- Handles cycles and self-intersections.

7.2.2 Painter's Algorithm

This algorithm is based on the idea of the repeated drawing of all surfaces, from the most secluded up to the polygons placed close to the table; similarly, as a painter over lays gradually individual layers of surfaces. The surface drawn later will overlie the surface below it. The visibility solution is then carried out during the arrangement of surfaces based on the distance of the observer. Because the surfaces are 3D objects, it is not possible to arrange definitely according to the distances. Therefore, we proceed as follows: firstly, we arrange, meaning the indexing of surfaces according to the lowest z-coordinate of the relevant surface. This series cannot be drawn directly because there can appear a case when a surface with a lower z-coordinate would overlap another surface with a larger *z*-coordinate. The painter algorithm is suitable for more simple scenes with limited conjunctions of objects, due to its low calculation and memory demands.

After indexing all surfaces, we can start the method of drawing itself. Firstly, the surface from a list is marked as active, and then, we test it with regard to overlapping with other surfaces.

Displaying Spatial Graphs

Special displaying method on drawing spatial graphs, where visibility is determined by the floating horizon method. The algorithm uses two auxiliary fields (the upper horizon HOR, and the bottom horizon DOL) with the dimensions equal to the number of pixels of the table in the horizontal direction.

Algorithm

1. Initialize HOR by the value plus infinity.
2. Initialize DOL by the value minus infinity.
3. For every y from <y min, y max>
4. For every x from <x min, x max>
5. To calculate transformation of projection (x, y, z) to <xi, yi>
6. If y_i > HOR[x_i] then draw the relevant point in the table and HOR[xi]=yi
7. If y_i < DOL[y_i] then draw the relevant point in the table and HOR[xi]=yi

7.2.3 Binary Space Partitioning (BSP)

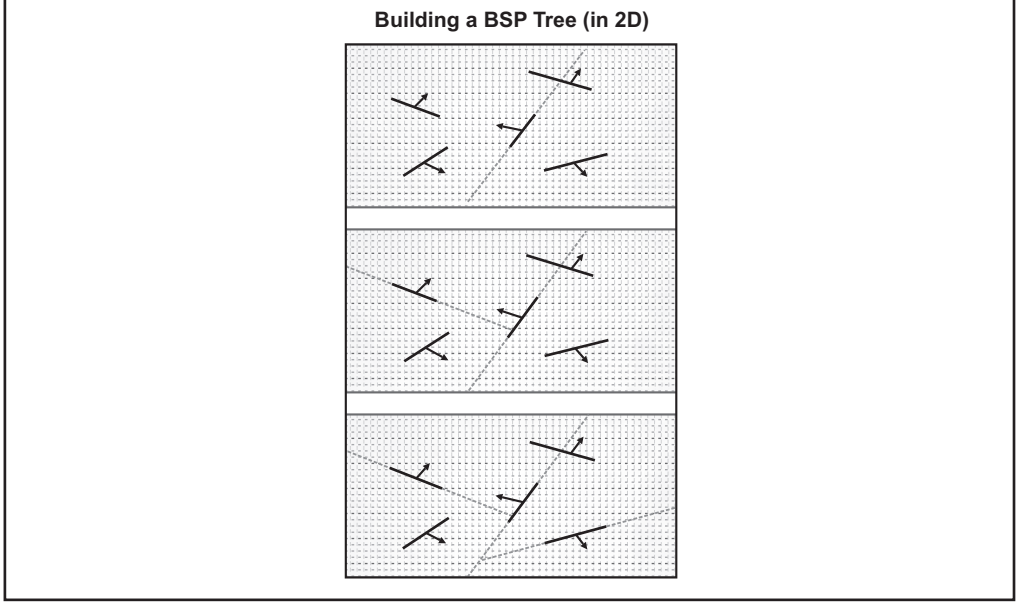
Binary space partition trees (BSP trees) are extremely helpful for people who work with 2D and 3D computer graphics. A BSP tree works by dividing up and organizing the parts of a game scene and then sorting the parts of the sight into a useful form from which information on how the parts relate to each other can be quickly accessed.

Building a BSP Tree (in 2D)

```
BSP Tree Construction
BSP tree make BSP( L: list of polygons )
{
if L is empty
{
return the empty tree
}
```

Notes

Figure 7.1: Building a BSP Tree (in 2D)



Choose a polygon P from L to serve as root

Split all polygons in L according to P

return new Tree Node(

P,

Make BSP(polygons on negative side of P),

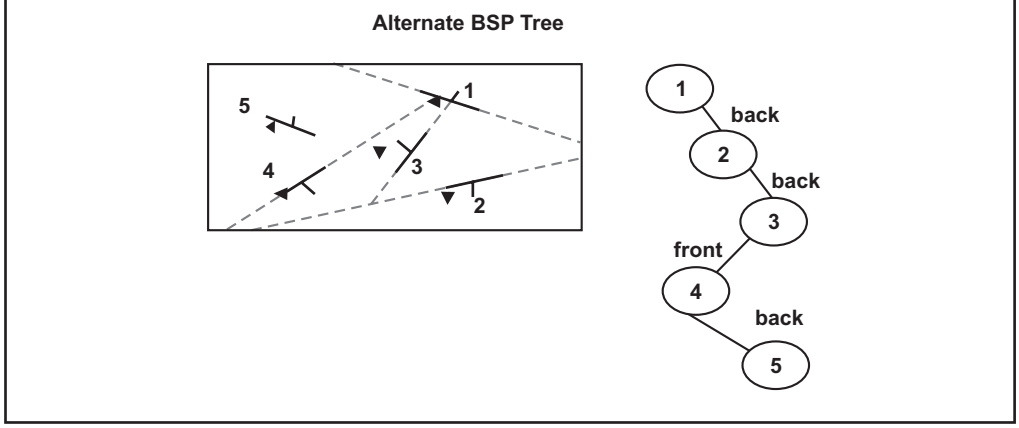
Make BSP(polygons on positive side of P))

}

Splitting polygons is expensive! It helps to choose P wisely at each step.

Example: choose five candidates; keep the one that splits the fewest polygons

Figure 7.2: Alternate BSP Tree



BSP Tree Display **Notes**

```
showBSP( v: Viewer, T: BSPtree )
{
if T is empty then return
P := root of T
if viewer is in front of P
{
showBSP( back subtree of T )
draw P
showBSP( front subtree of T )
} else {
showBSP( front subtree of T )
draw P
showBSP( back subtree of T )
}
}
```

BSP Tree Analysis

Categorization:

- Easy to implement
- Hardware implementation
- Incremental drawing calculations (uses coherence)
- Pre-processing required
- On-line (does not need all objects before drawing begins)
- Memory intensive
- Handles transparency
- Handles refraction
- Polygon-based
- Extra work for moving objects
- Extra work for moving viewer
- Efficient shading
- Handles cycles and self-intersections


Did u know?

Binary Space Partition Trees (or BSP trees for short) where introduced by Fuchs, Kedem, and Naylor around 1980.

7.2.4 Ray Tracing

Ray tracing is, in many ways, a standardized technique. Almost all ray tracing programs (known as "ray tracers") use the same basic algorithm in one way or the other. This algorithm is presented below.

Notes

The Pre-rendering Section

This section is executed before the actual rendering of the scene takes place. It is the most implementation and programmer dependent of all the sections of the algorithm. Its structure and content relies entirely on how the programmer has chosen to describe and store his scene and what variables need to be initialized for the program to run properly. Some parts that are usually present in some form or the other include generation of efficiency structures, initialization of the output device, setting up various global variables etc. No standard algorithm exists for this section.

The Main Rendering Loop

```
Procedure RenderPicture()
    For each pixel on the screen,
        Generate a ray R from the viewing position through the point on the view
        plane corresponding to this pixel.
        Call the procedure RayTrace() with the arguments R and 0
        Plot the pixel in the colour value returned by RayTrace()
    Next pixel
End Procedure
Procedure RayTrace(ray R, integer Depth) returns colour
    Set the numerical variable Dis to a maximum value
    Set the object pointer Obj to null
    For each object in the scene
        Calculate the distance (from the starting point of R) of the nearest
        intersection of R with the object in the forward direction
    If this distance is less than Dis
        Update Dis to this distance
        Set Obj to point to this object
    End if
    Next object
    If Obj is not null
        Set the position variable Pt to the nearest intersection point of R and Obj
        Set the total colour C to black
        For each light source in the scene
            For each object in the scene
            If this object blocks the light coming from the light source to Pt
            Attenuate the intensity of the received light by the transmittivity
                of the object
            End if
            Next object
            Calculate the perceived colour of Obj at Pt due to this light source
                using the value of the attenuated light intensity
            Add this colour value to C
        Next light source
```

Notes

```
    If Depth is less than a maximum value
        Generate two rays Refl and Refr in the reflected and refracted directions,
        Starting from Pt
        Call RayTrace with arguments Refl and Depth + 1
        Add (the return value * reflectivity of Obj) to C
        Call RayTrace with arguments Refr and Depth + 1
        Add (the return value * transmittivity of Obj) to C
    End if
Else
    Set the total colour C to the background colour
End if
Return C
End Procedure
```

The Post-Rendering Section

This section is also more or less programmer dependent, but the functions performed usually fall into one or more of the following groups

- *Memory cleanup*: The space taken up for the scene and rendering data is released.
- *Applying post-process filters*: Special effects such as blur, edge highlighting, suppression of colour channels, gamma correction etc. may be added to the picture after the actual rendering has taken place. Anti aliasing, motion blur etc. are usually a part of the main rendering section.
- *Freeing output devices*: If the output device is a file, it is closed. If it is a hardware device, then any link established with it may be suspended.
- *Displaying statistics and other user data*: This is, of course, entirely up to the programmer. But it is useful to know the details of the rendering process.

7.2.5 The Warnock Algorithm

The Warnock algorithm is a hidden surface algorithm invented by John Warnock that is typically used in the field of computer graphics. It solves the problem of rendering a complicated image by recursive subdivision of a scene until areas are obtained that are trivial to compute. In other words, if the scene is simple enough to compute efficiently then it is rendered; otherwise it is divided into smaller parts which are likewise tested for simplicity.

Warnock's Algorithm

A divide and conquer algorithm

- Warnock(PolyList PL, ViewPort VP)
- If (PL simple in VP) then
- Draw PL in VP
- else
- Split VP vertically and horizontally into VP1,VP2,VP3,VP4
- Warnock(PL in VP1, VP1)
- Warnock(PL in VP2, VP2)

Notes

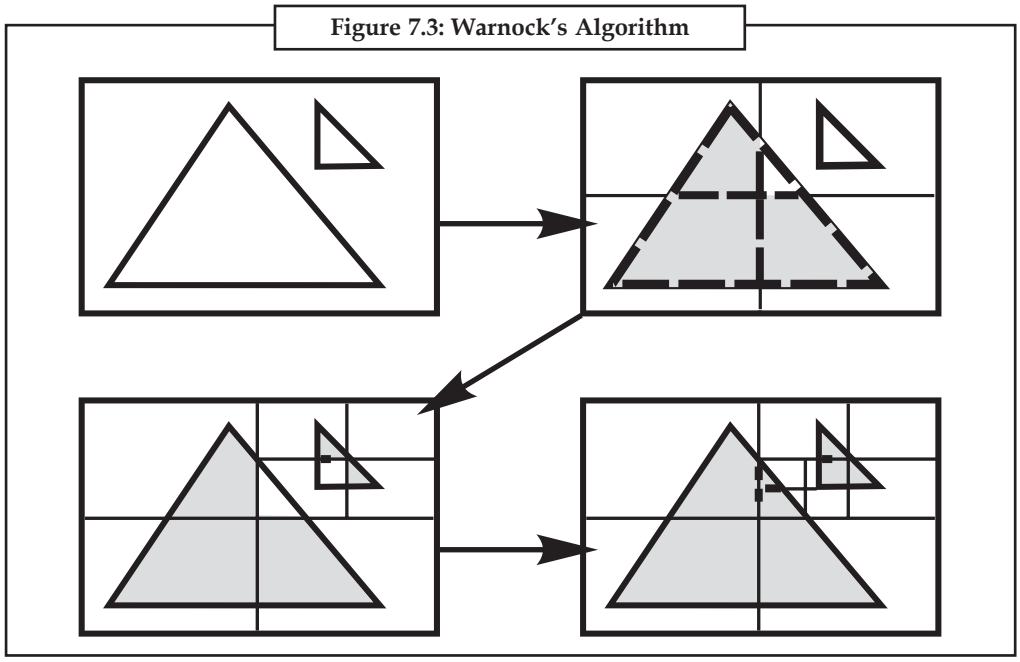
- Warnock(PL in VP3, VP3)
- Warnock(PL in VP4, VP4)

end

What does "simple" mean?

- No more than one polygon in viewport
- Scan convert polygon clipped to viewport
- Viewport only 1 pixel in size
- Shade pixel based on closest polygon in the pixel

Figure 7.3: Warnock's Algorithm



- Runtime: O (p*n)
  - *p*: number of pixels
  - *n*: number of polygons

Did u know?

The Warnock algorithm for hidden surface determination in computer graphics was invented by John Edward Warnock in 1969 in his doctoral thesis.

Self Assessment Questions

6. The process of hidden surface determination is sometimes called hiding, and such an algorithm is sometimes called …….
7. A set of pixels representing the …….. of the scene in the fixed given resolution, without any option to scale the scene.
8. ……. have an obtuse angle with the projection direction; and the reverse ones have an acute angle with the projection direction.
9. A conjunction between two near polygons, called ……..
10. A conjunction of two reverse polygons, called ……

Notes

11. A conjunction of the near polygon and the reverse polygon, called …….
12. The line scanning algorithm reduces memory demands of storage as here it is necessary to store only a field equal to one …….. of the screen
13. The painter algorithm is suitable for more simple scenes with limited conjunctions of objects, due to its low calculation and ……..
14. Ray tracing is, in many ways, a standardized technique. Almost all ray tracing programs use the same basic algorithm in one way or the other.

    (*a*) True (*b*) False

15. The z-buffer serves also to accept the situation where more distant objects are drawn last and in this way overlap objects standing closer to the observer.

    (*a*) True (*b*) False

7.3 Summary

- The process of hidden surface determination is sometimes called hiding, and such an algorithm is sometimes called a hider. The analogue for line rendering is hidden line removal. Hidden surface determination is necessary to render an image correctly.
- The Z-buffer serves also to avoid the situation where more distant objects are drawn last and in this way overlap objects standing closer to the observer.
- A disadvantage of the Z-buffer is its memory demand. The line scanning algorithm reduces memory demands of storage as here it is necessary to store only a field equal to one horizontal line of the screen.
- The painter algorithm is suitable for more simple scenes with limited conjunctions of objects, due to its low calculation and memory demands.
- Ray tracing programs use the same basic algorithm in one way or the other.

7.4 Keywords

Pixel: It is a physical point in a raster image, or the smallest, addressable element in a display device.

Polygons: A polygon is traditionally a plane figure that is bounded by a closed path, composed of a finite sequence of straight line segments.

Projection: A projection is the transformation of points and lines in one plane onto another plane by connecting corresponding points on the two planes with parallel lines.

Raster: A scan pattern in which an area is scanned from side to side in lines from top to bottom; also a pattern of closely spaced rows of dots that form an image.

Rendering: Rendering is the process of generating an image from a model by means of computer programs.

7.5 Review Questions

1. What is the object space algorithm?
2. How many type visibility algorithm categorizations?
3. Write a short note on hidden surface removal algorithms.

Notes

4. Solve the problem of hidden surface removal using Z-buffering algorithm.
5. Explain the Painter's algorithm.
6. Binary space partitioning algorithms using solve the on hidden surface removal.
7. Discuss the pre-rendering section and the post-rendering section.
8. Write a short note on Warnock's Algorithm.
9. Discuss Apple Algorithm.
10. What is the Hidden surface eliminator?

Answers for Self Assessment Questions

1.	(*b*)	2.	(*a*)	3.	(*d*)	4.	(*c*)	5.	(*b*)
6.	Hider			7.	specific points			8.	Polygons
9.	potential visible edge			10.	Invisible edge			11.	Visible contour edge
12.	horizontal line			13.	memory demands			14.	True
15.	False								

7.6 Further Readings


Books

Computer Graphics, by D.P Mukherjee and Debasish Jana

Comprehensive Computer Graphics, by V.K. Pachghare


Online link

www.siggraph.org/publications/newsletter/

Unit 8: Implementing of Scaling in 2D Transformation

Notes

CONTENTS

Objectives

After studying this chapter, you will be able to:

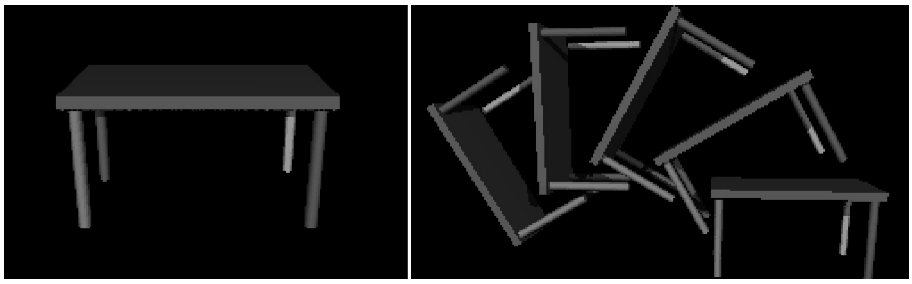
- Use transformation techniques to scale
- Define the 2D scaling from the origin
- Discuss the scaling about a particular point
- Describe the transformation between translation and scale
- Discuss the different functions of transformation
- Discuss homogeneous form of scale
- Discuss the transformation between translation and scale
- Describe the properties of scaling

Notes

Introduction

Transformations are a basic part of computer graphics. Transformations are used to place objects, to form objects, to change viewing positions, and even to change how something is viewed (e.g. the type of perspective that is used).(See figure 8.1)

Figure 8.1: 2D Transformation



In 3D graphics, we must use 3D transformations. However, 3D transformations can be quite confusing so it helps to first start with 2D.

8.1 3D Transformation

There are four main forms of transformations that one can execute in 2 dimensions:

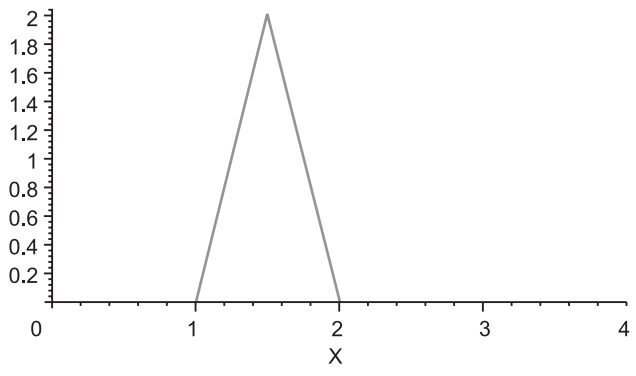
1. Translations
2. Scaling
3. Rotation
4. Shearing

These fundamental transformations can also be mingled to get more complex transformations. In order to make the demonstration of these complex transformations easier to recognize and more efficient, we introduce the idea of homogeneous coordinates.

Representation of Points/Objects

A point p in 2D is represented as a pair of numbers: p= (x, y) where x is the x-coordinate of the point p and y is the y-coordinate of p. 2D objects are often represented as a set of points (vertices), $\{p_1, p_2,...,p_n\}$, and an associated set of edges $\{e_1,e_2,...,em\}$. An edge is defined as a pair of points $e = \{p_i, p_j\}$. What are the points and edges of the triangle below? (See figure 8.2)

Figure 8.2: Representation of Points



Notes

We can also write points in vector/matrix notation as

$$p = \begin{bmatrix} x \\ y \end{bmatrix}$$

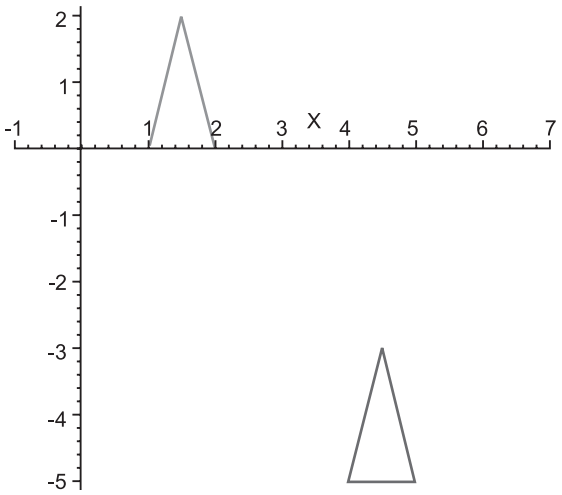
8.1.1 Translations

Suppose you are known a point at (x, y) = (2, 1). Where will the point be if you move it 3 units to the right and 1 unit up? Ans: (x′, y′) = (5, 2). How was this obtained? – (x′, y′) = (x + 3, y + 1). That is, to move a point by some amount dx to the right and dy up, you must add dx to the x-coordinate and add dy to the y-coordinate (See figure 8.3):

What was the required transformation to shift the green triangle to the red triangle? Here the green triangle is represented by 3 points

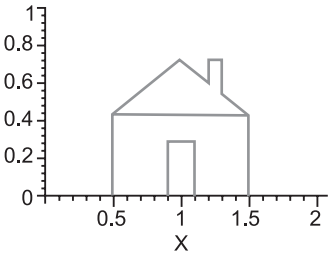
triangle = {p_1=(1,0), p_2=(2,0), p_3=(1.5,2) }

Figure 8.3: Translation



What are the points and edges in this picture of a house? Are the transformations required to move this house so that the peak of the roof is at the origin? What is compulsory to move the house as shown in animation? (See figure 8.4):

Figure 8.4: Translation

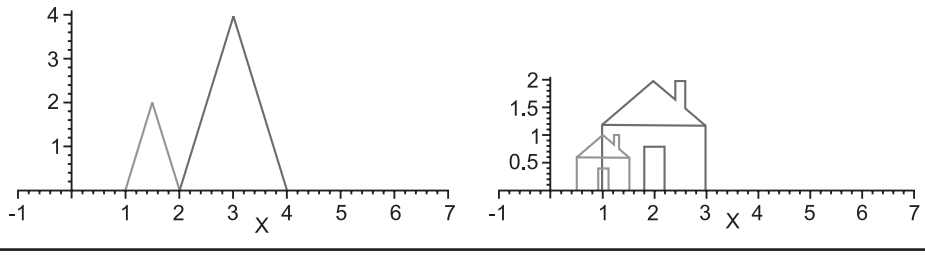


8.1.2 Scaling

Suppose we want to double the size of a 2-D object. What do we mean by double? Double in size, width only, height only, along some line only? When we talk about scaling we usually mean some total of scaling all along every dimension. That is, we must identify how much to change the size along each dimension. Below we see a triangle and a house that have been doubled in both width and height (note, the area is more than doubled). (See figure 8.5):

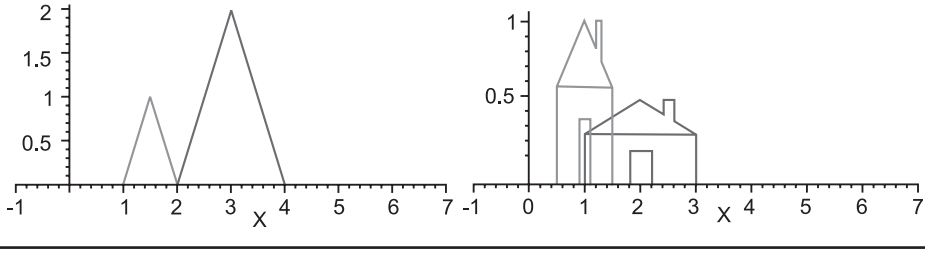
Notes

Figure 8.5: Scaling



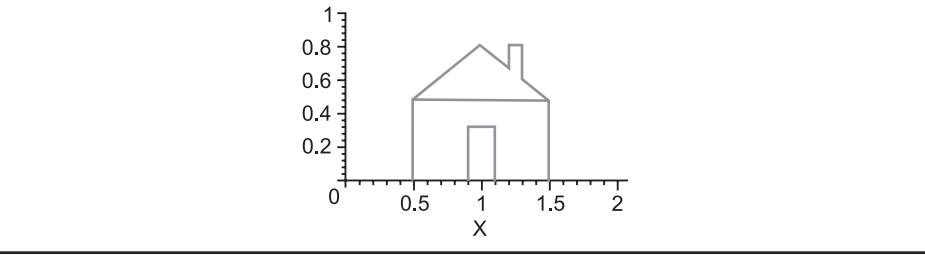
The scaling for the x dimension does not have to be the same as the y dimension. If these are different, then the object is distorted. What is the scaling in each dimension of the pictures below? (See figure 8.6):

Figure 8.6: Scaling



And if we double the size, where is the resulting object? In the pictures above, the scaled object is always shifted to the right. This is because it is scaled with respect to the origin. That is, the point at the origin is left fixed. Thus scaling by more than 1 move the object away from the origin and scaling of less than 1 moves the object toward the origin. This can be seen in the animation below (See figure 8.7):

Figure 8.7: Scaling



This is because of how basic scaling is done. The above objects have been scaled simply by multiplying each of its points by the appropriate scaling factor. For example, the point p = (1.5,2) has been scaled by 2 along x and .5 along y. Thus, the new point is

q = (2*1.5,.5*2) = (1,1).

Matrix/Vector Representation of Translations: Scaling transformations are represented by matrices. For example, the above scaling of 2 and .5 is represented as a matrix:

$$\text{scale matrix: } s = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & .5 \end{bmatrix}$$

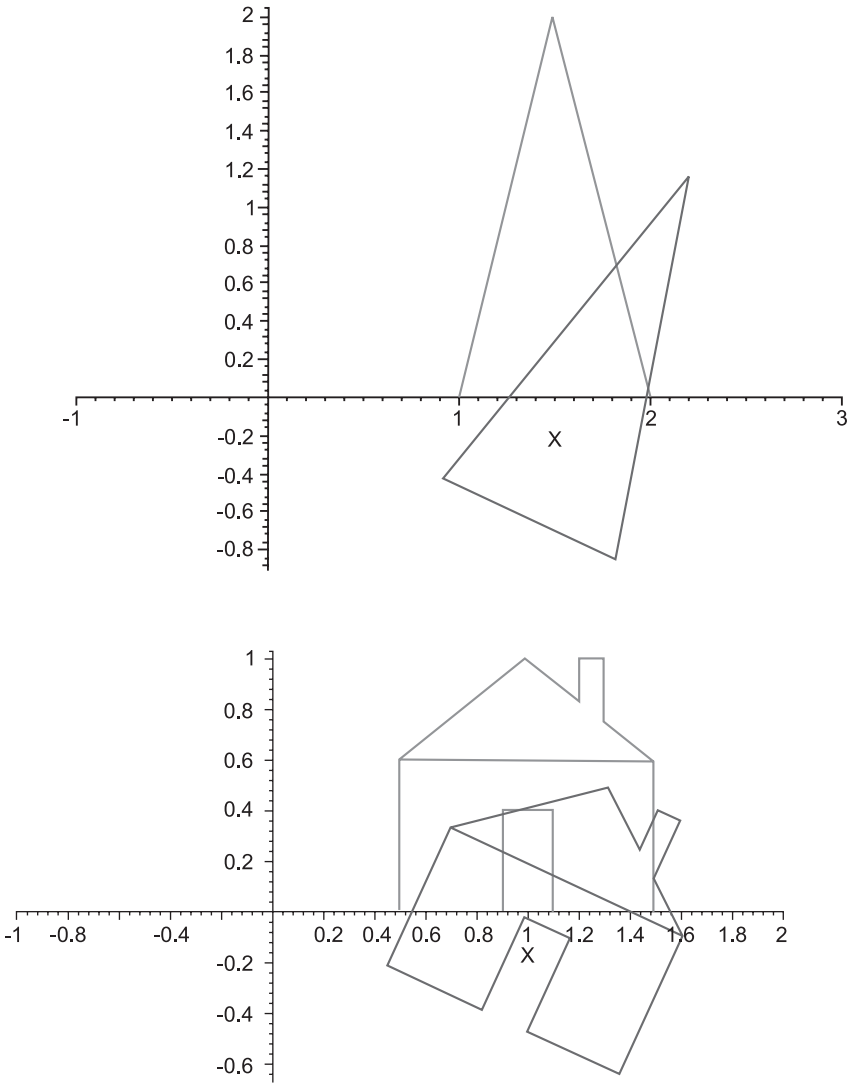
$$\text{new point: } q = s*p = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx & x \\ sy & y \end{bmatrix}$$

Notes

8.1.3 Rotation

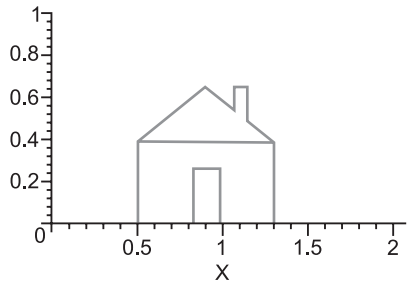
Below, we see objects that have been spin by 25 degrees (See Figure 8.8):

Figure 8.8: Rotation



Again, we see that basic rotations are regarding the origin (See Figure 8.9):

Figure 8.9: Rotation

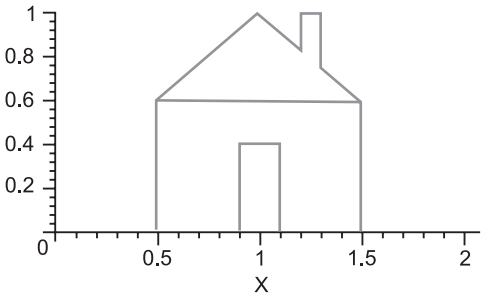


Notes

8.1.4 Shear

Figure 8.10 is viewing the point shearing.

Figure 8.10: Shearing



$$\text{shear along x axis} = \begin{bmatrix} 1 & shearx \\ 0 & 1 \end{bmatrix}$$

$$\text{shear along y axis} = \begin{bmatrix} 1 & 0 \\ sheary & 1 \end{bmatrix}$$

Self Assessment Questions

1. The …….. for the x dimension does not have to be the same as the y dimension.

   (*a*) rotations (*b*) translations

   (*c*) scaling (*d*) shearing

2. The basic …….. are with respect to the origin.

   (*a*) rotations (*b*) translations

   (*c*) scaling (*d*) shearing

3. Changing the size of an object is called a…….

   (*a*) vertex (*b*) scale

   (*c*) matrix (*d*) origin

4. Coordinates multiplied by the scaling factor:

   (*a*) Scale by (Sx Sy) (*b*) translate origin back

   (*c*) x′ = sx*x (*d*) None of these

5. In case of rotation, object can be rotated about x or y axis.

   (*a*) True (*b*) False

8.2 Concept of Implementing of Scaling in 2D Transformation

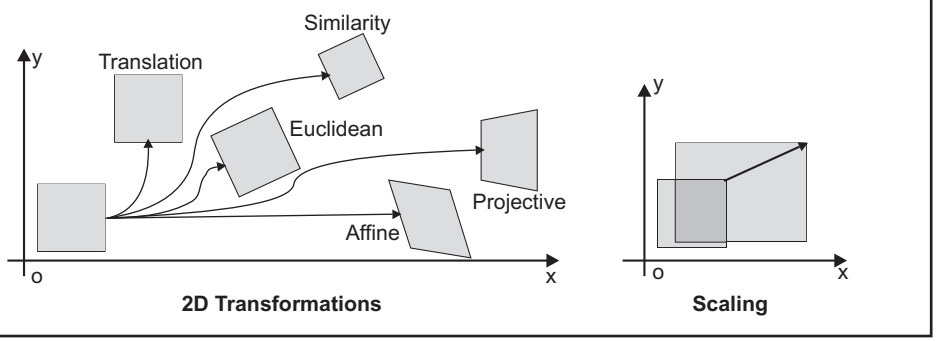
Changing the size of an object is called a scale. We scale an object by Changing the size of an object is called a scale.

Notes

8.2.1 Scaling

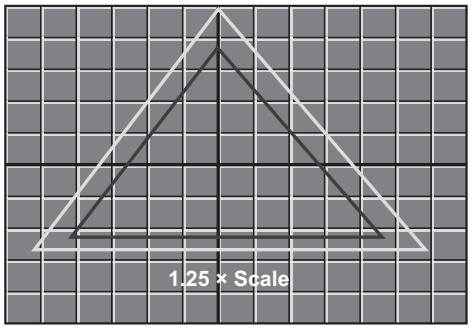
We scale an object by scaling the x and y coordinates of each vertex in the object (See Figure 8.11):

Figure 8.11: Scaling



This is a really neat feature to include if you are going to use 2d points for an overhead map because you can zoom in an out to get the amount of aspect you like. This function is also dependent on the fact that all points must be centered around (0, 0). If we do not follow this rule we will get object flying all over the place. Take a look at the equation and (See Figure 8.12).

Figure 8.12: Scaling




```
point2d Transform2d::Scale(float scale,point2d &point)
{
 point2d newpoint;
 newpoint.x = point.x*scale;
 newpoint.y = point.y*scale;
 return newpoint;
}
```

As you can see with this simple equation, if an object's points are not centered around (0,0) it will be hurled in a direction and look like its being translated and if it does handle to seem to be scaled, it would not look correct. So one more time, confirm that you objects are centered around (0,0)!!

8.2.2 Scaling Matrix

- Change size:
  1. x′ = sx * x
  2. y′ = sy * y

Notes

- Vector form:

1. P′ = S*P

$$S = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

8.2.3 2D Scaling From the Origin

General fixed point scaling (scaling about a point (a b) :

- Translate Origin To (a b).
- Scale by (Sx Sy).
- Translate Origin Back.

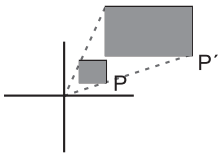
It can be clarify as:

Point P defined as P(x, y),

Perform a scale (stretch) to Point P′(x′, y′) by a factor S_x along the x-axis,

And S_y along the y-axis

$X' = S_x*X$, $Y' = S_y*Y$



So scaling around the origin:

$$[x' \quad y'] = [x \quad y] \begin{bmatrix} S_x & 0 \\ 0 & s_y \end{bmatrix}$$

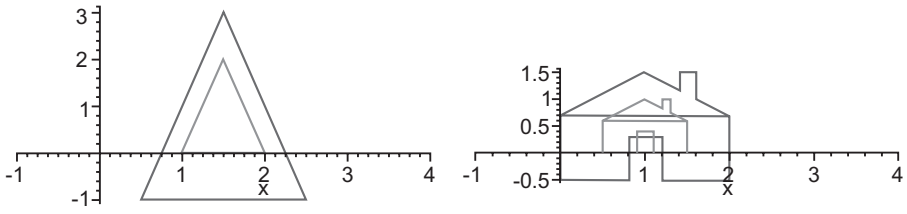
or

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

8.2.4 Scaling about a Particular Point

Figure 8.13 illustrates a scaling about a particular point.

Figure 8.13: Scaling about a Particular Point



Scaling in 2D

Coordinates multiplied by the scaling factor:

- x′ = sx*x
- y′ = sy*y

Scaling in 2D, matrix notation

Notes

- Scaling is a matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$P' = SP$$

8.2.5 Write a C Program to Implement 2D Transformations

Step By Step Procedural Algorithm

1. Enter the choice for transformation.
2. Perform the translation, rotation, scaling, reflection and shearing of 2D object.
3. Get the needed parameters for the transformation from the user.
4. In case of rotation, object can be rotated about x or y axis.
5. Display the transmitted object in the screen.

Source Code Programming 2D Transformations

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
int ch,x,y,az,i,w,ch1,ch2,xa,ya,ra,a[10],b[10],da,db;
float x1,y1,az1,w1,dx,dy,theta,x1s,y1s,sx,sy,a1[10],b1[10];
void main()
{
int gm ,gr;
clrscr();
detectgraph(&gm,&gr);
initgraph(&gm,&gr,"d:tcBGI");
printf("Enter the upper left corner of the rectangle:n");
scanf("%d%d",&x,&y);
printf("Enter the lower right corner of the rectangle:n");
scanf("%d%d",&az,&w);
rectangle(x,y,az,w);
da=az–x;
db=w–y;
a[0]=x;
b[0]=y;
a[1]=x+da;
b[1]=y;
a[2]=x+da;
b[2]=y+db;
a[3]=x;b[3]=y+db;
```

Notes

```
while(1)
{
printf("******2D Transformations*******n");
printf("1.Translationn2.Rotationn3.Scalingn4.Reflectionn5.Shearingn6.
ExitnEnter your choice:n");
scanf("%d",&ch);
switch(ch)
{
case 1:
detectgraph(&gm,&gr);
initgraph(&gm,&gr,"d:tcBGI");
rectangle(x,y,az,w);
printf("*******Translation*******nn");
printf("Enter the value of shift vector:n");
scanf("%f%f",&dx,&dy);
x1=x+dx;
y1=y+dy;
az1=az+dx;
w1=w+dy;
rectangle(x1,y1,az1,w1);
break;
case 2:
detectgraph(&gm,&gr);
initgraph(&gm,&gr,"d:tcBGI");
rectangle(x,y,az,w);
printf("*******Rotation*******nn");
printf("Enter the value of fixed point and angle of rotation:Enter the
value of fixed point and angle of rotation:n");
scanf("%d%d%d",&xa,&ya,&ra);
theta=(float)(ra*(3.14/180));
for(i=0;i<4;i++)
{
a1[i]=(xa+((a[i]–xa)*cos(theta)–(b[i]–ya)*sin(theta)));
b1[i]=(ya+((a[i]–xa)*sin(theta)+(b[i]–ya)*cos(theta)));
}
for(i=0;i<4;i++)
{
if(i!=3)
line(a1[i],b1[i],a1[i+1],b1[i+1]);
else
line(a1[i],b1[i],a1[0],b1[0]);
}
```

Notes

```
break;
case 3:
detectgraph(&gm,&gr);
initgraph(&gm,&gr,"d:tcBGI");
rectangle(x,y,az,w);
printf("********Scaling*******nn");
printf("Enter the value of scaling factor:n");
scanf("%f%f",&sx,&sy);
x1=x*sx;
y1=y*sy;
az1=az*sx;
w1=w*sy;
rectangle(x1,y1,az1,w1);
break;
case 4:
detectgraph(&gm,&gr);
initgraph(&gm,&gr,"d:tcBGI");
rectangle(x,y,az,w);
printf("*******Reflection*********n");
printf("1.About x–axisn2.About y-axisn3.About both axisnEnter your
choice:n");
scanf("%d",&ch1);
switch(ch1)
{
case 1:
printf("Enter the fixed pointn");
scanf("%d%d",&xa,&ya);
theta=(float)(90*(3.14/180));
for(i=0;i<4;i++)
{
a1[i]=(xa+((a[i]–xa)*cos(theta)–(–b[i]–ya)*sin(theta)));
b1[i]=(ya+((a[i]–xa)*sin(theta)+(–b[i]–ya)*cos(theta)));
}
for(i=0;i<4;i++)
{
if(i!=3)
line(a1[i],b1[i],a1[i+1],b1[i+1]);
else
line(a1[i],b1[i],a1[0],b1[0]);
}
break;
case 2:
```

Notes

```
printf("Enter the fixed pointn");
scanf("%d%d",&xa,&ya);
theta=(float)(270*(3.14/180));
for(i=0;i<4;i++)
{
a1[i]=(xa+((–a[i]–xa)*cos(theta)–(b[i]–ya)*sin(theta)));
b1[i]=(ya+((–a[i]–xa)*sin(theta)+(b[i]–ya)*cos(theta)));
}
for(i=0;i<4;i++)
{
if(i!=3)
line(a1[i],b1[i],a1[i+1],b1[i+1]);
else
line(a1[i],b1[i],a1[0],b1[0]);
}
break;
case 3:
printf("Enter the fixed pointn");
scanf("%d%d",&xa,&ya);
theta=(float)(180*(3.14/180));
for(i=0;i<4;i++)
{
a1[i]=(xa+((–a[i]–xa)*cos(theta)–(–b[i]–ya)*sin(theta)));
b1[i]=(ya+((–a[i]–xa)*sin(theta)+(–b[i]–ya)*cos(theta)));
}
for(i=0;i<4;i++)
{
if(i!=3)
line(a1[i],b1[i],a1[i+1],b1[i+1]);
else
line(a1[i],b1[i],a1[0],b1[0]);
}
break;
}
break;
case 5:
detectgraph(&gm,&gr);
initgraph(&gm,&gr,"d:tcBGI");
rectangle(x,y,az,w);
printf("*******Shearing******nn");
printf("1.x-direction shearn2.y-direction shearnEnter your choice:n");
scanf("%d",&ch2);
```

Notes

```
switch(ch2)
{
case 1:
printf("Enter the value of shear:n");
scanf("%f",&x1s);
x1=x+(y*x1s);
y1=y;
az1=az+(w*x1s);
w1=w;
rectangle(x1,y1,az1,w1);
break;
case 2:
printf("Enter the value of shear:n");
scanf("%f",&y1s);
x1=x;
y1=y+(x*y1s);
az1=az;
w1=w+(az*y1s);
rectangle(x1,y1,az1,w1);
break;
}
break;
case 6:
exit(0);
}
}
getch();
}
```

Example OUTPUT Result computer Graphics

```
Enter the upper left corner of the rectangle:
100
100
Enter the lower right corner of the rectangle:
200
200
******2DTransformations*******
1. Translation
2. Rotation
3. Scaling
4. Reflection
5. Shearing
6. Exit
```

Notes

Enter your choice: 1
*******Translation******* Enter the value of shift vector:
150
150
******2DTransformations*******
1. Translation
2. Rotation
3. Scaling
4. Reflection
5. Shearing
6. Exit
Enter your choice: 2
*******Rotation*******
Enter the value of fixed point and angle of rotation:
300
300
70
******2DTransformations******* 1.Translation
2. Rotation
3. Scaling
4. Reflection
5. Shearing
6. Exit
Enter your choice: 3
********Scaling*******
Enter the value of scaling factor:
2
2
******2DTransformations*******
1. Translation
2. Rotation
3. Scaling
4. Reflection
5. Shearing
6. Exit
Enter your choice: 4
*******Reflection*********
1. about x-axis
2. about y-axis
3. about both axis
Enter your choice: 1
Enter the fixed point

Notes

```
150
150
Enter your choice: 2
Enter the fixed point
150
150
Enter your choice: 3
Enter the fixed point
150
150
******2DTransformations******* 1.Translation
2. Rotation
3. Scaling
4. Reflection
5. Shearing
6. Exit
Enter your choice: 5
*******Shearing*********
1. x-direction shear
2. y-direction shear
Enter your choice: 1
Enter the value of shear: 2
Enter your choice: 2
Enter the value of shear: 2
```

RESULT: Thus the c program to implement 2D transformations was coded and executed successfully.

Did u know?

2×SaI, short for 2× Scale and Interpolation engine, was inspired by Eagle.

8.2.6 Homogeneous Coordinates

In general, when you want to execute a complex transformation, you usually compose it by combining a number of essential transformations. The above equation for q, however, is awkward to read because scaling is done by matrix multiplication and translation is done by vector addition. In order to represent all transformations in the same form, computer scientists have devised what are called homogeneous coordinates. Do not try to apply any exotic interpretation to them. They are merely a mathematical hoax to create the representation be more steady and easier to apply.

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Form of Scale:

Recall the (x, y) form of Scale:

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Notes

In homogeneous coordinates:

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous coordinates (HC) add an extra virtual dimension. Thus 2D HC are actually 3D and 3D HC are 4D. Consider a 2D point p = (x,y). In HC, we represent p as p = (x, y, 1). An extra coordinate is added whose value is always 1. This may seem odd but it allows us to now represent translations as matrix multiplication instead of as vector addition. A translation (dx, dy) which would normally be performed as q = (x,y) + (dx, dy) now is written as

$$q = Tp = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix}$$

Now, we can write the scaling about a fixed point as simply a matrix multiplication:

$$q = (-T)\ S\ T\ p = A\ p,$$

Where $A = (-T)\ S\ T$

The matrix A can be calculated once and then applied to all the points in the object. This is much more efficient than our previous representation. It is also easier to identify the transformations and their order when everything is in the form of matrix multiplication.

The matrix for scaling in HC is

$$S = \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And the matrix for rotation is

$$R = \begin{bmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

So scaling with homogeneous:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \rightarrow \begin{cases} x' = S_x x \\ y' = S_y y \\ w' = w \end{cases} \rightarrow \begin{cases} \frac{x'}{w'} = S_x \frac{x}{w} \\ \frac{y'}{w'} = S_y \frac{y}{w} \end{cases}$$

8.2.7 Transformation 2D (Scaling, Translation, Rotation)

```
#include "Stdio.h"
#include "conio.h"
#include "math.h"
#include "graphics.h"
void intgraph();
void display_cordinate(char bg_color,char line_color);
Create_poly(int poly[][2]);
void fill_poly(int poly[][2],int points,char line_color,char fill_color);
void Identity(float Matrix[3][3]);
void Translate_to_Relative(int poly[][2],int points,int Xdis,int Ydis);
void main()
```

Notes

```
{
 int poly[3][2],i,points,Xdis,Ydis;
 float Sx,Sy,Matrix[3][3],temp,Cos,Sin;
 char ch;
 intgraph();
 setfillstyle(8,RED);
 points=Create_poly(poly);
 display_cordinate(DARKGRAY,WHITE);
 fill_poly(poly,points,WHITE,RED);
 getch();
 Identity(Matrix);
 Xdis=0–poly[0][0];
 Ydis=0–poly[0][1];
 Translate_to_Relative(poly,points,Xdis,Ydis);
 Menu:
 restorecrtmode();
 do
 {
 clrscr();
 printf("================================");
 printf("
    1. Scaling");
 printf("
    2. Translation");
 printf("
    3. Rotation");
 printf("
    4. Reflection [ X axis ]");
 printf("
    5. Reflection [ Y axis ]");
 printf("
    6. Reflection [ X=Y axis ]");
 printf("
    7. Draw ");
 printf("
    8. EXIT ");
 printf("================================");
 printf("
 Choose Ur Destiny:- ");
 ch=getche();
 }while((ch< '1')||(ch> '8'));
 switch(ch)
```

Notes

```
{
case ‘1’:
printf("
Enter Scaling Ratio:- ");
printf("
Sx:- ");
scanf("%f",&Sx);
printf(" Sy:- ");
scanf("%f",&Sy);
for(i=0;i<3;i++)
{
Matrix[i][0]=Matrix[i][0]*Sx;
Matrix[i][1]=Matrix[i][1]*Sy;
}
goto Menu;
case ‘2’:
printf(" Enter Translation [Relative]:- ");
printf(" Tx:- ");
scanf("%f",&Sx);
printf("   Ty:- ");
scanf("%f",&Sy);
Matrix[2][0]=Matrix[2][0]+Sx;
Matrix[2][1]=Matrix[2][1]+Sy;
goto Menu;
case ‘3’:
printf("
Enter Rotation Angle [Degree]:- ");
scanf("%f",&Sx);
Sx=(Sx*3.14)/180;
Cos=cos(Sx);
Sin=sin(Sx);
for(i=0;i<=2;i++)
{
temp=Matrix[i][0]*Cos–Matrix[i][1]*Sin;
Matrix[i][1]=Matrix[i][0]*Sin+Matrix[i][1]*Cos;
Matrix[i][0]=temp;
}
goto Menu;
case ‘4’:
Matrix[1][1]=Matrix[1][1]*–1;
printf("
```

Notes

```
    Done");
 getch();
 goto Menu;
 case '5':
 Matrix[0][0]=Matrix[0][0]*-1;
 printf("
    Done");
 getch();
 goto Menu;
 case '6':
 Matrix[0][1]=1;
 Matrix[1][1]=1;
 printf("
    Done");
 getch();
 goto Menu;
 case '7':
 for(i=0;i<=points;i++)
 {
 temp=poly[i][0]*Matrix[0][0]+poly[i][1]*Matrix[1][0]+Matrix[2][0];
poly[i][1]=poly[i][0]*Matrix[0][1]+poly[i][1]*Matrix[1][1]+Matrix[2][1];
 poly[i][0]=temp;
 }
 case '8':
 getch();
// closegraph();
 exit(0);
 }
 setgraphmode(2);
 display_cordinate(DARKGRAY,WHITE);
 Translate_to_Relative(poly,points,-Xdis,-Ydis);
 fill_poly(poly,points,CYAN,RED);
 getch();
 restorecrtmode();
 do
 {
 clrscr();
 printf("
    Do you Want to Switch to MENU [Y|N]:- ");
 ch=getche();
 }while( (ch!= 'Y') && (ch!= 'N') );
 if(ch== 'Y')
```

Notes

```
{
Identity(Matrix);
Xdis=0–poly[0][0];
Ydis=0–poly[0][1];
Translate_to_Relative(poly,points,Xdis,Ydis);
goto Menu;
}
closegraph();
}
void intgraph()
{
int g=DETECT,d;
initgraph(&g,&d,"c:tcbgi");
}
void fill_poly(int poly1[][2],int points,char line_color,char fill_color)
{
int pol[20],i;
char str[2];
for(i=0;i<=points;i++)
{
pol[i*2]=poly1[i][0];
pol[i*2+1]=poly1[i][1];
}
pol[i*2]=poly1[0][0];
pol[i*2+1]=poly1[0][1];
setcolor(line_color);
setfillstyle(8,fill_color);
fillpoly(points+1,pol);
setcolor(fill_color);
settextstyle(1,0,3);
for(i=0;i<=points;i++)
{
sprintf(str,"%c",i+ ‘a’);
outtextxy(poly1[i][0],poly1[i][1],str);
}
}
void Identity(float Matrix[3][3])
{
int i,j;
for(i=0;i<=2;i++)
{
for(j=0;j<=2;j++)
```

Notes

```
{
if(i==j)
Matrix[i][j]=1;
else
Matrix[i][j]=0;
}
}
}
int Create_poly(int poly[][2])
{
poly[0][0]=200;
poly[0][1]=20;
poly[1][0]=300;
poly[1][1]=150;
poly[2][0]=130;
poly[2][1]=280;
return 2;
}
void display_cordinate(char bg_color,char line_color)
{
int i;
setbkcolor(bg_color);
setcolor(line_color);
for(i=0;i<=640;i+=50)
{
line(i,0,i,480);
}
for(i=0;i<=480;i+=50)
{
line(0,i,640,i);
}
rectangle(0,0,639,479);
}
void Translate_to_Relative(int poly[][2],int points,int Xdis,int Ydis)
{
int i;
for(i=0;i<=points;i++)
{
poly[i][0]=poly[i][0]+Xdis;
poly[i][1]=poly[i][1]+Ydis;
}
}
```

Notes

8.2.8 Combining Transformations

We saw that the essential scaling and rotating transformations are always regarding the origin. To scale or rotate about an exacting point (the fixed point) we should first translate the object so that the fixed point is at the origin. We then perform the scaling or rotation and then the opposite of the original translation to move the set point back to its original position. For example, if we want to scale the triangle by 2 in each direction about the point fp = (1.5,1), we first translate all the points of the triangle by T = (–1.5,1), scale by 2 (S) , and then translate back by T = (1.5,1). Mathematically this looks like.

$$q = \begin{bmatrix} x^2 \\ y^2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \left(\begin{bmatrix} x1 \\ y1 \end{bmatrix} + \begin{bmatrix} -1.5 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 1.5 \\ 1 \end{bmatrix}$$

Order Matters!

Notice the order in which these transformations are performed. The first (rightmost) transformation is T and the last (leftmost) is –T. If you apply these transformations in a different order then you will get very different results. For example, what happens when you first apply T followed by –T followed by S? Here T and –T cancel each other out and you are simply left with S. Sometimes (but be careful) order does not matter, For example, if you apply multiple 2D rotations, order makes no difference:

R1 R2 = R2 R1

But this will not necessarily be true in 3D.

8.2.9 Transformation between Translation and Scale

Scale then Translate: p′ = T (S p) = TS p

$$TS = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 3 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Translate then Scale: p′ = S (T p) = ST p

$$ST = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 6 \\ 0 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

8.2.10 Concatenation of Scales

The matrix product S (sx_1,*sy_1) S (sx_2,*sy_2) is:

$$\begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Only diagonal matrix in the elements easy to multiply!

8.2.11 Other Properties of Scaling

- It does not preserve lengths in objects.
- It does not preserve angles between parts of objects (except when scaling is uniform, $x = sy$).
- If it is not at origin, translates house relative to origin– often not desired.

8.2.12 Conclusion: 2D Transformations

1. Simple, consistent matrix notation
   - using homogeneous coordinates

- all transformations expressed as matrices

2. Used by the window system:
   - for conversion from model to window
   - for conversion from window to model
3. Used by the application:
   - for modelling transformations

Self Assessment Questions

6. There is simply a …… trick to make the representation be more consistent and easier to use.
7. Homogeneous coordinates (HC) add an extra ……. dimension.
8. To scale or rotate about a particular point we must first ……. the objects so that the fixed point is at the origin.
9. Scaling does not preserve …….. in objects.
10. If scaling is not at origin, translates …… relative to origin often not desired.
11. 2D transformations are used by the window system for conversion from model to ……..
12. We scale an object by scaling the …… of each vertex in the object.
13. Scaling is use to changing the ……. of an object.
14. 2D transformations are simple in using …….
15. A linear transformation is a function between two vector spaces that preserves the operations of vector addition and scalar multiplication.

    (*a*) True (*b*) False

8.3 Summary

- Transformations are used to position objects, to shape objects, to change viewing positions, and even to change how something is viewed.
- Scale is changing the size of an object.
- A linear transformation is a function between two vector spaces that preserves the operations of vector addition and scalar multiplication.
- 2D transformations are used by the window system for conversion from window to model.
- 2D transformations are simple in using homogeneous coordinates.
- If scaling is not at origin, translates house relative to origin often not desired.
- There is simply a mathematical trick to make the representation be more consistent and easier to use.
- 2D transformations are used by the application for modelling transformations.
- In 2D transformations, all transformations expressed as matrices.

8.4 Keywords

Homogeneous Coordinates: It usually used to perform a complex transformation.

Linear Transformation: It is a function between two vector spaces that preserves the operations of vector addition and scalar multiplication.

Notes

Modelling Transformation: 2D transformations are used by the application for modeling transformations.

Object: We scale an object by scaling the x and y coordinates of each vertex in the object.

Scale: It is changing the size of an object.

Scaling: It does not preserve angles between parts of objects (except when scaling is uniform $x=sy$).

Transformations: These are used to position objects, to shape objects, to change viewing positions, and even to change how something is viewed.

8.5 Review Questions

1. What are the points and edges in this picture of a house?
2. What are the transformations is required to move this house so that the peak of the roof is at the origin?
3. What are the different functions of transformation?
4. What is required to move the house as shown in animation?
5. Write a C program to implement 2D transformations.
6. Describe the transformation between translation and scale.
7. Define the 2D scaling from the origin.
8. What are the properties of scaling?
9. Discuss the scaling about a Particular Point.
10. Discuss the Homogeneous form of scale.

Answers for Self Assessment Questions

1.	(*c*)	2.	(*a*)	3.	(*b*)	4.	(*c*)	5.	(*a*)
6.	mathematical			7.	virtual	8.	translate	9.	lengths
10.	house	11.	window/model			12.	x and y coordinates		
13.	size	14.	homogeneous coordinates			15.	**		

8.6 Further Readings

Books

"*Computer Graphics*", by Steven Harington

"*Principles of Interactive Computer Graphics*", by Newman and Sproull

"*Computer Graphics*", by A.P. Godse

Online link

http://web.iitd.ac.in/~achawla/public_html/429/transformations/2d-transformations.pdf

Unit 9: Translation

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the 2-dimensional translation in c program
- Describe translating with matrices
- Explain the 3-dimensional translation in c program
- Describe translating addresses
- What is the inverse of a translation?

Introduction

Translation is one of the easiest transformations. A translation goes all points of an object a fixed distance in a particular direction. It can also be stated in words of two frames by expressing the coordinate system of object in terms of translated frames. We can develop the matrix involved in a straightforward manner by considering the translation of a single frame. If we are given a frame, $F = (\vec{u}, \vec{v}, \vec{w}, o)$ a translated frame would be one that is given by $F = (\vec{u}, \vec{v}, \vec{w}, o)$ that is, the origin is moved, and the vectors stay the same. If we write $\mathbf{O}'$ in terms of the previous frame by

If we write O′ in terms of the previous frame by

$$\mathbf{O}' = a\vec{u} + b\vec{v} + c\vec{w} + \mathbf{O}$$

then we can write the frame $\mathcal{F}'$ in terms of the frame $\mathcal{F}$ by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathrm{O} \end{bmatrix} = \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathrm{O}' \end{bmatrix}$$

Notes

So a 4×4 matrix implements a frame to frame transformation for translated frames, and any matrix of this type (for arbitrary a, b, c) will translate the frame F. We call any matrix

$$T_{a,b,c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix}$$

A translation matrix and utilize matrices of this type to implement translations.

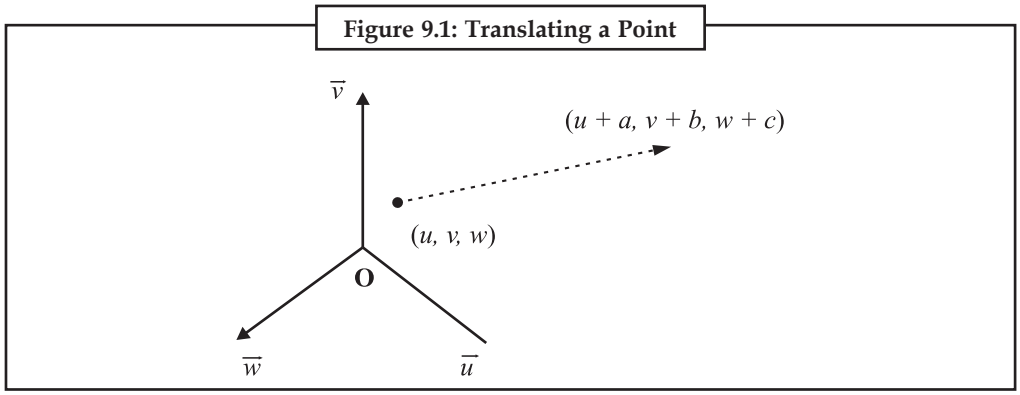
Transformation Directly to the Coordinates of Point

Given a frame $F = (\vec{u}, \vec{v}, \vec{w}, o)$ and a point P that has coordinates (u, v, w) in F, if we apply the transformation to the coordinates of the point we obtain

$$[u\ v\ w\ 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{bmatrix} = [u + a \quad v + b \quad w + c \quad 1]$$

That is, we can translate the point *within the frame* F. An illustration of this is shown in Figure 9.1

Figure 9.1: Translating a Point



Translation is a simple transformation that is calculated directly from the conversion matrix for two frames, one a translate of the other. The translation matrix is most frequently applied to all points of an object in a local coordinate system resulting in an action that moves the object within this system.

```
#include<stdio.h>
#include<string.h>
#include<alloc.h>
#include<graphics.h>
#include<stdlib.h>
#include<conio.h>
#include<bios.h>
#include<math.h>
#include<dos.h>
#defineLTARROW0x4B
#defineRTARROW0x4D
#defineUPARROW0x48
#defineDNARROW0x50
#defineCR0x0d
```

Notes

```
#defineESC0x1b
#defineALT_X0x2d
longsize=0;
voidfar*buffer2=0;
voidfar*buffer1=0;
voiddraw_pixelbackground()
{
inti,xpos,ypos,color;
intxmax=getmaxx();
intymax=getmaxy();
setbkcolor(BLUE);
setcolor(WHITE);
rectangle(0,0,xmax,ymax);
for(i=0;i<5000;i++)
{
xpos=rand()%xmax;
ypos=rand()%ymax;
color=rand()%12;
putpixel(xpos,ypos,color);
}
}
voiddraw_object(intleft,inttop,intright,intbottom)
{
size=imagesize(left,top,right,bottom);
buffer1=farmalloc(size);
buffer2=farmalloc(size);
getimage(left,top,right,bottom,buffer1);
rectangle(left,top,right,bottom);
line(left,top,right,bottom);
line(left,bottom,right,top);
}
voidmove_object(intleft,inttop,intright,intbottom,intdestx,intdesty)
{
intwidth=right-left;
intheight=bottom-top;
getimage(left,top,right,bottom,buffer2);
putimage(left,top,buffer1,COPY_PUT);
getimage(destx,desty,destx+width,desty+height,buffer1);
putimage(destx,desty,buffer2,COPY_PUT);
}
voidmain()
{
```

Notes

```
intgrd,grm;
intkey;
intx1=10,y1=30,x2=100,y2=70;
intnewx,newy;
intxoffset=10,yoffset=5;
detectgraph(&grd,&grm);
initgraph(&grd,&grm,"");
draw_pixelbackground();
draw_object(x1,y1,x2,y2);
while(1)
{
key=bioskey(0);
if((key&0x00FF)>0)
key=key&0x00FF;
else
key=(key&0xFF00)>>8;
switch(key)
{
caseLTARROW:
newx=x1-xoffset;
newy=y1;
move_object(x1,y1,x2,y2,newx,newy);
x1-=xoffset;
x2-=xoffset;
break;
caseRTARROW:
newx=x1+xoffset;
newy=y1;
move_object(x1,y1,x2,y2,newx,newy);
x1+=xoffset;
x2+=xoffset;
break;
caseDNARROW:
newx=x1;
newy=y1+yoffset;
move_object(x1,y1,x2,y2,newx,newy);
y1+=yoffset;
y2+=yoffset;
break;
caseUPARROW:
newx=x1;
```

Notes

```
newy=y1-yoffset;
move_object(x1,y1,x2,y2,newx,newy);
y1-=yoffset;
y2-=yoffset;
break;
caseALT_X:
caseESC:
caseCR:
closegraph();
exit(0);
}
}
}
```

9.1 2–Dimensional Translation in C program

Translation is an easy directly line movement of the object in x and y direction. Transformation is refer to transform from one location to another location depends upon their transformation it is classified the Scaling, Shearing, Reflection and Rotation see the source code in C coding.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include<graphics.h>
#include<conio.h>
#include<math.h>
void draw2d(int,int [],int [],int,int);
void main()
{
int gd = DETECT, gm;
int x[20], y[20], tx=0, ty=0, i, fs;
initgraph(&gd, &gm,"");
printf("No of sides:");
scanf("%d", &fs);
printf("Co-ordinates :");
for(i=0;i<fs;i++)
{
printf("(x%d,y%d)", i, i);
scanf("%d%d", &x[i], &y[i]);
}
draw2d(fs, x, y, tx, ty);
printf("translation (x, y):");
scanf("%d%d", &tx, &ty);
```

Notes

```
draw2d(fs, x, y, tx, ty);
getch();
}
void draw2d(int fs, int x[20], int y[20], int tx, int ty)
{
int i;
for(i=0;i<fs;i++)
{
if(i!=(fs–1))
line(x[i] + tx, y[i]+ty, x[i+1]+tx, y[i+1]+ty);
else
line(x[i]+tx, y[i]+ty, x[0]+tx, y[0]+ty);
}
}
```

9.2 3–Dimensional Translation in C program

3Dimensional transformation it has three axis x, y, z depending upon their coordinate it will execute their translation, rotation, scaling. The translation can be performed by changing the symbol of the translation components Tx, Ty, and Tz.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include<graphics.h>
#include<conio.h>
void draw3d(int fs, int x[20], int y[20], int tx, int ty, int d);
void draw3d(int fs, int x[20], int y[20], int tx, int ty, int d);
{
int i,j,k=0;
for(j=0;j<2;j++)
{
for(i=0;i<fs;i++)
{
if(i!=fs–1)
line(x[i]+tx+k, y[i]+ty–k, x[i+1]+tx+k, y[i+1]+ty–k);
else
line(x[i]+tx+k,y[i]+ty–k,x[0]+tx+k,y[0]+ty–k);
}
k=d;
}
for(i=0;i<fs;i++)
{
```

Notes

```
line(x[i]+tx,y[i]+ty,x[i]+tx+d,y[i]+ty–d);
}
}
void main()
{
int gd=DETECT,gm;
int x[20],y[20],tx=0,ty=0,i,fs,d;
initgraph(&gd,&gm,"");
printf("no of sides (front view only) : ");
scanf("%d",&fs);
printf("co-ordinates : ");
for(i=0;i<fs;i++)
{
printf("(x%d,y%d)",i,i);
scanf("%d%d",&x[i],&y[i]);
}
printf("Depth :");
scanf("%d",&d);
draw3d(fs,x,y,tx,ty,d);
printf("translation (x,y)");
scanf("%d%d",&tx,&ty);
draw3d(fs,x,y,tx,ty,d);
getch();
}
```

9.3 Translating with Matrices

This function obtains as factors an orientation to the matrix that grasps the current state of the transformation and the X and Y translation values. First, the function loads a local matrix with the worth that makes a translation matrix, after which it multiplies the translation matrix times, the main transformation matrix. The effect of the multiplication, accumulated in the local matrix m2, is then copied into the transformation matrix.

Matrix/Vector Representation of Translations

A translation can also be represented by a pair of numbers, t=(tx, ty) where tx is the change in the x-coordinate and ty is the modify in y coordinate. To translate the point p by t, we simply add to obtain the new (translated) point q = p + t

$$q = p + t = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix} = \begin{bmatrix} x + tx \\ y + ty \end{bmatrix}$$

Did u know?

APL (form of metrics), designed by Ken Iverson, was the first programming language to provide array programming capabilities.

Notes

Self Assessment Questions

1. Translation is a simple transformation that is calculated directly from the conversion matrix for ……., one translate of the other.

   (*a*) one frames (*b*) two frames

   (*c*) three frames (*d*) None of these

2. Transformation is classified:

   (*a*) Matrix (*b*) Frame

   (*c*) Scaling (*d*) None of these

3. 3-Dimensional transformation it has three axes Tx, Ty, Tz depending upon their coordinate.

   (*a*) True (*b*) False

4. A translation can also be represented by a pair of numbers……..

   (*a*) t = (tx, ty) (*b*) t = (x, y)

   (*c*) t = (x + y) (*d*) None of these

5. Where Tx is the change in the y-coordinate.

   (*a*) True (*b*) False

9.4 Translating Addresses

Things are really bad than just physical and virtual addresses. Though, on many setups, there are really three different ways of gazing at memory addresses, and here we actually want the third, the so-called "bus address".

Essentially, the three ways of addressing memory are:

- CPU untranslated. This is the "physical" address, ie physical address 0 is what the CPU sees when it drives zeroes on the memory bus.
- CPU translated address. This is the "virtual" address, and is completely internal to the CPU itself with the CPU doing the appropriate translations into "CPU untranslated".
- Bus address. This is the address of memory as seen by other devices, not the CPU. Now, in theory there could be many different bus addresses, with each device seeing memory in some device-specific way, but happily most hardware designers are not actually actively trying to make things any more complex than necessary, so you can assume that all external hardware sees the memory the same way.

Now, on normal PC's, the bus address is exactly the same as the physical address and things are very simple indeed. However, they are that simple because the memory and the devices share the same address space, and that is not generally necessarily true on other PCI/ISA setups.

Translation of Virtual Addresses

Computer graphics processor able of reading from, and writing to, virtual memory. The invention provides a graphics processing unit which includes, among other things, a graphic processor in the form of an address generator which retrieves data from memory locations, and writes data to memory locations. The address generator retrieves data from memory locations memory access request directly to a memory control unit, which retrieves the contents of the memory location. Prior to issuing the request, the address generator sends the address to a virtual translation unit, which translates the virtual address to a physical address. The virtual translation/FIFO

control unit also contains three translation buffers, in which are stored the most recently accessed virtual addresses, which, in many situations, enables the virtual translation/FIFO control unit to translate the virtual address using less memory accesses.

Notes

9.5 The Inverse of a Translation

To undo a translation by t_x , t_y, t_z apply the matrix

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{bmatrix}$$

We can now complete scaling about an arbitrary fixed point and rotation about an arbitrary turn. To scale about F= [x_f y_f z_f] use the composition of matrices

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_f & -y_f & -z_f & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_f & y_f & z_f & 1 \end{bmatrix}$$

when multiplied out yields?

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ x_f(1-s_x) & y_f(1-s_y) & z_f(1-s_z) & 1 \end{bmatrix}$$

So a scaled point [X Y Z 1] becomes,

$$[x' \; y' \; z' \; 1] = [x \; y \; z \; 1] \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ x_f(1-s_x) & y_f(1-s_y) & z_f(1-s_z) & 1 \end{bmatrix}$$

In a similar manner you can determine that rotation about a pivot R = [Xr Y_r] results in

$$x' = x_r + (x - y_r) \cos \theta - (y - y_r) \sin \theta$$

$$y' = y_r + (x - y_r) \cos \theta - (x - x_r) \sin \theta$$

Self Assessment Questions

6. Translation is one of the …… a translation moves all points of an object.
7. Simple straight line is movement of the …….. in x and y direction.
8. This function takes as parameters a reference to …… that holds the current state of the transformation and the X and Y translation values.
9. ……..... capable of reading from, and writing to virtual memory.
10. We can develop the matrix involved in a straight forward manner by considering the translation of a ……...
11. CPU translated address is known as ……

    (*a*) virtual address. (*b*) physical address.

    (*c*) bus address. (*d*) None of these.

Notes

12. Corrects meaning is the address of memory…….

    (*a*) physical address. (*b*) bus address.

    (*c*) Both a and b (*d*) None of these.

13. A bilinear transformation can be simulated by the ……

    (*a*) transformation, rotation and stretching

    (*b*) translation and rotation

    (*c*) rotation, stretching and inversion

    (*d*) rotation, stretching, inversion and translation.

14. What is the translating address?

    (*a*) Physical and virtual addresses (*b*) Bus address.

    (*c*) Physical address. (*d*) Virtual address.

15. What is address generator retrieves data from memory locations?

    (*a*) Memory locations. (*b*) Memory control unit

    (*c*) Bus address (*d*) None of these.

9.6 Summary

- Translation is one of the simplest transformations. A translation moves all points of an object a fixed distance in a specified direction.
- Translation is a simple straight line movement of the object in x and y direction. Transformation is refer to transform from one position to another position depends upon their transformation.
- The address generator retrieves data from memory locations memory access request directly to a memory control unit.
- The bus address is exactly the same as the physical address, and things are very simple indeed.

9.7 Keywords

Bus Address: This is the address of memory as seen by other devices, not the CPU.

Coordinates of Point: Given a frame $F = (\vec{u}, \vec{v}, \vec{w}, o)$ and a point P that has coordinates (u, v, w) in F, if we apply the transformation to the coordinates of the point.

Matrix/Vector of the Translations: A translation can also be represented by a pair of numbers, t=(tx, ty) where tx is the change in the x-coordinate and ty is the change in y coordinate.

Translation Matrix: This function takes as parameters a reference to the matrix that holds the current state of the transformation and the X and Y translation values.

9.8 Review Questions

1. Write a C program in 2Dimensional of the translation.
2. Write a C program in 3Dimensional of the translation.

3. How the 4×4 matrix does implements a frame?
4. What is the process of translating addresses?
5. Explain the translation of virtual addresses.
6. Discuses the inverse of a translation.
7. Differences between matrix and vector.

Notes

Answers for Self Assessment Questions

1. (*b*) 2. (*c*) 3. (*b*) 4. (*c*) 5. (*b*)
6. simplest transformation 7. Object 8. the matrix.
9. Computer graphics processor 10. single frame.
11. (*a*) 12. (*b*) 13. (*d*) 14. (*a*) 15. (*b*)

9.10 Further Readings

Books *Computer Graphics, C Version*, by Hearn

Computer Graphics: *Principles & Practice In. C*, by Foley

Online link http://books.google.co.in/books?id=dvrTIelDhJUC&printsec=frontcover&dq=translation+in+computer+graphics+in+c&source=bl&ots=NE1jThlVlr&sig=i9ehKHHCQWKGZPnINDbdZ_Oe1kw&hl=en&sa=X&ei=KtAPUL6tLMjirAeRu4CICQ&ved=0CDAQ6AEwAA#v=onepage&q=translation%20in%20computer%20graphics%20in%20c&f=false

Notes

Unit 10: Shearing

CONTENTS

Objectives

After studying this unit, you will be able to:

- Define shearing with C programming
- Explain rotation with C programming
- Describe reflection with C programming

Introduction

A shearing transformation revolves one axis in order that the x-axis and y-axis are no longer perpendicular. The coordinates of the node are moved by the specified multipliers. To shear, use the Shear class or the shear function of the Transform class. In the Xylophone application, you can shear the xylophone by pulling the mouse while holding Shift and pressing the left mouse button.

10.1 Shearing

Shearing transformations in three-dimensions alter two of the three coordinate values proportionally to the value of the third coordinate.

10.1.1 The X-Shear Transformation

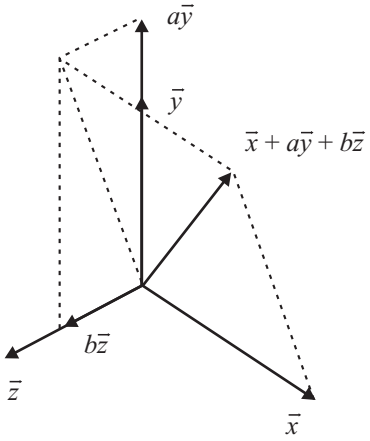
Given a frame $F = (\vec{u}, \vec{v}, \vec{w}, O)$ we "x-shear" a frame by modifying the first vector of the frame by including to it a linear arrangement of the other two vectors. The frame transformation takes the following form:

Notes

$$\begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix} \rightarrow \begin{bmatrix} \vec{u} + a\vec{v} + b\vec{w} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix}$$

An illustration of this process is given in the following Figure 10.1.

Figure 10.1: The X-shear Transformation



This transform can be implemented by the following 4 × 4 matrix:

$$\begin{bmatrix} 1 & a & b & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix} = \begin{bmatrix} \vec{u} + a\vec{v} + b\vec{w} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix}$$

And so we define the x-shear transformation by

$$\boldsymbol{H}_{x;a,b} = \begin{bmatrix} 1 & a & b & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If this transformation is applied to the point (u, v, w), we obtain:

$$[u \quad v \quad w \quad 1] \begin{bmatrix} 1 & a & b & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [u \quad au + v \quad bu + w \quad 1]$$

and thus objects can be shared by applying this matrix to all points of the object.

10.1.2 The Y-shear Transformation

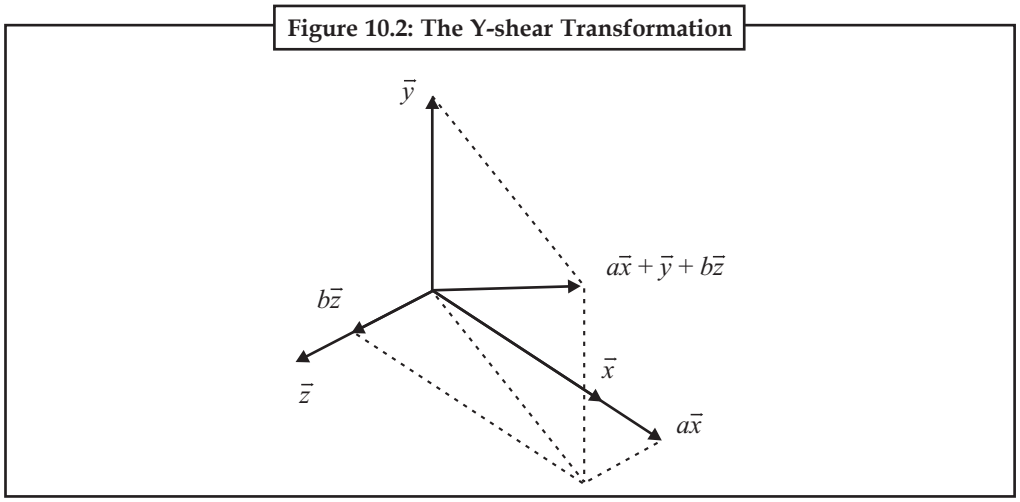
Given a frame $F = (\vec{u}, \vec{v}, \vec{w}, O)$ we "y-shear" a frame by transforming the second vector by adding a linear combination of the other two vectors. The frame transformation takes the following form:

$$= \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix} \rightarrow \begin{bmatrix} \vec{u} \\ a\vec{u} + \vec{v} + b\vec{w} \\ \vec{w} \\ \mathbf{O} \end{bmatrix}$$

Notes

An illustration of this process is given in the following Figure 10.2.

Figure 10.2: The Y-shear Transformation



This transform can be implemented by the following 4 × 4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ a & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathrm{O} \end{bmatrix} = \begin{bmatrix} \vec{u} \\ a\vec{u}+\vec{v}+b\vec{w} \\ \vec{w} \\ \mathrm{O} \end{bmatrix}$$

and so we define the y-shear transformation by

$$H_{y;a,b} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ a & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If this transformation is applied to the point (u, v, w), we obtain:

$$[u \quad v \quad w \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ a & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [u + av \quad v \quad w + bv \quad 1]$$

and thus objects can be sheared by applying this matrix to all points of the object.

10.1.3 The Z-shear Transformation

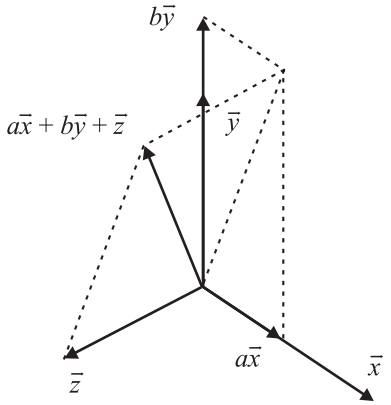
Given a frame F = $(\vec{u}, \vec{v}, \vec{w}, O)$ we "z-shear" a frame by transforming the third vector by adding a linear combination of the other two vectors. The frame transformation gets the following form

$$\begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix} \rightarrow \begin{bmatrix} \vec{u} \\ \vec{v} \\ a\vec{u}+b\vec{v}+\vec{w} \\ \mathbf{O} \end{bmatrix}$$

This is illustrated by Figure 10.3.

Notes

Figure 10.3: The Z-shear Transformation



This transform can be implemented by the following 4 × 4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \\ \mathbf{O} \end{bmatrix} = \begin{bmatrix} \vec{u} \\ \vec{v} \\ a\vec{u} + b\vec{v} + \vec{w} \\ \mathbf{O} \end{bmatrix}$$

and so we define the z-shear transformation by

$$H_{z;a,b} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If this transformation is applied to the point (u, v, w), we obtain

$$[u \quad v \quad w \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [v + aw \quad v \quad bw + w \quad 1]$$

and thus objects can be sheared by applying this matrix to all points of the object.

10.1.4 2D Transformations

To write a C program to perform 2D transformations such as shearing Algorithm:

(a) Input the shearing factors shx and shy.

(b) Shearing related to x axis: Transform coordinates x_1=x+shx*y and y_1=y.

(c) Shearing related to y axis: Transform coordinates x_1=x and y_1=y+shy*x.

(d) Input the x ref and y ref values.

(e) X axis shear related to the reference line y–y ref is:

x_1 = x + shx(y – y ref) and y_1 = y.

(f) Y axis shear related to the reference line x = x ref is x_1 = x and

(g) Display the object after shearing.

Notes

Source code for Shearing Algorithm C Programming

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
void shearing ()
{
int sh;
float xn1, yn1, xn2, yn2;
printf ("Enter the value of Shearing");
scanf("%d",&sh);
cleardevice();
outtextxy(500,100, "SHEARING");
xn1=x1+sh*y1;
yn1=y1;
xn2=x2+sh*y2;
yn2=y2;
line(x1,y1,x2,y2);
line(xn1,yn1,xn2,yn2);
getch();
}
```

10.2 Rotation

Rotation of a planar body is the movement when points of the body travel in circular trajectories around a fixed point called the centre of rotation. For a three-dimensional body, the rotation is around an axis it quantity to rotation in each plane perpendicular to the axis around the intersection of the plane and the axis.

An example of rotation of a planar figure around a point is the movement of the propeller of an aircraft. A door attached to the wall by two or more hinges rotates around the axis going through the hinges.

If the axis of rotation is within the body, the body is said to rotate upon itself, or spin. Among such rotations, the simplest case is that of constant angular frequency.

A rotation is a rigid body movement which keeps a point fixed; unlike a translation. This definition is applicable both for rotations in a plane (two dimensions) and in space (three dimensions). It turns out that a rotation in the three-dimensional space keeps fixed not just a single point, but rather an entire line; that is to say, any rotation in the three dimensional space is a rotation around an axis.

If one does a rotation around a point (axis), followed by another rotation around the same point (axis), the total result is yet another rotation. The reverse (inverse) of a rotation is also a rotation. It follows that the rotations around a point or axis form a group. If however one performs rotation around a point (axis) followed by rotation around another point (axis), the overall movement may not be a rotation anymore.

Notes

Rotations around the x, y and z axes are called principal rotations. Rotation around any axis can be performed by taking a rotation around the x axis, followed by a rotation around the y axis, and followed by a rotation around the z axis. That is to say, any spatial rotation can be decomposed into a combination of principal rotations.

2D Rotation Program Using C Programming

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<process.h>
#include<math.h>
void Triangle(int x1,int y1,int x2,int y2,int x3,int y3);
void Rotate(int x1,int y1,int x2,int y2,int x3,int y3);
void main ()
{
int gd=DETECT,gm;
int x1,y1,x2,y2,x3,y3;
initgraph(&gd,&gm, "");
printf("Enter the 1st point for the triangle:");
scanf("%d%d",&x1,&y1);
 printf("Enter the 2nd point for the triangle:");
scanf("%d%d",&x2,&y2);
 printf("Enter the 3rd point for the triangle:");
scanf("%d%d",&x3,&y3);
TriAngle(x1,y1,x2,y2,x3,y3);
getch();
cleardevice();
Rotate(x1,y1,x2,y2,x3,y3);
setcolor(1);
TriAngle(x1,y1,x2,y2,x3,y3);
getch();
}
void TriAngle(int x1,int y1,int x2,int y2,int x3,int y3)
{
line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x1,y1);
}
void Rotate(int x1,int y1,int x2,int y2,int x3,int y3)
{
int x,y,a1,b1,a2,b2,a3,b3,p=x2,q=y2;
float Angle;
printf("Enter the angle for rotation:");
```

Notes

```
scanf("%f",&Angle);
cleardevice();
Angle=(Angle*3.14)/180;
a1=p+(x1 – p)*cos(Angle) – (y1 – q)*sin(Angle);
b1=q+(x1 – p)*sin(Angle)+(y1 – q)*cos(Angle);
a2=p+(x2 – p)*cos(Angle) – (y2 – q)*sin(Angle);
b2=q+(x2 – p)*sin(Angle)+(y2 – q)*cos(Angle);
```

Self Assessment Questions

1. Shearing transformations in two-dimensions alter two of the three coordinate values proportionally to the value of the third coordinate.

   (*a*) True (*b*) False

2. Given a frame we …….. a frame by transforming the third vector by adding a linear combination of the other two vectors.

   (*a*) (x,y,z)-shear (*b*) x-shear

   (*c*) y-shear (*d*) z-shear

3. Shearing related to x axis: Transform coordinates ……and y1=y.

   (*a*) x1=shx*y (*b*) x1=x+shx*y

   (*c*) x1=x+shx*x (*d*) None of these

4. Rotations around the x, y and z axes are called……...

   (*a*) axis of rotation (*b*) center of rotation

   (*c*) origin of rotation (*d*) principal rotations

5. X axis shear related to the reference line y–y ref is x1=x+shy(y – y ref) and y1=y.

   (*a*) True (*b*) False

10.3 Reflection

Reflection in computer graphics is used to track reflective objects alike to mirrors and shiny surfaces. Reflection is achieved in a ray draw renderer by subsequent a ray from the eye to the mirror and then calculating where it bounces from, and continuing the process awaiting no surface is found, or a non-reflective surface is found. Reflection on a shiny surface like wood or tile can add to the photorealistic effects of a 3D rendering.

Polished: A Polished Reflection is an undisturbed reflection, like a mirror or chrome.

Blurry: A Blurry Reflection means that tiny random bumps on the surface of the material cause the reflection to be blurry.

Metallic: A reflection is metallic if the highlights and reflections retain the colour of the reflective object.

Glossy: This term can be misused. Sometimes it is a setting which is the opposite of Blurry. (When "Glossiness" has a low value, the reflection is blurry.) However, some people use the term "Glossy Reflection" as a synonym for "Blurred Reflection." Glossy used in this context means that the reflection is actually blurred.

10.3.1 2D Transformations Such As Reflection Algorithms

Notes

1. Start
2. Initialize the graphics mode.
3. Construct a 2D object (use Drawpoly()) e.g. (x, y)
   a. About X axis
      i. x′= x, y′ = –y
      ii. Plot (x′, y′)
   b. About Y axis
      i. x′= –x, y′=y
      ii. Plot (x′, y′)

Source Code for Reflection Algorithm C Programming

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<graphics.h>
int x1,y1,x2,y2;
void reflection()
{
int xn1,yn1,xn2,yn2;
cleardevice();
outtextxy(300,100, "REFLECTION");
if((x1<y1)^(x2<y2))
{
xn1=x1+50;
xn2=x2+50;
yn1=y1;
yn2=y2;
}
else
{
xn1=x1;
xn2=x2;
yn1=y1+50;
yn2=y2+50;
}
line(x1,y1,x2,y2);
line(xn1,yn1,xn2,yn2);
getch();
}
```

Notes

Self Assessment Questions

6. A ……. is metallic if the highlights and reflections retain the colour of the reflective object.
7. The basic element of a picture in volume graphics is ……..
8. A circle, if scaled only in one direction becomes a …….
9. Reflection of a point about x-axis, followed by a counter-clockwise rotation of 900, is equivalent to reflection about the line ……
10. A …….. Reflection is an undisturbed reflection, like a mirror or chrome.
11. Reflection on a shiny surface like wood or tile can add to the photorealistic effects of a …… rendering.
12. If the axis of rotation is within the body, the body is said to rotate upon itself, or spin.

    (*a*) True (*b*) False

13. Rotations around the x, y and z axes are called principal rotations.

    (*a*) True (*b*) False

14. If one does a rotation around a point (axis), followed by another rotation around the same point (axis), the total result is yet another rotation.

    (*a*) True (*b*) False

15. A shearing transformation rotates one axis so that the …….. and …….. are no longer perpendicular.

10.4 Summary

- Rotation of a planar body is the movement when points of the body travel in circular trajectories around a fixed point called the centre of rotation.
- A Blurry reflection means that tiny random bumps on the surface of the material cause the reflection to be blurry.
- A polished reflection is an undisturbed reflection, like a mirror or chrome.
- A reflection is metallic if the highlights and reflections retain the colour of the reflective object.

10.5 Keywords

Blurry: A Blurry Reflection means that tiny random bumps on the surface of the material cause the reflection to be blurry.

Metallic: A reflection is metallic if the highlights and reflections retain the colour of the reflective object.

Polished: A Polished Reflection is an undisturbed reflection, like a mirror or chrome.

Rotation: Rotation of a planar body is the movement when points of the body travel in circular trajectories around a fixed point called the center of rotation.

10.6 Review Questions

Notes

1. Define the shearing.
2. What is rotation?
3. What is reflection?
4. Define the 2D rotation program using c programming.
5. Write the reflection algorithms.
6. Discuss the Z-shear transformation.
7. Write the source code for reflection algorithm C programming.
8. Write the source code for shearing algorithm C programming.
9. Explain X- shears transformation.
10. Define Y-shear transformation.

Answers for Self Assessment Questions

1.	(*a*)	2.	(*d*)	3.	(*b*)	4.	(*d*)	5.	(*b*)
6.	reflection	7.	voxel	8.	ellipse	9.	x = y	10.	Polished
11.	3D	12.	(*a*)	13.	(*a*)	14.	(*a*)	15.	x-axis, y-axis

10.7 Further Readings


Books

Computer Graphics by, D.P Mukherjee, Debasish Jana

Mathematics for Computer Graphics, by John A. Vince


Online link

http://books.google.co.in/booss?id=ShmRZtw9gBwC&printsec=frontcover&dq=Further+Reading+Sharing+in+computer+graphics&source=bl&ots=mKcYRjcph5&sig=fgzZIehXVpElZIbnWX_cW5Z1oOk&hl=en&sa=X&ei=_v8QUMTvGs2rrAe96YDoBA&ved=0CFkQ6AEwBg#v=onepage&q&f=false

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in