M.C.A.

SEMESTER - IV (CBCS)

COMPUTER GRAPHICS AND IMAGE PROCESSING

SUBJECT CODE : MCAL402

Prof. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai.

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai.

Prof. Prakash Mahanwar
Director
IDOL, University of Mumbai.

Programme Co-ordinator: **Prof. Mandar Bhanushe**
Head, Faculty of Science & Technology,
IDOL, University of Mumbai - 400 098.

Course Co-ordinator : **Ms. Reshma Kurkute**
Assistant Professor B.Sc.IT, IDOL,
IDOL, University of Mumbai- 400098.

Course Writers : **Mr. Sandeep Kamble**
Assistant Professor, Cosmopolitan's Valia College.

: **Ms. Geeta Sahu**
Assistant Professor, Vidyalankar School of Information Technology.

: **Mr. Ashish Shah**
BSc. I.T Coordinator J. M. PATEL College of Commerce.

: **Mr. Ahtesham Shaikh**
Assistant Professor, Anjuman-i-Islam's Akbar Peerbhoy College.

: **Ms. Anjali A Gaikwad**
Assistant Professor, JES college of commerce, science and I.T.

: **Mrs. Jayshri R Parab**
Assistant Professor, Lala Lajpatrai College.

: **Mr. Rohan K Parab**
AssistantProfessor, Guru Nanak Khalsa College.

May 2022, Print I

Published by
Director
Institute of Distance and Open learning ,
University of Mumbai,Vidyanagari, Mumbai - 400 098.

DTP COMPOSED AND PRINTED BY
Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai - 400098.

CONTENT

Chapter No.	Title	Page No.
Module I		
1.	Introduction To Graphics	1-11
2.	Demonstration of Simple Graphic Inbuilt Function	12-22
Module II		
3.	Output Primitives & Its Algorithms	23-38
Module III		
4.	Output Primitives & Its Algorithms	39-54
Module IV		
5.	Output Primitives & Its Algorithms	55-67
Unit V		
6.	Output Primitives & Its Algorithm	68-80
Module VI		
7.	Output Primitives & Its Algorithms	81-96
Module VII		
8.	2d Geometric Transformations & Clipping	97-114
Module VIII		
9.	2d Geometric Transformations & Clipping	115-139
Module IX		
10.	Implementation of 3d Transformations (Only Coordinates Calculation)	140-148
Unit X		
11.	Output Primitives & Its Algorithm	149-162
Module XI		
12.	Introduction To Animation	163-171
Module XII		
13.	Image Enhancement Transformation	172-177
14.	Image Enhancement Transformation	178-187
15.	Image Enhancement Transformation	188-198

*****

Syllabus

Course Code	Course Name
MCAL402	Computer Graphics and Image Processing

Sr. no	Module	Detailed Contents	Hours
01	**Introduction**	Introduction to graphics coordinates system and demonstration of simple inbuilt graphic functions	**01**
02	**Output primitives & its algorithms**	Implementation of line generation A. A. DDA line B. Bresenhams line C. Application of Line drawing algos.	**02**
03	**Output primitives & its Algorithms**	Implementation of circle drawing A. Midpoint circle B. Application of Circle drawing algos.	**03**
04	**Output primitives & its Algorithms**	Implementation of ellipse drawing A. Midpoint Ellipse	**04**
05	**Output primitives & its Algorithms**	Implementation of curve drawing A. Bezier Curve	**05**
06	**Output primitives & its Algorithms**	Implementation of filling algorithms A. Boundary fill B. Flood fill C. Scan line D. application of Circle drawing algos.	application of Circle drawing algos.
07	**2DGeometric Transformations & Clipping**	Implementation of two dimensional transformations A. Translation, Rotation & Scaling B. Shear & Reflection	**6**
08	**2D Geometric Transformations& Clipping**	Implementation of clipping algorithms A. Cohen Sutherland Line clipping B. Midpoint Subdivision C. Sutherland Hodgeman Polygon Clipping	**10**
09	**Basic 3D Concepts & Fractals**	Implementation of 3D Transformations (only coordinates calculation)	**2**
10	**Basic 3D Concepts &**	Implementation of fractal generation	**6**

	Fractals	A. Koch curve/Snowflake B. Sirepenski Triangle	
11	**Introduction of Animation**	Implementation of animation programs (using basic inbuilt Graphical functions)	**4**
12	**Image Enhancement Techniques**	Implementation of Basic Intensity Transformations A. Image negative B. Log transformation C. Power law Transformation	**6**
13	**Image Enhancement Techniques**	Implementation of Piecewise-Linear Transformation Functions A. Contrast Stretching B. Grey level Slicing C. Bit plane slicing	**8**
14	**Image Enhancement Techniques**	Implementation of histogram equalization A. Image histogram & histogram Equalization B. Image Subtraction C. Image averaging	10

*****

MODULE I

1

INTRODUCTION TO GRAPHICS

Unit Structure

1.0 Objectives
1.1 Introduction
1.2 Summary
1.3 References
1.4 Unit End Exercises

1.0 OBJECTIVE

Graphics are defined as any sketch or a drawing or a special network that pictorially represents some meaningful information. Computer Graphics is used where a set of images needs to be manipulated or the creation of the image in the form of pixels and is drawn on the computer. Computer Graphics can be used in digital photography, film, entertainment, electronic gadgets, and all other core technologies which are required. It is a vast subject and area in the field of computer science. Computer Graphics can be used in UI design, rendering, geometric objects, animation, and many more. In most areas, computer graphics is an abbreviation of CG. There are several tools used for the implementation of Computer Graphics. The basic is the <graphics.h> header file in Turbo-C, Unity for advanced and even OpenGL can be used for its Implementation. It was invented in 1960 by great researchers Verne Hudson and William Fetter from Boeing.

Computer Graphics refers to several things:

- The manipulation and the representation of the image or the data in a graphical manner.
- Various technology is required for the creation and manipulation.
- Digital synthesis and its manipulation.

1.1 INTRODUCTION

Types of Computer Graphics:

- **Raster Graphics:** In raster, graphics pixels are used for an image to be drawn. It is also known as a bitmap image in which a sequence of images is into smaller pixels. Basically, a bitmap indicates a large number of pixels together.

- **Vector Graphics:** In vector graphics, mathematical formulae are used to draw different types of shapes, lines, objects, and so on.

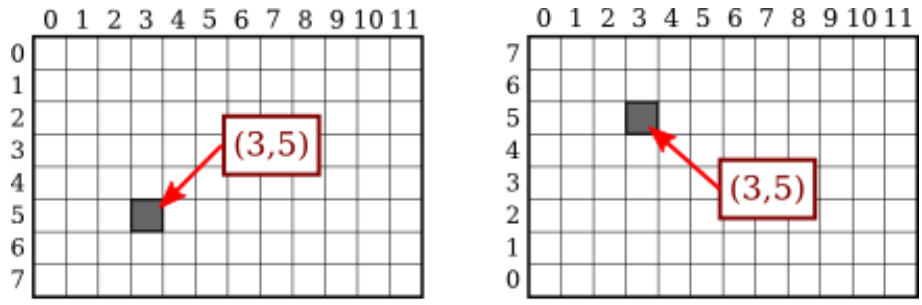
Applications:

- **Computer Graphics are used for an aided design for engineering and architectural system:** These are used in electrical automobiles, electro-mechanical, mechanical, electronic devices. For example gears and bolts.
- **Computer Art:** MS Paint.
- **Presentation Graphics:** It is used to summarize financial statistical scientific or economic data. For example- Bar chart, Line chart.
- **Entertainment:** It is used in motion pictures, music videos, television gaming.
- **Education and training:** It is used to understand the operations of complex systems. It is also used for specialized system such for framing for captains, pilots and so on.
- **Visualization:** To study trends and patterns.For example- Analyzing satellite photo of earth.

1.2 SUMMARY

Pixel Coordinates:

A digital image is made up of rows and columns of pixels. A pixel in such an image can be specified by saying which column and which row contains it. In terms of coordinates, a pixel can be identified by a pair of integers giving the column number and the row number. For example, the pixel with coordinates (3,5) would lie in column number 3 and row number 5. Conventionally, columns are numbered from left to right, starting with zero. Most graphics systems, including the ones we will study in this chapter, number rows from top to bottom, starting from zero. Some, including OpenGL, number the rows from bottom to top instead.



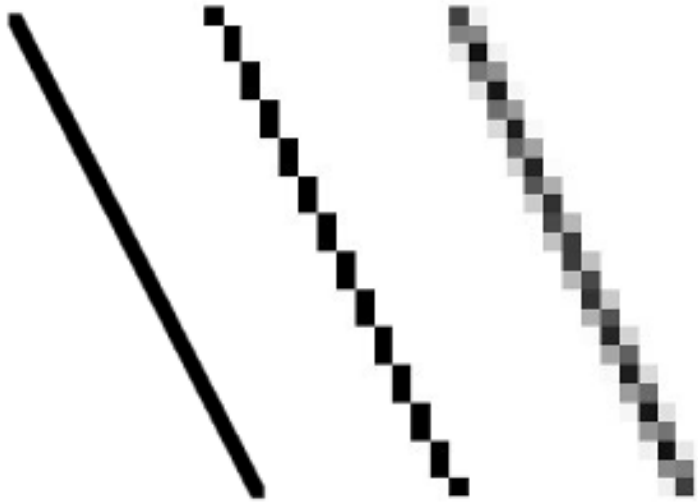
12-by-8 pixel grids, shown with row and column numbers.
On the left, rows are numbered from top to bottom,
on the right, they are numberd bottom to top.

Note in particular that the pixel that is identified by a pair of coordinates (x,y) depends on the choice of coordinate system. You always need to know what coordinate system is in use before you know what point you are talking about.

Row and column numbers identify a pixel, not a point. A pixel contains many points; mathematically, it contains an infinite number of points. The goal of computer graphics is not really to color pixels—it is to create and manipulate images. In some ideal sense, an image should be defined by specifying a color for each point, not just for each pixel. Pixels are an approximation. If we imagine that there is a true, ideal image that we want to display, then any image that we display by coloring pixels is an approximation. This has many implications.

Suppose, for example, that we want to draw a line segment. A mathematical line has no thickness and would be invisible. So we really want to draw a thick line segment, with some specified width. Let's say that the line should be one pixel wide. The problem is that, unless the line is horizontal or vertical, we can't actually draw the line by coloring pixels. A diagonal geometric line will cover some pixels only partially. It is not possible to make part of a pixel black and part of it white. When you try to draw a line with black and white pixels only, the result is a jagged staircase effect. This effect is an example of something called "aliasing." Aliasing can also be seen in the outlines of characters drawn on the screen and in diagonal or curved boundaries between any two regions of different color. (The term aliasing likely comes from the fact that ideal images are naturally described in real-number coordinates. When you try to represent the image using pixels, many real-number coordinates will map to the same integer pixel coordinates; they can all be considered as different names or "aliases" for the same pixel.)

Antialiasing is a term for techniques that are designed to mitigate the effects of aliasing. The idea is that when a pixel is only partially covered by a shape, the color of the pixel should be a mixture of the color of the shape and the color of the background. When drawing a black line on a white background, the color of a partially covered pixel would be gray, with the shade of gray depending on the fraction of the pixel that is covered by the line. (In practice, calculating this area exactly for each pixel would be too difficult, so some approximate method is used.) Here, for example, is a geometric line, shown on the left, along with two approximations of that line made by coloring pixels. The lines are greatly magnified so that you can see the individual pixels. The line on the right is drawn using antialiasing, while the one in the middle is not:



Note that antialiasing does not give a perfect image, but it can reduce the "jaggies" that are caused by aliasing (at least when it is viewed on a normal scale).

There are other issues involved in mapping real-number coordinates to pixels. For example, which point in a pixel should correspond to integer-valued coordinates such as (3,5)? The center of the pixel? One of the corners of the pixel? In general, we think of the numbers as referring to the top-left corner of the pixel. Another way of thinking about this is to say that integer coordinates refer to the lines between pixels, rather than to the pixels themselves. But that still doesn't determine exactly which pixels are affected when a geometric shape is drawn. For example, here are two lines drawn using HTML canvas graphics, shown greatly magnified. The lines were specified to be colored black with a one-pixel line width

The top line was drawn from the point (100,100) to the point (120,100). In canvas graphics, integer coordinates corresponding to the lines between pixels, but when a one-pixel line is drawn, it extends one-half pixel on either side of the infinitely thin geometric line. So for the top line, the line as it is drawn lies half in one row of pixels and half in another row. The graphics system, which uses antialiasing, rendered the line by coloring both rows of pixels gray. The bottom line was drawn from the point (100.5,100.5) to (120.5,120.5). In this case, the line lies exactly along one line of pixels, which gets colored black. The gray pixels at the ends of the bottom line have to do with the fact that the line only extends halfway into the pixels at its endpoints. Other graphics systems might render the same lines differently.

All this is complicated further by the fact that pixels aren't what they used to be. Pixels today are smaller! The resolution of a display device can be measured in terms of the number of pixels per inch on the display, a quantity referred to as PPI (pixels per inch) or sometimes DPI (dots per inch). Early screens tended to have resolutions of somewhere close to 72 PPI. At that resolution, and at a typical viewing distance, individual pixels are clearly visible. For a while, it seemed like most displays had about 100 pixels per inch, but high resolution displays today can have 200, 300 or even 400 pixels per inch. At the highest resolutions, individual pixels can no longer be distinguished.

The fact that pixels come in such a range of sizes is a problem if we use coordinate systems based on pixels. An image created assuming that there are 100 pixels per inch will look tiny on a 400 PPI display. A one-pixel-wide line looks good at 100 PPI, but at 400 PPI, a one-pixel-wide line is probably too thin.

In fact, in many graphics systems, "pixel" doesn't really refer to the size of a physical pixel. Instead, it is just another unit of measure, which is set by the system to be something appropriate. (On a desktop system, a pixel is usually about one one-hundredth of an inch. On a smart phone, which is usually viewed from a closer distance, the value might be closer to 1/160 inch. Furthermore, the meaning of a pixel as a unit of measure can change when, for example, the user applies a magnification to a web page.)

Pixels cause problems that have not been completely solved. Fortunately, they are less of a problem for vector graphics, which is mostly what we will use in this book. For vector graphics, pixels only become an issue during rasterization, the step in which a vector image is converted into pixels for display. The vector image itself can be created using any convenient coordinate system. It represents an idealized, resolution-independent image. A rasterized image is an approximation of that ideal image, but how to do the approximation can be left to the display hardware.

Real-number Coordinate Systems:

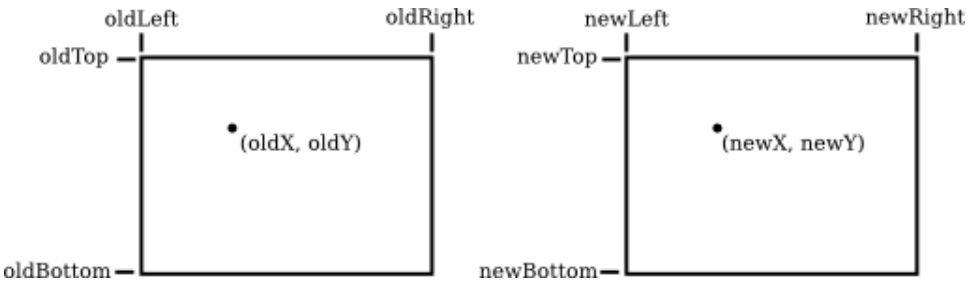
When doing 2D graphics, you are given a rectangle in which you want to draw some graphics primitives. Primitives are specified using some coordinate system on the rectangle. It should be possible to select a coordinate system that is appropriate for the application. For example, if the rectangle represents a floor plan for a 15 foot by 12 foot room, then you might want to use a coordinate system in which the unit of measure is one foot and the coordinates range from 0 to 15 in the horizontal direction and 0 to 12 in the vertical direction. The unit of measure in this case is feet rather than pixels, and one foot can correspond to many pixels in the image. The coordinates for a pixel will, in general, be real numbers rather than integers. In fact, it's better to forget about pixels and just think about points in the image. A point will have a pair of coordinates given by real numbers.

To specify the coordinate system on a rectangle, you just have to specify the horizontal coordinates for the left and right edges of the rectangle and the vertical coordinates for the top and bottom. Let's call these values left, right, top, and bottom. Often, they are thought of as xmin, xmax, ymin, and ymax, but there is no reason to assume that, for example, top is less than bottom. We might want a coordinate system in which the vertical coordinate increases from bottom to top instead of from top to bottom. In that case, top will correspond to the maximum y-value instead of the minimum value.

To allow programmers to specify the coordinate system that they would like to use, it would be good to have a subroutine such as

setCoordinateSystem(left,right,bottom,top)

The graphics system would then be responsible for automatically transforming the coordinates from the specfiied coordinate system into pixel coordinates. Such a subroutine might not be available, so it's useful to see how the transformation is done by hand. Let's consider the general case. Given coordinates for a point in one coordinate system, we want to find the coordinates for the same point in a second coordinate system. (Remember that a coordinate system is just a way of assigning numbers to points. It's the points that are real!) Suppose that the horizontal and vertical limits are oldLeft, oldRight, oldTop, and oldBottom for the first coordinate system, and are newLeft, newRight, newTop, and newBottom for the second. Suppose that a point has coordinates (oldX,oldY) in the first coordinate system. We want to find the coordinates (newX,newY) of the point in the second coordinate system



Formulas for newX and newY are then given by

newX = newLeft +

((oldX - oldLeft) / (oldRight - oldLeft)) * (newRight - newLeft))

newY = newTop +

((oldY - oldTop) / (oldBottom - oldTop)) * (newBotom - newTop)

the same fraction of the distance from newLeft to newRight. You can also check the formulas by testing that they work when oldX is equal to oldLeft or to oldRight, and when oldY is equal to oldBottom or to oldTop. As an example, suppose that we want to transform some real-number

coordinate system with limits left, right, top, and bottom into pixel coordinates that range from 0 at left to 800 at the right and from 0 at the top 600 at the bottom. In that case, newLeft and newTop are zero, and the formulas become simply

newX = ((oldX - left) / (right - left)) * 800

newY = ((oldY - top) / (bottom - top)) * 600

Of course, this gives newX and newY as real numbers, and they will have to be rounded or truncated to integer values if we need integer coordinates for pixels. The reverse transformation—going from pixel coordinates to real number coordinates—is also useful. For example, if the image is displayed on a computer screen, and you want to react to mouse clicks on the image, you will probably get the mouse coordinates in terms of integer pixel coordinates, but you will want to transform those pixel coordinates into your own chosen coordinate system.

Example:

Aim: Write a program to draw a circle in C graphics

Code:

The header file graphics.h contains **circle()** function which draws a circle with center at (x, y) and given radius.

Syntax:

circle(x, y, radius);

where,

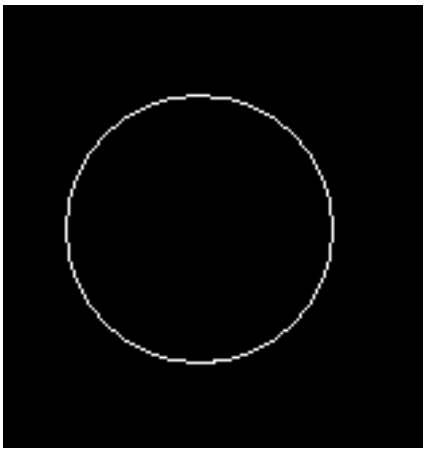
(x, y) is center of the circle.

'radius' is the Radius of the circle.

Examples:

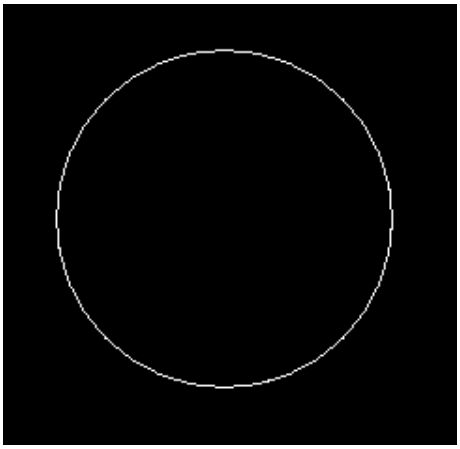
Input: x = 250, y = 200, radius = 50

Output:



Input: x = 300, y = 150, radius = 90

Output:

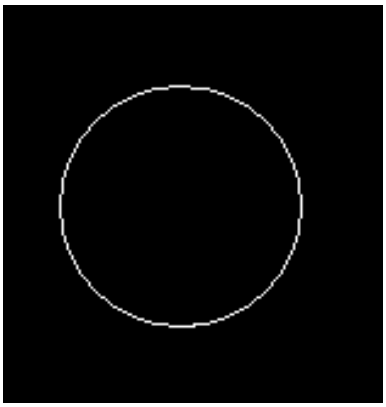


Below is the implementation to draw circle in C:

```
// C Implementation for drawing circle
#include <graphics.h>
//driver code
int main()
{
// gm is Graphics mode which is
// a computer display mode that
// generates image using pixels.
// DETECT is a macro defined in
// "graphics.h" header file
int gd = DETECT, gm;
// initgraph initializes the
// graphics system by loading a
// graphics driver from disk
initgraph(&gd, &gm, "");
// circle function
circle(250, 200, 50);
getch();
// closegraph function closes the
// graphics mode and deallocates
// all memory allocated by
// graphics system .
```

```
23. closegraph();
24. return 0;
25. }
```

Output:



Example:

Aim: Write a code to draw a line in C graphics

graphics.h library is used to include and facilitate graphical operations in program. graphics.h functions can be used to draw different shapes, display text in different fonts, change colors and many more. Using functions of graphics.h you can make graphics programs, animations, projects and games. You can draw circles, lines, rectangles, bars and many other geometrical figures. You can change their colors using the available functions and fill them.

Examples:

For line 1, Input: x1 = 150, y1 = 150, x2 = 450, y2 = 150

For line 2, Input: x1 = 150, y1 = 200, x2 = 450, y2 = 200

For line 2, Input: x1 = 150, y1 = 250, x2 = 450, y2 = 250

Output:

Explanation: The header file graphics.h contains line() function which is described below:

Declaration: void line(int x1, int y1, int x2, int y2);

line function is used to draw a line from a point(x1,y1) to point(x2,y2) i.e. (x1,y1) and (x2,y2) are end points of the line.The code given below draws a line.

Code:

```
1.  // C++ Implementation for drawing line
2.  #include <graphics.h>
3.  // driver code
4.  int main()
5.  {
6.  // gm is Graphics mode which is a computer display
7.  // mode that generates image using pixels.
8.  // DETECT is a macro defined in "graphics.h" header file
9.  int gd = DETECT, gm;
10. // initgraph initializes the graphics system
11. // by loading a graphics driver from disk
12. initgraph(&gd, &gm, "");
13. // line for x1, y1, x2, y2
14. line(150, 150, 450, 150);
15. // line for x1, y1, x2, y2
16. line(150, 200, 450, 200);
17. // line for x1, y1, x2, y2
18. line(150, 250, 450, 250);
19. getch();
20. // closegraph function closes the graphics
21. // mode and deallocates all memory allocated
22. // by graphics system .
23. closegraph();
24. }
```

Output:

1.3 REFERENCE

1] Introduction to Computer Graphics: A Practical Learning Approach By Fabio Ganovelli, Massimiliano Corsini, Sumanta Pattanaik, Marco Di Benedetto

2] Computer Graphics Principles and Practice in C: Principles & Practice in C Paperback – 1 January 2002 by Andries van Dam; F. Hughes John; James D. Foley; Steven K. Feiner (Author)

1.4 UNIT END EXERCISE

- **Write a Program to draw basic graphics construction like line, circle, arc, ellipse and rectangle.**

*****

2

DEMONSTRATION OF SIMPLE GRAPHIC INBUILT FUNCTION

Unit Structure

2.0 Objectives
2.1 Introduction
2.2 Summary
2.3 References
2.4 Unit End Exercises

2.0 OBJECTIVES

C++ GRAPHICS Functions relating to graphics are used to create different shapes in different colors. The graphics functions require a graphics monitor (nowadays almost all computers have graphics monitors) and a graphics card such as VGA, SVGA or EGA. Color monitor is recommended for viewing graphics in colors.

The graphics include:

- Lines in different colors and styles.
- Different shapes like circles, rectangles in different styles.
- Text in different fonts, sizes, and styles.

The following files are needed with the C++ compiler to work in graphics:

- The Header File "graphics.h" that contains built-in graphic functions. This header file is included in the program.
- Borland Graphics Interface (BGI) files. These files contain graphics driver programs that initialize the computer monitor into graphics mode. These files have BGI extension.
- Character font style files having extension "chr". These files are used to process text or characters in graphics mode.

Display Mode:

The output of a program can be displayed on the screen in two modes. These modes are:

1. Text Mode
2. Graphics Mode

C++ Text Mode:

In a text mode, the screen is normally divided into 80 columns and 25 rows. The topmost row is 0 and the bottom-most row is 24. Similarly. the leftmost column is 0 and the rightmost column is 79. In-text mode, only text can be displayed. The images and one graphics object cannot be displayed.

C++ Graphics Mode:

Images and other graphic objects are displayed on the SC graphics mode. In this mode, the output is displayed on the computer screen in points or pixels.

2.1 INTRODUCTION

In graphics mode, the screen is divided into small dots called For example, in the VGA monitor, the screen is divided into 480 row 640 columns of dots. Thus, the VGA monitor screen is divide 640×480 pixels. The number of dots per inch is called resolution screen. The dots are very close to each other. The more the number pixels, the more clearer the graphics is.

The monitor types (display adapter) and their respective resolution are given below.

Monitor type	Resolution	Color
CGA	640×200	16
VGA	640×480	16
SVGA	800×600	256

Initializing C++ Graphics Mode:

The computer display system must be initialized into graphics mode before calling the graphics function.

The "initgraph" function is used to initialize the display into graphics mode. This function is a part of the "graphics.h" header file. So this file is included in the program before executing "initgraph" function.

The syntax of initgraph" function is:

initgraph(&driver, &mode, "path");:

Where:

Driver:

Represents the graphics driver installed on the computer. It may be an integer variable or an integer constant identifier, e.g. CGA, EGA, SVGA, etc.

The-graphics driver can also be automatically detected by using the keyword "DETECT". Letting compiler detect the graphic driver is known as auto-detect.

If the driver is to be automatically detected, the variable driver is declared as of integer type and DETECT value is assigned to it as shown below.

int driver, mode;

driver = DETECT;

This statement must be placed before "initgraph" function. When the above statement is executed. he computer automatically detects the graphic driver and the graphics mode.

Mode:

Represents output resolution on the computer screen. The normal mode for VGA is VGAHI. It gives the highest resolution

If the driver is on auto-detected, then its use is optional. The computer automatically detects the driver as well as the mode.

&

represents the addresses of constant numerical identifiers of driver and mode. If constants (e.g., VGA, VGAHI) are used, then "&" operator is not used as shown below:

initgraph (VGA, VGAHI, "path");

Path:

Represents the path of graphic drivers. It is the directory on the dish where the BGI files are located. Suppose the BGI files are stored in "C:\TC\BGI", then the complete path is written as:

initgraph (VGA, VGAHI, "C:\TC\\BGI");

Use of double backslash "\" is to be noted. One backslash is used as escape character and other for the directory path. If the BGI files are in the current directory, then the path is written as:

initgraph (VGA, VGAHI, " ");

```
1  #include<graphics.h>
2  main()
3  {
4  int d, m;
5  d= DETECT;
6  Initgraph(&d, &m, "c:\\tc");
7  }
```

In the above example, the BGI files must be in the specified directory, 1.e., in "c:\tclbgi". If the BGI files are in the directory path"c:\tc" then the above statement is written as:

initgraph(&d, &m, "C:\\TC");

The "cleardevice" Function:

The "cleardevice" function is used to clear the screen in graphics mode. It is similar to "clrscr" function that is used to clear the screen text mode. Its syntax is:

cleardevice();

Closing Graphics Mode:

The "closegraph" function is used to restore the screen to the text mode.

When graphics mode is initialized, memory is allocated to the graphics system. When "closegraph" function is executed, it de-allocates all memory allocated to the graphics system. This function is usually used a the end of the program. Its syntax is:

closegraph();

Text in Graphics Mode:

In graphics node: text can also be written in different fonts, styles. sizes, colors, and directions. The graphic functions commonly used to create and print text are described below.

The "outtext" Function

The "outtext" function is used to print text on the computer screen in graphics mode. The text is printed at the current cursor position on the screen. Its syntax is:

outtext(string);

Where:

string:

Represents the characters that are to be printed on the screen. It may be a string variable or string constant. The string constant is enclosed in double-quotes.

Example how to use cleardevice, closegraph and outtext function and print Electronic Clinic into C++ graphic mode.

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
```

```
4   {
5   int d, m;
6   d=DETECT;
7   initgraph (&d, &m, “”);
8   cleardevice();
9   outtext(“electronic clinic”);
10  getch();
11  closegraph();
12  }
```

The “moveto” Function:

The “moveto” function is used to move the current cursor position to a specified location on the screen where the output is to be printed. It is similar to “gotoxy” function used in text mode. Its syntax is:

moveto (x, y);

Where:

x

Represents the x-coordinate of the screen. It is the horizontal distance in pixels from the left of the screen. It may be an unsigned int type value or variable. For VGA, its value is from 0 to 639.

y

represents the y-coordinate of the scren. It is the vertical distance in pixels from the top of the screen. It may be an unsigned int type value or variable. For VGA, its value is from 0 to 479.

Example of how to use moveto function using C++ graphics.

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
4   {
5   int d,m;
6   d= DETECT;
7   initgraph(&d, &m, “”);
8   cleardevice();
9   moveto(400,200);
10  outtext(“Electronic Clinic”);
```

```
11  getch();
12  closegraph();
13  }
```

The "outtextxy" Function:

The "outtextxy" function is similar to the outtext" function but it is used to print text on the screen at a specified location. This function serves the purpose of both the "moveto" and "outtext" functions. Its syntax is:

outtextxy (x, y, string);

where:

x

represents the x-coordinate of the screen. It is the horizontal distance in pixels from the left of the screen. It may be unsigned int type value or variable. For VGA, its value is from 0 to 639.

Y

represents the y-coordinate of the screen. It is the vertical distance in pixels from the top of the screen. It may be unsigned int type value or variable. For VGA, its value is from 0 to 479.

String

represents the string of characters that is to be printed on the computer screen. It may be a string variable or a string. constant. The string constant is enclosed in double quotes.

Examples of using outtextxy in C++ graphics.

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
4   {
5   int d,m;
6   d= DETECT;
7   initgraph(&d, &m, "");
8   cleardevice();
9   outtextxy(100,200, "electronic clinic");
10  getch();
11  closegraph();
12  }
```

The "settextstyle" Function:

The "settextstyle" function is used to define the text style in graphics mode. The text style includes the font type, font size and text direction. The syntax of this function is given as: settextstyle (style, dir, size);

All the three parameters are of int type. These may be int type values or variables.

Where:

Style:

specifies the font style. Its value range is from 0 to 10.

Dir:

Specifies the direction of the text in which it is to be displayed. Its value is from 0 to 1. It may be a numerical constant identifier. It is HORIZ DIR (for horizontal direction) or VERT_DIR (for vertical direction).

Size:

Specifies the font size of the text. Its value is from 0 to 72.

Example of settextstyle:

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
4   {
5   int d,m,c;
6   d= DETECT;
7   initgraph(&d, &m, "");
8   cleardevice();
9   for(c=0; c<=10; c++)
10  {
11  setextstyle(c,0,0);
12  outtextxy(100,20+c*20, "electronic clinic")
13  }
14  getch();
15  closegraph();
16  }
```

2.2 SUMMARY

The "setcolor" Function:

The setcolor" function is used to define color of the objects and the text in graphics mode. Its syntax is: setcolor (co);

where:

co

Represents the color. It may be an int type value or variable. For VGA, its value is from 0 to 15. It may also be a numerical constant identifier, e.g. BLUE, GREEN, RED etc.

The "setbkcolor" Function

The "setbkcolor" function is used in graphics mode to define the background color of the screen. Its syntax is: setbkcolor(co);

Where:

co

Specifies the color. It may be an it type value or variable. For VGA. Its value is from 0 to 15. It may also be numerical constant identifier eg blue, green and red etc.

Example how to use setcolor and setbkcolor function and print Electronic Clinic into C++ graphic mode

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
4   {
5   int d,m,co;
6   d= DETECT;
7   initgraph(&d, &m, "");
8   cleardevice();
9   for(co=0; co<=15; co++)
10  {
11  setbkcolor(co);
12  setcolor(co+1);
13  settextstyle(0,0,2);
14  outtextxy(100,10+co*20, "electronic clinic");
15  outtextxy(200, 200,"press any key…");
```

```
16  getch();
17  }
18  closegraph();
19  }
```

Creating Objects in C++ Graphics Mode:

Different objects, e.g. lines, circles, rectangles and many other shapes are created in graphics mode using various built-in functions. Following are the functions that are commonly used to create graphics objects:

The “circle” Function

The “circle” function is used to draw a circle on the screen. Its syntax is:

circle(x, y, radius);

All the three parameters are of int type. These may be int type values or variables.

Where

x & y

Specifies the center point of the circle. These are the x- coordinate and y-coordinate of the center of the circle on the screen.

Radius

Specifies the radius of the circle.

Example how to make a circle using circle function in C++ graphics mode:

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
4   {
5   int d,m,r,c;
6   d= DETECT;
7   initgraph(&d, &m, “”);
8   cleardevice();
9   for(c=1; c<= 15; c++)
10  {
11  setcolor(c);
12  circle(300,200,c*10);
13  }
```

```
14  getch();
15  closegraph();
16  }
```

The "arc" Function:

The arc function is used to draw a circular arc starting from a specified angle and up to another specified angle. Its syntax is:

arc (x, y, stangle, endangle, radius);

All the five parameters are of int types. These may be constants or variables.

Where:

x & y

specify the center point of the circle. These are the x- coordinate and y-coordinate of the center of the arc on the screen.

Stangle:

Specifies the starting angle of the arc in degree.

Endangle:

Specifies the ending angle of the arc in degree.

Radius:

Specifies the radius of the arc.

Note:

The arc function can also be used to draw a circle by giving the starting angle 0 and ending angle 360. Similarly, it can also be used to draw line by giving the same values for starting and ending angles.

Example on how to use arc in C++

```
1   #include<graphics.h>
2   #include<conio.h>
3   main()
4   {
5   int d,m,c;
6   d= DETECT;
7   initgraph(&d, &m, "");
8   cleardevice();
9   for(c=1; c<=15; c++)
```

```
10  {
11  setcolor(c);
12  arc(300,200,45,145,c*10);
13  }
14  getch();
15  closegraph();
16
17  }
```

2.3 REFERENCES

1] Introduction to Computer Graphics: A Practical Learning Approach By Fabio Ganovelli, Massimiliano Corsini, Sumanta Pattanaik, Marco Di Benedetto

2] Computer Graphics Principles and Practice in C: Principles & Practice in C Paperback – 1 January 2002 by Andries van Dam; F. Hughes John; James D. Foley; Steven K. Feiner (Author)

2.4 UNIT END EXERCISE

- **How to make lines in C++ using line function in graphic mode?**

*****

MODULE II

3

OUTPUT PRIMITIVES & ITS ALGORITHMS

Unit Structure

3.0 Objectives
3.1 Introduction
3.2 Scan Conversion Algorithms
3.3 Line drawing algorithms
3.3.1 Digital Differential Analyser (DDA) Line Drawing Algorithm
3.3.2 Bresenham's Line Drawing Algorithm
3.4 Summary
3.5 Unit End Exercise
3.6 References for Future Reading

3.0 OBJECTIVES

The objective of scan conversion is to determine the intersection of rows & columns to find an area called pixel, to paint any object.

3.1 INTRODUCTION

It is a field of computer science that refers to the creation, storage, manipulation & drawing of pictures in digital form. It also manipulates visual contents which helps in manipulating images & objects in two & three dimension.

3.2 SCAN CONVERSION ALGORITHMS

It is a term used for drawing methods for two dimensional pictures such as lines, polygons and text that creates raster images. The objective of scan conversion is to determine the intersection of rows & columns to find an area called pixel, to paint any object. In a high resolution system, the adjacent pixels are so closely spaced that the approximate line pixels lie very close to the actual line path & hence the plotted lines appear to be smoother. In a low resolution the same approximated line pixels appear as a "zig-zag or staircase" which are not smooth.

3.3 LINE DRAWING ALGORITHMS

Several line drawing algorithms are used with an objective to create smooth images with a high resolution & this can be achieved as follows:

3.3.1 Digital Differential Analyser (DDA) Line Drawing Algorithm:

It is an incremental scan conversion method to determine points on a line.

Algorithm:

Step1: Input the coordinates of the two end points A(x1,y1) & B(x2,y2) for the line AB, such that A and B are not equal.

Step2: Calculate dx=(x2-x1) and dy=(y2-y1)

Step3: Calculate the length (L)

If abs(x2-x1) ≥ abs(y2-y1) then

L= abs(x2-x1)

Else

L= abs(y2-y1)

Step4: Calculate the increment factor

dx=$\frac{(x2-x1)}{L}$ and dy= $\frac{(y2-y1)}{L}$

Step5: Initialize the initial point on the line and plot

xnew=x1+0.5 and ynew = y1+0.5

Plot(Integer xnew, Integer ynew)

Step6: Obtain the new pixel on the line and plot the same

Initialize i to 1

While (i≤L)

{

xnew = xnew +dx
ynew=ynew + dy
plot(Integer xnew, Integer ynew)
i=i+1

}

Step7: Stop

Solved Example:

Q.1. Consider a line AB with A(0,0) and B(8,4) apply a simple DDA algorithm to calculate the pixels on this line.

Solution:

1. A(0,0) B(8,4) x1=0 y1=0 x2=8 y2=4

2. dx = 8-0 = 8, dy = 4-0 = 4
3. L=8, dx>dy
4. dx = $\frac{(x2-x1)}{L}$ =$\frac{8-0}{8}$ = $\frac{8}{8}$ = 1
   dy = $\frac{(y2-y1)}{L}$ = $\frac{(4-0)}{8}$ =$\frac{4}{8}$ =0.5
5. xnew = x1+ 0.5 = 0+0.5 =0.5

   ynew = y1+0.5 = 0+0.5 = 0.5

Plot (0,0)

6. i=1

while (1 ≤ 8)

{

    xnew = 0.5 + 1 =1.5
    ynew= 0.5+0.5 =1.0
    Plot (1,1)
    i=i+1

}

while (2 ≤ 8)

{

    xnew = 1.5 + 1 =1.5
    ynew= 1.0+0.5 =1.5
    Plot (2,1)
    i=i+1

}

while (3 ≤ 8)

{

    xnew = 2.5 + 1 =3.5
    ynew= 1.5+0.5 =2.0
    Plot (3,2)
    i=i+1

}

while (4 ≤ 8)

{

    xnew = 3.5 + 1 =4.5

ynew= 2.0+0.5 =2.5
Plot (4,2)
i=i+1

}

while (5 ≤ 8)

{

xnew = 4.5 + 1 =5.5
ynew= 2.5+0.5 =3.0
Plot (5,3)
i=i+1

}

while (6 ≤ 8)

{

xnew = 5.5 + 1 =6.5
ynew= 3.0+0.5 =3.5
Plot (6,3)
i=i+1

}

while (7 ≤ 8)

{

xnew = 6.5 + 1 =7.5
ynew= 3.5+0.5 =4.0
Plot (7,4)
i=i+1

}

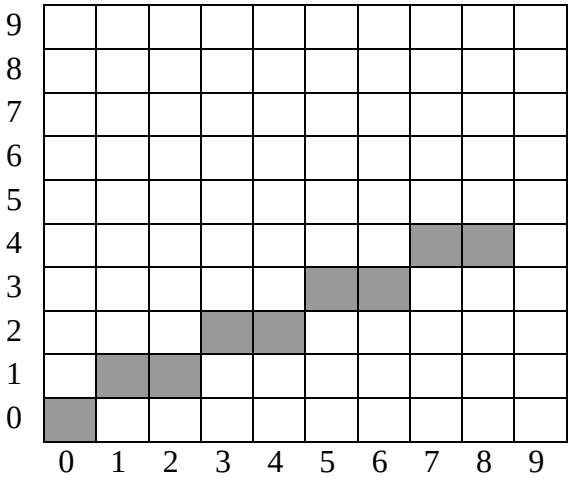
while (8 ≤ 8)

{

xnew = 7.5 + 1 =8.5
ynew= 4.0+0.5 =4.5
Plot (8,4)
i=i+1

}

i	Plot	Xnew	ynew
-	(0,0)	0.5	0.5
1	(1,1)	1.5	1.0
2	(2,1)	2.5	1.5
3	(3,2)	3.5	2.0
4	(4,2)	4.5	2.5
5	(5,3)	5.5	3.0
6	(6,3)	6.5	3.5
7	(7,4)	7.5	4.0
8	(8,4)	8.5	4.5



Plot of the Line AB

Q.2. Use DDA Line drawing algorithm draw a line AB for the endpoints A (1,1) and B(5,3)

Solution:

1. A(1,1) and B(5,3), x1=1 y1=1 x2= 5 y2=3
2. dx=5-1 = 4 , dy =3-1 =2
3. L= 4, dx>dy
4. dx = $\frac{5-1}{4}$ = $\frac{4}{4}$ = 1, dy = $\frac{2}{4}$ =0.5
5. xnew = 1+0.5 =1.5

   ynew = 1+0.5 =1.5

Plot(1,1)

6. i=1

while (1 ≤ 4)

{

xnew = 1.5 +1=2.5
ynew=1.5+0.5 =2.0
Plot(2,2)
i=i+1

}

while (2 ≤ 4)

{

xnew = 2 +1.5=3.5
ynew=2+0.5 =2.5
Plot(3,2)
i=i+1

}

while (3 ≤ 4)

{

xnew = 3 +1.5=4.5
ynew=2.5+0.5 =3.0
Plot(4,3)
i=i+1

}

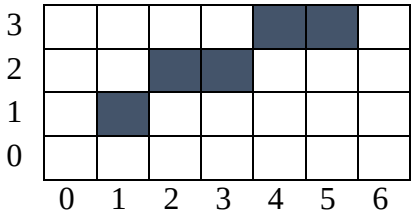
while (4 ≤ 4)

{

xnew = 4 +1.5=5.5
ynew=3.0+0.5 =3.5
Plot(5,3)
i=i+1

}

i	**Plot**	**xnew**	**ynew**
-	(1,1)	1.5	1.5
1	(2,2)	2.5	2.0
2	(3,2)	3.5	2.5
3	(4,3)	4.5	3.0
4	(5,3)	5.5	3.5



Q.3. Consider a line AB with A (2,3) and B (6,8). Apply a simple DDA algorithm and calculate the pixels on the line

Solution:

1. A(2,3) and B(6,8) x1=2, y1 =3 x2=6 y2 =8
2. dx = x2-x1 = 6-2 =4

   dy = y2-y1 = 8-3 =5
3. L=5
4. dx = $\frac{x2-x1}{L}$ = $\frac{4}{5}$ 0.8

   dy= $\frac{y2-y1}{L}$ = $\frac{5}{5}$ = 1
5. xnew =x1 +0.5 = 2+0.5 =2.5

   ynew = y1+0.5 = 3+0.5 =3.5

Plot (2,3)

6. i=1

while (1≤ 5)

{

xnew = xnew +dx

=2.5+0.8

3.3

ynew =ynew +dy

=3.5+1

=4.5

Plot(3,4)

i=i+1

}

while (2≤ 5)

{

xnew = xnew +dx

=3.3+0.8
=4.1
ynew =ynew +dy =4.5+1
=5.5
Plot(4,5)
i=i+1
}
while (3≤ 5)
{
xnew =xnew +dx
=4.1+0.8
=4.9
Ynew =5.5+1
=6.5
Plot(4,6)
i=i+1
}
while (4≤ 5)
{
xnew = xnew +dx
=4.9+0.8
=5.7
ynew =ynew +dy
=6.5+1
=7.5
Plot(5,7)
i=i+1
}
while (5≤ 5)
{
xnew = xnew +dx
=5.7+0.8
=6.5
ynew =ynew +dy

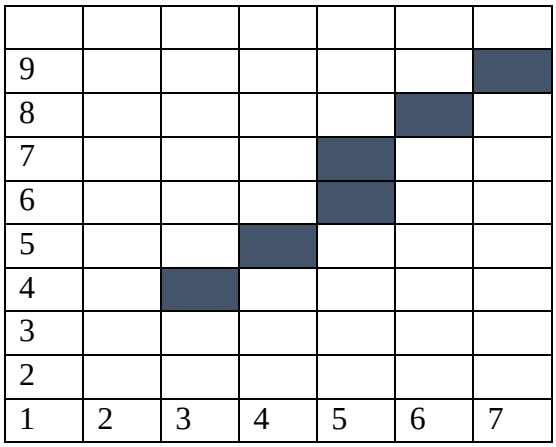
=7.5+1

=8.5

Plot(6,8)

}

i	Plot	Xnew	ynew
-	(2,3)	2.5	3.5
1	(3,4)	3.3	4.5
2	(4,5)	4.1	5.5
3	(4,6)	4.9	6.5
4	(5,7)	5.7	7.5
5	(6,8)	6.5	8.5



3.3.2 Bresenham's Line Drawing Algorithm:

It was developed by Jack Bresenham in 1965. It uses floating point arithmetic to calculate the slope of the line and error term.

Algorithm:

Step1: Initialize the end points of the line AB with A(x1,y1) and B(x2,y2). The two end points are assumed to be distinct.

Step2: Calculate dx and dy such that:

dx=x2-x1

dy=y2-y2

Step3: Initialize error term (e)

e= 2*dy-dx

xnew=x1

ynew=y1

Step4: Determine the first pixel on the line and update the error term

For i=1 to dx

Plot(Integer xnew, Integer ynew)

While(e≥0)

{

ynew=ynew+1
e=e-2*dx

}

xnew=xnew+1

e=e+2*dy

Next i

End for loop

Step5: Stop

Solved Examples:

Q.1. Consider a line AB with coordinates A(5,5) and B(13,9). Determine the line segment using Bresenham's line drawing algorithm.

Solution:

1. A(x1,y1) = A(5,5)

   B(x2,y2)=B(13,9)

2. dx=x2-x1 =13-5 =8

   dy = y2-y1 = 9-5 =4

3. e=2*dy-dx = 2*4 -8 = 8-8 =0

   xnew=x1 ynew=y1

   xnew=5 ynew=5

4. for i=1 to 8

   Plot(5,5)

While (e≥0)

{

ynew=ynew+1=5+1=6

e=e-2*dx=0-2*8 = -16

}

xnew=xnew+1 =5+1=6

e=e+2*dy = -16+2*4 = -16+8 = -8

for i=2 to 8

plot(6,6)

while(e≥0)

{

}

xnew=xnew+1 =6+1=7

e=e+2*dy =-8+2*4 = 0

for i=3 to 8

plot(7,6)

while(e≥0)

{

ynew=ynew+1=6+1=7

e=e-2*dx = 0-2*8 = - 16

}

xnew=xnew+1=7+1=8

e=e+2*dy = -16+2*4 = -16+8 = -8

for i=4 to 8

plot(8,7)

while(e≥0)

{

}

xnew=xnew+1 = 8+1 =9

e=e+2*dy = -8+2*4 = -8+8 =0

for i=5 to 8

plot(9,7)

while(e≥0)

{

ynew=ynew +1=7+1 =8

e=e-2*dx = 0-2*8 =-16

}

xnew=xnew+1=9+1 =10

e=e+2*dy =-16+2*4 = -8

for i=6 to 8
plot(10,8)
while(e≥0)
{
}
xnew=xnew+1 = 10+1 = 11
e=e+2*dy = -8+2*4 = 0

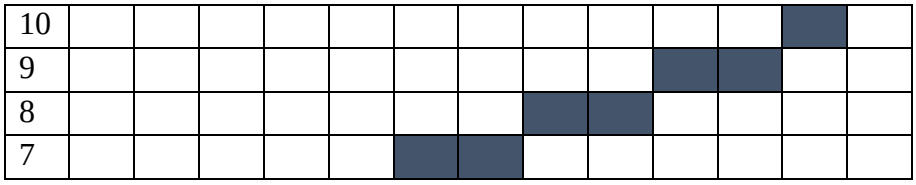
for i=7 to 8
Plot(11,8)
ynew=ynew+1 = 8+1 =9
e=e-2*dx = 0-2*8 = -16

}

xnew=xnew+1 =11+1 =12
e=e+2*dy = -16+2*4 =-8

for i=8 to 8
plot(12,9)
while(e≥0)
{
}
Xnew=xnew+1 =12+1 =13
e=e+2*dy =-8+2*4 =0

i	plot	xnew	ynew	e
-	-	5	5	0
1	(5,5)	6	6	-8
2	(6,6)	7	6	0
3	(7,6)	8	7	-8
4	(8,7)	9	7	0
5	(9,7)	10	8	-8
6	(10,8)	11	8	0
7	(11,8)	12	9	-8
8	(12,9)	13	9	0



6													
5													
4													
3													
2													
1													
1	2	3	4	5	6	7	8	9	10	11	12	13	

Q.2. Consider the line coordnates A(0,0) and B(8,4). Determine the line segment using Bresenham's algorithm.

Solution:

1. A(0,0) and B(8,4) x1=0 y1=0 and x2 =8 y2 = 4
2. dx = x2-x1 = 8-0 dy= y2-y1 = 4-0
3. e=2*dy-dx = 2*4-8 =8-8 =0

   xnew=x1=0 ynew= y1=0
4. for (i=1) to 8

Plot(0,0)

while(e≥0)

{

ynew=0+1=1

e = 0-2*8 = - 16

}

xnew=0+1=1

e=-16+2*4 = -8

for i=2 to 8

plot(1,1)

while(e≥0)

{

}

xnew=1+1 =2

enew= -8+2*4 =0

for i=3 to 8

Plot (2,1)

while(e≥0)

{

ynew=1+1=2

e=0-2*8 = -16

}

xnew =2+1 =3

e= -16+2*4 = -8

for i=4 to 8

Plot(3,2)

while (e≥0)

{

}

xnew = 3+1=4

e=8+2*4 = 0

for i=5 to 8

plot(4,2)

while(e≥0)

{

ynew=2+1 =3

e=0-2*8 =-16

}

xnew =4+1=5

e=-16+2*4 = -8

for i=6 to 8

plot(5,3)

while(e≥0)

{

}

xnew=5+1=6

e= -8+2*4 =0

for i=7 to 8

plot(6,3)

while(e≥0)

{

ynew= 3+1=4

e=0-2*8 = -16

}

xnew=6+1 =7

e=-16+2*4 =-8

For i=8 to 8

Plot(7,4)

while(e≥0)

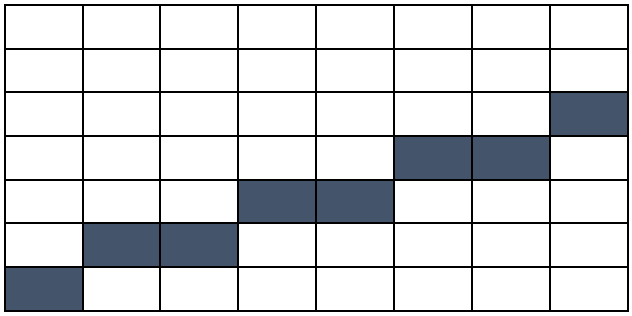
{

}

xnew=7+1=8

e= -8+2+4 =0

i	Plot	xnew	ynew	e
-	-	0	0	0
1	(0,0)	1	1	-8
2	(1,1)	2	1	0
3	(2,1)	3	2	-8
4	(3,2)	4	2	0
5	(4,2)	5	3	-8
6	(5,3)	6	3	0
7	(6,3)	7	4	-8
8	(7,4)	8	4	0



3.4 SUMMARY

In this chapter line algorithms are used to describe how a line can be drawn using each and pixel. Two important line drawing algorithms are explained- DDA & Bresenham's line drawing algorithm.

3.5 UNIT END EXERCISE

1. Define scan conversion. Write an algorithm of DDA line drawing.
2. Write an algorithm of DDA Line drawing.
3. Consider a line AB with A(0,0) and B(8,4). Apply a simple DDA to calculate the pixels of this line.
4. Consider a line from (0,0) to B(6,6). Using simple DDA to calculate the points of this line.
5. Consider a line from (0,0) to (5,5). Using simple DDA to calculate the points of this line.
6. Use DDA to draw pixels of the line AB with A(1,1) and B(5,3).
7. Use Bresenham's line drawing algorithm to draw pixels of the line XY(5,5) and Y (13,9)
8. Use Bresenham's line drawing algorithm to draw pixels of the line XY(0,0) and Y (8,4)
9. Write the applications of line drawing algorithm.

3.6 REFERENCES FOR FUTURE READING

- Computer_Graphics_C_Version_by_Donald_Hearn_and_M_Pauline_Baker_II_Edition
- Computer graphics by Atul P. Godse, Dr. Deepali A. Godse

*****

MODULE III

4

OUTPUT PRIMITIVES & ITS ALGORITHMS

Unit Structure

4.0 Introduction
4.1 Implementation of circle drawing Midpoint circle
4.2 Application of Circle drawing algorithm
4.3 Unit End Exercise

4.0 INTRODUCTION

What is scan conversion and how it is utilized for drawing a Circle:

Converting the unbroken graphical object as a group of distinct objects is called as scan conversion.

In the process of "scan –converting a circle", the circle is divided into eight equal parts, one part is called as octant, and if one part is generated, then it is easy to replicate the other seven parts; so computing one octant is enough to determine the complete circle.

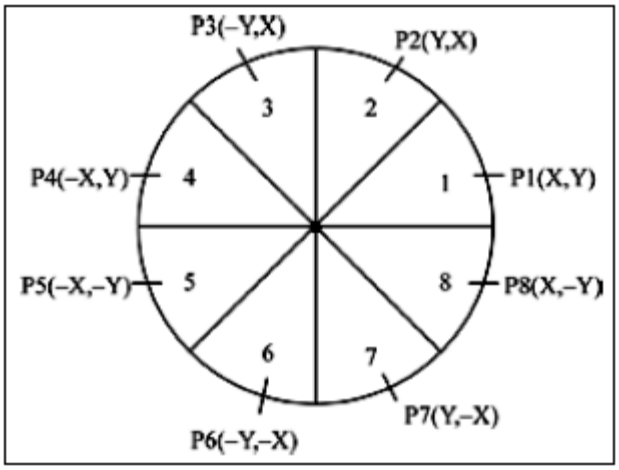


Fig. 3.4.0 The eight-way of Symmetry of Circle

Techniques used to compute octant of Circle:

- **Digital Differential analyzer (DDA):**

○ **Direct or polynomial approach:** second degree polynomial equation is used to form the circle octant.

- **Parametric or Trigonometric approach:** parametric polar representation is used to form the circle octant.

- **Bresenham's Algorithm:** This algorithm can form the entire circle; to avoid time of calculation one octant is computed and symmetry is used form the other seven octants.

- **Mid-point Circle algorithm:** It is better than polynomial and parametric approach; it adopts integer operation and so avoids trigonometric and square root calculations.

4.1 IMPLEMENTATION OF CIRCLE DRAWING MIDPOINT CIRCLE

Before studying Mid-Point Circle algorithm, we must study Bresenham's Algorithm for circle drawing as below:

To draw Circle, we first have to look at properties of circle:

Properties of Circle:

1. Circle function $f(x, y) = x^2 + y^2 – r^2$.
2. Any point (x, y) on the boundary of the circle with radius 'r' satisfies the equation circle (x, y) = 0.
3. If the point is in the interior of the circle, the circle function is negative.
4. If the point is outside of the circle, the circle function is positive.

Value of Circle (x, y)	Position of pixel w.r.t. circle
Less than 0	if (x, y) is inside the circle boundary
Equal to 0	if (x, y) is on the circle boundary
Greater than 0	if (x, y) is outside the circle boundary

- Midpoint between candidate pixels at sampling position xk+1 along a circular path. (Ref. Fig. 3.4.1 below)

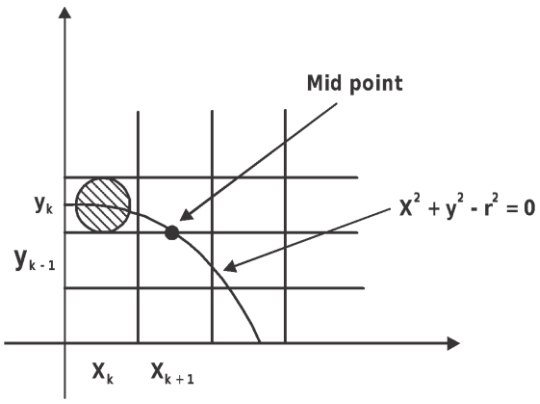


Fig. 3.4.1 Sampling position of candidate point in circular path

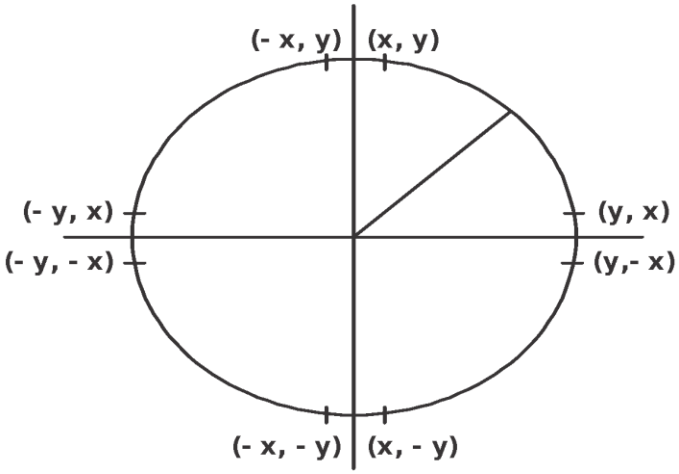


Fig. 3.4.2 For pixel (x,y) all possible pixels in 8 (eight) octants.

Circle algorithm is based on The circle equation is given by,

$(x - xc)^2 + (y - yc)^2 = r^2$ (where xc & yc are coordinates of the center of the Circle)

However, above equation is non linear, so that square-root evaluations would be required to compute pixel distances from circular path. This algorithm avoids this square-root calculations by comparing the squares of the pixel separation distances.

A method for direct distance comparison is to test the halfway position between two pixels to determine if this midpoint is inside or outside the circle boundary. This method is more easily applied to other conics, and for an integer circle radius, the midpoint approach generates the same pixel.

How Mid Point Circle differs from above define Bresenham's algorithm?

As per above Bresenham's Algorithm, we deal with integers, so it occupies less memory and lesser time for execution as well as it is reliable, accurate and efficient as it avoids using round function or floating-point calculations, where as in Mid-**point circle algorithm** also avoids square root or trigonometric calculation by applying integer operation only. This algorithm checks the nearest integer by computing the middle point of the pixels nearer to the given point on the circle.

MID POINT CIRCLE ALGORITHM:

(1) Input radius r and circle center (xc, yc) and obtain the first point on the circumference of a circle centered on the origin as (x0, y0) = (0,r)

(2) Calculate the initial value of the decision parameter as p0 = 5/4 – r

(3) At each xk position, starting at k = 0, perform the following test:

If pk<0, the next point along the circle centered on (0, 0) is

(xk+1, yk) and

pk+1 = pk +2xk+1 +1

Otherwise,

the next point along the circle is (xk+1, yk –1) and

pk+1 = pk + 2xk+1 +1 – 2yk+1

Where 2xk+1= 2xk +2, 2yk+1=2yk-2

(4) Determine symmetry points in the other seven octants.

(5) Move each calculated pixel position (x, y) onto the circular path centered on (xc, yc) and plot the coordinate values x = x + xc, y = y + yc

(6) Repeat steps 3 through 5 until x ≥ y.

Example: Consider a circle is centered on origin and the radius is 10.

First circle octant in the first quadrant from x = 0 to x = y

Initial value of the decision parameter is p0 =1 – r = –9

(x0, y0) = (0, 10), 2x0 =0 and 2y0 =20

Let us do calculation as shown in table below:

K	pk	(xk+1,yk+1)	2xk+1	2yk+1
0	-9	(1,10)	2	20
1	-6	(2,10)	4	20
2	-1	(3,10)	6	20
3	6	(4,9)	8	18
4	-3	(5,9)	10	18
5	8	(6,8)	12	16
6	5	(7,7)	14	14

By plotting graph from above calculated value, we get as shown below.

Note: Table. 3.4.3 above is showing only two symmetrical points (x, y) and (y, x).

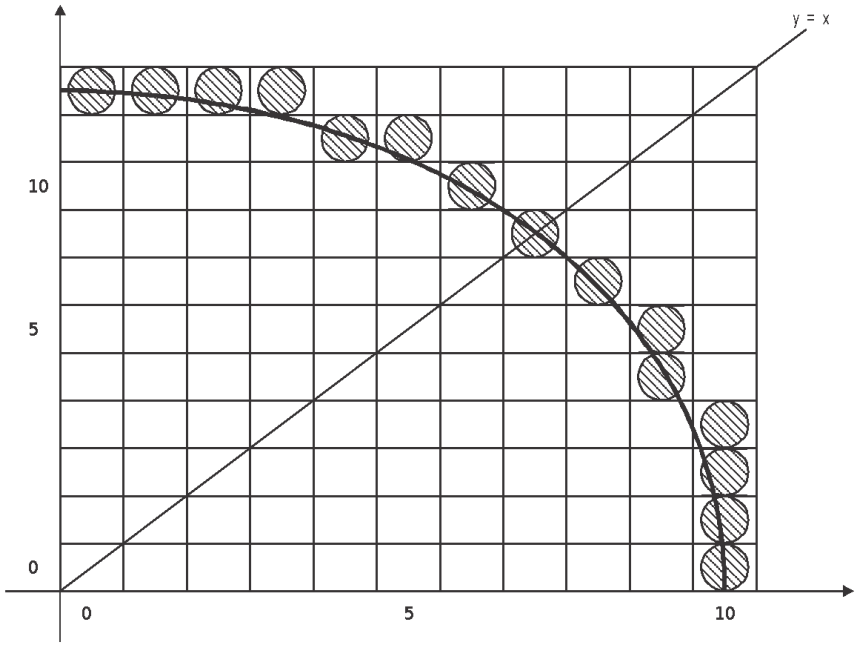


Fig. 3.4.3 Demonstration for Mid-point Circle Algorithm

Midpoint Circle Algorithm in C++:

```
#include<graphics.h>
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<conio.h>
void main()
{
        int x,y,r;
        int gd=DETECT,g;
        initgraph(&gd,&gm,"c:\\TurboC3\\bgi");
        cleardevice();
cout<<"Enter the centre co-ordinates:";
cin>>x>>y;
        cout<<"Enter radius of circle:";
        cin>>r;
        circlemidpoints (x,y,r);
        getch();
        closegraph();
}
```

```
void circlemidpoints (int xcenter, int ycenter, int radius)
{       int x=0;
        int y=radius;
        int p=1-radius;
        void circleplotpoints (int ,int ,int ,int );
        circleplotpoints (xcenter,ycenter,x,y);
        while(x<y)
        {       x++;
                if(p<0)
                        p=p+2*x+1;
                else
                {       y--;
                        p=p+2*(x-y)+1;
                }
                circleplotpoints (xcenter,ycenter,x,y);
        }       return(0);
}


void circleplotpoints (int xcenter,int ycenter,int x,int y)
{
        putpixel(xcenter+x,ycenter+y,5);
        putpixel(xcenter-x,ycenter+y,5);
        putpixel(xcenter+x,ycenter-y,5);
        putpixel(xcenter-x,ycenter-y,5);
        putpixel(xcenter+y,ycenter+x,5);
        putpixel(xcenter-y,ycenter+x,5);
        putpixel(xcenter+y,ycenter-x,5);
        putpixel(xcenter-y,ycenter-x,5);
}
```

4.2 APPLICATION OF CIRCLE DRAWING ALGORITHM

Circle drawing algorithm is used by many applications, some of them are listed in this section.

1. Concentric Circle Application:

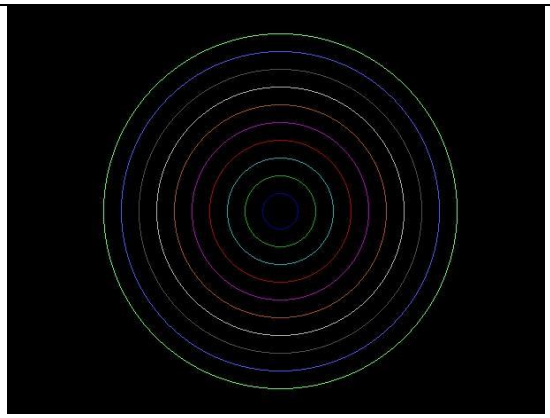
In graphic designing in many application areas drawing of concentric circle is required.

Below is program written using C++ to draw a concentric circle with different colors and at periodic interval.

Program: Write a C++ program to draw a concentric circle of different colors at periodic interval of time.

```
#include<iostream.h>
#include<conio.h>
#include<graphics.h>
main()
{
 int i,gd,gc,xcen,ycen,color=1;

gd=DETECT;

initgraph(&gd,&gc,"C:\\TURBOC3\\BGI");
 xcen=getmaxx()/2;
ycen=getmaxy()/2;
for(i=20;i<=200;i+=20)
{
        setcolor(color++);
        circle(xcen,ycen,i);
}
getch();
}
```

Output of the above code is as shown below:

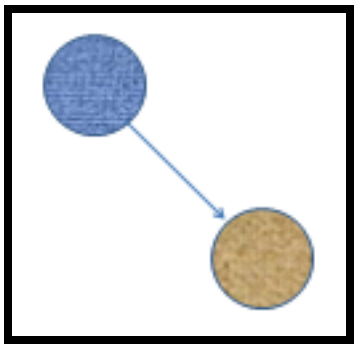


2. For creating a Bouncing ball application, Circle drawing algorithm is also utilized as follows:

```
#include<graphics.h>
#include<dos.h>
#include<conio.h>
main()
{
        int gd,gm,x=10,y=10,xinc=10,yinc=10,c=1,f=1;
        gd=DETECT;
        initgraph(&gd,&gm,"C:\\TC\\BGI");
        while(!kbhit())
        {
                x = x + xinc;
                y = y + yinc;
                if(x<=0 || x>=getmaxx())
                        xinc=-xinc;
                if(y<=0 || y>=getmaxy())
                        yinc=-yinc;
                        c++;
                        f++;
                if(c>=15)
```

```
                c=1;
            if(f>=12)
                f=0;
            setfillstyle(c,f);
            fillellipse(x,y,10,10);
        delay(100);
        cleardevice();
        } }
```

Output of above program is as below:



3. To display an analog clock, circle drawing algorithm is utilized as follows:

```
#include<dos.h>
#include<math.h>
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#define x 3.1415

void tick(); /*Produce Tic-Tic with every second*/
struct time t; /*Structure to get time from the computer Bios*/

void main()
```

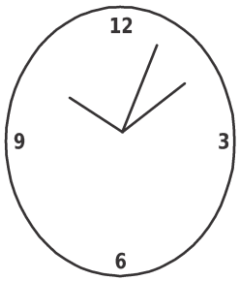
```
{
        int gdriver=DETECT,gmode;
        float sec_x,sec_y,min_x,min_y,hour_x,hour_y,h=0,m=0,s=0;

        /*initialize graphics mode*/
        initgraph(&gdriver,&gmode,"c:\\tc\\bgi");

        gettime(&t);        /* Obtain the system time*/
        h=t.ti_hour;
        if(h>=12)h=h-12;
        m=t.ti_min;
        s=t.ti_sec;

        outtextxy(170,20,":Demonstartion of animating Clock:");
        circle(getmaxx()/2,getmaxy()/2,120); //Draw Border
        circle(getmaxx()/2,getmaxy()/2,123); //Draw Border
        setfillstyle(SOLID_FILL,BLUE);   //Fill Interior of the clock
        floodfill(320,240,WHITE);

        while(!kbhit())
        {
                setcolor(WHITE);
                outtextxy(420,240,"3");
                outtextxy(210,240,"9");
                outtextxy(310,130,"12");
                outtextxy(310,340,"6");

                sec_x=100*cos(2*x/60*s-x/2)+getmaxx()/2;   //Compute
Second Needle Coordinate
```

```
            sec_y=100*sin(2*x/60*s-x/2)+getmaxy()/2;

        min_x=90*cos(2*x/60*m-x/2)+getmaxx()/2;          //Compute
Minute Needle Coordinate

        min_y=90*sin(2*x/60*m-x/2)+getmaxy()/2;

hour_x=60*cos(2*x/12*(h+m/60)-x/2)+getmaxx()/2;       //Compute
Hour Needle Coordinate

        hour_y=60*sin(2*x/12*(h+m/60)-x/2)+getmaxy()/2;

            setcolor(RED);
            line(getmaxx()/2,getmaxy()/2,sec_x,sec_y);
            setcolor(WHITE);
            line(getmaxx()/2,getmaxy()/2,min_x,min_y);
            setcolor(YELLOW);
            line(getmaxx()/2,getmaxy()/2,hour_x,hour_y);

            tick();
            delay(1000);

            setcolor(BLUE);
            line(getmaxx()/2,getmaxy()/2,sec_x,sec_y);
            line(getmaxx()/2,getmaxy()/2,min_x,min_y);
            line(getmaxx()/2,getmaxy()/2,hour_x,hour_y);

            s=s+1;
            if(s>=60)
            {
```

```
                    s=0;
                    m=m+1;
                    h=h+1/60;
                }

        }
        nosound();
        getch();
        closegraph();
}
/* simulate clock tick sound*/
void tick()
{
        int i;
        for(i=3500;i<=6500;i++)
        sound(i);
        nosound();
}
```

Output of above code is as shown below:

Demonstrating Analog Clock:



4. Circle drawing algorithm is also used to create an application to with four (04) bouncing ball, each ball is bouncing back in the respective quadrant in which it is placed.

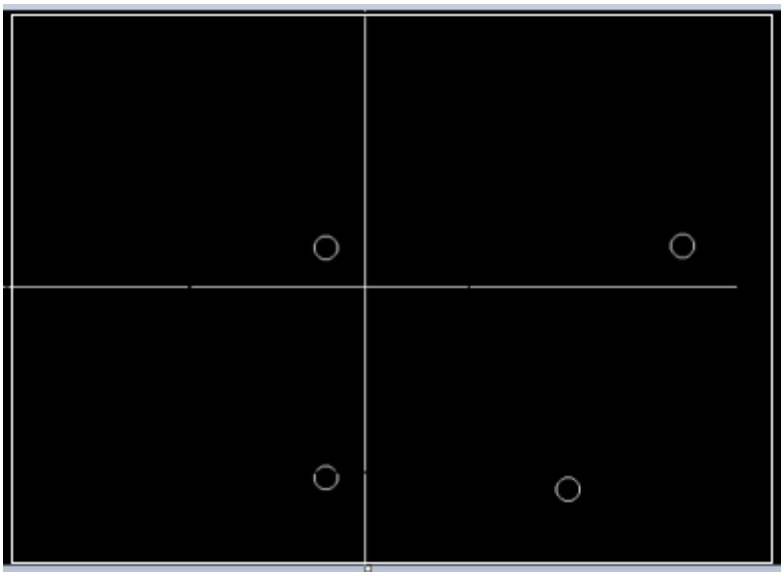
Note; Divide an output screen in four equal quadrants .

Program:

```
#include<conio.h>
#include<graphics.h>
main()
{
        int gd=DETECT,gm,xc=10,yc=10,xctr=5,yctr=5,
        xc2=(int)getmaxx()/2+10,yc2=10,xctr2=5,yctr2=5,
         //xc3=(int)getmaxx()/-2+10,yc3=10,xctr3=5,yctr3=5;
        xc3=10,yc3=getmaxy()/2,xctr3=5,yctr3=5,xc4,yc4,xctr4=5,yc
tr4=5;
        initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
                xc2=(int)getmaxx()/2+10;yc2=10;xctr2=5;yctr2=5;
                yc3=getmaxy()/2+10;
                xc4=getmaxx()/2;
                yc4=getmaxy()/2;
        while(!kbhit())
        {
                line(getmaxx()/2,0,getmaxx()/2,getmaxy());
                line(0,getmaxy()/2,getmaxx(),getmaxy()/2);

                circle(xc,yc,10);
                if(xc<=0 || xc>=getmaxx()/2-20)
                        xctr=-xctr;
                if(yc<=0 || yc>=getmaxy()/2-20)
                        yctr=-yctr;
                xc += xctr;
                yc += yctr;
```

```
            circle(xc2,yc2,10);
             if(xc2-4<=(int)getmaxx()/2 || xc2>=getmaxx()-20)
                    xctr2=-xctr2;
            if(yc2<=0 || yc2>=getmaxy()/2-20)
                    yctr2=-yctr2;
            xc2 += xctr2;
            yc2 += yctr2;

            circle(xc3,yc3,10);
            if(xc3<=0 || xc3>=getmaxx()/2-20)
                    xctr3=-xctr3;
            if(yc3<getmaxy()/2 || yc3>=getmaxy())
                    yctr3=-yctr3;
            xc3 += xctr3;
            yc3 += yctr3;

            circle(xc4,yc4,10);
            if(xc4<getmaxx()/2 || xc4>=getmaxx()-20)
                    xctr4=-xctr4;
            if(yc4<getmaxy()/2 || yc4>=getmaxy())
                    yctr4=-yctr4;
            xc4 += xctr4;
            yc4 += yctr4;

            delay(10);
            cleardevice();
      }

}
```

Output of the above program is as below:



Apart from above application of circle algorithm, there are many other applications which are mention below.

1. For drawing symbol that can be like as an Olympic ring like below in OpenGL software:

Fig. 3.4.3 Olympic ring in OpenGL software

3. To Construct a simulator for representation of Solar System
4. To Create various animation-based games
5. To Create various animated short films or videos
6. To represent waves generated by object like spring

4.3 UNIT END EXERCISE

Answer the following:

Q-1. List various algorithm, which are used for drawing a circle using computer graphics.

Q-2. Explain the difference between Bresenham's Algorithm and Mid-Point Circle Algorithm for drawing a Circle.

Q-3. Explain in brief Mid-Point Circle Algorithm.

Q-4. For a Center x and y coordinates as 20 and 20 respectively and having radius 10 units, plot a scanning a table using Mid-Point circle algorithm.

Q-5. Based on scanning table received in Q-4, plot a Circle on a graph paper.

Q-6. List a various application where Mid-Point Circle algorithm is used.

Q-7. Write a CPP using computer graphics to demonstrate how it can be utilized to develop various application such as Analog Clock.

*****

MODULE IV

5

OUTPUT PRIMITIVES & ITS ALGORITHMS

Unit Structure

5.0 Objectives
5.1 Introduction
5.2 Mid-Point Ellipse
5.3 Implementation of Ellipse Drawing A. Midpoint Ellipse
5.4 Summary
5.5 References for Future Reading

5.0 OBJECTIVES

After this Chapter a learner will be able to implement the Mid-Point Ellipse algorithm using computer program.

5.1 INTRODUCTION

Ellipse is defined as the geometric figure which is the set of all points on a plane whose distance from two fixed points known as the foci remains a constant.

It consists of two axes: major and minor axes where the major axis is the longest diameter and minor axis is the shortest diameter.

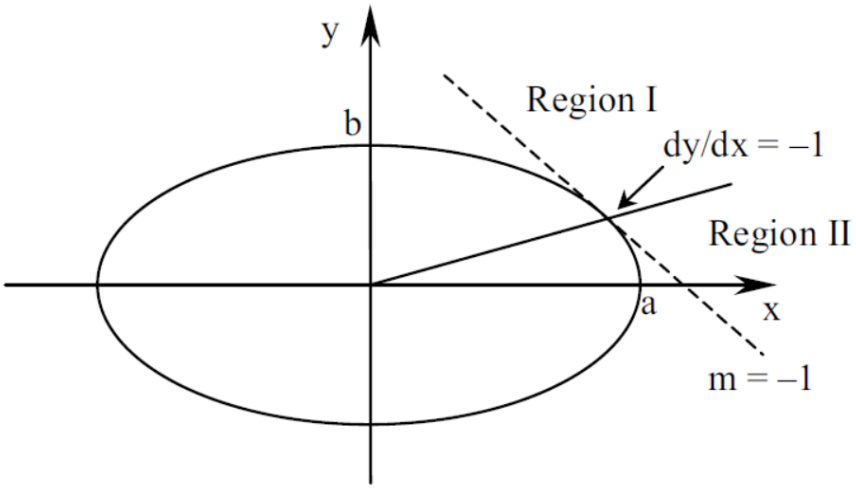
Unlike circle, the ellipse has four-way symmetry property which means that only the quadrants are symmetric while the octants are not.

5.2 MID-POINT ELLIPSE

Mid-Point Ellipse is an incremental method for scan converting an ellipse that is centered at the origin in standard position i.e., with the major and minor axis parallel to coordinate system axis. It is very similar to the midpoint circle algorithm. Because of the four-way symmetry property we need to consider the entire elliptical curve in the first quadrant.

The **mid-point ellipse drawing algorithm** is used to calculate all the perimeter points of an ellipse. In this algorithm, the mid-point between the two pixels is calculated which helps in calculating the decision parameter. The value of the decision parameter determines whether the mid-point lies inside, outside, or on the ellipse boundary and the then position of the mid-point helps in drawing the ellipse.

let us consider the elliptical curve in the first quadrant.



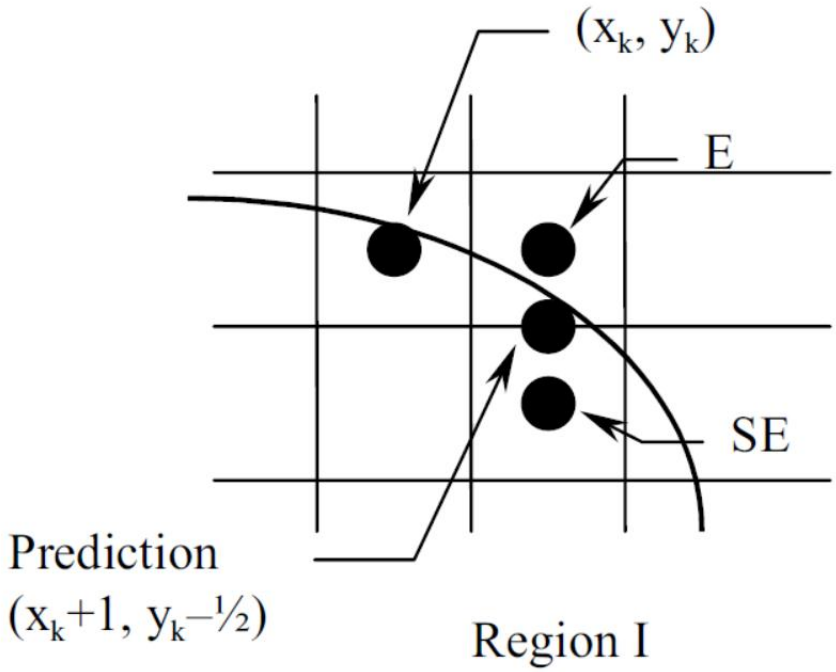
Consider the general equation of an ellipse,

$b^2 x^2 + a^2 y^2 – a^2 b^2 = 0$

where a is the horizontal radius and b is the vertical radius, we can define an function f(x,y) by which the error due to a prediction coordinate (x,y) can be obtained. The appropriate pixels can be selected according to the error so that the required ellipse is formed. The error can be confined within half a pixel.

Set $f(x,y) = b^2 x^2 + a^2 y^2 – a^2 b^2$

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 P_k can be

defined as follows:

$P_k = f(x_k+1, y_k–½)$

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

$= b^2(x_k^2 + 2x_k +$

1. Input rx, ry, and ellipse center (xc, yc,), and plot the first point as

$(x_0,y_0)=(0,r_y)$

Region 1:

2. Then, Calculate the initial value of the decision parameter in region 1 as

$p_0 = r_y^2 – r_x^2r_y + r_x^2/4$

3. At each x_k, position in region 1, from k = 0, check the condition

If $p_k < 0$, the next point is (x_k+1, y_k) and

$p_k+1 = p_k + 2r_y^2x_k + 1 + r_y^2$

Otherwise, the next point is $(x_k + 1, y_k – 1)$ and

$p_k+1 = p_k + 2r_y^2x_k + 1 + r_y^2 – 2\, r_x^2\, y_k+1$

and continue until $2r_y^2x >= 2r_x^2\, y$

Region 2:

4. Calculate initial decision parameter in region 2 with the last point (x0, y0) calculated in region 1 as

$p_0 = r_y^2(x_0 + 1/2)^2 + r_x^2(y_0 – 1)^2 – r_x^2r_y^2$

5. At each yk position in region 2, starting at k = 0, check:

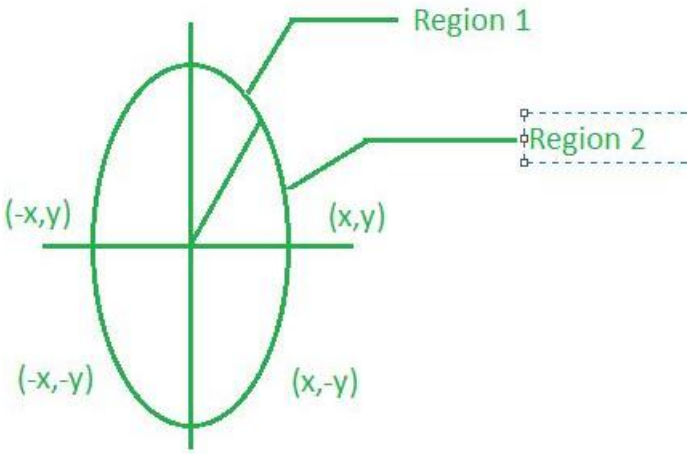
If $p_k > 0$, the next point is $(x_k, y_k - 1)$

$p_{k+1} = p_k - 2 r_x^2 y_{k+1} + r_x^2$

and Otherwise, the next point is $(x_k + 1, y_k - 1)$

$p_{k+1} = p_k + 2r_y^2 x_k + 1 + r_x^2 - 2 r_x^2 y_{k+1}$

6. Find out symmetry points in the other three quadrants.
7. Translate each calculated pixel position (x, y) by adding (xc, yc) and plot:
8. Repeat the steps for region 1 until $2r_y^2 x >= 2r_x^2 y$



5.3 IMPLEMENTATION OF ELLIPSE DRAWING A. MIDPOINT ELLIPSE

The algorithm described above shows how to obtain 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).

C Implementation:

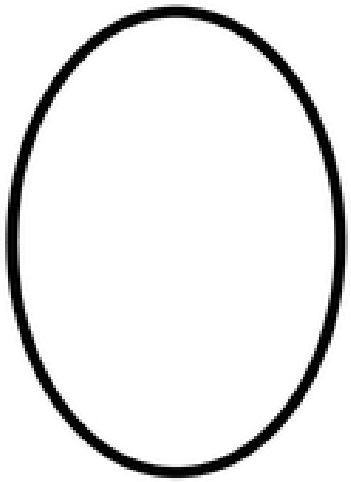
```
#include <graphics.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
```

```
class Mid-Point
{
   float x,y,a, b,r,p,h,k,p1,p2;
   public:
   void get ();
   void cal ();
};
   void main ()
   {
   Mid-Point b;
   b.get ();
   b.cal ();
   getch ();
  }
   void Mid-Point :: get ()
  {
   cout<<"\n ENTER CENTER OF ELLIPSE";
   cout<<"\n ENTER (h, k) ";
         cin>>h>>k;
   cout<<"\n ENTER LENGTH OF MAJOR AND MINOR AXIS";
   cin>>a>>b;
  }
void Mid-Point ::cal ()
{
   /* request auto detection */
   int gdriver = DETECT,gmode, errorcode;
   int midx, midy, i;
   /* initialize graphics and local variables */
   initgraph (&gdriver, &gmode, " ");
   /* read result of initialization */
   errorcode = graphresult ();
   if (errorcode ! = grOK)    /*an error occurred */
   {
      printf("Graphics error: %s \n", grapherrormsg (errorcode);
      printf ("Press any key to halt:");
```

```
    getch ();
    exit (1); /* terminate with an error code */
}
x=0;
y=b;
// REGION 1
p1 =(b * b)-(a * a * b) + (a * a)/4);
{
    putpixel (x+h, y+k, RED);
    putpixel (-x+h, -y+k, RED);
    putpixel (x+h, -y+k, RED);
    putpixel (-x+h, y+k, RED);
    if (p1 < 0)
        p1 += ((2 *b * b) *(x+1))-((2 * a * a)*(y-1)) + (b * b);
    else
    {
        p1+= ((2 *b * b) *(x+1))-((2 * a * a)*(y-1))-(b * b);
        y--;
    }
    x++;
}
//REGION 2
p2 =((b * b)* (x + 0.5))+((a * a)*(y-1) * (y-1))-(a * a *b * b);
while (y>=0)
{
    If (p2>0)
    p2=p2-((2 * a * a)* (y-1))+(a *a);
    else
    {
    p2=p2-((2 * a * a)* (y-1))+((2 * b * b)*(x+1))+(a * a);
    x++;
    }
    y--;
    putpixel (x+h, y+k, RED);
    putpixel (-x+h, -y+k, RED);
```

```
        putpixel (x+h, -y+k, RED);
        putpixel (-x+h, y+k, RED);
    }
    getch();
}
```

Output:



Java Implementation:

```
// Java program for implementing
// Mid-Point Ellipse Drawing Algorithm
import java.util.*;
import java.text.DecimalFormat;
class GFG
{
static void midptellipse(float rx, float ry, float xc, float yc)
{

        float dx, dy, d1, d2, x, y;
        x = 0;
        y = ry;

        // Initial decision parameter of region 1
        d1 = (ry * ry) - (rx * rx * ry) +        (0.25f * rx * rx);
```

```
dx = 2 * ry * ry * x;
dy = 2 * rx * rx * y;
DecimalFormat df = new DecimalFormat("#,###,##0.00000");
        // For region 1
while (dx < dy)
{

        // Print points based on 4-way symmetry
        System.out.println(df.format((x + xc)) +
                                        ", "+df.format((y + yc)));
        System.out.println(df.format((-x + xc)) +
                                        ", "+ df.format((y + yc)));
        System.out.println(df.format((x + xc)) +
                                        ", "+ df.format((-y + yc)));
        System.out.println(df.format((-x + xc)) +
                                        ", "+df.format((-y + yc)));

        // Checking and updating value of
        // decision parameter based on algorithm
        if (d1 < 0)
        {
                x++;
                dx = dx + (2 * ry * ry);
                d1 = d1 + dx + (ry * ry);
        }
        else
        {
                x++;
                y--;
```

```
            dx = dx + (2 * ry * ry);
            dy = dy - (2 * rx * rx);
            d1 = d1 + dx - dy + (ry * ry);
        }
    }

    // Decision parameter of region 2
    d2 = ((ry * ry) * ((x + 0.5f) * (x + 0.5f)))
        + ((rx * rx) * ((y - 1) * (y - 1)))
        - (rx * rx * ry * ry);

    // Plotting points of region 2
    while (y >= 0) {

        // printing points based on 4-way symmetry
        System.out.println(df.format((x + xc)) +
                                        ", " + df.format((y + yc)));
        System.out.println(df.format((-x + xc)) +
                                        ", "+ df.format((y + yc)));
        System.out.println(df.format((x + xc)) +
                                        ", " + df.format((-y + yc)));
        System.out.println(df.format((-x + xc)) +
                                        ", " + df.format((-y + yc)));

        // Checking and updating parameter
        // value based on algorithm
        if (d2 > 0) {
            y--;
            dy = dy - (2 * rx * rx);
```

```
                d2 = d2 + (rx * rx) - dy;
            }
            else {
                y--;
                x++;
                dx = dx + (2 * ry * ry);
                dy = dy - (2 * rx * rx);
                d2 = d2 + dx - dy + (rx * rx);
            }
        }
}

// Driver code
public static void main(String args[])
{
        // To draw a ellipse of major and
        // minor radius 15, 10 centered at (50, 50)
        midptellipse(10, 15, 50, 50);
}
}
```

Python Implementation:

```
# Python3 program for implementing
# Mid-Point Ellipse Drawing Algorithm

def midptellipse(rx, ry, xc, yc):

        x = 0;
        y = ry;
```

```
# Initial decision parameter of region 1
d1 = ((ry * ry) - (rx * rx * ry) +
                        (0.25 * rx * rx));
dx = 2 * ry * ry * x;
dy = 2 * rx * rx * y;

# For region 1
while (dx < dy):

    # Print points based on 4-way symmetry
    print("(", x + xc, ",", y + yc, ")");
    print("(",-x + xc,",", y + yc, ")");
    print("(",x + xc,",", -y + yc ,")");
    print("(",-x + xc, ",", -y + yc, ")");

    # Checking and updating value of
    # decision parameter based on algorithm
    if (d1 < 0):
        x += 1;
        dx = dx + (2 * ry * ry);
        d1 = d1 + dx + (ry * ry);
    else:
        x += 1;
        y -= 1;
        dx = dx + (2 * ry * ry);
        dy = dy - (2 * rx * rx);
        d1 = d1 + dx - dy + (ry * ry);

# Decision parameter of region 2
```

```
    d2 = (((ry * ry) * ((x + 0.5) * (x + 0.5))) +
          ((rx * rx) * ((y - 1) * (y - 1))) -
          (rx * rx * ry * ry));

    # Plotting points of region 2
    while (y >= 0):

        # printing points based on 4-way symmetry
        print("(", x + xc, ",", y + yc, ")");
        print("(", -x + xc, ",", y + yc, ")");
        print("(", x + xc, ",", -y + yc, ")");
        print("(", -x + xc, ",", -y + yc, ")");

        # Checking and updating parameter
        # value based on algorithm
        if (d2 > 0):
            y -= 1;
            dy = dy - (2 * rx * rx);
            d2 = d2 + (rx * rx) - dy;
        else:
            y -= 1;
            x += 1;
            dx = dx + (2 * ry * ry);
            dy = dy - (2 * rx * rx);
            d2 = d2 + dx - dy + (rx * rx);
# Driver code
# To draw a ellipse of major and
# minor radius 15, 10 centered at (50, 50)
midptellipse(10, 15, 50, 50);
```

5.4 SUMMARY

Mid-point Ellipse algorithm is used to draw an ellipse in computer graphics. Midpoint ellipse algorithm plots(finds) points of an ellipse on the first quadrant by dividing the quadrant into two regions.

5.5 REFERENCES FOR FUTURE READING

- Donald Hearn and M Pauline Baker, Computer Graphics C Version -- Computer Graphics, C Version, 2/E, Pearson Education.
- David F. Rogers, James Alan Adams, Mathematical elements for computer graphics , McGraw-Hill, 1990
- Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing (3rd Edition), Pearson Education.
- S. Sridhar-Digital image Processing, Second Edition, Oxford University Press
- Anil K. Jain -Fundamentals of digital image processing. Prentice Hall, 1989

*****

UNIT V

6

OUTPUT PRIMITIVES & ITS ALGORITHM

Unit Structure

6.0 Objective
6.1 Introduction
6.2 Implementation of curve
6.2.1 Bezier curve and surface
6.2.2 Properties of Bezier curve
6.2.3 Design techniques using Bezier curve
6.2.4 Cubic Bezier curve
6.2.5 Bezier surface
6.3 Summary
6.4 Unit End Exercise
6.5 References for Future Reading

6.0 OBJECTIVE

This chapter will able you to understand the following concept:

- Bezier curve and surface
- properties of Bezier curve
- Design techniques using Bezier curve
- Cubic Bezier curve
- Bezier surface

6.1 INTRODUCTION

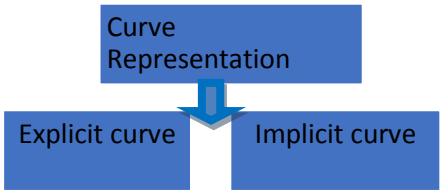
The object which we can see around us is of different shapes either visible as 2D or 3D. Modelling of objects in computer graphics uses the primitives such as lines, circle, ellipse etc. Modeling of geometric objects generally combines these basic primitives to create another object. For example, to generate curves in computer graphic, multiple lines are conducted with each other using same data points.

Dots + Lines > curve

We can see that in graphics modeling, the curve generated by connecting multiple lines are not smooth but in real world curves appears smooth. To design curve which also have some smoothness we require high design

approximation, which can be represented as explicit, implicit parametric and non-parametric.

Curves are represented mainly in two ways:



Explicit Representation: The explicit form of curve is in two dimensions gives the value of one variable i.e. the dependent variable, in terms of other independent variable.

In x, y plane it is written as y=f(x)

Implicit Representation: in two dimension implicit curve can be represented by the equation f(x, y)= 0

- The implicit form I less coordinate system dependent than is the explicit form.
- In three dimensions, the implicit form f(x, y,z)=0
- Curve in three dimensions are not as easily represented in implicit curve.
- We can represent a curve as the intersection. If it exists, of the two surface: f(x, y, z) = 0, g(x, y, z) = 0

6.2 IMPLEMNTATION OF CURVE

The spline approximation method was developed by the French engineer Pierre Bezier for use in the design of Renault automobile bodies. Bezier splines have a multiple property that one can use very conveniently for curve and surface design. They are also easy to implement. The above listed all reasons makes most use of Bezier splines in CAD systems, in general graphics packages (such as GL on Silicon Graphics systems), and in assorted drawing and painting packages (such as Aldus Super Paint and Cricket Draw). A most important property of a Bezier curve is that it always passes between the first and last control of the point

Curve Function:

Routines for circle, splines, and other commonly used curves are included in many graphics packages. The PHIGS standard does not provide explicit functions for these curve, but it does include

greneralizedDrawingPrimitive (n,wcpoint, id, datalist)

where wcpoints is a list of n coordinate positions, datalist contain noncoordinate data value, and parameter id select the desired function. At a particular installation, a circle might be referenced with id=1, an ellipse with id=2, and so on.

As an example of the definition of curve through this PHIGS function a, circle (id=1, say) could be specified by assigning the two center coordinate Val-uses to wcpoints and assigning the radius value to datalist. The generalized drawing primitive would then reference the appropriate algorithm, such as file midpoint method, to generate the circle. With interactive input, a circle could be defined two coordinate points: the center position and a point on the circumference. Similarly, interactive specification of an ellipse can be done with three points. The two foci and a point on the ellipse boundary, all stored in wcpoints. For an ellipse in standard position wcpoint could be assigned only the center coordinate, with datalist assigned the value for rx and ry. Splines defined with control points would be generated by assigning the control point coordinate to wcpoints.

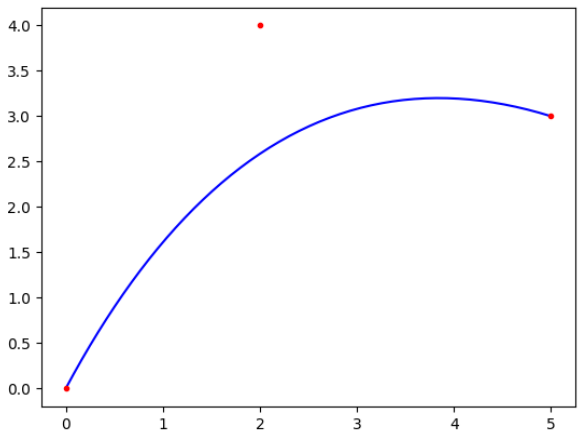
Function to generate circle and ellipse often include the capability of drawing curve section by specifying parameter for the line endpoints. Expanding the parameter list allow specification of the beginning and ending angular value for an arc, as illustrated in fig 3-27. Another method for designation a circular or elliptical arc is to input the beginning and ending coordinate positions of the arc.

1. Quadratic Bézier Curve:

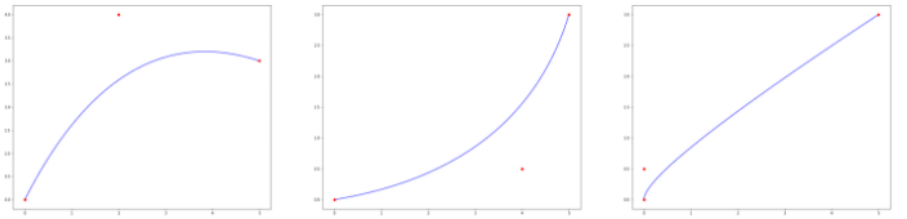
With 3 control points the quadratic Bézier curve having the degree of 2-point p(t).

Note that P(t) will not return a numeric, but it will return a point on the curve. Now we have to choose three control points and it generate line between that three points from the range 0-1.

In the diagram we can see that the curve initiate and ends at the starting and ending control points. Any number point it will draw a curve. The selected t ranges from 0 to 1, hence one can prove this by taking P(t) at t=0 and t=1.



The curve goes from p0 and p1 and it will determine the slope of the curve also the shape of the curve. It will always contain polygon shape formed by specified control points. Hence the specific polygon is called as control polygon or Bézier polygon, as it is controlled by the points determine by the curve. With this property any number of control points can holds the polygon.



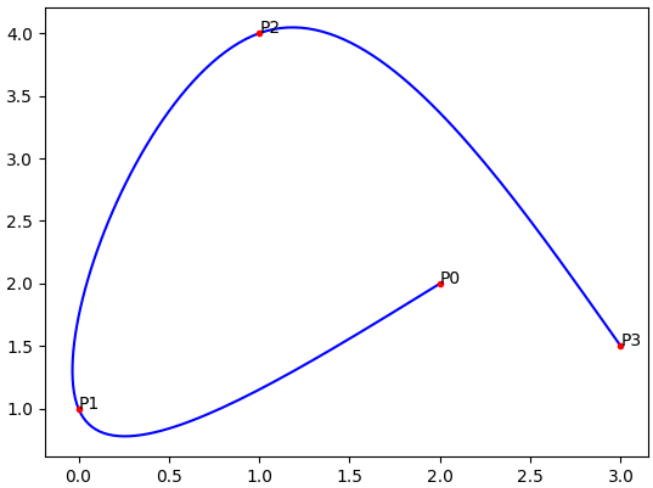
2. Matrix representation:

Using matrix multiplication, we can actually represent the Bézier curve, which we can use in splitting the Bézier curve.

Matrix M cane be used for holding all the information about the quadratic Bézier curve into one matrix. The steps proceeded by matrix we need to take the coefficients of that matrix in all these steps, hence the coefficients of the matrix are connected to the polynomial in front of each Pi, here we have to expanded form of the Bernstein polynomial.

3. Interpolation:

Bézier curves can be drawn with smooth points hence the drawn curve is appearing smooth with a predefined set of points. The formula used for this is of P(t) produces points and is not of the form y=f(x), so one x can have multiple y's (basically a function that can "go backward").



6.2.1 Bezie Curve S And Surface:

Any number of control points are available in brazier curve. One can determine the degree of brazier polynomial with the number of control points to their relative positions. A Bezier curve can be specified with

boundary conditions with the interpolation splines, with a characterizing matrix, or with blending functions. The blending function specification is the most convenient in brazier curve.

Suppose we are given n+1 control-point positions: pk = (xk, yk, zk), with k varying from 0 to n. the following position vector P(u) can be produced by blended coordinate points, which describes the path of an approximating Bezier polynomial function between P0 and Pn.

$$P\ (u) = \sum_{k=0}^{0}\ (p_k(u))\ 0 \le \mathrm{u} \le 1$$

The Bezier blending functions BEZk,n(u) are the Bernstein polynomials:

$$BEZ_{K,N}(u) = C(n,k)\ \mathrm{u}^k (1-u)^{n-k}$$

Where the C(n,k) are the binomial coefficients:

$$C\ (n,k) = \frac{n:}{k:(n-k):}$$

Equivalently, we can define Bezier blending functions with the recursive calculation

$$BEZ_{k,n}(u) = (1-u)BEZ_{k,n-1}(u) + uBEZ_{k-1,n-1}(u),\ n > k \ge 1$$

with BEZK,K = uk, and BEZ0,k = (1-u)k. Vector equation 1 represents a set of three parameters equations for the individual curve coordinates:

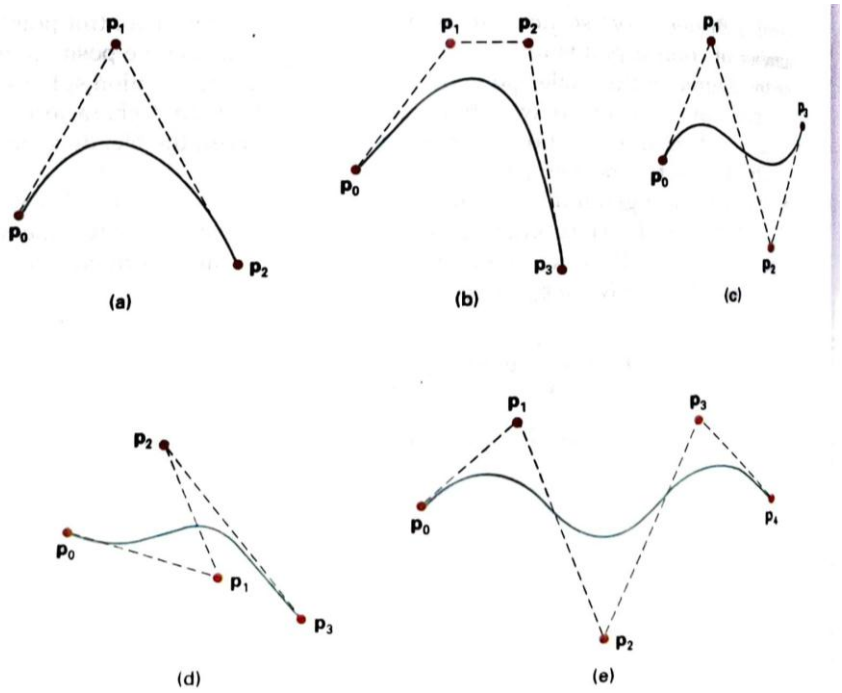
$$x(u) = \sum_{k=0}^{0}\quad (\mathrm{x}_k(u))$$

$$y(u) = \sum_{k=0}^{0}\quad (\mathrm{y}_k(u))$$

$$z(u) = \sum_{k=0}^{0}\quad (\mathrm{z}_k(u))$$

in a Bezier curve the degree of polynomial is one less than the number of control points used in Bezier curve. Parabola can be created by 3 points, with 4 points its generate cubic curve and so on. Following figure demonstrates the appearance of some Bezier curves for various selections of control points in the xy plane (z=0). The degenerated Bezier curve can be determining by certain control-point placements. For example, with three collinear control points a straight line segment is generated a Bezier curve. The control points at one location can produce Bezier curve in a single point.

Painting and drawing packages contain Bezier curves which is mostly used, as they are very easy to implement and powerful to draw a curve hence it can also use in CAD systems. Recursive calculation can be used for determining coordinate position of Bezier curve. For example, proceeded binomial coefficients can be counted as



$$C(n,k) = \frac{n-k+1}{k}\, C(n, k-1)$$

For n > k. The following example program illustrates a method for generating Bezier curves.

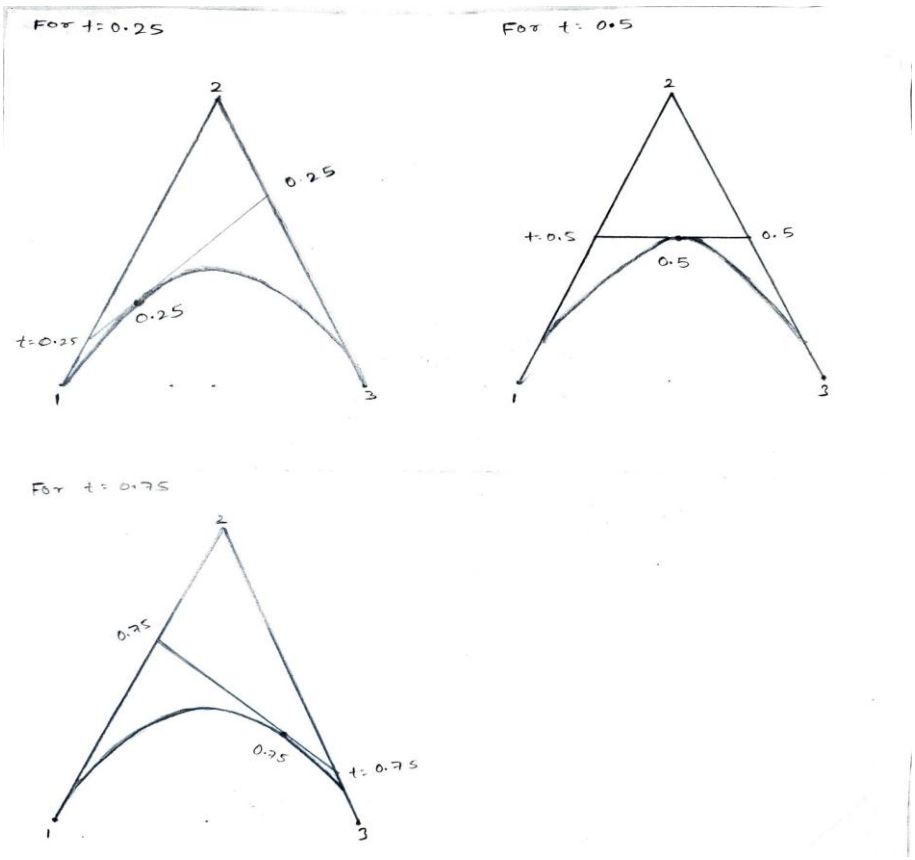
De Casteliau's Algorithm:

1. Draw control points. As named are labeled 1, 2, 3.
2. Draw the line between control points $1 \rightarrow 2 \rightarrow 3$.
3. The parameter t moves from 0 to 1. In the illustration above the step0.05 is used the curve goes over 0,0.05,0.1,0.15,.0.95, 1. For each of these values of t.

On proportional distance draw each line on point t located from its starting. we've two points, so draw two lines.

For illustration, for t = 0 – both points will be at the starting of points, and for t = 0.25 – on the 25% of length from the starting, for t = 0.5 – 50 (the middle), for t = 1 – in the end of line. Connect the points. For t = 0.25 For t = 0.5

4. Now in the line take a point on the distance proportional to the same value of t. That is, for t = 0.25 (the left picture) we've a point at the end of the left quarter of the line, and for t = 0.5 (the right picture) – in the middle of the line.
5. As t runs from 0 to 1, every value of t adds a point to the wind. The set of similar points forms the Bezier curve.



6.2.2 Properties Of Bezier Curve:

A most important property of a Bezier curve is that it always passes between the first and last control of the point. Which specify the boundary conditions at both ends of the curve are

$$P(0) = p_0$$

$$P(1) = p_n$$

Parametric first derivatives value of a Bezier curve at the endpoints can be calculated from control-point coordinates as

$$P(0) = -np_0 + np_1$$

$$P(1) = -np_{n-1} + np_1$$

Thus, at the starting of the curve the slope of the line is going through the first two control points, and at the end, slope of the curve is along the line joining the last two endpoints. Similarly, the calculation of the second parametric derivatives of a Bezier curve at the endpoints

$$P''(0) = n(n-1)[(p_1 - p_2)-(p_1 - p_0)]$$

$$P''(0) = n(n-1)[(p_{n-2} - p_{n-1})-(p_{n-1} - p_n)]$$

One of the most attractive property of Bezier curve is that, All the control points of Bezier curve are lies within the convex (convex polygon boundary) of the control points. The blending function property is used in the above curve generation, hence they all are positive and their sum is always 1,

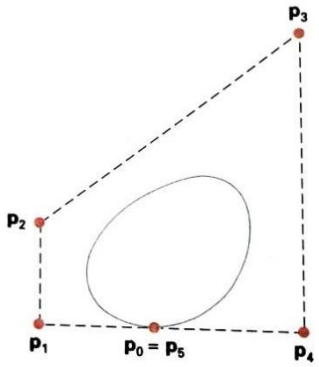
$$\sum_{k=0}^{n} (p_k(u)) = 1$$

The weighted sum of control points position can be simply specifying the curve position. The convex-hull property for a Bezier curve ensures that the polynomial smoothly follows the control panels without erratic oscillations.

Bezier curve Properties in Short:

- Joining the control points segments will decide the shape of the control polygon.
- It always travels between initial and end control points.
- They are contained in the convex hull of their defining control points.
- The degree of polynomial defining the curve segment is one less that the number of defining polygon point. Therefore, for 4 control point, the degree of the polynomial is 3, i.e. cubic polynomial.
- Shape of the polygon can generally draw with the Bezier curve representation.
- The tangent vector direction aims at the end points is same with the vector determined by initial and end segments.
- The control points of polynomial are smoothly followed by the Bezier curve ca make the convex hull property for it.
- No straight line intersects a Bezier curve more times than it intersects it control polygon.
- They are invariant under an affine transformation.
- Bezier curves can handle the global changes like, moving a control point alters the shape of the whole curve.
- A given Bezier curve can be subdivided at a point u=u0 int two Bezier segments which join together at the point corresponding to the parameter value u=u0.

6.2.3 Design Technique Using Bezier Curve:

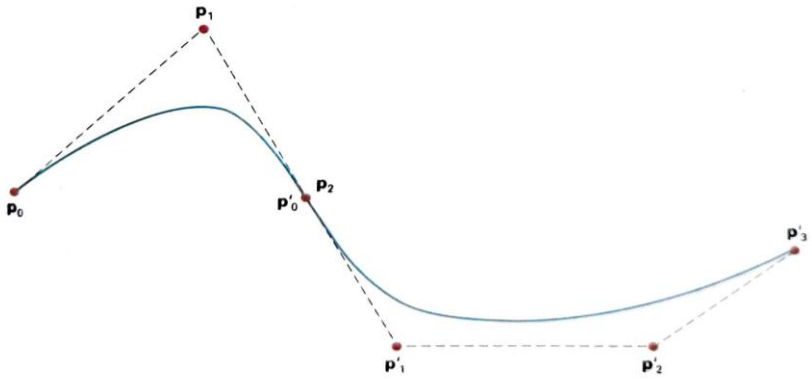


The first and the last control points at the same position can be called as closed Bezier curve, as in figure shown.

Also, at a single coordinate position multiple control points gives more weight to that position. In below figure, two control points can be taken by a single coordinate position as input, and the output curve is puled nearer to this position.

With the help of polynomial calculation function of higher degree, we can draw Bezier curve with n number of control points. Complicated Bezier curve ca be generated with doing fractals of lower degree together to draw a curve, to get the better control we do the smaller section of the curve and with this we can get better control over the shape of the curve in small region. With the help of property of Bezier curves pass through endpoints, it is easy to match curve sections (zero-order continuity). also, the property of Bezier curves the tangent to the curve at an endpoint is along the line joining that endpoint to the adjacent control point. Therefore, to obtain first-order continuity between curve sections, we can pick control points p'0 and p'1 of a new section to be along the same straight line as control points pn-1 and pn of the previous section , when the number of control points are same for two curve sections, we can obtain C1 continuity by choosing the first control point of the new section as the last control point of the previous section and by positioning the second control point of the new section at position

$$p_n + (p_n - p_{n-1})$$



In the above figure, Two Bezier section can have formed Piecewise approximation curve. Zero-order and first-order continuity are attained between curve sections by setting $p'_0 = p_2$ and by making points p_1, p_2, and p'_1 collinear.

Thus, the number of collinear and equally spaced control points are three.

We obtain C^2 continuity between two Bezier section by calculating the position of the third control point of a new section in terms of the positions of the last three control points of the previous section as

$$P_{n-2} + 4(p_n - p_{n-1})$$

In cubic curve, second order continuity of Bezier curve requirement is specifying unnecessarily restrictive. In four control points cubic curve this is specially indicate. In this case, we have to fix the position of first three control points and the other one point will use to adjust the shape of the curve segment in second-order continuity.

6.2.4 Cubic Bezier Curve:

Many Graphics packages provide only cubic spline functions. This given reasonable flexibility while avoiding the increased calculations needed with higher-order polynomials. with four control points the cubic Bezier curve is generated. The four blending functions for cubic Bezier curves, obtained by substituting n=3 in Eq.

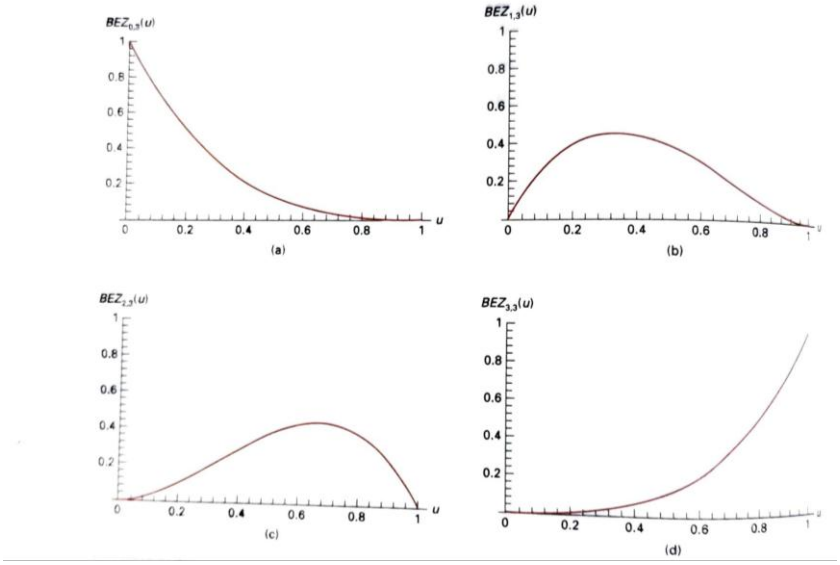
$$BEZ_{0,3}(u)=(1-u)^3$$

$$BEZ_{1,3}(u)=3u(1-u)^2$$

$$BEZ_{2,3}(u)=3u^2(1-u)$$

$$BEZ_{3,3}(u)=u^3$$

As you can see in the diagram first we have to plots of the four cubic Bezier blending functions are given in Fig. The shape of the curve is decided from the blending function which shows the influence of the control points control points influence range from 0 to 1. At u=0, the only nonzero blending function is $BEZ_{0,3}$, which has the value 1.At u=1, the only nonzero function is $BEZ_{3,3}$,with a value of 1 at that point. Thus, the cubic Bezier curve will always pass through control points p_0 and p_3. The other functions, $BEZ_{1,3}$ and $BEZ_{2,3}$, influence the shape of the curve at intermediate values of parameter u, so that the resulting curve tends toward points p_1 and p_2. Blending function $BEZ_{1,3}$ is maximum at u=1/3, and $BEZ_{2,3}$ is maximum at u=2/3.



We note in above figure with considering the range of parameter u, the four blending functions are non-zero. Thus, Bezier curves do not allow for local control of the curve shape. The entire curve will be affected if we try to reposition one of the control point.

At the end positions of the cubic Bezier curve, the parametric first derivatives are

$$P'(0)=3\ (p_1-p_0), \quad P'(1)=3(p_3-p_2)$$

And the parametric second derivatives are

$$P''(0)=6(p_0-2p_1+p_2), \quad P''(1)=6(p_1-2p_2+p_3)$$

To construct continuity between sections We can use these expressions for the parametric derivatives C^1or C^2.

The blending function can be extended with the polynomial expressions, the cubic Bezier point can be written in matrix form

$$P(u)= [u^3\ u^2\ u\ 1].\ M_{BEZ}.\ [p0 \quad p1 \quad p2\ p3\ ]$$

Where the Bezier matrix is

$$M_{BEZ}= [\ -1 \quad 3-3 \quad 1 \quad 3-3 \quad 3 \quad 0-3 \quad 3 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0\ ]$$

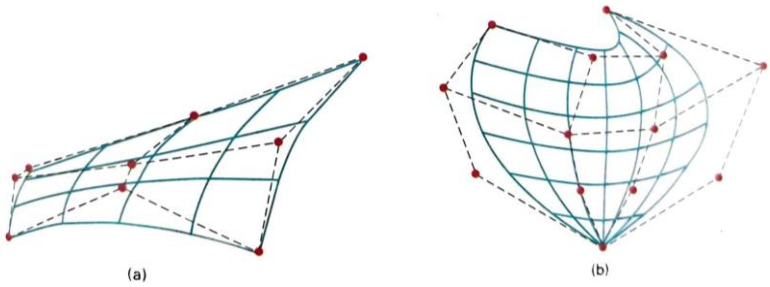
The additional parameter can also allow to do adjustment of curve "tension" and "bias", as we did with interpolating splines. But the more useful B-splines, as well as β-splines, provide this capability.

6.2.5 Bezier Surface:

Two sets of orthogonal Bezier curves can be used to design an object surface by specifying by an input mesh of control points. The parametric vector function for the Bezier surface is formed as the Cartesian product of Bezier blending functions:

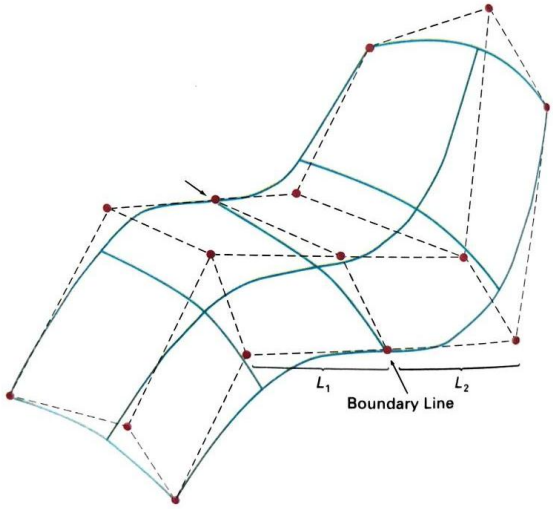
$$P(u,v) = \sum_{j=0}^{m} \sum_{k=0}^{n} (p_{j,k}(v) BEZ_{k,n}(u))$$

With pj,k specifying the location of the (m+1) by (n+1) control points.



The above figure illustrates two Bezier surface plots. The dashed lines are used to connect the control points, and constant v can be defined by the solid lines. The 0-1 ranges interval can be varying by plotted Each curve of constant u over v, with u fixed at one of the values in this unit interval. Constant v by Curves plotted similarly.

Bezier curve and Bezier surfaces shares the same properties, and they also help to achieve interactive design application with convenient method. For each surface patch, we can select a mesh of controls points in the xy "ground" plane, then we choose elevations above the ground plane for the z-coordinate values of the control points. Using the boundary constraints, the Patches can then be pieced together.



The figure illustrates a surface formed with two Bezier sections. With the zero-order continuity at the boundary line one can get smooth transition from one section to the other curve. matching control points at the boundary can specify the Zero-order continuity. First-order continuity is obtained by choosing control points along a straight line across the boundary and by maintaining a constant ratio of collinear line segments for each set of specified control points across section boundaries.

6.3 SUMMARY

The spline approximation method was developed by the French engineer Pierre Bezier for use in the design of Renault automobile bodies. Bezier splines have a multiple property that one can use very conveniently for curve and surface design. Painting and drawing packages contain Bezier curves which is mostly used, as they are very easy to implement and powerful to draw a curve hence it can also use in CAD systems. Many Graphics packages provide only cubic spline functions. Bezier curve and Bezier surfaces shares the same properties, and they also help to achieve interactive design application with convenient method. With the help of polynomial calculation function of higher degree, we can draw Bezier curve with n number of control points. Bezier curve and Bezier surfaces shares the same properties, and they also help to achieve interactive design application with convenient method. The blending function can be extended with the polynomial expressions, the cubic Bezier point can be written in matrix form

6.4 UNIT END EXCERCISE

- Explain the concept of Bezier curve and surface.
- Explain De Casteliau's Algorithm.
- Explain properties of Bezier curve in detail.
- How design can be done through Bezier curve.
- Explain cubic Bezier curve.
- Explain implementation of Bezier curve
- Explain Curve function in detail.

6.5 REFERENCES FOR FUTURE READING

- Computer Graphics C version 2nd Edition by Donald D. Hearn and M. Pauline Baker
- Computer Graphics A programming approach 2nd Edition by Steven Harrington McGraw Hill
- Fundamental of Computer Graphics 3rd Edition by Peter Shirley an Steven Marschner
- Computer Graphics from Scratch: A Programmer's Introduction to 3D Rendering by Gabriel Gambetta

*****

MODULE VI

7

OUTPUT PRIMITIVES & ITS ALGORITHMS

Unit Structure

7.1 Objectives
7.2 Definition
7.3 Introduction
7.4 Polygon Filling
7.5 Seed fill algorithms
7.6 Boundary Fill Algorithm
7.7 Flood Fill Algorithm
7.8 Scan Line Algorithm
7.9 Summary
7.10 Unit End Exercise
7.11 Reference for further reading

7.1 OBJECTIVES

- The purpose is to colour entire area of pixels connected with each other & then finally get the shape of the object on the screen.
- Basically filling up of polygons using horizontal lines or scanlines.
- The purpose is to fill the interior pixels of a polygon given only the vertices of the figure.

7.2 DEFINITION

The process of colouring or highlighting the pixels with any colour which lies inside the polygon is known as polygon filing.

7.3 INTRODUCTION

The process of colouring or highlighting the pixels with any colour which lies inside the polygon is known as polygon filing. For polygon filing the requirements are:

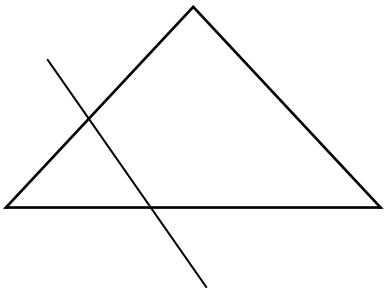
A digital representation of the shape must be closed.

A test for determining if a point is inside or outside of the shape.

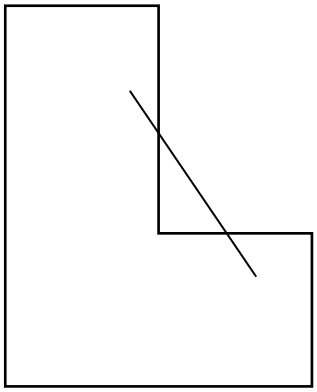
A rule or procedure for determining the colours of each point inside the shape.

Introduction to Solid Area Scan Conversion:

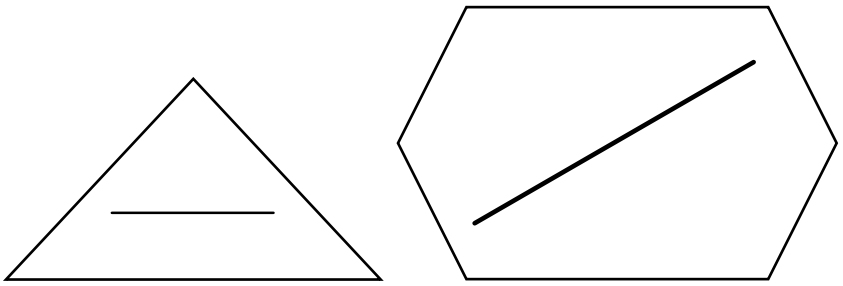
Polygon: It is a figure which is formed by connecting line segments in a closed manner. Polygons are categorised as concave & convex polygon.



Concave Polygon: A line segment joining any two points within the polygon which is not completely inside the polygon is called a concave polygon.



Convex Polygon: A line segment joining any two points within the polygon which is completely inside the polygon is called a convex polygon.



Inside & Outside Test of a Polygon:

This test is used to check whether a point is inside or outside a polygon. There are to methods:

a) Even –odd Test:

Take a point & draw a line extend till the outside of a polygon. Count the point of intersection of the segment with the edge of the polygon. If the

number of polygon edges crossed by the line is odd, then this point is an interior point. Otherwise point is an exterior point.

In the diagram point P1 has one intersection which is odd so P1 is an interior point. Point P2 has two intersections which is even so P2 is an exterior point.

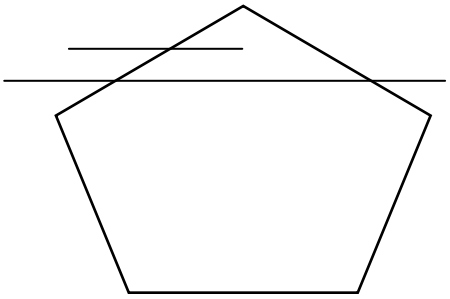
When the line segment intersects the vertex of a polygon then the following rules are used:

Count is even: If the other end points o0f the two segments meet at the intersecting vertex.

Count is odd: if both the end points lies on the opposite side of the line segment.

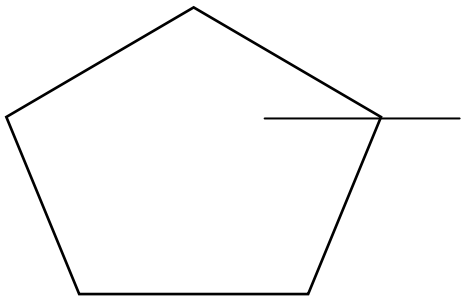
P1

P2



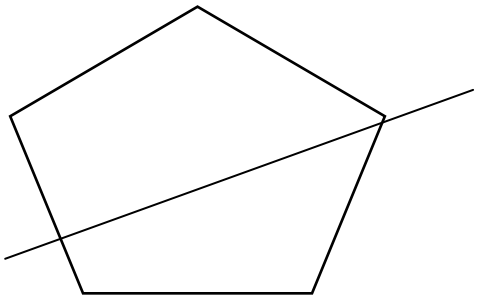
For Vertices: If the (intersection) point is the vertex of the polygon. Then check the edges. If they are in the same side of the polygon direction, then it is counted as even number of intersection.

P3 count (1+2)
So odd count hence point is inside the polygon



If the intersecting edges are in the opposite direction, then the intersection point is counted as odd number of intersection.

P4=count is odd + one (odd) =even =point is exterior to the polygon

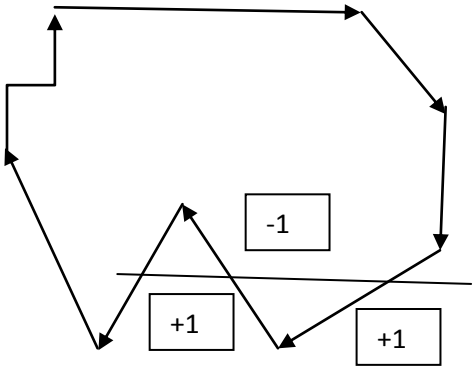


b) Winding Number Test:

In this picture, a line segment running from outside the polygon to the point given in the question & consider a polygon sides which it crosses. Assign direction numbers to the boundary line crossed & sum these direction numbers.

Let P1 is a test point & a line segment is drawn from the outside of the polygon up to the point P1. The edge can be drawn starting below the line, cross it & end above the line (direction number-1) and starting above the line, cross it & end below the line (direction number 1)

Take the sum of these direction numbers, if the value is non-zero then the point is inside a polygon otherwise the point is outside a polygon.



Sum=+1-1+1 =+1

Sum ≠ 0 (inside the polygon)

Sum = non zero

Sum=0 (outside the polygon)

Point P1 is inside a polygon.

7.4 POLYGON FILLING

There are two approaches to fill the polygon:

1. Seed fill algorithms
2. Scan Line Algorithms

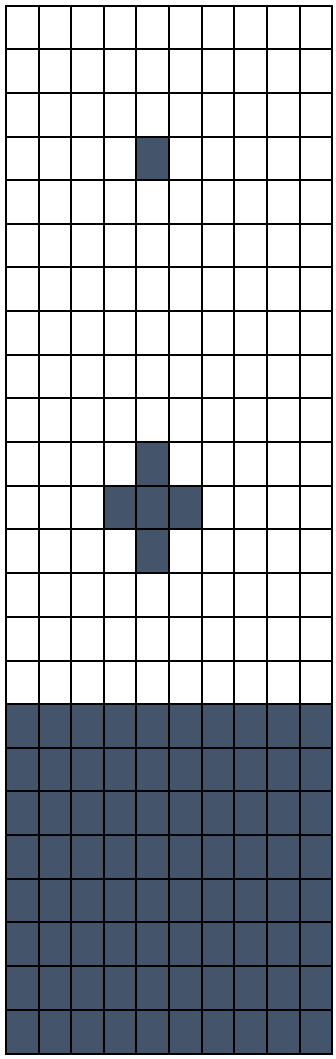
7.5 SEED FILL ALGORITHMS

In this fill a polygon starts with a point known as a seed from inside polygon & highlight outwards from this point through neighbouring pixels until boundary pixels are encountered. This is known as seed fill because color flows from seed pixel until reaching the polygon boundary.

Algorithm:

1. Select a seed point inside the region

2. Move outwards from the seed point
3. If pixel is not set, set pixel.
4. Process each neighbour of pixel that is inside the region.



Seed point or seed pixel

Move towards its neighbouring pixel left, right, top & bottom seed fill algorithm

Stop when the entire region is filled with the pixel color

It is further classified as flood fill algorithm (that fills an interior region) & boundary fill algorithm (that fills the boundary defined region.

Filled Area Primitives:

Region filling is the process of filling image or region. Filling can be of boundary or interior region as shown in fig. Boundary Fill algorithms are used to fill the boundary and flood-fill algorithm are used to fill the interior.

7.6 BOUNDARY FILL ALGORITHM

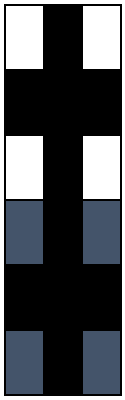
Starting with the seed point, i.e. any point inside the polygon examine the neighbouring pixel to check whether boundary pixel is reached. If boundary pixels are not reached pixels are highlighted and the process is continued until boundary pixel is reached.

Algorithm:

1. Region described by a set of bounding pixels.
2. A seed pixel is set inside the boundary
3. Check if this pixel is a boundary pixel or has already been filled.
4. If no to both, then fill it & make neighbours new seeds.

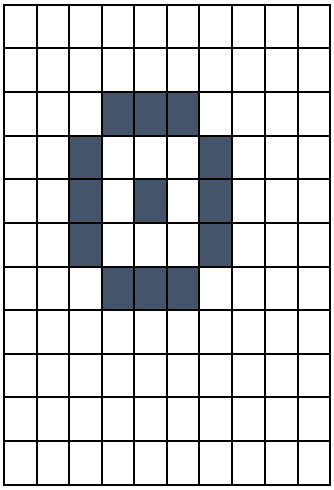
It is defined either with four connected or eight connected regions. In four connected regions every pixel can be reached by a combination of moves in four directions – left, right, top & bottom.

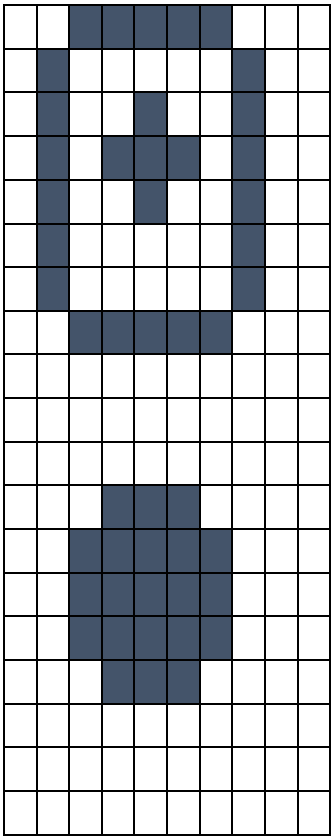
In eight connected regions every pixel can be reached by a combination of moves in two horizontals, two vertical & four diagonal directions.



4-connected region 8-connected region

Example of boundary fill using 4- connected region





seed pixel, boundary pixel	Move towards neighbouring pixels left, right, top & bottom	Stop when the entire boundary region is filled with colours

Pseudo code of 4-connected region boundary fill:

```
Void boundaryfill(int x, int y, int fill, int boundary)
{
Int current;
Current=getPixel(x,y)
If ((current! =boundary) && (current! =fill))
{
setColor(fill);
setPixel(x,y);
boundaryfill(x+1, y, fill, boundary);
boundaryfill(x-1, y, fill, boundary);
boundaryfill(x, y+1, fill, boundary);
boundaryfill(x, y-1, fill, boundary);
}
}
```

C++ program to fill the rectangle using boundary fill algorithm:

```
#include<iostream.h>
#include<conio.h>
#include<dos.h>
#include<graphics.h>
void b_fill(int x, int y, int bc, int fc)
{
int p;
p=getpixel(x,y)
if((p!=bc) && (p!=fc)
        {
        putpixel(x,y,fc);
        b_fill(x,y+1,bc,fc);
        b_fill(x,y-1,bc,fc);
        b_fill(x+1,y,bc,fc);
        b_fill(x-1,y,bc,fc)
        }
}
void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"c:\\tc\\bgi");
settextstyle(5,HORIZ_DIR,3);
outtextxy(100,100,"Program to boundary fill");
setcolor(10);
rectangle(260,200,310,260);
delay(1000);
b_fill(280,250,10,12);
getch();
}
```

Advantages of Boundary-Fill over Flood-Fill:

1. **Flood:** fill regions are defined by the whole of the region. All pixels in the region must be made the same colour when the region is being created. The region cannot be translated, scaled or rotated.

2. **4-connected boundary:** fill regions can be defined by lines and arcs. By translating the line and arc endpoints we can translate, scale and rotate the whole boundary-fill region. Therefore 4-connected boundary-fill regions are better suited to modelling.

Disadvantages of Boundary-Fill over Flood-Fill:

1. **In boundary:** fill algorithms each pixel must be compared against both the new colour and the boundary colour. In flood-fill algorithms each pixel need only be compared against the new colour. Therefore, flood-fill algorithms are slightly faster.

2. **Boundary:** fill algorithms can leak. There can be no leakage in flood-fill algorithms.

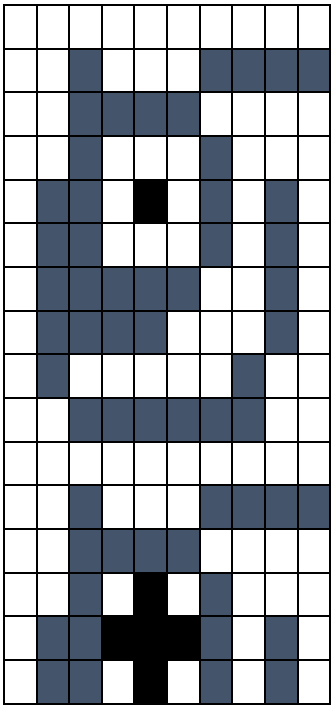
7.7 FLOOD FILL ALGORITHM

Sometimes it is required to fill in an area that is not defined within a single colour boundary. In such cases area can be filled by replacing a specified interior colour instead of searching for a boundary. This approach is known as flood fill.

Algorithm:

1. Region is a patch of like-coloured pixels.
2. A seed pixel is set and a range of colours is defined.
3. Check if the pixel is in the colour range.
4. If yes, fill it and make the neighbours new seed.

Example of Flood fill using 4-connected region:



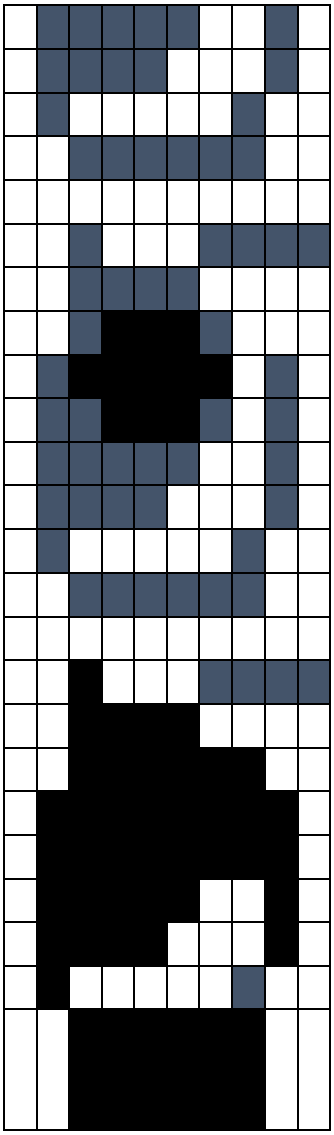


Image after 4-connected flood fill

C++ program to fill the rectangle using flood fill algorithm:

```
#include<iostream.h>
#include<conio.h>
#include<dos.h>
#include<graphics.h>
void flood(int,int,int,int)
void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"c:\\tc\\BGI");
rectangle(50,50,100,100);
flood(55,55,0,4);
```

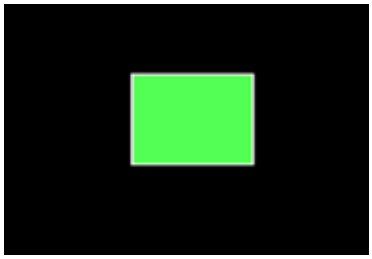
```
getch();
}
void flood(int x, int y, int old, int new1);
{
int current;
current=getpixel(x,y);
if(current==old && current!=new1)
{
setcolor(new1);
delay(10);
putpixel(x,y,new1);
flood(x+1,y,old,new1);
flood(x-1,y,old,new1);
flood(x,y+1,old,new1);
flood(x,y-1,old,new1);
}
}
```

Program: To implement four-connected flood fill algorithm

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<dos.h>
void flood(int,int,int,int);
void main()
{
   intgd=DETECT,gm;
   initgraph(&gd,&gm,"C:/TURBOC3/bgi");
   rectangle(50,50,250,250);
   flood(55,55,10,0);
   getch();
}
void flood(intx,inty,intfillColor, intdefaultColor)
{
   if(getpixel(x,y)==defaultColor)
   {
```

```
        delay(1);
        putpixel(x,y,fillColor);
        flood(x+1,y,fillColor,defaultColor);
        flood(x-1,y,fillColor,defaultColor);
        flood(x,y+1,fillColor,defaultColor);
        flood(x,y-1,fillColor,defaultColor);
    }
}
```

Output:



Program: To implement 8-connected flood fill algorithm:

```
#include<stdio.h>
#include<graphics.h>
#include<dos.h>
#include<conio.h>
void floodfill(intx,inty,intold,intnewcol)
{
        int current;
        current=getpixel(x,y);
        if(current==old)
        {
                delay(5);
                putpixel(x,y,newcol);
                floodfill(x+1,y,old,newcol);
                floodfill(x-1,y,old,newcol);
                floodfill(x,y+1,old,newcol);
                floodfill(x,y-1,old,newcol);
                floodfill(x+1,y+1,old,newcol);
                floodfill(x-1,y+1,old,newcol);
                floodfill(x+1,y-1,old,newcol);
```

```
                floodfill(x-1,y-1,old,newcol);
        }
}
void main()
{
        intgd=DETECT,gm;
        initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
        rectangle(50,50,150,150);
        floodfill(70,70,0,15);
        getch();
        closegraph();
}
```

Output:

Advantages of Flood Fill:

Flood fill colors an entire area in an enclosed figure through interconnected pixels using a single color.

It is an easy way to fill color in the graphics. One just takes the shape and starts flood fill.

The algorithm works in a manner so as to give all the pixels inside the boundary the same color leaving the boundary and the pixels outside. Flood Fill is also sometimes referred to as Seed Fill as you plant a seed and more and more seeds are planted by the algorithm.

Each seed takes the responsibility of giving the same color to the pixel at which it is positioned. There are many variations of Flood Fill algorithm that are used depending upon requirements.

Flood fill Vs Boundary fill:

Though both Flood fill and Boundary fill algorithms color a given figure with a chosen color, they differ in one aspect. In Flood fill, all the connected pixels of a selected color get replaced by a fill color. On the other hand, in Boundary fill, the program stops when a given color boundary is found.

Advantages of rendering polygons by scan line method:

i. The max and min values of the scan were simply found.

ii. The intersection of scan lines with edges is simply calculated by a simple incremental method.

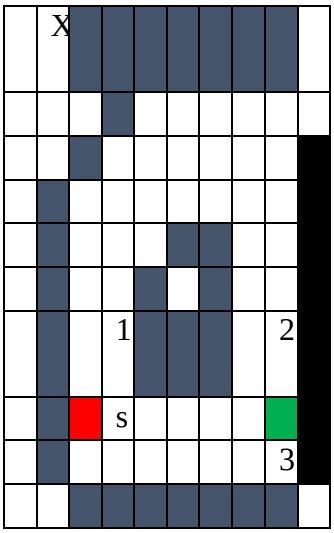
iii. The depth of the polygon at each pixel is simply calculated by an incremental method.

7.8 SCAN LINE ALGORITHM

This algorithm takes one pixel at a time out of unfilled span of pixels. It processes pixels in raster pattern i.e from left to right moving along top to bottom in the scan line region.

Algorithm:

1. A seed pixel is selected and colour it.
2. The left, right, top, bottom line of the seed pixel is filled until a boundary is found.
3. The extreme left and extreme right unprocessed pixel in the span are saved as xleft and xright.
4. The scan line above and below the current scan line are examined in the range of xleft to xright in any contiguous span of either boundary pixel. If any span is found cross over.



Red color – xleft

Green color –xright

s-seed pixel

Black color- Boundary region filled With color.

Advantages of Bresenham's Circle Drawing Algorithm:

1. The Bresenhem's circle drawing algorithm uses integer arithmetic which makes the implementation less complex.
2. Due to its integer arithmetic, it is less time-consuming.
3. This algorithm is more accurate than any other circle drawing algorithm as it avoids the use of round off function.

Disadvantages of Bresenham's Circle Drawing Algorithm:

1. This algorithm does not produce smooth results due to its integer arithmetic as it fails to diminish the zigzags completely.
2. The Bresenhem's circle drawing algorithm is not accurate in the case of drawing of complex graphical images.

7.9 SUMMARY

How to fill the polygon using seed fill algorithms are explained along with flood fill & boundary fill algorithm?

Scan line algorithm is also explained.

7.10 UNIT END EXERCISE

1. Explain the steps required to fill the polygon using seed fill algorithm.
2. What is polygon filling? What are the requirements to fill a polygon?
3. How a seed fill algorithm is used to fill a polygon?
4. Explain the steps required to fill the polygon using Flood Fill algorithm.
5. Explain Boundary fill and Flood fill algorithm with the help of a diagram?
6. Write a c++ program to fill the rectangle using Flood fill algorithm.
7. Write a c++ program to fill the rectangle using Boundary fill algorithm.

7.11 REFERENCES FOR FURTHER READING

- https://www.geeksforgeeks.org/boundary-fill-algorithm/

- https://techdifferences.com/difference-between-flood-fill-and-boundary-fill-algorithm.html
- Computer_Graphics_C_Version_by_Donald_Hearn_and_M_Pauline_Baker_II_Edition
- Computer graphics by Atul P. Godse, Dr. Deepali A. Godse
- https://www.tutorialspoint.com/computer_graphics/polygon_filling_algorithm.htm
- https://www.javatpoint.com/computer-graphics-boundary-filled-algorithm

*****

MODULE VII

8

2D GEOMETRIC TRANSFORMATIONS & CLIPPING

Unit Structure

8.1 Implementation of Two Dimensional Transformations:

8.1.1 What is Transformation?

8.2 Translation

8.2.1 What is Cohen Sutherland Line Clipping algorithm?

8.2.2 Source Code

8.2.3 Output

8.3 Rotation

8.3.1 What is Rotation

8.3.2 Source Code

8.3.3 Output

8.4 Shearing

8.4.1 What is Rotation

8.4.2 Source Code

8.4.3 Output

8.5 Scaling

8.5.1 What is Rotation

8.5.2 Source Code

8.5.3 Output

8.6 Reflction

8.6.1 What is Rotation

8.6.2 Source Code

8.6.3 Output

8.7 References For Future Reading

8.1 IMPLEMENTATION OF TWO DIMENSIONAL TRANSFORMATIONS

Transformation means changes in orientation, size and shape of the object they are used to position the object. To change the position of the object how the object is viewed

There are different types of Transformations namely:

1. Translation

2. Scaling
3. Rotation
4. Shearing
5. Reflection

We will discuss each of the above topics in detail

8.2 TRANSLATION

8.2.1 What is Translation?:

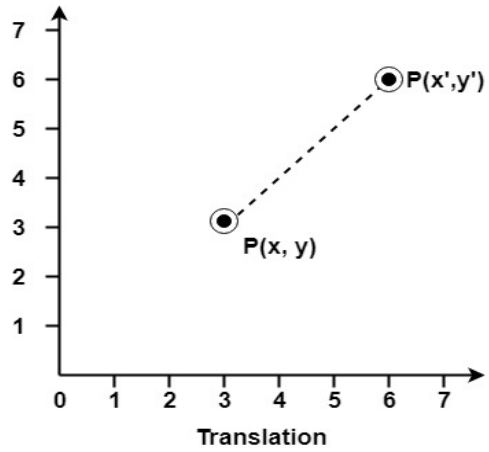
- It is repositioning an object along the straight-line path from one coordinate location to another.
- The translation is a rigid body transformation that moves objects without deformation.

How do Translation works?:

- To translate a point from coordinate position (x, y) to another (x',y'), we add algebraically the translation distances T_x and T_y to the original coordinate.

$$x' = x + T_x$$
$$y' = y + T_y$$

- The translation pair (T_x, T_y) is called a shift-vector.



Matrix representation will be:

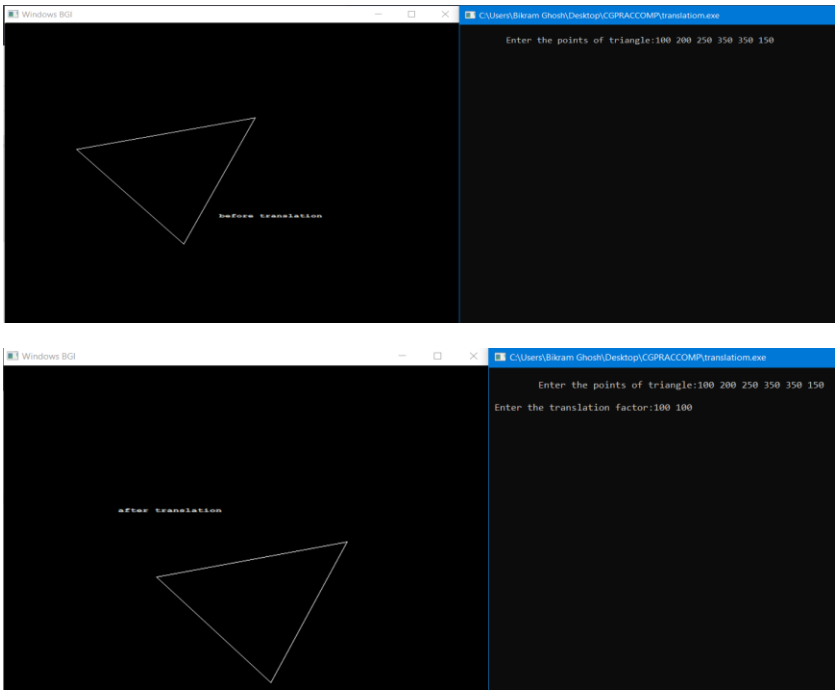
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

8.2.2 Source code:

```
#include<graphics.h>
#include<stdlib.h>
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
int main(){
int gd=DETECT,gm;
int x1,x2,x3,y1,y2,y3,nx1,nx2,nx3,ny1,ny2,ny3,c;
int sx,sy,xt,yt,r;
float t;
initgraph(&gd,&gm,"");
cout<<" \n \t Enter the points of triangle:";
setcolor(15);
cin>>x1>>y1>>x2>>y2>>x3>>y3;
line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x1,y1);
outtextxy(300,300,"before translation");
getch();
cleardevice();
outtextxy(150,200,"after translation");
cout<<" \n Enter the translation factor:";
cin>>xt>>yt;
nx1=x1+xt;
ny1=y1+yt;
nx2=x2+xt;
ny2=y2+yt;
nx3=x3+xt;
ny3=y3+yt;
line(nx1,ny1,nx2,ny2);
```

```
line(nx2,ny2,nx3,ny3);
line(nx3,ny3,nx1,ny1);
getch();
closegraph();
}
```

8.2.3.Output:

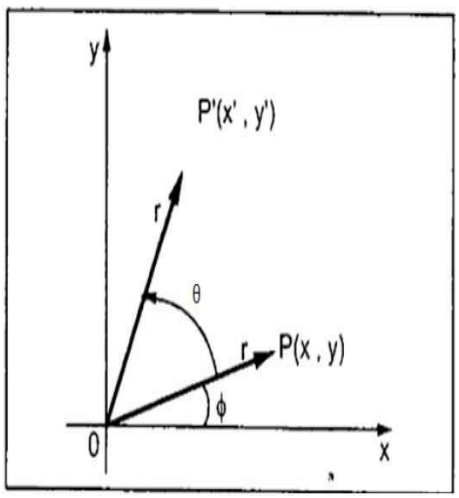


8.3 ROTATION

8.3.1 What is rotation?:

- Here we rotate an object with a particular angle θ from origin.

How does it works?:



From the following figure, we can see that the point P (x,y) is located at angle φ from the horizontal X coordinate with distance r from the origin.

Let us suppose you want to rotate point P with angle θ. After rotating it to a new location, we will get a new point P'(x',y')

Coordinates of point P can be represented as

$$x=r\cos\phi \ldots.1$$

$$y=r\sin\phi \ldots\ldots 2$$

Similarly coordinates of point p' can be represented as

$$x'=r\cos(\phi+\theta)=r\cos\phi\cos\theta-r\sin\phi\sin\theta \ldots\ldots.(3)$$

$$y'=r\sin(\phi+\theta)=r\cos\phi\sin\theta+r\sin\phi\cos\theta \ldots\ldots.(5)$$

Substituting equation 1 and 2 in 3 and 5, we will get

$$x'=x\cos\theta-y\sin\theta$$

$$y'=x\sin\theta+y\cos\theta$$

Representing the above equation in matrix form,

$$[X'Y'] = [XY]\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix}$$

P' = P . R

Where R is the rotation matrix

$$R = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix}$$

The rotation angle can be positive and negative.

For positive rotation angle, we can use the above rotation matrix. However, for negative angle rotation, the matrix will change as shown below

$$R = \begin{bmatrix} cos(-\theta) & sin(-\theta) \\ -sin(-\theta) & cos(-\theta) \end{bmatrix}$$

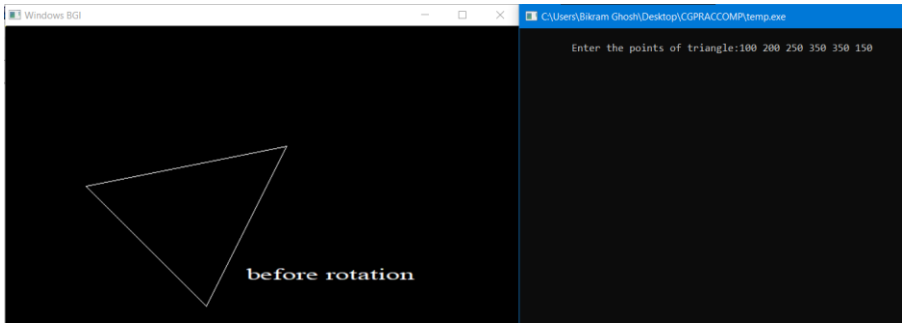
$$= \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} (\because cos(-\theta) = cos\theta \; and \; sin(-\theta) = -sin\theta)$$

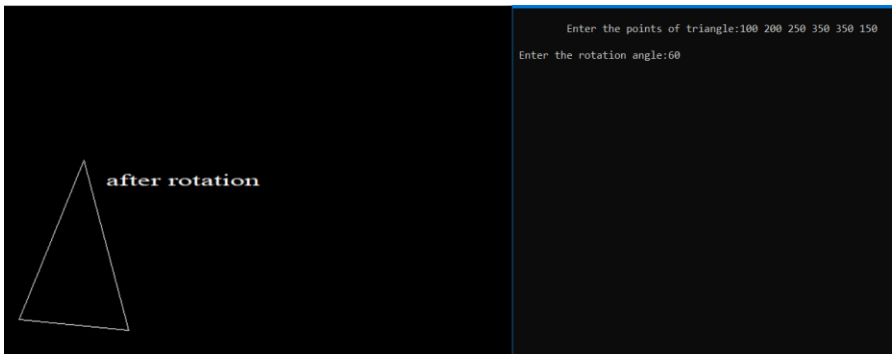
8.3.2 Source code:

```
//scaling
#include<graphics.h>
#include<stdlib.h>
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
int main(){
int gd=DETECT,gm;
int x1,x2,x3,y1,y2,y3,nx1,nx2,nx3,ny1,ny2,ny3,c;
int sx,sy,xt,yt,r;
float t;
initgraph(&gd,&gm,"");
settextstyle(1,0,2);
cout<<" \n \t Enter the points of triangle:";
setcolor(15);
cin>>x1>>y1>>x2>>y2>>x3>>y3;
line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x1,y1);
outtextxy(300,300,"before rotation");
getch();
cleardevice();
outtextxy(150,200,"after rotation");
cout<<" \n Enter the rotation angle:";
cin>>r;
t=3.15*r/170;
nx1=abs(x1*cos(t)-y1*sin(t));
ny1=abs(x1*sin(t)+y1*cos(t));
nx2=abs(x2*cos(t)-y2*sin(t));
ny2=abs(x2*sin(t)+y2*cos(t));
nx3=abs(x3*cos(t)-y3*sin(t));
ny3=abs(x3*sin(t)+y3*cos(t));
line(nx1,ny1,nx2,ny2);
```

```
line(nx2,ny2,nx3,ny3);
line(nx3,ny3,nx1,ny1);
getch();
}
```

8.3.3 Output:





8.4 SHEARING

8.4.1 What is Shearing?:

- Shearing means changing the shape and size of a 2D object along the x and y-axis
- It is similar to sliding the layers in one direction to change the size of an object
- There are two types of shearing. Shearing along the x-axis and along the y-axis
- Shearing can be done on both axes.
- Shearing is also termed Skewing.

Types of Shearing:

i. X-Axis Shearing:

The X-Shear preserves the Y coordinate and changes are made to X coordinates, which causes the vertical lines to tilt right or left

Shearing in X-axis is achieved by using the following shearing equations

$$X_{new} = X_{old} + S_{hx} * Y_{old}$$

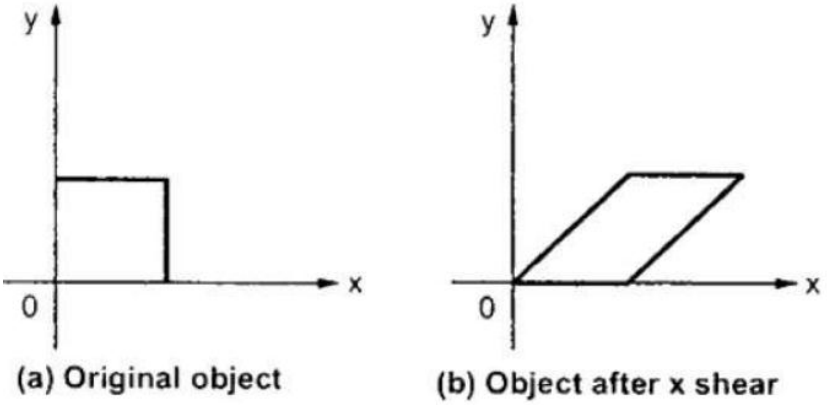
$$Y_{new} = Y_{old}$$

In Matrix form, the above shearing equations may be represented as:

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1 & Sh_x \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

Shearing Matrix

(In X axis)



ii. Y-Axis Shearing:

The Y-Shear preserves the X coordinates and changes the Y coordinates which causes the horizontal lines to transform into lines that slopes up or down

Shearing in Y-axis is achieved by using the following shearing equations:

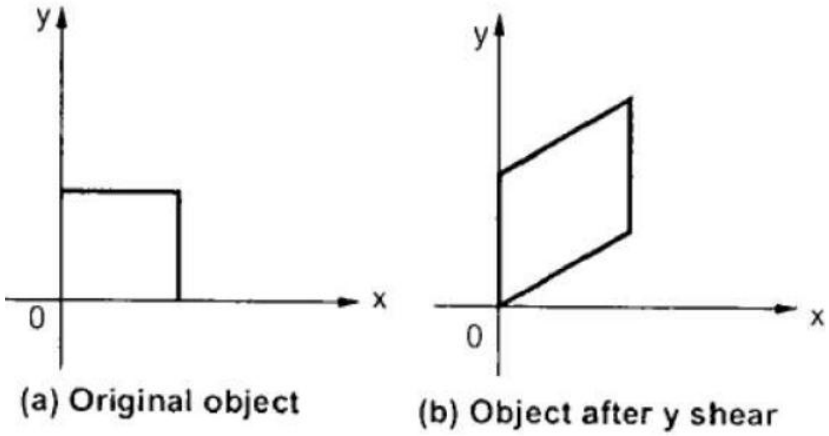
$$X_{new} = X_{old}$$
$$Y_{new} = Y_{old} + S_{hy} * X_{old}$$

In Matrix form, the above shearing equations may be represented as:

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ Sh_y & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

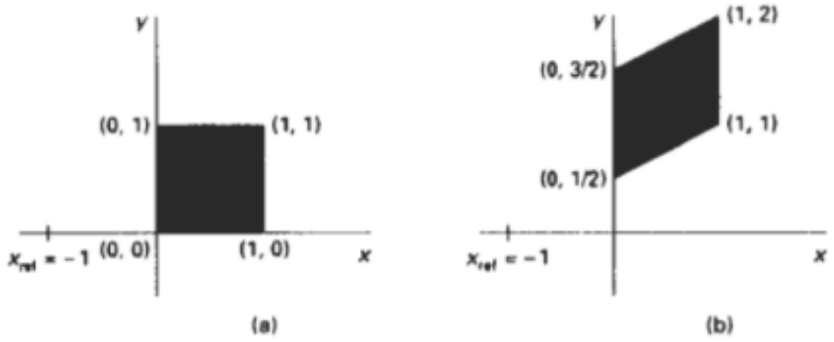
Shearing Matrix

(In Y axis)



iii. Shearing on both axes:

Shearing on both Axes changes both X and Y coordinates of the new point where the object is to be transformed



How do Shearing works?:

Shearing works by using two factors in terms of x and y-axis If we want an object to be sheared in the x-axis without distorting the y -xis we simply add the Shearing factor (Sh_x) of x by which object is to be sheared. Vice-versa for the object to be sheared in Y-axis the Shearing factor is denoted by (Sh_y).

Shearing by any axis works by keeping the one axis constant for example X axis shearing the y axis is kept constant and the shearing factor is multiplied by old coordinates of x

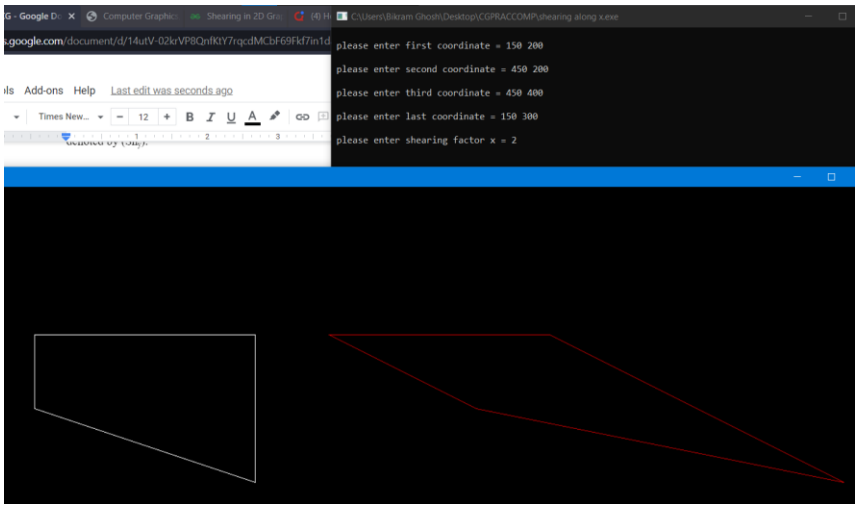
8.4.2 Source Code:

```
#include<stdio.h>

#include<graphics.h>

#include<conio.h>

main()

{

int gd=DETECT,gm;
```

```
int x,y,x1,y1,x2,y2,x3,y3,shear_f;
initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
cout<<("\n please enter first coordinate = ");
scanf("%d %d",&x,&y);
cout<<("\n please enter second coordinate = ");
scanf("%d %d",&x1,&y1);
cout<<("\n please enter third coordinate = ");
scanf("%d %d",&x2,&y2);
cout<<("\n please enter last coordinate = ");
scanf("%d %d",&x3,&y3);
cout<<("\n please enter shearing factor x = ");
scanf("%d",&shear_f);
cleardevice();
line(x,y,x1,y1);
line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x,y);

setcolor(RED);
x=x+ y*shear_f;
x1=x1+ y1*shear_f;
x2=x2+ y2*shear_f;
x3=x3+ y3*shear_f;

line(x,y,x1,y1);
line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x,y);
getch();
closegraph();
}
```

8.4.3 Output:

Shearing Along Y-Axis



8.5 SCALING

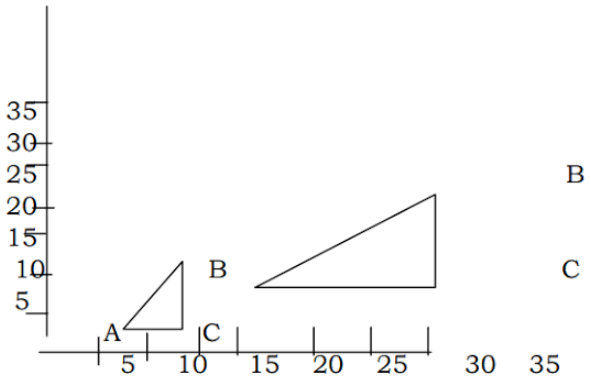
8.5.1 What is Scaling?:

- Scaling is a transformation technique used for changing the size of a 2D object along the x and y-axis
- In the Scaling process, the size of the 2-D object is either increased or decreased

How does Scaling work?:

- Scaling is done by multiplying the coordinates of each vertex of a polygon with Scaling factor
- The scaling factor is denoted by "S"
- if the Scaling factor S is less than 1 we reduce the size of the object and if it is greater than 1 we increase the size of the object

Scaling process:



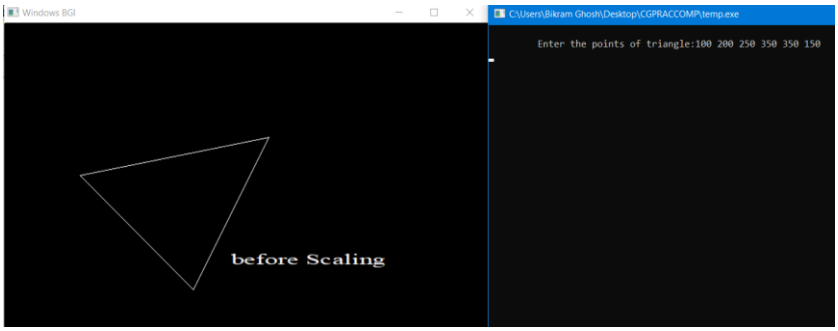
- Now lets consider a triangle having vertices A,B,C
- vertex A has coordinates x,y
- x’ = x * sx and y’ = y * sy.
- The scaling factor sx, sy scales the object in X and Y direction respectively. So, the above equation can be represented in matrix form:
- $$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$
- or,
- A'(x'y')=A(x,y).S(Sx,Sy)
  B(x'y')'=B(x,y).S(Sx,Sy)
  C'(x'y')=C(x,y).S(Sx,Sy)

8.5.2 Source Code:

```
//scaling
#include<graphics.h>
#include<stdlib.h>
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
int main(){
int gd=DETECT,gm;
int x1,x2,x3,y1,y2,y3,nx1,nx2,nx3,ny1,ny2,ny3,c;
int sx,sy,xt,yt,r;
float t;
initgraph(&gd,&gm,"");
settextstyle(1,0,2);
cout<<" \n \t Enter the points of triangle:";
setcolor(15);
cin>>x1>>y1>>x2>>y2>>x3>>y3;
line(x1,y1,x2,y2);
line(x2,y2,x3,y3);
line(x3,y3,x1,y1);
outtextxy(300,300,"before Scaling");
```

```
getch();
cleardevice();
outtextxy(150,200,"after scaling");
cout<<" \n Enter the scaling factor:";
cin>>sx>>sy;
nx1=x1*sx;
ny1=y2*sy;
nx2=x2*sx;
ny2=y2*sy;
nx3=x3*sx;
ny3=y3*sy;
line(nx1,ny1,nx2,ny2);
line(nx2,ny2,nx3,ny3);
line(nx3,ny3,nx1,ny1);
getch();
}
```

8.5.3 Outputs:



8.6 2D REFLECTION

8.6.1 What is 2D reflection?:

- 2D Reflection is a kind of rotation, where the angle of rotation is 170 degree.
- The reflected image is always formed on the other side of the mirror.
- The size of the reflected image formed is the same as the original object.

Types of 2D reflection:

- Reflection on X-axis
- Reflection on Y-axis
- Reflection on the axis perpendicular to the XY plane and passing through the origin.
- Reflection on Y = X.

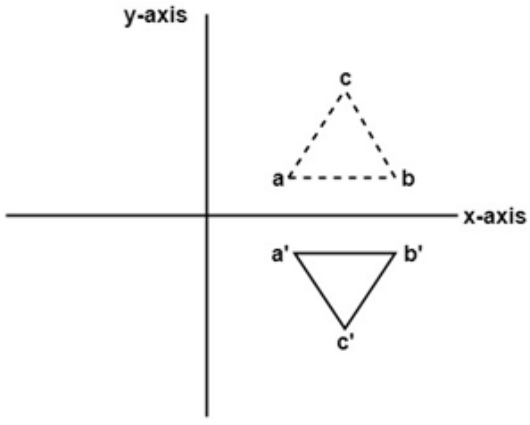
Consider a point object O that has to be reflected in a 2D plane. Let Initial coordinates of the object O is (X1, Y1) and New coordinates of the reflected object O after reflection is (X2, Y2).

Reflection on X-axis:

In this transformation the value of x will remain the same whereas the value of y will become negative. The object will lie on another side of the x-axis.

$$X1 = X2$$

$$Y1 = -Y2$$



In matrix form, the above reflection matrix may be represented as a 3 x 3 matrix as-

Reflection on Y-axis:

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ 1 \end{bmatrix}$$

Reflection Matrix

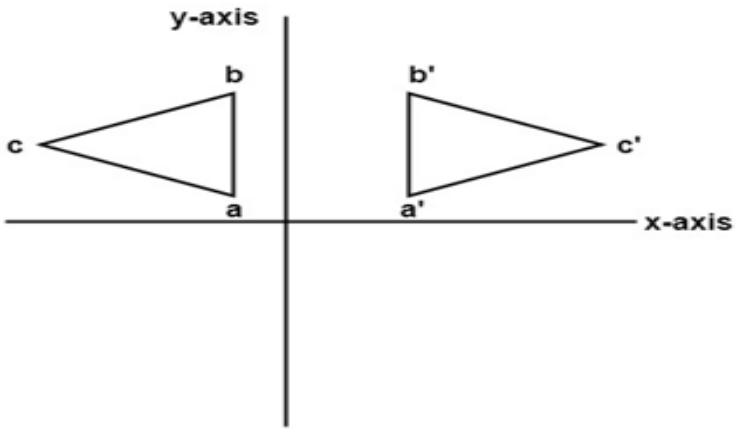
(Reflection Along X Axis)

(Homogeneous Coordinates Representation)

In this transformation, the value of y will remain the same whereas the value of x will become negative. The object will lie on another side of the y-axis.

$X1 = -X2$

$Y1 = Y2$



In matrix form, the above reflection matrix may be represented as a 3 x 3 matrix as:

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ 1 \end{bmatrix}$$

Reflection Matrix

(Reflection Along Y Axis)

(Homogeneous Coordinates Representation)

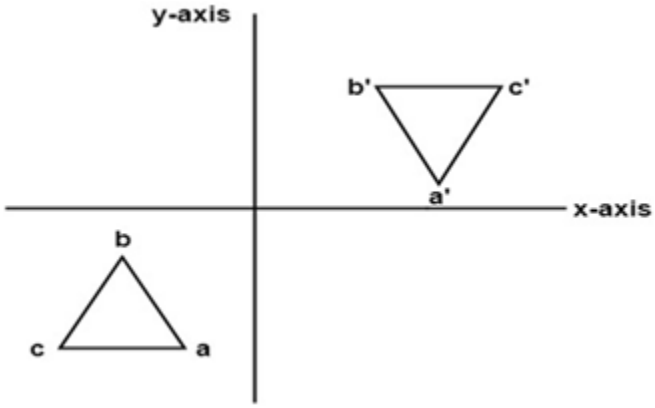
Reflection on the axis perpendicular to the XY plane and passing through the origin.

In this transformation,, the value of y and the value of x will become negative. The object will lie on the opposite side of the axis.

$$X1 = -X2$$

$$Y1 = -Y2$$

In this value of x and y, both will be reversed. This is also called the half revolution about the origin.

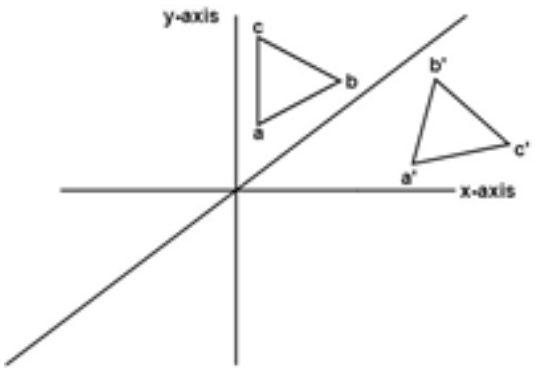


In the matrix form of this transformation will be,

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

- Reflection on Y = X.

First of all, the object is rotated at 55°. The direction of rotation is clockwise. After it reflection is done concerning the x-axis. The last step is the rotation of y=x back to its original position that is counterclockwise at 55°



The object may be reflected about line y = x with the help of following transformation matrix

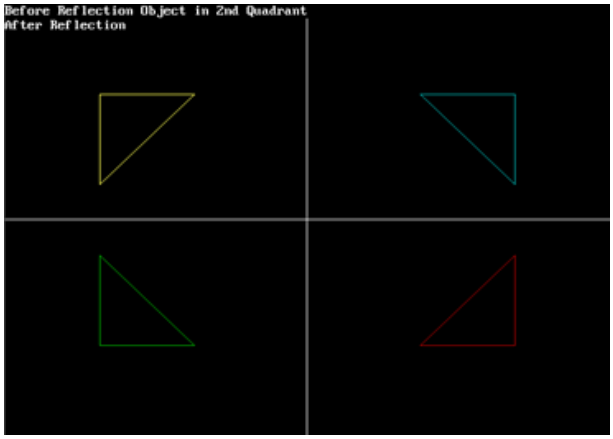
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

8.6.2 Source code:

```
#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <iostream.h>
// Driver Code
void main()
{
        // Initialize the drivers
        int gm, gd = DETECT, ax, x1 = 100;
        int x2 = 100, x3 = 200, y1 = 100;
        int y2 = 200, y3 = 100;
        initgraph(&gd, &gm, "C:\\turboc3\\bgi");
        cleardevice();
        // Draw the graph
        line(getmaxx() / 2, 0, getmaxx() / 2,getmaxy());
        line(0, getmaxy() / 2, getmaxx(),getmaxy() / 2);
        cout<<"Before Reflection Object in 2nd Quadrant";
        setcolor(15);
        line(x1, y1, x2, y2);
        line(x2, y2, x3, y3);
        line(x3, y3, x1, y1);
        getch();
        // After reflection
        cout<<"\nAfter Reflection";
setcolor(5);
        line(getmaxx() - x1, getmaxy() - y1,getmaxx() - x2, getmaxy() - y2);
        line(getmaxx() - x2, getmaxy() - y2,getmaxx() - x3, getmaxy() - y3);
        line(getmaxx() - x3, getmaxy() - y3,getmaxx() - x1, getmaxy() - y1);
```

```
        setcolor(3);
        line(getmaxx() - x1, y1,getmaxx() - x2, y2);
        line(getmaxx() - x2, y2,getmaxx() - x3, y3);
        line(getmaxx() - x3, y3,getmaxx() - x1, y1);
        setcolor(2);
        line(x1, getmaxy() - y1, x2,getmaxy() - y2);
        line(x2, getmaxy() - y2, x3,getmaxy() - y3);
line(x3, getmaxy() - y3, x1,getmaxy() - y1);
        getch();

        // Close the graphics
        closegraph();
}
```

8.6.3 Output:



8.7 REFERENCES FOR FUTURE READING

1. https://www.tutorialspoint.com/computer_graphics/2d_transformation.htm
2. https://programmerbay.com/c-program-to-perform-shearing-on-a-rectangle/
3. https://www.geeksforgeeks.org/translation-objects-computer-graphics-reference-added-please-review/

*****

MODULE VIII

9

2D GEOMETRIC TRANSFORMATIONS & CLIPPING

Unit Structure

9.1 Midpoint Subdivision Algorithm
- 9.1.1 What is Midpoint Subdivision Algorithm
- 9.1.2 Algorithm
- 9.1.3 Source Code
- 9.1.4 Output

9.2 Cohen Sutherland Line Clipping Algorithm
- 9.2.1 What is Cohen Sutherland Line Clipping Algorithm?
- 9.2.2 Algorithm
- 9.2.3 Source Code
- 9.2.4 Output

9.3 Sutherland-Hodgman Polygon Clipping Algorithm
- 9.3.1 What is Sutherland-Hidgman polygon clipping Algorithm
- 9.3.2 Algorithm
- 9.3.3 Source Code
- 9.3.4 Output

9.4 Conclusion

9.5 References for Future Reading

9.1 MIDPOINT SUBDIVISION ALGORITHM

9.1.1 What is Midpoint Subdivision Algorithm:

This algorithm is made by basic algorithm i.e. subdivision algorithm. Subdivision algorithm is first line clipping algorithm. In this we subdivide the line randomly.

Midpoint subdivision algorithm is an extension of the Cyrus Beck algorithm. This algorithm is mainly used to compute visible areas of lines that are present in the view port are of the sector or the image. It follows the principle of the bisection method and works similarly to the Cyrus Beck algorithm by bisecting the line in to equal halves. But unlike the Cyrus Beck algorithm, which only bisects the line once, Midpoint Subdivision Algorithm bisects the line numerous times..

9.1.2 Algorithm:

Step1: Calculate the position of both endpoints of the line

Step2: Perform OR operation on both of these endpoints

Step3: If the OR operation gives 0000

then Line is guaranteed to be visible

else

Perform AND operation on both endpoints.

If AND ≠ 0000

then the line is invisible

else

AND=0000

then the line is clipped

case.

Step4: For the line to be clipped. Find midpoint

$X_m=(x_1+x_2)/2$

$Y_m=(y_1+y_2)/2$

X_m is midpoint of X coordinate.

Y_m is midpoint of Y coordinate.

Step5: Check each midpoint, whether it nearest to the boundary of a window or not.

Step6: If the line is totally visible or totally rejected not found then repeat step 1 to 5.

Step7: Stop algorithm.

9.1.3 Source Code:

```
//MIDPOINT SUBDIVISION LINE CLIPPING

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<dos.h>

#include<math.h>

#include<graphics.h>
```

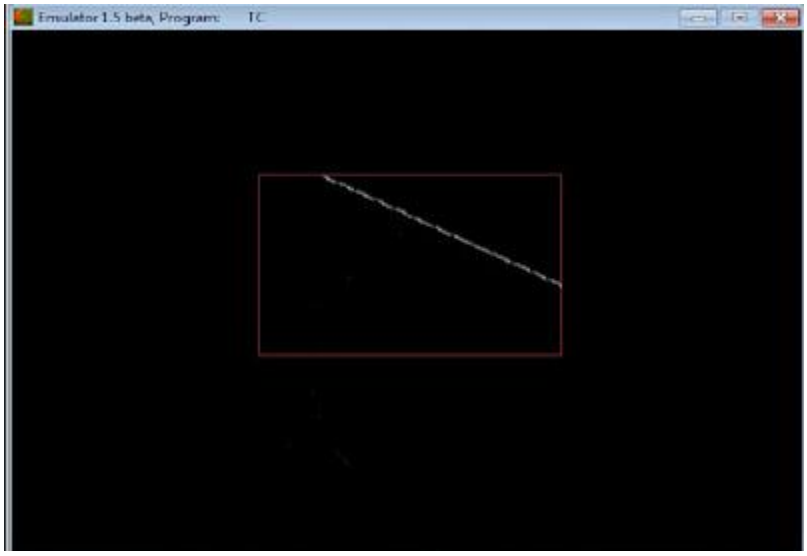
```
typedef struct coordinate
{
int x,y;
char code[4];
}PT;
void drawwindow();
void drawline (PT p1,PT p2,int cl);
PT setcode(PT p);

int visibility (PT p1,PT p2);
PT resetendpt (PT p1,PT p2);
main()
{
        int gd=DETECT, gm,v;
        PT p1,p2,ptemp;
        initgraph(&gd,&gm,"C:\\TC\\BGI ");
        cleardevice();
        printf("\n\n\t\tENTER END-POINT 1 (x,y): ");
scanf("%d,%d",&p1.x,&p1.y);
        printf("\n\n\t\tENTER END-POINT 2 (x,y): ");
        scanf("%d,%d",&p2.x,&p2.y);
        cleardevice();
        drawwindow();
        getch();
        drawline(p1,p2,15);
        getch();
        cleardevice();
        drawwindow();
        midsub(p1,p2);
        getch();
        closegraph();
        return(0);
}
midsub(PT p1,PT p2)
{
```

```
        PT mid;
        int v;
        p1=setcode(p1);
        p2=setcode(p2);
        v=visibility(p1,p2);
        switch(v)
        {
                case 0:  /* Line conpletely visible */
                        drawline(p1,p2,15);
                        break;
                case 1:  /* Line completely invisible */
                        break;
                case 2:  /* line partly visible */
                        mid.x = p1.x + (p2.x-p1.x)/2;
                        mid.y = p1.y + (p2.y-p1.y)/2;
                        midsub(p1,mid);
                        mid.x = mid.x+1;
                        mid.y = mid.y+1;
                        midsub(mid,p2);
                        break;
        }
}
void drawwindow()
{
setcolor(RED);
line(150,100,450,100);
line(450,100,450,400);
line(450,400,150,400);
line(150,400,150,100);
}

void drawline (PT p1,PT p2,int cl)
{
setcolor(cl);
line(p1.x,p1.y,p2.x,p2.y);
```

```
}
PT setcode(PT p)
{
PT ptemp;
if(p.y<=100)
ptemp.code[0]='1'; /* TOP */
else
ptemp.code[0]='0';
if(p.y>=400)
ptemp.code[1]='1'; /* BOTTOM */
else
ptemp.code[1]='0';
if (p.x>=450)
ptemp.code[2]='1'; /* RIGHT */
else
ptemp.code[2]='0';
if (p.x<=150)      /* LEFT */
ptemp.code[3]='1';
else
ptemp.code[3]='0';
ptemp.x=p.x;
ptemp.y=p.y;
return(ptemp);
}
int visibility (PT p1,PT p2)
{
int i,flag=0;
for(i=0;i<4;i++)
{
if((p1.code[i]!='0')||(p2.code[i]!='0'))
flag=1;
}
if(flag==0)
return(0);
for(i=0;i<4;i++)
```

```
{
if((p1.code[i]==p2.code[i]) &&(p1.code[i]=='1'))
flag=0;
}
if(flag==0)
return(1);
return(2);
}
```

9.1.4 OUTPUT:



9.2 COHEN SUTHERLAND LINE CLIPPING ALGORITHM

9.2.1 What is Cohen Sutherland Line Clipping algorithm?:

Cohen Sutherland algorithm was developed by Danny Cohen and Ivan Sutherland in the year 1967 during flightSimulator work

The time and space complexity of Cohen Sutherland line clipping algorithm are:

- worst case time complexity: 0(n)
- Average case time complexity:0(n)
- Best case time complexity:0(n)
- space complexity:0(1) Here n=number of lines.

The Cohen–Sutherland algorithm is a computer-graphics algorithm used for line clipping. The algorithm divides a two-dimensional space into 9

regions and then efficiently determines the lines and portions of lines that are visible in the central region of interest (the viewport).

In the algorithm, first of all, it is detected whether line lies inside the screen or it is outside the screen.

Sutherland Line clipping have 3 categories:

All lines come under any one of the following categories:

1. Visible
2. Not Visible
3. Clipping Case

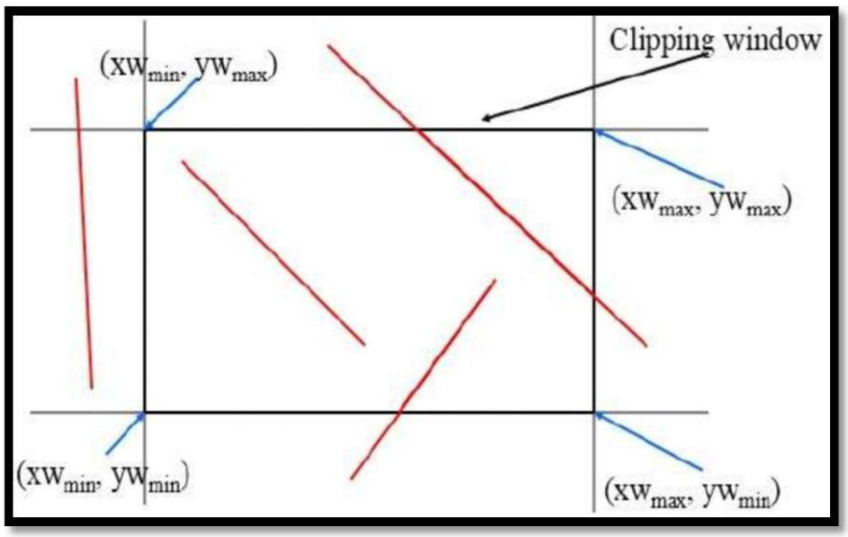
1. Visible: If a line lies within the window, i.e., both endpoints of the line lies within the window. A line is visible and will be displayed as it is.

2. Not Visible: If a line lies outside the window it will be invisible and rejected. Such lines will not display. If any one of the following inequalities is satisfied, then the line is considered invisible. Let A (x1,y2) and B (x2,y2) are endpoints of line.

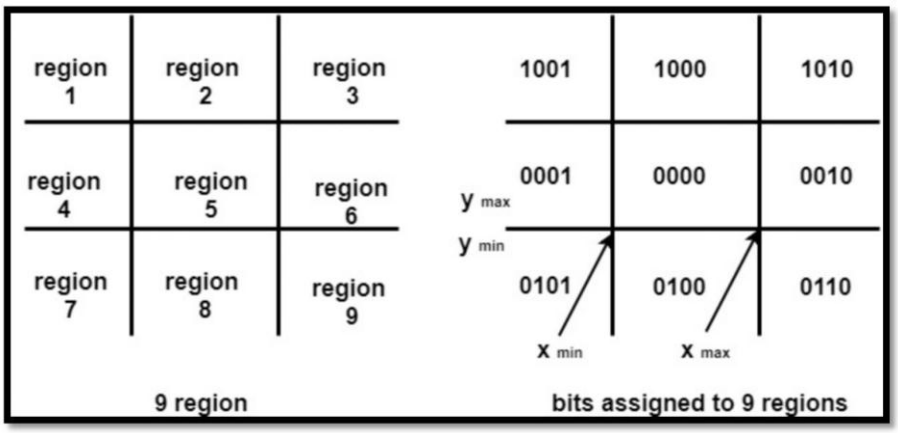
3. Clipping Case: If the line is neither visible case nor invisible case. It is considered to be clipped case. First of all, the category of a line is found based on nine regions given below. All nine regions are assigned codes. Each code is of 4 bits. If both endpoints of the line have end bits zero, then the line is considered to be visible.

The center area is having the code, 0000, i.e., region 5 is considered a rectangle window.

I. This algorithm uses the clipping window as shown in the following figure. The minimum coordinate for the clipping region is(XWmin,YWmin)(XWmin,YWmin) and the maximum coordinate for the clipping region is (XWmax,YWmax)(XWmax,YWmax).



II. We will use 4-bits to divide the entire region. These 4 bits represent the Top, Bottom, Right, and Left of the region as shown in the following figure. Here, the TOP and LEFT bit is set to 1 because it is the TOP-LEFT corner.



There are 3 possibilities for the line:

- Line can be completely inside the window (This line should be accepted).
- Line can be completely outside of the window (This line will be completely removed from the region).
- Line can be partially inside the window (We will find intersection point and draw only that portion of line that is inside region).

9.2.2 Algorithm:

1. Read two end points of the line say P1(x1, y1) and P2 (x2, y2).
2. Read two corners (left-top and right-bottom)of the window, say (Wx1, Wy1 and Wx2,Wy2).
3. Assign the region codes for two endpoints P1 and P2 using following steps:

Initialize code with bits 0000

Set Bit 1 - if (x < Wx1)

Set Bit 2 - if (x> Wx2)

Set Bit 3 - if (y < Wy2)

Set Bit 4 - if (y> Wy1)

4. Check for visibility of line P1 P2.
   a) If region codes for both endpoints P1 and P2 are zero then the line is completely visible.Hence draw the line and go to step 9.

b) If region codes for endpoints are not zero and the logical AND of them is also nonzero then the line is completely invisible, so reject the line and go to step 9.

c) If region codes for two endpoints do not satisfy the conditions in 4a) and 4b) the line is partially visible.

5. Determine the intersecting edge of the clipping window by inspecting the region codes of two endpoints.

   a) If region codes for both the end points are non-zero, find intersection points P1'and P2' with boundary edges of clipping window with respect to point P1 and point P2, respectively

   b) If region code for anyone end point is non- zero then find intersection point P1' or P2' with the boundary edge of the clipping window with respect to it.

6. Divide the line segments considering intersection points.

7. Reject the line segment if any one end point of it appears outsides the clipping window.

8. Draw the remaining line segments.

9. Stop.

9.2.3 Program:

```
#include<iostream>
#include<stdlib.h>
#include<math.h>
#include<graphics.h>
#include<dos.h>
using namespace std;
 typedef struct coordinate
 {
int x,y; char code[4];
 }PT;
 void drawwindow();
void drawline(PT p1,PT p2);
 PT setcode(PT p);
int visibility(PT p1,PT p2);
 PT resetendpt(PT p1,PT p2);
int main()
{
```

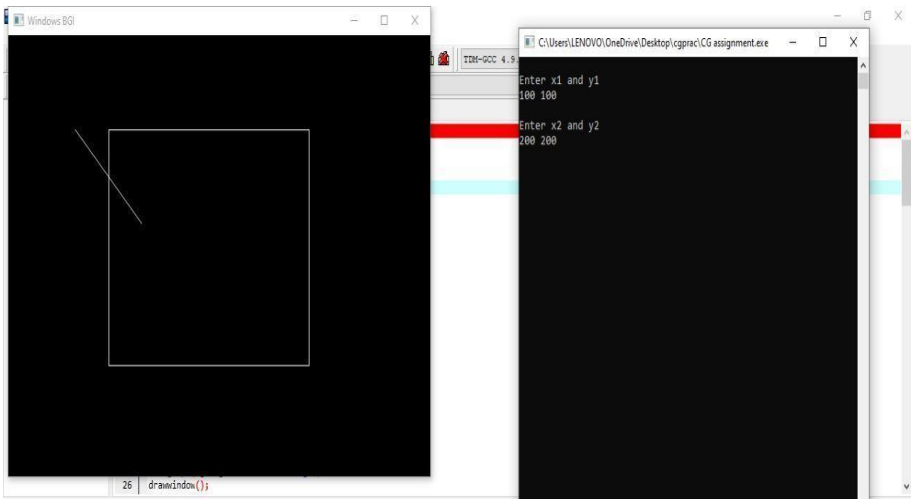
```
int gd=DETECT,v,gm;
initgraph(&gd,&gm,(char*)"");
settextstyle(1,0,2);
PT p1,p2,p3,p4,ptemp;
 cout<<"\nEnter x1 and y1\n";
cin>>p1.x>>p1.y;
 cout<<"\nEnter x2 and y2\n";
 cin>>p2.x>>p2.y;
 drawwindow();
 delay(1500);
 drawline(p1,p2);
 delay(1500);
cleardevice();
 delay(1500);
 p1=setcode(p1);
 p2=setcode(p2);
v=visibility(p1,p2);
 delay(1500);
 switch(v)
{
case 0:
 drawwindow();
delay(1500);
drawline(p1,p2);
 break;
case 1:
drawwindow();
 delay(1500);
 break;
 case 2:
p3=resetendpt(p1,p2);
 p4=resetendpt(p2,p1);
drawwindow();
delay(1500);
drawline(p3,p4);
```

```
 break;
}
settextstyle(1,0,2);
delay(5000);
closegraph();
}
void drawwindow()
{
 line(150,100,450,100);
line(450,100,450,350);
 line(450,350,150,350);
line(150,350,150,100);
}
 void drawline(PT p1,PT p2)
{
line(p1.x,p1.y,p2.x,p2.y);
}
PT setcode(PT p) //for setting the 4 bit code
{
PT ptemp;
if(p.y<100)
ptemp.code[0]='1'; //Top
else
ptemp.code[0]='0';
if(p.y>350)
ptemp.code[1]='1'; //Bottom
else
ptemp.code[1]='0';
if(p.x>450)
ptemp.code[2]='1'; //Right
else
ptemp.code[2]='0';
if(p.x<150)
 ptemp.code[3]='1'; //Left
else ptemp.code[3]='0';
```

```
 ptemp.x=p.x;
ptemp.y=p.y;
return(ptemp);
}

int visibility(PT p1,PT p2)
{
int i,flag=0;
for(i=0;i<4;i++)
{
if((p1.code[i]!='0') || (p2.code[i]!='0'))
flag=1;
 }
if(flag==0)
 return(0);
for(i=0;i<4;i++)
{
if((p1.code[i]==p2.code[i]) && (p1.code[i]=='1'))
flag='0';
}
if(flag==0)
return(1);
return(2);
}
PT resetendpt(PT p1,PT p2)
{
PT temp;
int x,y,i;
float m,k; if(p1.code[3]=='1')
x=150;
if(p1.code[2]=='1')
x=450;
if((p1.code[3]=='1') || (p1.code[2]=='1'))
{
m=(float)(p2.y-p1.y)/(p2.x-p1.x); k=(p1.y+(m*(x-p1.x)));
```

```
temp.y=k;
temp.x=x;
for(i=0;i<4;i++)
 temp.code[i]=p1.code[i];
if(temp.y<=350 && temp.y>=100)
return (temp);
}
if(p1.code[0]=='1')
y=100;
if(p1.code[1]=='1')
y=350;
if((p1.code[0]=='1') || (p1.code[1]=='1'))
{
m=(float)(p2.y-p1.y)/(p2.x-p1.x);
k=(float)p1.x+(float)(y-p1.y)/m;
temp.x=k;
temp.y=y;
for(i=0;i<4;i++)
temp.code[i]=p1.code[i];
 return(temp);
}
else
return(p1);
}
```

9.2.4 OUTPUT:



The primary use of clipping in computer graphics is to remove objects, lines, or line segments that are outside the viewing pane. The viewing transformation is insensitive to the position of points relative to the viewing volume − especially those points behind the viewer − and it is necessary to remove these points before generating the view. The algorithm includes, excludes or partially includes the line based on whether:

Both endpoints are in the viewport region (Bitwise OR of endpoints = 0000): trivial accept.

Both endpoints share at least one non-visible region, which implies that the line does not cross the visible region. (Bitwise AND of endpoints ≠ 0000): trivial reject.

Both endpoints are in different regions: in case of this nontrivial situation the algorithm finds one of the two points that is outside the viewport region (there will be at least one point outside). The intersection of the outpoint and extended viewport border is then calculated (i.e. with the parametric equation for the line), and this new point replaces the outpoint. The algorithm repeats until a trivial accept or reject occurs.

This algorithm uses a four digit (bit) code to indicate which of nine regions contain the end point of line. The four bit codes are called region codes or outcodes. These codes identify the Location of the point relative to the boundaries of the clipping rectangle

Each bit position in the region code is used to indicate one of the four relative co-ordinate positions of the point with respect to the clipping window: to the left, right, top or bottom. The rightmost bit is the first bit and the bits are set to 1 based on the following scheme:

Set Bit 1: if the end point is to the left of the window.

Set Bit 2: if the end point is to the right of the window.

Set Bit 3: if the end point is below the window Set Bit 4 - if the end Point is above the window

Otherwise, the bit is set to zero

Advantages & Disadvantages:

Advantages:

1. Implementation manageable
2. This can happen if the clipping rectangle is very large or very small
3. Well suited to hardware
4. It can clip pictures much large than screen size

5. It calculates end-points very quickly and rejects and accepts lines quickly.

Disadvantages:

1. Clipping window region can only be in rectangular shape. It does not allow any other polygonal shaped window.
2. This method requires a considerable amount of memory due to lot of operations. So wastage of memory for storing intermediate polygons.
3. Sutherland Hodgeman clipping algorithm can't produce connected areas.
4. X-axis and Y-axis has to be parallel to the edges of rectangular shaped window
5. If end points of line segment lies diagonally i.e one at R.H.S other at L.H.S., and on one the at top and other at the bottom then, even if the line doesn't pass through the clipping region it will have logical intersection of 0000 indirectly .

9.3 SUTHERLAND-HODGMAN POLYGON CLIPPING ALGORITHM

9.3.1 What Is Sutherland-Hodgman Polygon Clipping Algorithm:

- **Introduction:** Polygon clipping is one of those humble tasks computers do all the time .To clip a polygon , Sutherland-Hodgman clipping algorithm is applied. A polygon clipping algorithm receives a polygon and a clipping window as input. To clip a polygon, Sutherland - Hodgman clipping algorithm is applied on a polygon. There is a clipping window, those lines of a polygon which lies inside a window are accepted to be display on graphic screen and those which lies outside the clipping window are rejected to be display on graphic screen.

- **Limitations:**

1. Clipping window region can be rectangular in shape only and no other polygonal shaped window is allowed.
2. Edges of rectangular shaped clipping window has to be parallel to the x-axis and y axis.
3. If end points of line segment lies in the extreme limits i.e., one at R.H.S other at L.H.S., and on one the at top and other at the bottom (diagonally) then, even if the line doesn't pass through the clipping region it will have logical intersection of 0000 implying that line segment will be clipped but infect it is not so.

- **Uses:**

1. Sutherland Hodgeman polygon clipping algorithm is used for polygon clipping.
2. .In this algorithm, all the vertices of the polygon are clipped against each edge of the clipping window.
3. In this algorithm the polygon is clipped against the left edge of the polygon window to get new vertices of the polygon
4. These new vertices are used to clip the polygon against three edges i.e right edge, top edge, bottom edge, of the clipping window.

- **Advantages:**

1 It is very useful for clipping polygons. It clips a polygon against all edges of the clipping region.

2 It is easy to implement.

3 It works by extending each line of the convex clip polygon. It steps from vertex to vertex and adds 0, 1, or 2 vertices at each step to the output list.

4 It selects only those vertices which are on the visible side of the subject polygon.

- **Disadvantages:**

1. It clips to each window boundary one at a time.
2. It has a “Random” edge choice
3. It has Redundant edge-line cross calculations

9.3.2 Explanation with Examples:

Let's understand by an example:

Clip a line A (-1,5) and B (3,8) using the Cohen Sutherland algorithm with window coordinates (-3,1) and (2,6).

Here xmin-3 and ymin-1 with xmax -1 and ymax-6 with x1-1, y1-5 and x2-3, y2-8

S-1: Find the region code of the line point x1--1y1-5 as follows:

- Bit 1-Sign of (y-ymax) - Sign of (5-6)- Negative so bit 1 would be 0
- Bit 2-Sign of tymin-y)-Sign of (1-6)- Negative, so bit 2 would be 0 - Bit 3-Sign of (x-xmax) - Sign of (-1-1)-Negative, so bit 3 would be 0.
- Bit 4-Sign of (xxmin-x) - Sign of (3-(-1))- Negative, so bit 4 would be 0.

FIND THE REGION CODE OF THE LINE POINT X2=3.Y2-8 AS FOLLOWS:

- Bit 1-Sign of y-ymax -Sign of (8-6) - Positive, so bit 1 would be 1
- Bit 2-Sign of (ymin-y) Sign of (1-B)-Negative so bit 2 would be d
- Bit 3-Sign of (x-xmax) - Sign of (3-1) Positive, so Bit 3 would be 1
- Bit 4-Sign of (min-x-Sign of (33)-Negative so bit 4 would be al The (x1y) has the code (0000) and (x2y2) has the code (1010). Rember the coldedore written from left to right. Left is bit -1.

S-2: Which category this line belongs to: Find, does it require clipping?: D of both the region codes are (0000) * (1010) = (0*1,0*0,0°1,0'0) = (0000). Since the logical AND of codes is zero, so the line belongs to third category- Clipping category=>Computer would clip it

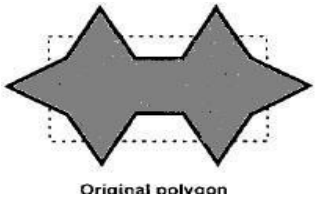
S-3: The Line "Needs Clipping" based on the calculation above.

S-4: Since Bit 1 is one therefore intersection point is y-ymax and x-x1+(y1ymax)/m=> y=6 and x=-1+(6-5)/(3/4) => -1+4/3 => 1/3. So the intersection point would be (x=1/3,y-6).

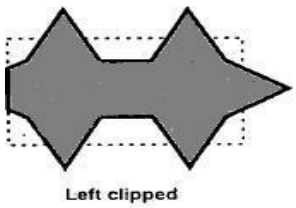
S-5: Find the region code of newfound point, C(x-1/3 and y-6) =>(0000)

Since the starting point, A(-1,9 has (0000) and new Point C is also (0000) that means both endpoints are visible. The line does not need more clipping. The algorithm would stop here.

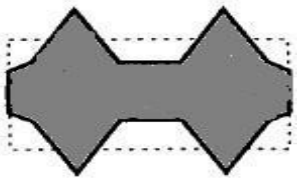
- **Steps:** There are four steps to be applied on polygon while cipping using SutherlandHodgman Algorithm,which are given as follows:



Step 1: Left Clip: Clip a line of a polygon which lies outside, on the left side of the clipping window.

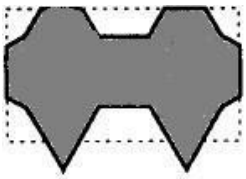


Step 2: Right Clip: Clip a line of a polygon which lies outside, on the right side of the clipping window.



Right clipped

Step 3: Top Clip: Clip a line of a polygon which lies outside, on the upper part of the clipping window.



Top clipped

Step 4: Bottom Clip: Clip a line of a polygon which lies outside, on the bottom part of the clipping window.

Bottom clipped

9.3.3 Program:

```
#include<iostream.h>
#include<conio.h>
#include<graphics.h>
#define round(a) ((int)(a+0.5))
int k;
float xmin,ymin,xmax,ymax,arr[20],m;
void clipl(float x1,float y1,float x2,float y2)
{
if(x2-x1)
        m=(y2-y1)/(x2-x1);
    else
        m=100000;
    if(x1 >= xmin && x2 >= xmin)
    {
        arr[k]=x2;
        arr[k+1]=y2;
```

```
            k+=2;
        }

if(x1 < xmin && x2 >= xmin)
        {
            arr[k]=xmin;
            arr[k+1]=y1+m*(xmin-x1);
            arr[k+2]=x2;
            arr[k+3]=y2;
            k+=4;
        }

if(x1 >= xmin  && x2 < xmin)
        {
            arr[k]=xmin;
            arr[k+1]=y1+m*(xmin-x1);
            k+=2;
        }
}
void clipt(float x1,float y1,float x2,float y2)
{
        if(y2-y1)
        {
            m=(x2-x1)/(y2-y1);
        }
else
        {
            m=100000;
        }
        if(y1 <= ymax && y2 <= ymax)
        {
            arr[k]=x2;
            arr[k+1]=y2;
            k+=2;
        }
```

```
        if(y1 > ymax && y2 <= ymax)
        {
                arr[k]=x1+m*(ymax-y1);
                arr[k+1]=ymax;
                arr[k+2]=x2;
                arr[k+3]=y2;
                k+=4;
        }
        if(y1 <= ymax  && y2 > ymax)
        {
                arr[k]=x1+m*(ymax-y1);
                arr[k+1]=ymax;
                k+=2;
        }
}
void clipr(float x1,float y1,float x2,float y2)
{
        if(x2-x1)
        {
m=(y2-y1)/(x2-x1);
        }

else
        {
                m=100000;
}

if(x1 <= xmax && x2 <= xmax)
        {
                arr[k]=x2;
                arr[k+1]=y2;
                k+=2;
        }
   if(x1 > xmax && x2 <= xmax)
        {
```

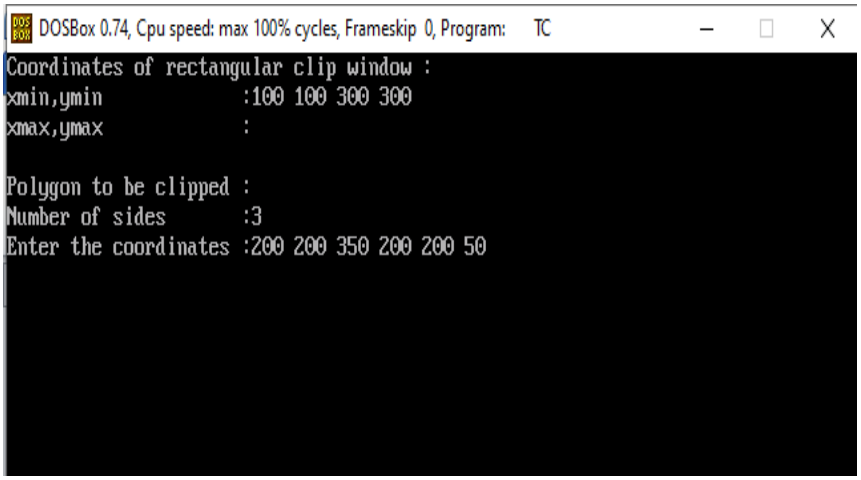
```
                arr[k]=xmax;
                arr[k+1]=y1+m*(xmax-x1);
                arr[k+2]=x2;
                arr[k+3]=y2;
                k+=4;
        }
    if(x1 <= xmax  && x2 > xmax)
        {
                arr[k]=xmax;
                arr[k+1]=y1+m*(xmax-x1);
                k+=2;
        }
}
void clipb(float x1,float y1,float x2,float y2)
{
        if(y2-y1)
        {
m=(x2-x1)/(y2-y1);
        }
else
        {
        m=100000;
}
        if(y1 >= ymin && y2 >= ymin)
        {
                arr[k]=x2;
                arr[k+1]=y2;
                k+=2;
        }
        if(y1 < ymin && y2 >= ymin)
        {
                arr[k]=x1+m*(ymin-y1);
                arr[k+1]=ymin;
                arr[k+2]=x2;
                arr[k+3]=y2;
```

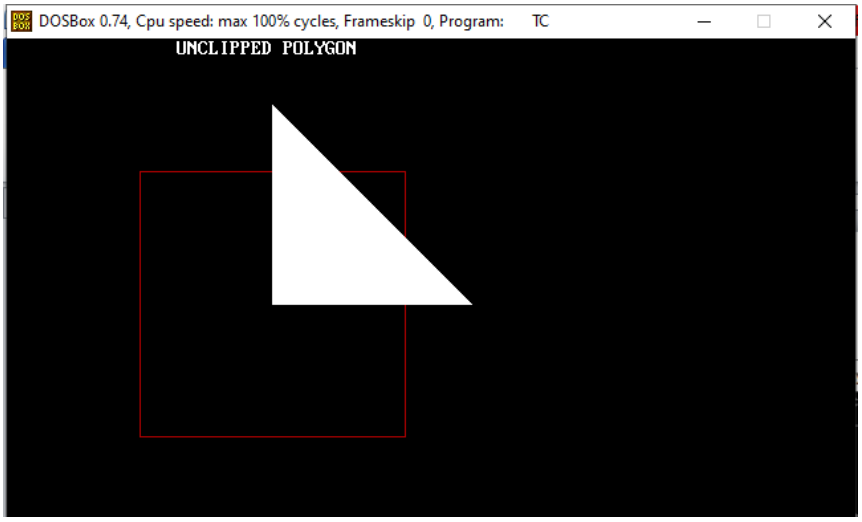
```
            k+=4;
        }
        if(y1 >= ymin  && y2 < ymin)
        {
            arr[k]=x1+m*(ymin-y1);
            arr[k+1]=ymin;
            k+=2;
        }
}
void main()
{
        int gdriver=DETECT,gmode,n,poly[20];
        float xi,yi,xf,yf,polyy[20];
        clrscr();
        cout<<"Coordinates of rectangular clip window :\nxmin,ymin
:";
        cin>>xmin>>ymin;
        cout<<"xmax,ymax            :";
        cin>>xmax>>ymax;
        cout<<"\n\nPolygon to be clipped :\nNumber of sides      :";
        cin>>n;
        cout<<"Enter the coordinates :";
        for(int i=0;i<2*n;i++)
cin>>polyy[i];
        polyy[i]=polyy[0];
        polyy[i+1]=polyy[1];
        for(i=0;i < 2*n+2;i++)
        poly[i]=round(polyy[i]);
        initgraph(&gdriver,&gmode,"C:\\TC\\BGI");
        setcolor(RED);
        rectangle(xmin,ymax,xmax,ymin);
        cout<<"\t\tUNCLIPPED POLYGON";
        setcolor(WHITE);
        fillpoly(n,poly);
        getch();
```

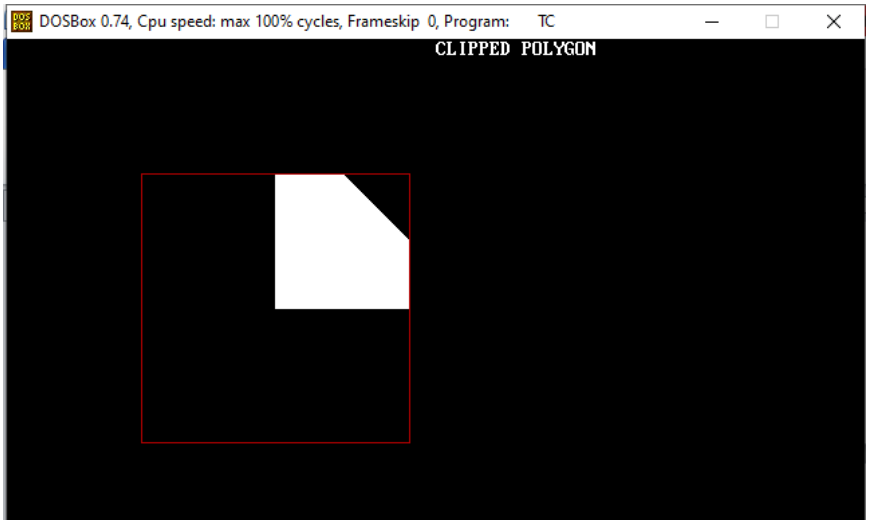
```
cleardevice();
k=0;
for(i=0;i < 2*n;i+=2)
clipl(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
n=k/2;
for(i=0;i < k;i++)
polyy[i]=arr[i];
polyy[i]=polyy[0];
polyy[i+1]=polyy[1];
k=0;
for(i=0;i < 2*n;i+=2)
clipt(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
n=k/2;
for(i=0;i < k;i++)
polyy[i]=arr[i];
polyy[i]=polyy[0];
polyy[i+1]=polyy[1];
k=0;
for(i=0;i < 2*n;i+=2)
clipr(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
n=k/2;
for(i=0;i < k;i++)
polyy[i]=arr[i];
polyy[i]=polyy[0];
polyy[i+1]=polyy[1];
k=0;
for(i=0;i < 2*n;i+=2)
clipb(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
for(i=0;i < k;i++)
poly[i]=round(arr[i]);
if(k)
fillpoly(k/2,poly);
setcolor(RED);
rectangle(xmin,ymax,xmax,ymin);
cout<<"\tCLIPPED POLYGON";
```

```
    getch();
    closegraph();
}
```

9.3.4 Output:







9.4 CONCLUSION

Polygon clipping is an important operation that computers execute all the time. Often, it is possible to feed a weird polygon to an algorithm and retrieve an incorrect result. One of the vertices may disappear, or a ghost vertex may be created. However, none of them is totally perfect. Therefore, the hunt for the perfect clipping algorithm is still open.

9.5 REFERENCES FOR FUTURE READING

- https://techzzers.wordpress.com/midpoint-subdivision-lineclipping-algorithm/
- https://www.javatpoint.com/sutherland-hodgeman-polygon-clipping
- https://www.geeksforgeeks.org/polygon-clipping-sutherland-hodgman-algorithm-please-change-bmp-imagesjpeg-png/
  https://www.ques10.com/p/11168/explain-sutherland-hodgeman-algorithm-for-polygo-1/
- Liang, Y. and Barsky, B. (1983) "An analysis and algorithm for polygon clipping", Commun. ACM 26, 11 (Nov), 868-877.

*****

MODULE IX

10

IMPLEMENTATION OF 3D TRANSFORMATIONS (ONLY COORDINATES CALCULATION)

Unit Structure

10.1 Objectives
10.2 Definition
10.3 Introduction
10.4 3-D Transformations
10.4.1 Geometric transformation
10.5 Translation
10.6 Scaling
10.7 Rotation
10.8 Reflection
10.9 Shearing
10.10 Summary
10.11 Unit End Exercise
10.12 Reference for further reading

10.1 OBJECTIVES

In Computer graphics:

- Transformation is a process of modifying and re-positioning the existing graphics.
- 3D Transformations take place in a three dimensional plane.
- 3D Transformations are important and a bit more complex than 2D Transformations.
- Transformations are helpful in changing the position, size, orientation, shape etc of the object.

10.2 DEFINITION

2-D geometry transformation is useful in showing charts, graphs, etc. various objects can be shown by 3-Dimension. Since those 3-Dimensions are:

x-coordinate-which shows width

y-coordinate-which shows height

z-coordinate-which shows depth

All these 3-axis are perpendicular to each other hence it is not easy to display an object of 3-D into 2-D.

Thus there should be a projection from 3-D to 2-D plane this is known as geometric transformation.

10.3 INTRODUCTION

3-D Transformations:

There are 2 types of transformation:

a. Geometric transformation

b. Coordinate transformation

10.4 3-D TRANSFORMATIONS

10.4.1 Geometric transformation:

In this an object is transformed to a new position or size, keeping the coordinate system stationary.

10.5 TRANSLATION

It means shifting the position of an object from one place to another without changing its shape.

T= [1 0 0 0 0 1 0 0 0 0 1 0 tx ty tz 1]

The homogenous coordinates are:

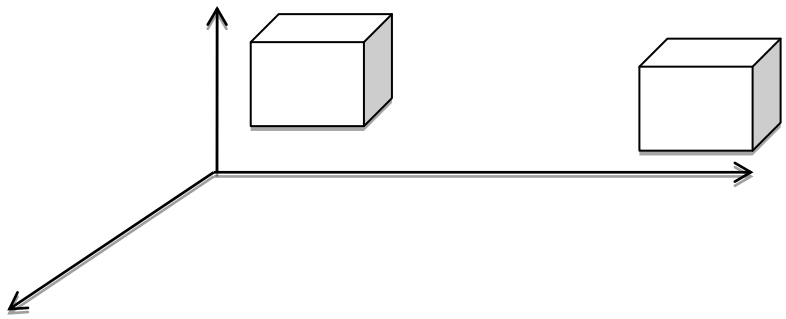
[x' y' z' 1] = [x y z 1] [1 0 0 0 0 1 0 0 0 0 1 0 tx ty tz 1]

[x' y' z' 1] = [x+tx y+ty z+tz 1]

x'=x+tx, where tx = translation in x-direction

y'=y+ty, where ty = translation in y-direction

z'=z+tz, where tz= translation in z-direction



Translating an object i.e shifting its position

Q. Perform a translation of an object having coordinates (2,3,4) with translation distance(2,4,6)

Solution:

T= [1 0 0 0 0 1 0 0 0 0 1 0 tx ty tz 1]

tx= translation distance =2

ty= translation distance =4

tz= translation distance =6

Object Coordinates are A=2, B= 3 & C=4

The homogenous coordinates are:

[A' B' C' 1] = [2 3 4 1] [1 0 0 0 0 1 0 0 0 0 1 0 2 4 6 1]

[A' B' C' 1] = [2+2 3+4 4+6 1] = [4 7 10 1]

A'= 4,

B'= 7

C'=10

After translation object position is (4,7,10)

Q. Perform a translation of an object having coordinates (4,10,12) with translation distance(2,3,4)

Solution:

T= [1 0 0 0 0 1 0 0 0 0 1 0 tx ty tz 1]

tx= translation distance =2

ty= translation distance =3

tz= translation distance =4

Object Coordinates are A=4, B= 10 & C=12

The homogenous coordinates are:

[A' B' C' 1] = [4 10 12 1] [1 0 0 0 0 1 0 0 0 0 1 0 2 3 4 1]

[A' B' C' 1] = [4+2 10+3 12+4 1] = [6 13 16 1]

A'= 6,

B'= 13

C'=16

After translation object position is (6,13,16)

10.6 SCALING

It is defined as a process of compressing and expanding an object. The scaling factor is given by: Sx,Sy, Sz.

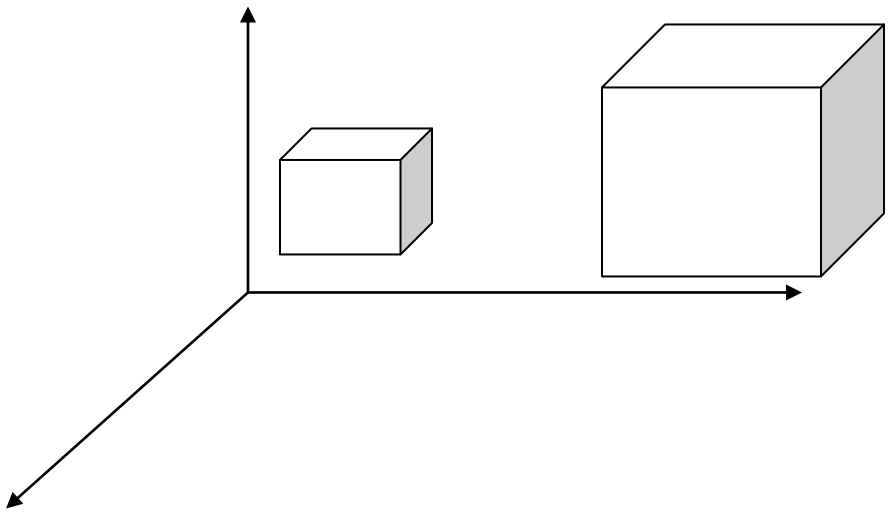
If Sx=Sy=Sz=S <1 then scaling is called as Magnification.

The 3-dimensional transformation matrix for scaling in homogenous coordinates is given by:

S = [Sx 0 0 0 0 Sy 0 0 0 0 Sz 0 0 0 0 1]

[x' y' z' 1] = [x y z 1] * [Sx 0 0 0 0 Sy 0 0 0 0 Sz 0 0 0 0 1]

=[x*Sx y*Sy z*Sz 1]



Scaling an object

Q. Perform scaling on an object with coordinates (2,2,1) with scaling factors (1,2,2)

Solution:

S = [Sx 0 0 0 0 Sy 0 0 0 0 Sz 0 0 0 0 1]

Object Coordinate positions are = 2,2,1

Scaling factors are Sx = 1, Sy=2, Sz= 2

[P' Q' R' 1] = [2 2 1 1] * [1 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1]

=[2*1 2*2 1*2 1]

=[2 4 2 1]

Therefore, the coordinates after scaling are (2,4,2)

10.7 ROTATION ABOUT AN ORIGIN:

The 3-dimensional transformation matrix of rotation in anticlockwise direction for each axis is given by:

(Rotation about x-axis):

Rx =

$$[1\ 0\ 0\ 0\ 0\ \cos\cos\theta\ \ \sin\sin\theta\ \ 0\ 0\ -\sin\sin\theta\ \ \cos\cos\theta\ \ 0\ 0\ 0\ 0\ 1]$$

(Rotation about y axis):

Ry =

$$[\cos\cos\theta\ \ 0\ \ -\sin\sin\theta\ \ 0\ 0\ 1\ 0\ 0\ \sin\sin\theta\ \ 0\ \ \cos\cos\theta\ \ 0\ 0\ 0\ 0\ 1]$$

(Rotation about z axis):

Rz =

$$[\cos\cos\theta\ \ \sin\sin\theta\ \ 0\ 0\ -\sin\sin\theta\ \ \cos\cos\theta\ \ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1]$$

Type equation here.

Note: For rotation in clockwise direction change the sign of theta (θ) in its opposite sign.

i.e if cos θ then cos (-θ) = cos θ

if sin θ then sin (- θ) = -sin θ

if –sin θ then-□sin sin (-θ) = -(-□sin sin θ)=θ

Q. Perform a rotation with an angle of 450 about y-axis followed by a rotation of 450 about x-axis of the given matrix [1 0 2 0 1 3 0 0 3 0 2 0 0 3 4 1]

Solution:

Rotation with an angle of 45^0 about y-axis is given by:

$$[\cos\cos 45\ \ 0\ \ -\sin\sin 45\ \ 0\ 0\ 1\ 0\ 0\ \sin\sin 45\ \ 0\ \ \cos\cos 45\ \ 0\ 0\ 0\ 0\ 1]$$

$$\left[\frac{1}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ 0\ 1\ 0\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$[1\ 0\ 2\ 0\ 1\ 3\ 0\ 0\ 3\ 0\ 2\ 0\ 0\ 3\ 4\ 1] \quad *$$

$$\left[\frac{1}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ 0\ 1\ 0\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$\left[\frac{1}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 3\ \frac{-1}{\sqrt{2}}\ 0\ \frac{3}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ 0\ -\frac{3}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ 0\ \frac{4}{\sqrt{2}}\ 3\ \frac{4}{\sqrt{2}}\ 1\right]$$

$$\left[\frac{3}{\sqrt{2}}\quad 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\quad 3\ \frac{-1}{\sqrt{2}}\ 0\ \frac{5}{\sqrt{2}}\quad 0\ -\frac{1}{\sqrt{2}}\ 0\ \frac{4}{\sqrt{2}}\quad 3\ \frac{4}{\sqrt{2}}\ 1\right]$$

Rotation with an angle of 450 about x-axis is given as:

Rx =
$[1\ 0\ 0\ 0\ 0\ \cos\cos\theta\ \ \sin\sin\theta\ \ 0\ 0\ -\sin\sin\theta\ \ \cos\cos\theta\ \ 0\ 0\ 0\ 0\ 1] =$
$[1\ 0\ 0\ 0\ 0\ \cos 45\ \ \sin\sin 45\ \ 0\ 0\ -\sin\sin 45\ \ \cos\cos 45\ \ 0\ 0\ 0\ 0\ 1]$

$$\left[\frac{3}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 3\ \frac{-1}{\sqrt{2}}\ 0\ \frac{5}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ \frac{4}{\sqrt{2}}\ 3\ \frac{4}{\sqrt{2}}\ 1\right]*$$
$$\left[1\ 0\ 0\ 0\ 0\ \frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ 0\ 0\ -\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$\left[\frac{3}{\sqrt{2}}\ -\frac{1}{2}\ \frac{1}{2}\ 0\ \frac{1}{\sqrt{2}}\quad \frac{3}{\sqrt{2}}+\frac{1}{2}\ \frac{3}{\sqrt{2}}\ -\frac{1}{2}\ 0\ \frac{5}{\sqrt{2}}\ \frac{1}{2}\ -\frac{1}{2}\ 0\ \frac{4}{\sqrt{2}}\ \frac{3}{\sqrt{2}}\ -\frac{4}{2}\quad \frac{3}{\sqrt{2}}+\frac{4}{2}\ 1\right]$$

$$\left[\frac{3}{\sqrt{2}}\ -\frac{1}{2}\ \frac{1}{2}\ 0\ \frac{1}{\sqrt{2}}\quad \frac{3}{\sqrt{2}}+\frac{1}{2}\ \frac{3}{\sqrt{2}}\ -\frac{1}{2}\ 0\ \frac{5}{\sqrt{2}}\ \frac{1}{2}\ -\frac{1}{2}\ 0\ \frac{4}{\sqrt{2}}\ \frac{3}{\sqrt{2}}\ -\frac{4}{2}\quad \frac{3}{\sqrt{2}}+\frac{4}{2}\ 1\right] =$$

Q. Perform a rotation with an angle of 450 about x-axis followed by a rotation of 450 about y-axis of the given matrix [1 0 2 0 1 3 0 0 3 0 2 0 0 3 4 1]

Solution:

Rotation with an angle of 45^0 about x-axis

Rx =
$[1\ 0\ 0\ 0\ 0\ \cos\cos\theta\ \ \sin\sin\theta\ \ 0\ 0\ -\sin\sin\theta\ \ \cos\cos\theta\ \ 0\ 0\ 0\ 0\ 1] =$
$[1\ 0\ 0\ 0\ 0\ \cos 45\ \ \sin\sin 45\ \ 0\ 0\ -\sin\sin 45\ \ \cos\cos 45\ \ 0\ 0\ 0\ 0\ 1]$

$$= [1\ 0\ 2\ 0\ 1\ 3\ 0\ 0\ 3\ 0\ 2\ 0\ 0\ 3\ 4\ 1]$$
$$* \left[1\ 0\ 0\ 0\ 0\ \frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ 0\ 0\ -\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$\left[1\ -\frac{2}{\sqrt{2}}\ \frac{2}{\sqrt{2}}\quad 0\quad 1\quad \frac{3}{\sqrt{2}}\ \frac{3}{\sqrt{2}}\quad 0\ 3\ -\frac{2}{\sqrt{2}}\ \frac{2}{\sqrt{2}}\quad 0\ 0\ \frac{3}{\sqrt{2}}\ -\frac{4}{\sqrt{2}}\quad \frac{3}{\sqrt{2}}+\frac{4}{\sqrt{2}}\quad 1\right]$$

$$=$$

$$\left[1\ -\frac{2}{\sqrt{2}}\quad \frac{2}{\sqrt{2}}\quad 0\quad 1\quad \frac{3}{\sqrt{2}}\quad \frac{3}{\sqrt{2}}\quad 0\ 3\ -\frac{2}{\sqrt{2}}\quad \frac{2}{\sqrt{2}}\quad 0\ 0\ -\frac{1}{\sqrt{2}}\quad \frac{7}{\sqrt{2}}\quad 1\right]$$

Rotation of 45^0 about y-axis:

$$[cos\ cos\ 45\ \ 0\ \ -sin\ sin\ 45\ \ 0\ 0\ 1\ 0\ 0\ \ sin\ sin\ 45\ \ 0 \quad cos\ cos\ 45\ \ 0\ 0\ 0\ 0\ 1]$$

$$\left[\frac{1}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ 0\ 1\ 0\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$= \begin{matrix}\left[1\ -\frac{2}{\sqrt{2}}\ \ \frac{2}{\sqrt{2}}\ \ 0\ \ \ 1\ \ \ \frac{3}{\sqrt{2}}\ \ \frac{3}{\sqrt{2}}\ \ 0\ 3\ -\frac{2}{\sqrt{2}}\ \ \frac{2}{\sqrt{2}}\ \ 0\ 0\ -\right.\\ \left.\frac{1}{\sqrt{2}}\ \ \frac{7}{\sqrt{2}}\ \ 1\right]\end{matrix}$$

$$* \left[\frac{1}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ 0\ 1\ 0\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$= \begin{matrix}\left[\frac{1}{\sqrt{2}}+\frac{2}{2}\ \ \frac{-2}{\sqrt{2}}-\frac{1}{\sqrt{2}}+\frac{2}{2}\ \ \ 0\ \ \ \frac{1}{\sqrt{2}}+\frac{3}{2}\ \ \ \frac{3}{\sqrt{2}}\ \ -\frac{1}{\sqrt{2}}+\frac{3}{2}\ \ \ 0\ \frac{3}{\sqrt{2}}+\right.\\ \left.\frac{2}{2}-\frac{2}{\sqrt{2}}\ \ -\frac{3}{\sqrt{2}}+\frac{2}{2}\ \ \ 0\ \frac{7}{\sqrt{2}}-\frac{1}{\sqrt{2}}\ \ \frac{7}{2}\ \ \ \ 1\right]\end{matrix}$$

Q. Perform a rotation with an angle of 450 about y-axis followed by a rotation of 450 about z-axis of the given matrix
$[1\ 0\ 2\ 0\ 1\ 3\ 0\ 0\ 3\ 0\ 2\ 0\ 0\ 3\ 4\ 1]$

Solution:

Rotation with an angle of 45^0 about y-axis is given by:

$$[cos\ cos\ 45\ \ 0\ \ -sin\ sin\ 45\ \ 0\ 0\ 1\ 0\ 0\ \ sin\ sin\ 45\ \ 0 \quad cos\ cos\ 45\ \ 0\ 0\ 0\ 0\ 1]$$

$$\left[\frac{1}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ 0\ 1\ 0\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$= [1\ 0\ 2\ 0\ 1\ 3\ 0\ 0\ 3\ 0\ 2\ 0\ 0\ 3\ 4\ 1]\ * \left[\frac{1}{\sqrt{2}}\ 0\ -\frac{1}{\sqrt{2}}\ 0\ 0\ 1\ 0\ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ 0\ 0\ 0\ 0\ 1\right]$$

$$\begin{matrix}\left[\frac{1}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ \ \ 0\ \ \ -\frac{1}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ \ \ 3\ \frac{-1}{\sqrt{2}}\ 0\ \frac{3}{\sqrt{2}}+\frac{2}{\sqrt{2}}\ \ \ 0\ \ \ -\frac{3}{\sqrt{2}}\right.\\ \left.+\frac{2}{\sqrt{2}}\ 0\ \frac{4}{\sqrt{2}}\ \ \ 3\ \frac{4}{\sqrt{2}}\ 1\right]\end{matrix}$$

$$\left[\frac{3}{\sqrt{2}}\ \ \ 0\ \frac{1}{\sqrt{2}}\ 0\ \frac{1}{\sqrt{2}}\ \ \ 3\ \frac{-1}{\sqrt{2}}\ 0\ \frac{5}{\sqrt{2}}\ \ \ 0\ -\frac{1}{\sqrt{2}}\ 0\ \frac{4}{\sqrt{2}}\ \ \ 3\ \frac{4}{\sqrt{2}}\ 1\right]$$

Rotation of 45^0 about z-axis:

Rz =

$$[cos\,cos\,\theta \quad sin\,sin\,\theta \quad 0\ 0 \quad -sin\,sin\,\theta \quad cos\,cos\,\theta \quad 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1]$$

$$= \left[\frac{1}{\sqrt{2}} \ \frac{1}{\sqrt{2}} \ 0\ 0 \ -\frac{1}{\sqrt{2}} \ \frac{1}{\sqrt{2}} \ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\right]$$

$$= \left[\frac{3}{\sqrt{2}} \quad 0 \ \frac{1}{\sqrt{2}} \ 0 \ \frac{1}{\sqrt{2}} \quad 3 \ \frac{-1}{\sqrt{2}} \ 0 \ \frac{5}{\sqrt{2}} \quad 0 \ -\frac{1}{\sqrt{2}} \ 0 \ \frac{4}{\sqrt{2}} \quad 3 \ \frac{4}{\sqrt{2}} \ 1\right] * \left[\frac{1}{\sqrt{2}} \ \frac{1}{\sqrt{2}} \ 0\ 0 \ -\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\right]$$

10.8 REFLECTION

The 3-dimensional transformation matrix of reflection is given by:

Reflection about xy plane Rxy = $[1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 1]$

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] * [1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 1]$$

$$= [x\ y\ -z\ 1]$$

Reflection about yz plane Ryz $[-1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1]$

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] * [-1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1]$$

$$= [-x\ y\ z\ 1]$$

Reflection about xz plane Rxz = $[1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1]$

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] * [1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1]$$

$$= [x\ -y\ z\ 1]$$

Reflection about the origin Rxzy = $[-1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 1]$

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] * [-1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 0\ -1\ 0\ 0\ 0\ 1]$$

$$= [-x\ -y\ -z\ 1]$$

10.9 SHEARING

Shearing is done along x-axis, y-axis and z-axis

Shearing along x-axis-In this shearing factors are Shxy and Shxz

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] * [1\ Shxy\ Shxz\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1]$$

$$= [x\ x.Shxy + y\ x.Shxz + z\ 1\ ]$$

Shearing along y-axis-In this shearing factors are Shyz and Shyx

$$[x'\ y'\ z'\ 1\ ] = [x\ y\ z\ 1\ ] * [0\ 0\ 0\ 0\ Shx\ 1\ Shyz\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ ]$$

$$= [y.Shx\ y\ y.Shyz + z\ 1\ ]$$

Shearing along z-axis-In this shearing factors are Shzx and Shzy

$$[x'\ y'\ z'\ 1\ ] = [x\ y\ z\ 1\ ] * [1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ Shzx\ Shzy\ 0\ 0\ 0\ 0\ 0\ 1\ ]$$

$$= [x + z.Shzx\ y + z.Shzy\ 0\ 1\ ]$$

10.10 SUMMARY

3-D Transformation is the process of manipulating the view of a three-D object with respect to its original position by modifying its physical attributes through various methods of transformation like Translation, Scaling, Rotation, Shear, etc.

10.11 UNIT END EXERCISE

1. What is 3-D Transformation
2. What is geometric transformation
3. What is coordinate transformation
4. What is meant by Translation.
5. What is meant by rotation.
6. What is scaling explain with the help of an example.
7. Explain all types of rotation
8. Explain shearing in detail
9. Explain reflection in detail
10. What are the different types of geometric transformation?

10.12 REFERENCE FOR FURTHER READING

- Computer_Graphics_C_Version_by_Donald_Hearn_and_M_Pauline _Baker_II_Edition
- Computer graphics by Atul P. Godse, Dr. Deepali A. Godse
- https://www.geeksforgeeks.org/computer-graphics-3d-translation-transformation/#:~:text=3%2DD%20Transformation%20%3A,%2C%20Rotation%2C%20Shear%2C%20etc.

*****

UNIT X

11

OUTPUT PRIMITIVES & ITS ALGORITHM

Unit Structure

11.0 Objective
11.1 Introduction
11.2 Fractals and self-similarity overview
  11.2.1 Geometric Fractals
  11.2.2 Generating fractals
  11.2.3 Classification of fractals
  11.2.4 Characteristics of fractals
  11.2.5 Elements of fractals
  11.2.6 Application of fractals
  11.2.7 Fractals in real life
  11.2.8 Algorithms of fractals
11.3 Koch curve
  11.3.1 Construction of Koch curve
11.4 Sierpinski Triangle
11.5 Summary
11.6 Unit End Exercise
11.7 References for Future Reading

11.0 OBJECTIVE

This chapter will able you to understand the following concept:

- Fractals and self-similarity
- Characteristics of fractals
- Elements of fractals
- Fractals algorithm
- Fractals applications
- Types of fractals Koch curve and Sirpenski Triangle.
- Koch curve construction and implementation
- Sierpinski Triangle way of fractals

11.1 INTRODUCTION

Manmade or artificial objects usually have either flat surface, which can be described with polygons or smooth curved surfaces, which we have just studied. But objects accruing in nature often have rough, jagged, random edges. Attempting to draw things like maintains, trees, rivers or lightning bolts directly with lines or polygons require lots of specification. It is describing to let the machine do the work and draw the jagged lines. We would just give the endpoints to the computer and let the machine draw the jagged lined between them. The lines should be closely approximating the behavior of nature, so it will look better. This all scenario we called it as fractals.

We can use the computer to easily generate self-similar fractal curves. The self-similar drawing can be done by a self-referencing procedure. A curve is composed of N self-similar pieces, each scaled by 1/s. of course, a computer routine should terminate, which a true fractal does not.

11.2 FRACTALS AND SELF SIMILARITY

A fractal line is fine for the path of a lightning the bolt, but for something like a three dimensional mountain range, we need a fractal surface. There are several ways to extend the fractal idea to surface. The one can present is based on triangles.

We can use the computer to easily generate self-similar fractal curves. The self-similar drawing can be done by a self-referencing procedure. A curve is composed of N self-similar pieces, each scaled by 1/s. of course, a computer routine should terminate, which a true fractal does not.

Being able to use the computer to generate fractal curve means that the user can easily generate realistic cost line and mountains peak or lightning bolts without concern for all the small bends and wiggles, and the user need only give the endpoints.

The algorithm presented not very efficient in that it calculated each point twice. Care must be taken with the seed values to ensure that the same fractal edge is generated for two bordering triangles.

11.2.1 Geometric Fractals:

A fractal is " it is considered as geometric shape that is rough or fragmented and that can bifurcate in small parts, every part which is in reduced/size copy of the whole". The term was originated by Benoît Mandelbrot in 1975 and was come from the Latin word fractus meaning broken or fractured.

A Geometric object fractal has the following features:

- Arbitrarily small scales are at fine structure

- In traditional Euclidean geometric language, it is described as too irregular and it is easily accessible.
- Its nature is self-similar (at least approximatively or stochastically)
- It has a topological dimension less than Hausdorff dimension (but this requirement is not met by space-filling curves such as the Hilbert curve)
- it has recursive and simple function definition.

Because of same feature of appearing similar at all the levels of magnification, fractals are sometime often considered as 'infinitely complex'. Like example of mountain ranges, clouds and lightning bolts.

However, in case of all object it is not all self-similar objects can be a fractal such as the real line like a straight Euclidean line which is formally self-similar in nature but fails to have other fractal characteristics.

11.2.2 Generating Fractals:

Following are some main techniques for generating fractals are:

1. **Escape-time fractals**: The recurrence relation is defined by an object at each point to bifurcate the object in to self-similarity for a complex problem such as the Lyapunov fractal, Julia set Mandelbrot set, the Burning Ship fractal.
2. **Iterated function systems**: in this type of fractals the replacement theory is used to describe the geometric object. For example, Harter-Highway dragon curve, Cantor set, Sierpinski carpet, T-Square, Menger , Sierpinski gasket, Piano curve, Koch snowflake, sponge.
3. **Random fractals:** it is not following the deterministic approach it is just following the stochastic feature, there are some example of, Brownian tree, trajectories of the Brownian motion, Lévy flight, fractal landscapes.

In the latter stage it is called asmass- or dendritic fractals, for example, reaction-limited aggregation clusters or diffusion-limited aggregation.

11.2.3 Classification of Fractals:

With property of their self-similarity fractals can be classified into three main categories.

- **Exact self-similarity:** This is one of the main self-similarity activity and it is well versed; the fractal appearance is different for all object at all different scales. Iteration property of fractals defined can be display by exact self-similarity in some function.
- **Quasi-self-similarity:** This activity is one of the weakest activity of self-similarity; most of the time the fractal appearance is not exactly but approximately and it is again different at different scales. In

Quasi-self-similar fractals, it presents the small copies of the entire object fractal in distorted and degenerate forms. Fractals can be represented by recurrence relations and it is usually quasi-self-similar but not exactly self-similar.

- **Statistical self-similarity**: This is one more weakest type of self-similarity; these self-similarity fractal has statistical or numerical calculated numbers which are preserved across scales. Most of the time the definitions will be true for "fractal" which the implication of some form of statistical self-similarity. continuous Fractal dimension is a numerical measure which is preserved across scales whereas in case of random fractals are not exactly or Quasi self-similar but it is statistically self-similar, but neither exactly nor quasi-self-similar.

- **Invariant fractal sets**: with the help of nonlinear transformation invariant fractals are formed. self-squaring fractals are the main property of this type, in which squaring function is used. Squering function is self-inverse fractals which can be used for complex problem such as the mendelbrot set.

11.2.4 Characteristics of Fractals:

Fractal characteristics firstly introduced by ‘Alain Boutot’. All scales and observations are taken into consideration to do the fractals. In language of Euclidian geometry, it cannot be described as it is irregular locally and globally.

1. **Self-Similarity:** Each part is same as whole parts are available in the object.

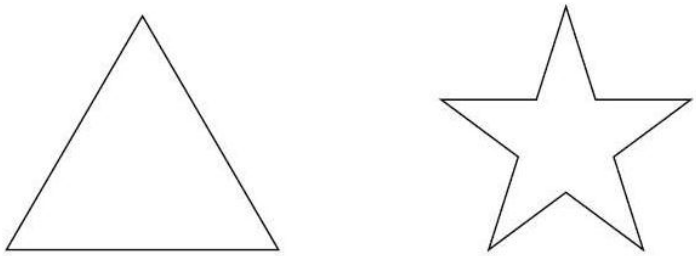
2. **Scaling:** All spatial resolution is same as other spatial resolution as it follows the self-similarity feature. Smaller feature can inherit the feature of larger element.

3. **Bounded Infinity:** within the finite boundary one can illustrate the infinite elements of same shape and size. The Koch curve or snowflake can illustrate the bounded infinity very clearly.

4. **Fractal dimensions:** in self-similarity fractal dimension can be defined as repeated number of action to be taken in the object drawing.

11.2.5 Elements of Fractals:

Fractal can be performing with the two main elements initiator and generator. With initiator it can starts and draw equilateral triangle and it can be divide into same fractal with the help of generator. Initiator will divide the line segment into three equal parts and the process is repeated until the generator generates a fine image.



dimension

11.2.6 Application of Fractals:

- Classification of histopathology slides in medicine
- Generation of new music
- Generation of various art forms
- Signal and image compression
- Seismology
- Computer and video game design, especially computer graphics for organic environments and as part of procedural generation
- Fractography and fracture mechanics
- Fractal antennas — Small size antennas using fractal shapes
- Neo-hippies t-shirts and other fashion.
- Generation of patterns for camouflage, such as MARPAT.
- Digital sundial

11.2.7 Fractals in real life:

- In pour day to day life fractals are easily found in our environment. Like in cloud or mountain nature object the self-similarity structure is extended in finite step or defines scale range. Another examples include river flow, snowflake, in vegetable it found in cauliflower or broccoli, and in a human body systems of blood vessels.
- As we know the fractals uses the recursive algorithm to make trees and ferns which are fractal in nature and can be drawn on screen with the same algorithm. These example shows the recursive nature used such as a branch of a tree can be used to make frond from a fern which can be replica of the main system but it is not identical, and it has a same shape in nature.
- In the example of mountain, with using a fractal the mountain surface can be modelled on a computer screen, it will initialised with a triangle in 3D space and each central point connect with the line

segment this will be done for each side, resulting in 4 triangles. The central points are then randomly moved up or down, within a defined range. This process is repetitive procedure for a finite step, with decomposing into iteration by half range at each stage. The property of recursive algorithm assures the self-similarity in the object which is statistically similar to each other.

- In some American artist painting like Jackson Pollock the Fractal patterns have been found, while in that Pollock's paintings the object is appear to be composed of chaotic dripping and splattering, while going through the analysis the computer has found fractal patterns in his work.

11.2.8 Algorithm of Fractals:

The below algorithm presents a procedure for drawing a fractal line segment from the current position to the specified position. It requires as arguments the endpoint a weight factor, and the desired recursion depth.

Algorithm FRACTAL-LINE-ABS-3 (X, Y, Z, W, N, FSEED) user routine from drawing fractals lines

Arguments X, Y, Z the point to which to draw a line

W described the roughness of the curve

N the desired depth of recursion

FSEED seed for fractal pattern

Global DF-PEN-X, DF-PEN-Y, DF-PEN-Z current pen position

SEED the seed used by the random number generator

Local L the approximate line length

BEGIN

SEED □ FSEED;

L□ | X – DF-PEN-X | + | Y – DF-PEN-Y | + | Z – DF-PEN-Z |;

FRACTAL – SUBDIVIDE (DF – PEN- X, DF – PEN- Y, DF – PEN- Z, X, Y, Z, L * W, N);

END;

The SEED is a number given to the random number generator. It is assumed the RND not only returns the random number calculated from SEED but also alters the value reason for giving the SEED so that on the next call a different random number will be returned. The fractal will depend upon the initial seed used.

```
Algorithm FRACTAL-LINE-SUBDIVIDE (X1, Y1, Z1, X2, Y2, Z2, S, N) Draws a fractal line between points X1, Y1, Z1 and X2, Y2, Z2

Arguments X1, Y1, Z1 the point to start the line

          X2, Y2, Z2 the point to stop the line

          S offset scale factor

          N the desired depth of recursion

          FSEED seed for fractal pattern

  Local   XMID, YMID, ZMID coordinates at which to break the line

  BEGIN

    IF N=0 THEN

      BEGIN

        Recursion stops, so just draw the line segment

        LINE-ABSA3 (X2, Y2, Z2)

  END

 ELSE

  BEGIN

    Calculate the halfway point

    XMID □ (X1 + X2)/2 + S * GAUSS;

    YMID □ (Y1 + Y2)/2 + S * GAUSS;

    ZMID □ (Z1 + Z2)/2 + S * GAUSS;

    Draw the two halves

    FRACTAL-SUBDIVIDE (X1, Y1, Z1, XMID, YMID, ZMID, S/2, N-1);

    FRACTAL-SUBDIVIDE (X2, Y2, Z2, XMID, YMID, ZMID, S/2, N-1);

END;

 RETURN;

END
```

We approximate a Gaussian distribution by averaging several uniformly random numbers. Half the numbers are added and half subtracted to provide for zero mean.

Algorithm GAUSS calculates an approximate Gaussian between -1 and 1.

```
Local    I for summing samples
BEGIN
    GAUSS □ 0;
    FOR I 1 TO 6 DO GAUSS □ GAUSS + RND – RND;
    GAUSS □ GAUSS / 6;
    RETURN;
END;
```

Classification of fractals:

Fractals can also be classified according to their self-similarity. There are three types of self-similarity found in fractals:

Exact self-similarity: This is the strongest type of self-similarity; the fractal appears identical at different scales. Fractals defined by iterated function systems often display exact self-similarity.

Quasi-self-similarity: This is a loose form of self-similarity; the fractal appears approximately (but not exactly) identical at different scales. Quasi-self-similar fractals contain small copies of the entire fractal in distorted and degenerate forms. Fractals defined by recurrence relations are usually quasi-self-similar but not exactly self-similar.

Statistical self-similarity: This is the weakest type of self-similarity; the fractal has numerical or statistical measures which are preserved across scales. Most reasonable definitions of "fractal" trivially imply some form of statistical self-similarity. (Fractal dimension itself is a numerical measure which is preserved across scales.) Random fractals are examples of fractals which are statistically self-similar, but neither exactly nor quasi-self-similar.

11.3 KOCH CURVE

In mathematic curve one of the oldest fractals described is, The Koch snowflake (also known as the Koch star and Koch island). The Koch snowflake is based on the Koch curve, which appeared in a 1904 paper titled "On a continuous curve without tangents, constructible from elementary geometry" by the Swedish mathematician Helge von Koch.

Self-similarity feature can be known as fractal and that fractal is also called as Koch curve. It is built from straight line segment and then it is divided into three equal parts; again the middle part id bifurcates in

equilateral triangle. Fractal objects can be achieved using one of the popular method that is L-system (Lindenmayer system). In this method the recursive function is used to create same size and same shape of the object several times.

This can be result into Koch curve with middle segment, middle can be varying every time and can be converted into a regular positive integers greater than or equal to 3.

The Koch curve can be built up iteratively with the specific iteration, in sequence stage. The starting is done with the equilateral triangle, and each successive stage the adding the construction of equilateral triangle to it. After each iteration the triangle size gets small. The size of area where the triangle constructed is increases without bond, consequently the snowflake is occupying the area but the area is finite.

Preliminaries:

It can be expressed by following rewrite

system (L-system)

Alphabet : F

Constant : +, -

Axiom : F

Production rule: F ->F +F --F +F

Here, F means "draw forward", + means "turn left

60°" and - means "turn right 60°".

To draw a Koch snowflake curve, the Prod. Rule 1 is

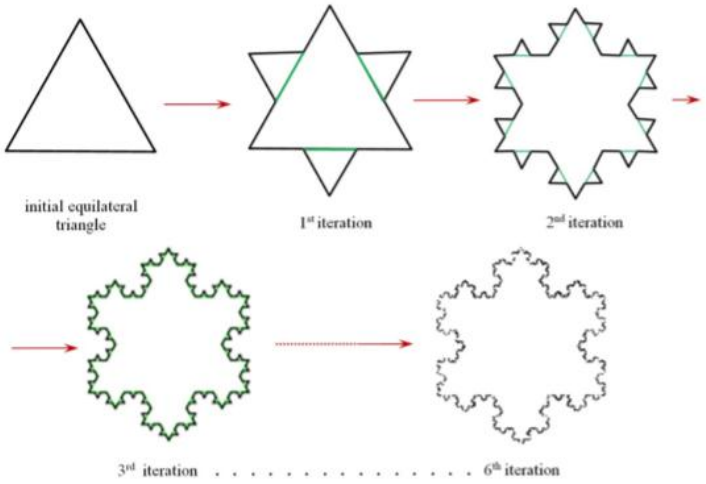
applied on axiom “F - - F - - F”

11.3.1 Construction of KOCH CURVE:

Koch curve can be constructed with an equilateral triangle, then recursively changing with the line segments that create a side of equilateral triangle as follows:

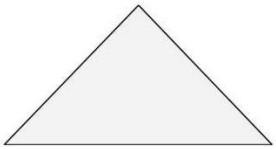
1. First, divide the line segment into three equal parts.
2. Draw an equilateral triangle with this three point which we get in step one with the 3 equal segments.
3. Remove the line segment that is the base of the triangle

4. After adding these three equilateral triangle the shape become the star
5. Follow the same process as much as you want the shape is to be fixed, but it should be finite in nature.

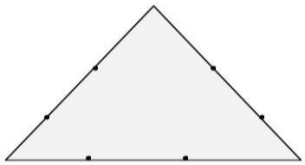


Construction:

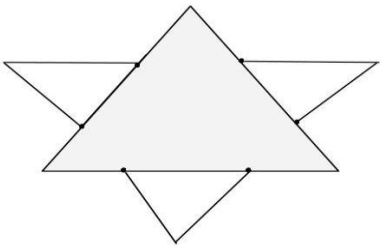
Step 1: first draw an equilateral triangle. It should divide in three



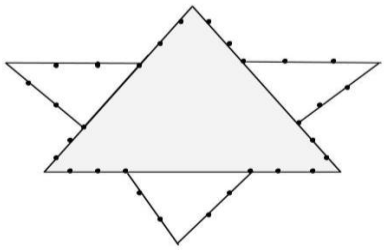
Step2: divide each of 3 side in equal parts



Step3: Three side which we have created draw an equilateral triangle in each middle part.

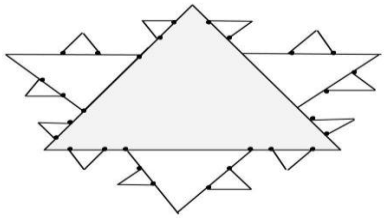


Step4: Divide each outer side into thirds. You can see the 2nd generation of triangles covers a bit of the first. These three line segments shouldn't be parted in three.



Step5:

Draw an equilateral triangle on each middle part.



11.4 SIRPENSKI TRIANGLE

The mathematician had taken much efforts on giving topological characterization of continuum and from this we got many example of topological space with some more properties. In which Sierpinski algorithm is most famous algorithm. The Sierpiński gasket is defined as follows: Take a solid equilateral triangle, divide it into four congruent equilateral triangles, and remove the middle triangle; then do the same with each of the three remaining triangles; and so on.

The simple continuous curve in the plain plane with some limit of fractal image is also known as Sierpiński construction. Repeated modification can have done in same manner or analogous manner can be formed koch snowflake. The base of equilateral triangle is the starting line segment of plane (initial curve).

Three iterated function of the Sierpinski gasket consists of three self-similar pieces in the iterated function system. One can zoom the object and can see the different parts of the gasket at at the same time, you will get the similar basic shape continually repeated for finite number of step.

Angle 120

Axiom F

F —> F+F−F−F+F

Let's take an example, we initiated with said axiom F, which is the various sides of the triangle. In the first iteration we have used the rule stated above which can be regenerated.

We can see the generation of the triangle with boundary and also seen how the rule used for removed inside triangle. In this concept iteration repetition will make more of the interior triangles but it will leave out the

other two sides of the main triangle. In case of L-system animation, you will notice in the animation that it seems to pause occasionally. some of the sides of the triangles more than once will traced by L-system.

Sierpinski's gasket generation by another L-system is given by

Angle 60

Axiom FX

F —> Z

X —> +FY-FX-FY+

Y —> -FX+FY+FX-

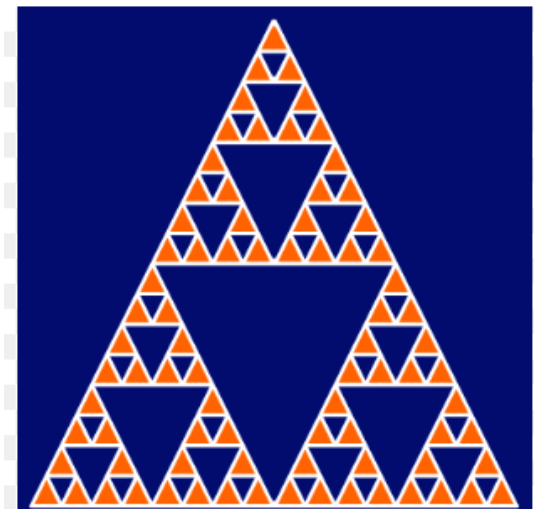
In the first L system each curve never intersects with itself in a different way, such as this construction will show that the Sierpinski gasket is actually a plane curve. The second iteration function is given in the corresponding system.

The fourth iteration of each L-system is given below in the figure.

Different mathematical tools are available to draw Sierpiński construction but the easiest way is to draw with pen and paper.

Steps:

1. Take pen and paper and draw a equilateral triangle.
2. Split the edges between two parts.
3. Divide that triangle into 4 smaller triangle
4. Repeat step 3 for the remaining triangle as much as you want.



At each recursive stage, replace each line segment on the curve with three shorter ones, each of equal length, such that:

1. the three line segments replacing a single segment from the previous stage always make 120° angles at each junction between two consecutive segments, with the first and last segments of the curve either parallel to the base of the given equilateral triangle or forming a 60° angle with it.

2. no pair of line segments forming the curve at any stage ever intersect, except possibly at their endpoints.

3. every line segment of the curve remains on, or within, the given equilateral the central downward pointing equilateral triangular regions that are external to the limiting curve.

We can describe the amount of variation in the object detail with a number called fractal dimension, unlike the Euclidian dimensions, this number is not necessarily an integer. The fractal dimension of an object is something referred to as the fractional dimension.

Special Properties:

The Sierpinski triangle curve is also called as Sierpinski gasket or Sierpinski triangle or both. It is derived by the Mandelbrot who first gave it the name "Sierpinski's gasket." Sierpinski described the construction to give an example of "a curve simultaneously Cantorian and Jordanian, of which every point is a point of ramification." Basically, this means that it is a curve that crosses itself at every point.

11.5 SUMMARY

A fractal line is fine for the path of a lightning the bolt, but for something like a three dimensional mountain range, we need a fractal surface. There are several ways to extend the fractal idea to surface. The one can present is based on triangles. Fractal characteristics firstly introduced by ‘Alain Boutot’. All scales and observations are taken into consideration to do the fractals. In language of Euclidian geometry, it cannot be described as it is irregular locally and globally. In mathematic curve one of the oldest fractals described is, The Koch snowflake (also known as the Koch star and Koch island). The Koch snowflake is based on the Koch curve, which appeared in a 1904 paper titled "On a continuous curve without tangents, constructible from elementary geometry" by the Swedish mathematician Helge von Koch. The mathematician had taken much efforts on giving topological characterization of continuum and from this we got many example of topological space with some more properties. In which Sierpinski algorithm is most famous algorithm.

11.6 UNIT END EXERCISE

1. Explain the concept of Koch curve in detail.

2. Explain Fractals various algorithm.

3. Explain Sierpinski algorithm in detail.

4. Explain construction of Koch curve in detail.
5. List and explain the application of fractals.
6. Explain characteristics of fractals.

11.7 REFERENCES FOR FUTURE READING

- Computer Graphics C version 2nd Edition by Donald D. Hearn and M. Pauline Baker
- Computer Graphics A programming approach 2nd Edition by Steven Harrington McGraw Hill
- Fundamental of Computer Graphics 3rd Edition by Peter Shirley and Steven Marschner
- Computer Graphics from Scratch: A Programmer's Introduction to 3D Rendering by Gabriel Gambetta

*****

MODULE XI

12

INTRODUCTION TO ANIMATION

Unit Structure

12.0 Objectives
12.1 Introduction
12.2 Summary
12.3 References
12.4 Unit End Exercises

12.0 OBJECTIVES

Animation refers to the movement on the screen of the display device created by displaying a sequence of still images. Animation is the technique of designing, drawing, making layouts and preparation of photographic series which are integrated into the multimedia and gaming products. Animation connects the exploitation and management of still images to generate the illusion of movement. A person who creates animations is called animator. He/she use various computer technologies to capture the pictures and then to animate these in the desired sequence.

Animation includes all the visual changes on the screen of display devices. These are:

1. Change of shape as shown in fig:

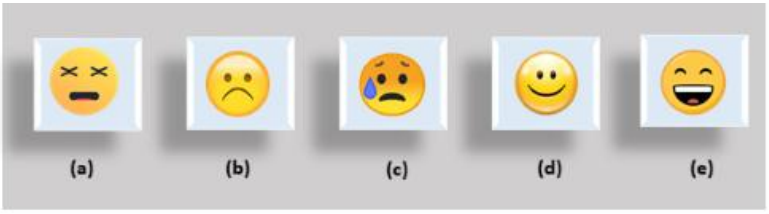


Fig: Change in Shape

2. Change in size as shown in fig:

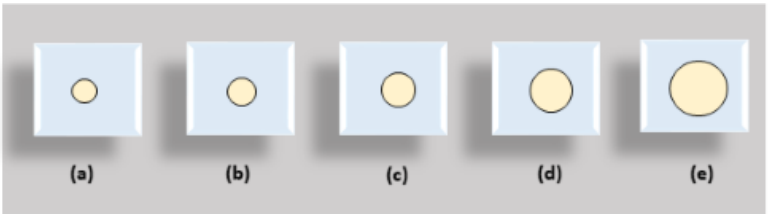


Fig: Change in Size

3. Change in color as shown in fig:

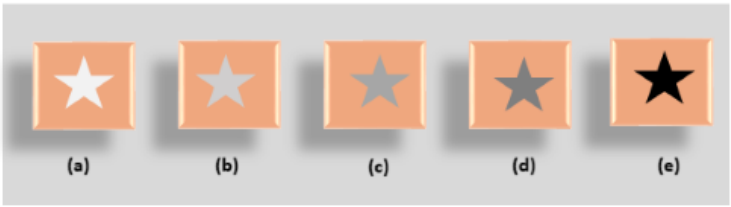


Fig: Change in Color

4. Change in structure as shown in fig:

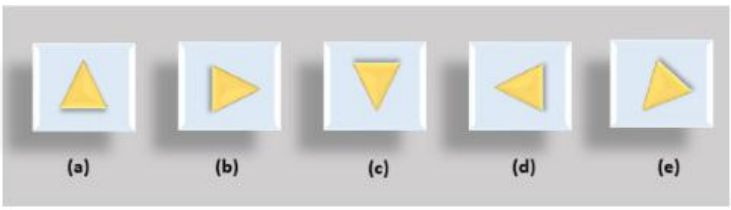


Fig: Change in Structure

5. Change in angle as shown in fig:

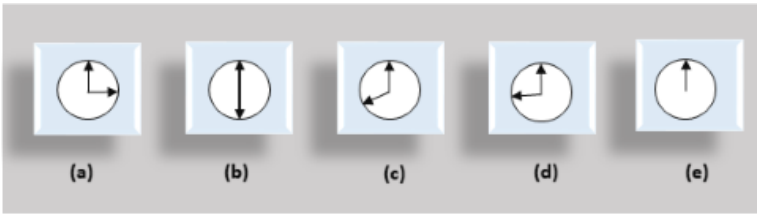


Fig: Change in angle

12.1 Introduction

Application Areas of Animation:

1. **Education and Training:** Animation is used in school, colleges and training centers for education purpose. Flight simulators for aircraft are also animation based.

2. **Entertainment:** Animation methods are now commonly used in making motion pictures, music videos and television shows, etc.

3. **Computer Aided Design (CAD):** One of the best applications of computer animation is Computer Aided Design and is generally referred to as CAD. One of the earlier applications of CAD was automobile designing. But now almost all types of designing are done by using CAD application, and without animation, all these work can't be possible.

4. **Advertising:** This is one of the significant applications of computer animation. The most important advantage of an animated advertisement is that it takes very less space and capture people attention.

5. **Presentation:** Animated Presentation is the most effective way to represent an idea. It is used to describe financial, statistical, mathematical, scientific & economic data.

Animation Functions:

1. **Morphing:** Morphing is an animation function which is used to transform object shape from one form to another is called Morphing. It is one of the most complicated transformations. This function is commonly used in movies, cartoons, advertisement, and computer games.

For Example:

1.Human Face is converted into animal face as shown in fig:

2. Face of Young person is converted into aged person as shown in fig:

The process of Morphing involves three steps:

1. In the first step, one initial image and other final image are added to morphing application as shown in fig: Ist & 4th object consider as key frames.

2. The second step involves the selection of key points on both the images for a smooth transition between two images as shown in 2nd object.

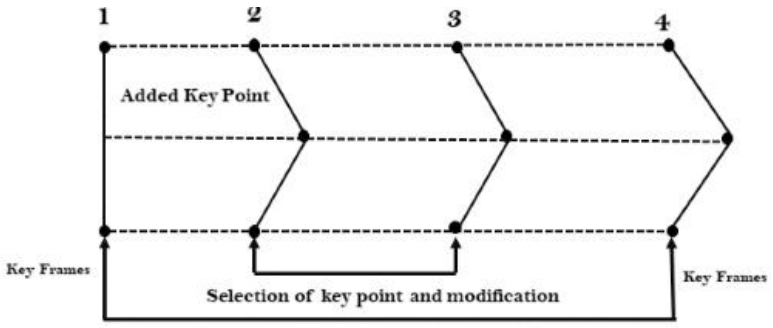


Fig: Process of Morphing

3. In the third step, the key point of the first image transforms to a corresponding key point of the second image as shown in 3rd object of the figure.

2. **Wrapping:** Wrapping function is similar to morphing function. It distorts only the initial images so that it matches with final images and no fade occurs in this function.

3. **Tweening:** Tweening is the short form of 'inbetweening.' Tweening is the process of generating intermediate frames between the initial & last final images. This function is popular in the film industry.

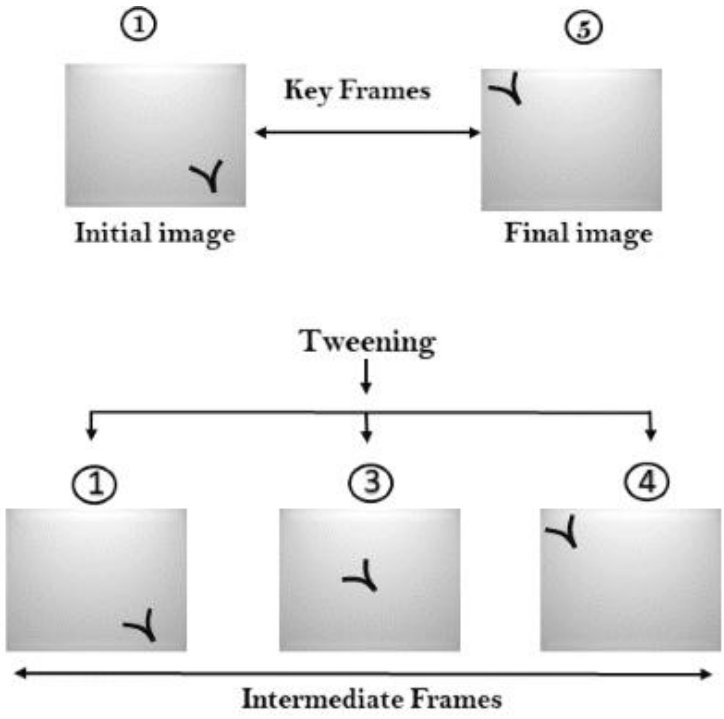


Fig: Tweening

4. **Panning:** Usually Panning refers to rotation of the camera in horizontal Plane. In computer graphics, Panning relates to the movement of fixed size window across the window object in a scene. In which direction the fixed sized window moves, the object appears to move in the opposite direction as shown in fig:

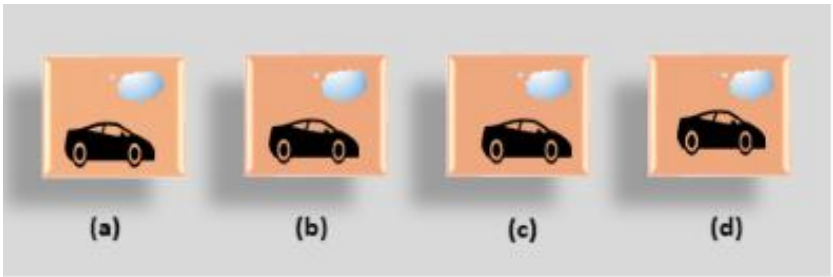


Fig: Panning

If the window moves in a backward direction, then the object appear to move in the forward direction and the window moves in forward direction then the object appear to move in a backward direction.

5. **Zooming:** In zooming, the window is fixed an object and change its size, the object also appear to change in size. When the window is made smaller about a fixed center, the object comes inside the window appear more enlarged. This feature is known as Zooming In.

   When we increase the size of the window about the fixed center, the object comes inside the window appear small. This feature is known as Zooming Out.

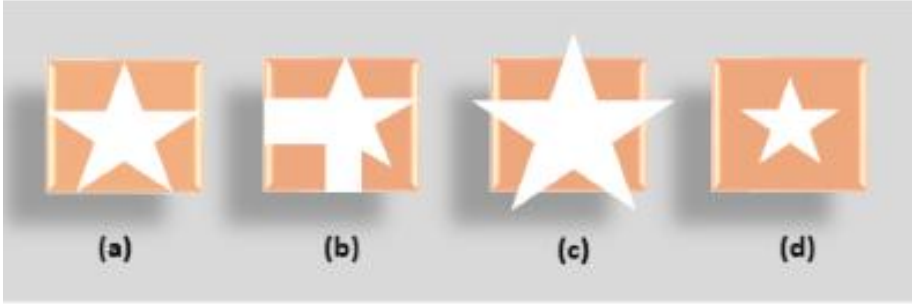


Fig: Zooming in & Zooming Out

6. **Fractals:** Fractal Function is used to generate a complex picture by using Iteration. Iteration means the repetition of a single formula again & again with slightly different value based on the previous iteration result. These results are displayed on the screen in the form of the display picture.

Examples:

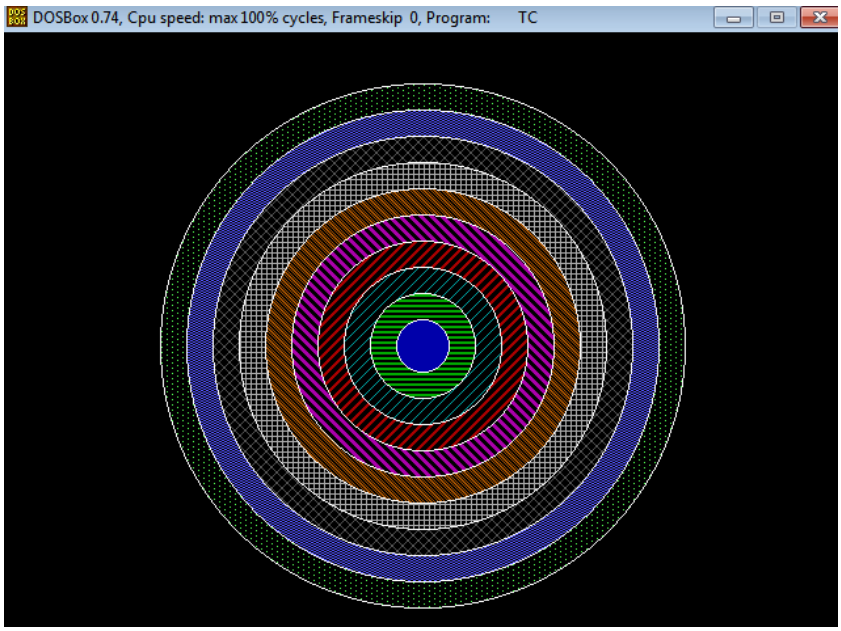
Aim: Write a Program to draw animation using increasing circles filled with different colors and patterns.

Code:

```
1.  #include<graphics.h>
2.  #include<conio.h>
3.  void main()
4.  {
5.      intgd=DETECT, gm, i, x, y;
6.      initgraph(&gd, &gm, "C:\\TC\\BGI");
7.      x=getmaxx()/3;
8.      y=getmaxx()/3;
9.      setbkcolor(WHITE);
10.     setcolor(BLUE);
```

```
11.     for(i=1;i<=8;i++)
12.         {
13.        setfillstyle(i,i);
14.        delay(20);
15.        circle(x, y, i*20);
16.        floodfill(x-2+i*20,y,BLUE);
17.     }
18.     getch();
19.     closegraph();
20.  }
```

Output:



12.2 SUMMARY

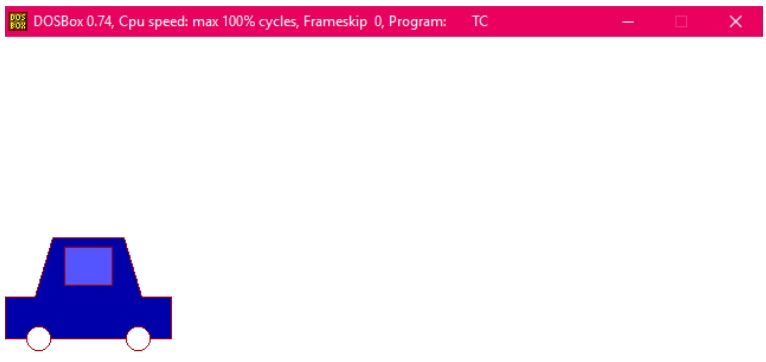
Aim: Write a Program to make a moving colored car using inbuilt functions.

Code:

```
1.  #include<graphics.h>
2.  #include<conio.h>
3.  int main()
4.  {
5.     intgd=DETECT,gm, i, maxx, cy;
6.     initgraph(&gd, &gm, "C:\\TC\\BGI");
7.     setbkcolor(WHITE);
8.     setcolor(RED);
```

```
9.      maxx = getmaxx();
10.     cy = getmaxy()/2;
11.     for(i=0;i<maxx-140;i++)
12.        {
13.        cleardevice();
14.        line(0+i,cy-20, 0+i, cy+15);
15.        line(0+i, cy-20, 25+i, cy-20);
16.        line(25+i, cy-20, 40+i, cy-70);
17.        line(40+i, cy-70, 100+i, cy-70);
18.        line(100+i, cy-70, 115+i, cy-20);
19.        line(115+i, cy-20, 140+i, cy-20);
20.        line(0+i, cy+15, 18+i, cy+15);
21.        circle(28+i, cy+15, 10);
22.        line(38+i, cy+15, 102+i, cy+15);
23.        circle(112+i, cy+15,10);
24.        line(122+i, cy+15 ,140+i,cy+15);
25.        line(140+i, cy+15, 140+i, cy-20);
26.        rectangle(50+i, cy-62, 90+i, cy-30);
27.        setfillstyle(1,BLUE);
28.        floodfill(5+i, cy-15, RED);
29.        setfillstyle(1, LIGHTBLUE);
30.        floodfill(52+i, cy-60, RED);
31.        delay(10);
32.         }
33.     getch();
34.     closegraph();
35.     return 0;
36.  }
```

Output:



Aim: C program for bouncing ball graphics animation

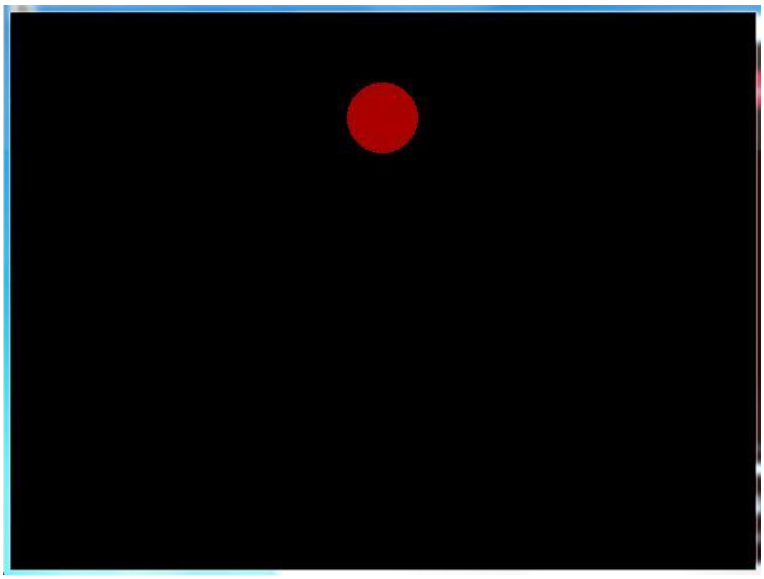
In this program, we first draw a red color ball on screen having center at (x, y) and then erases it using cleardevice function. We again draw this ball at center (x, y + 5), or (x, y - 5) depending upon whether ball is moving down or up. This will look like a bouncing ball. We will repeat above steps until user press any key on keyboard.

Code:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <dos.h>
int main()
int gd = DETECT, gm;
int i, x, y, flag=0;
initgraph(&gd, &gm, "C:\\TC\\BGI");
/* get mid positions in x and y-axis */
x = getmaxx()/2;
y = 30;
while (!kbhit()) {
if(y >= getmaxy()-30 || y <= 30)
flag = !flag;
/* draws the gray board */
setcolor(RED);
setfillstyle(SOLID_FILL, RED);
circle(x, y, 30);
floodfill(x, y, RED);
/* delay for 50 milli seconds */
delay(50);
/* clears screen */
cleardevice();
if(flag){
y = y + 5;
} else {
y = y - 5;
}
}
getch();
```

```
31. closegraph();
32. return 0;
33. }
```

34. Output:



12.3 REFERENCES

1] Introduction to Computer Graphics: A Practical Learning Approach By Fabio Ganovelli, Massimiliano Corsini, Sumanta Pattanaik, Marco Di Benedetto

2] Computer Graphics Principles and Practice in C: Principles & Practice in C Paperback – 1 January 2002
by Andries van Dam; F. Hughes John; James D. Foley; Steven K. Feiner (Author)

12.4 UNIT END QUESTIONS

- Write a program to make screen saver in that display different size circles filled with different colors and at random places.

*****

MODULE XII

13

IMAGE ENHANCEMENT TRANSFORMATION

Unit Structure

13.0 Objectives
13.1 Introduction
13.2 Summary
13.3 References
13.4 Unit End Exercises

13.0 OBJECTIVES

Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image.

13.1 INTRODUCTION

The following are commonly used intensity transformations:

1. Image Negatives (Linear)
2. Log Transformations
3. Power-Law (Gamma) Transformations

Spatial Domain Processes:

Image Negatives:

Mathematically, assume that an image goes from intensity levels 0 to (L-1). Generally, L = 256. Then, the negative transformation can be described by the expression s = L-1-r where r is the initial intensity level and s is the final intensity level of a pixel. This produces a photographic negative.

Log Transformations:

Mathematically, log transformations can be expressed as s = clog(1+r). Here, s is the output intensity, r>=0 is the input intensity of the pixel, and c is a scaling constant. c is given by 255/(log (1 + m)), where m is the maximum pixel value in the image. It is done to ensure that the final pixel value does not exceed (L-1), or 255.

Practically, log transformation maps a narrow range of low-intensity input values to a wide range of output values.

Consider the following input image.

Below is the code to apply log transformation to the image.

```
import cv2
import numpy as np
# Open the image.
img = cv2.imread('sample.jpg')


# Apply log transform.
c = 255/(np.log(1 + np.max(img)))
log_transformed = c * np.log(1 + img)


# Specify the data type.
log_transformed = np.array(log_transformed, dtype = np.uint8)


# Save the output.
cv2.imwrite('log_transformed.jpg', log_transformed)
```

Below is the log-transformed output.

13.2 SUMMARY

Power-Law (Gamma) Transformation:

"Gamma Correction", most of you might have heard this strange sounding thing. In this blog, we will see what it means and why does it matter to you?

The general form of Power law (Gamma) transformation function is

$$s = c*r\gamma$$

Where, 's' and 'r' are the output and input pixel values, respectively and 'c' and γ are the positive constants. Like log transformation, power law curves with $\gamma < 1$ map a narrow range of dark input values into a wider range of output values, with the opposite being true for higher input values. Similarly, for $\gamma > 1$, we get the opposite result which is shown in the figure below

This is also known as gamma correction, gamma encoding or gamma compression. Don't get confused.

The below curves are generated for r values normalized from 0 to 1. Then multiplied by the scaling constant c corresponding to the bit size used.

All the curves are scaled. Don't get confused (See below):

But the main question is why we need this transformation, what's the benefit of doing so?

To understand this, we first need to know how our eyes perceive light. The human perception of brightness follows an approximate power function(as shown below) according to Stevens' power law for brightness perception.

See from the above figure, if we change input from 0 to 10, the output changes from 0 to 50 (approx.) but changing input from 240 to 255 does not really change the output value. This means that we are more sensitive to changes in dark as compared to bright. You may have realized it yourself as well!

But our camera does not work like this. Unlike human perception, camera follows a linear relationship. This means that if light falling on the camera is increased by 2 times, the output will also increase 2 folds. The camera curve looks like this

So, where and what is the actual problem?:

The actual problem arises when we display the image.

You might be amazed to know that all display devices like your computer screen have Intensity to voltage response curve which is a power function with exponents(Gamma) varying from 1.8 to 2.5.

This means for any input signal(say from a camera), the output will be transformed by gamma (which is also known as Display Gamma) because of non-linear intensity to voltage relationship of the display screen. This results in images that are darker than intended.

To correct this, we apply gamma correction to the input signal(we know the intensity and voltage relationship we simply take the complement) which is known as Image Gamma. This gamma is automatically applied by the conversion algorithms like jpeg etc. thus the image looks normal to us.

This input cancels out the effects generated by the display and we see the image as it is. The whole procedure can be summed up as by the following figure

If images are not gamma-encoded, they allocate too many bits for the bright tones that humans cannot differentiate and too few bits for the dark tones. So, by gamma encoding, we remove this artifact.

Images which are not properly corrected can look either bleached out, or too dark.

Let's verify by code that γ <1 produces images that are brighter while γ >1 results in images that are darker than intended

Code:

```
1   import numpy as np
2   import cv2
3   # Load the image
4   img = cv2.imread('D:/downloads/forest.jpg')
5   # Apply Gamma=2.2 on the normalised image and then multiply by scaling constant (For 8 bit, c=255)
6   gamma_two_point_two = np.array(255*(img/255)**2.2,dtype='uint8')
7   # Similarly, Apply Gamma=0.4
8   gamma_point_four = np.array(255*(img/255)**0.4,dtype='uint8')
9   # Display the images in subplots
10  img3 = cv2.hconcat([gamma_two_point_two,gamma_point_four])
11  cv2.imshow('a2',img3)
12  cv2.waitKey(0)
13
```

Output:

Original Image

Gamma Encoded Images

Below is the Python code to apply gamma correction.

```
import cv2
import numpy as np

# Open the image.
img = cv2.imread('sample.jpg')

# Trying 4 gamma values.
for gamma in [0.1, 0.5, 1.2, 2.2]:

    # Apply gamma correction.
```

```
    gamma_corrected = np.array(255*(img / 255) ** gamma, dtype =
'uint8')

    # Save edited images.
    cv2.imwrite('gamma_transformed'+str(gamma)+'.jpg',
gamma_corrected)
```

Below are the gamma-corrected outputs for different values of gamma.

Gamma = 0.1:

Gamma = 0.5:

Gamma = 1.2:

Gamma = 2.2:

As can be observed from the outputs as well as the graph, gamma>1 (indicated by the curve corresponding to ‘nth power’ label on the graph), the intensity of pixels decreases i.e. the image becomes darker. On the other hand, gamma<1 (indicated by the curve corresponding to 'nth root' label on the graph), the intensity increases i.e. the image becomes lighter.

13.3 REFERENCES

1] Introduction to Computer Graphics: A Practical Learning Approach By Fabio Ganovelli, Massimiliano Corsini, Sumanta Pattanaik, Marco Di Benedetto

2] Computer Graphics Principles and Practice in C: Principles & Practice in C Paperback – 1 January 2002 by Andries van Dam; F. Hughes John; James D. Foley; Steven K. Feiner (Author)

13.4 UNIT END EXERCISE

- Write a python code to perform gamma transformation.

*****

14

IMAGE ENHANCEMENT TRANSFORMATION

Unit Structure

14.0 Objectives
14.1 Introduction
14.2 Summary
14.3 References
14.4 Unit End Exercises

14.0 OBJECTIVES

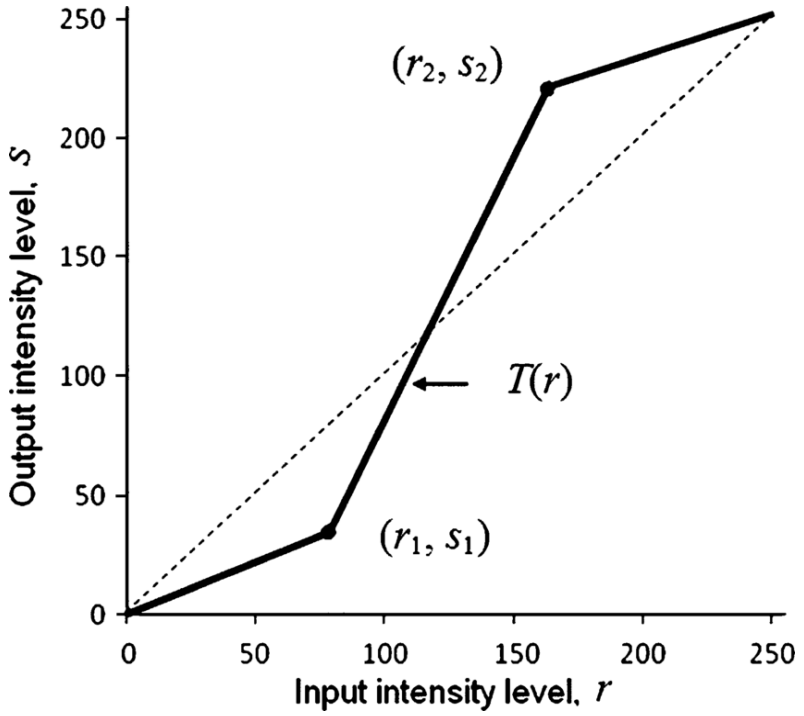
Piecewise-Linear Transformation Functions:

These functions, as the name suggests, are not entirely linear in nature. However, they are linear between certain x-intervals. One of the most commonly used piecewise-linear transformation functions is contrast stretching.

Contrast can be defined as:

Contrast = (I_max - I_min)/(I_max + I_min)

This process expands the range of intensity levels in an image so that it spans the full intensity of the camera/display. The figure below shows the graph corresponding to the contrast stretching.



With (r1, s1), (r2, s2) as parameters, the function stretches the intensity levels by essentially decreasing the intensity of the dark pixels and increasing the intensity of the light pixels. If r1 = s1 = 0 and r2 = s2 = L-1, the function becomes a straight dotted line in the graph (which gives no effect). The function is monotonically increasing so that the order of intensity levels between pixels is preserved.

14.1 INTRODUCTION

Below is the Python code to perform contrast stretching.

```
import cv2
import numpy as np

 # Function to map each intensity level to output intensity level.
def pixelVal(pix, r1, s1, r2, s2):
   if (0 <= pix and pix <= r1):
      return (s1 / r1)*pix
   elif (r1 < pix and pix <= r2):
      return ((s2 - s1)/(r2 - r1)) * (pix - r1) + s1
   else:
      return ((255 - s2)/(255 - r2)) * (pix - r2) + s2


# Open the image.
img = cv2.imread('sample.jpg')


# Define parameters.
r1 = 70
s1 = 0
r2 = 140
s2 = 255

# Vectorize the function to apply it to each value in the Numpy array.
pixelVal_vec = np.vectorize(pixelVal)

# Apply contrast stretching.
contrast_stretched = pixelVal_vec(img, r1, s1, r2, s2)


# Save edited image.
cv2.imwrite('contrast_stretch.jpg', contrast_stretched)
```

Output:

Piece-wise Linear Transformation:

Piece-wise Linear Transformation is type of gray level transformation that is used for image enhancement. It is a spatial domain method. It is used for manipulation of an image so that the result is more suitable than the original for a specific application.

Some commonly used piece-wise linear transformations are:

Contrast Stretching:

Low contrast image occurs often due to improper illumination or non-linearly or small dynamic range of an imaging sensor. It increases the dynamic range of grey levels in the image.

Contrast Stretching Transform is given by:

S = l.r, 0 <= r < a

S = m.(r-a) + v, a <= r < b

S = n.(r-b) + w, b <= r < L-1

where l, m, n are slopes

The Formula for Contrast stretch or Image Normalization:

Io = (Ii-Mini)*(((Maxo-Mino)/(Maxi-Mini))+Mino)

Io	-	Output pixel value
Ii	-	Input pixel value
Mini	-	Minimum pixel value in the input image
Maxi	-	Maximum pixel value in the input image

Mino - Minimum pixel value in the output image

Maxo - Maximum pixel value in the output image

Contrast stretch using Python and Pillow:

- The Python Image Processing Library supports point image operations through method point()of the Image module.
- The point()method takes a function as a parameter. The function passed in accepts one argument and typically this is the pixel value that is to be transformed.
- In case of contrast stretching of an image, the formula for contrast stretching can be implemented inside a function, which takes the pixel value as a parameter and returns the modified intensity of the pixel.

Code:

```
# Example Python Program for contrast stretching
from PIL import Image

# Method to process the red band of the image
def normalizeRed(intensity):
    iI      = intensity

    minI    = 86
    maxI    = 230

    minO    = 0
    maxO    = 255

    iO      = (iI-minI)*(((maxO-minO)/(maxI-minI))+minO)
    return iO

# Method to process the green band of the image
def normalizeGreen(intensity):
    iI      = intensity

    minI    = 90
    maxI    = 225

    minO    = 0
    maxO    = 255

    iO      = (iI-minI)*(((maxO-minO)/(maxI-minI))+minO)
    return iO
```

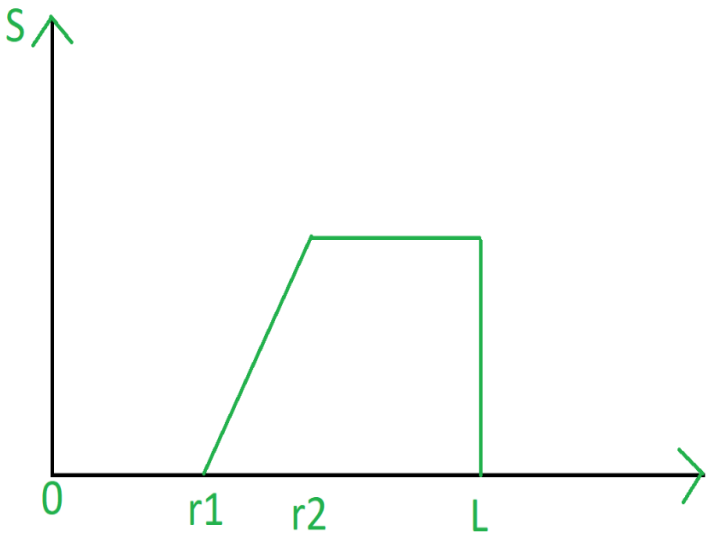
```
# Method to process the blue band of the image
def normalizeBlue(intensity):
    iI      = intensity

    minI    = 100
    maxI    = 210

    minO    = 0
    maxO    = 255

    iO      = (iI-minI)*(((maxO-minO)/(maxI-minI))+minO)
    return iO


# Create an image object
imageObject     = Image.open("./glare4.jpg")

# Split the red, green and blue bands from the Image
multiBands      = imageObject.split()

# Apply point operations that does contrast stretching on each color band
normalizedRedBand       = multiBands[0].point(normalizeRed)
normalizedGreenBand     = multiBands[1].point(normalizeGreen)
normalizedBlueBand      = multiBands[2].point(normalizeBlue)

# Create a new image from the contrast stretched red, green and blue
brands
normalizedImage = Image.merge("RGB", (normalizedRedBand,
normalizedGreenBand, normalizedBlueBand))


# Display the image before contrast stretching
imageObject.show()

# Display the image after contrast stretching
normalizedImage.show()
```

Input Image for Contrast Stretching Operation:

14.2 SUMMARY

Clipping:

A special case of contrast stretching is clipping where l=n=0. It is used for noise reduction when the input signal is known. It puts all grey levels below r1 to black(0) and above r2 to white(1).



numpy.clip() function is used to Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Syntax: numpy.clip(a, a_min, a_max, out=None)

Parameters:

a: Array containing elements to clip.

a_min: Minimum value.

- If None, clipping is not performed on lower interval edge. Not more than one of a_min and a_max may be None.

a_max: Maximum value.

- If None, clipping is not performed on upper interval edge. Not more than one of a_min and a_max may be None.
- If a_min or a_max are array_like, then the three arrays will be broadcasted to match their shapes.

out: Results will be placed in this array. It may be the input array for in-place clipping. out must be of the right shape to hold the output. Its type is preserved.

Return: clipped_array

Code:

```
# Python3 code demonstrate clip() function


# importing the numpy
import numpy as np


in_array = [1, 2, 3, 4, 5, 6, 7, 8 ]
print ("Input array : ", in_array)


out_array = np.clip(in_array, a_min = 2, a_max = 6)
print ("Output array : ", out_array)
```

Output:

```
Input array :  [1, 2, 3, 4, 5, 6, 7, 8]
Output array :  [2 2 3 4 5 6 6 6]
```

Code:

```
# Python3 code demonstrate clip() function


# importing the numpy
import numpy as np
```

```
in_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print ("Input array : ", in_array)

out_array = np.clip(in_array, a_min =[3, 4, 1, 1, 1, 4, 4, 4, 4, 4],
                                        a_max = 9)
print ("Output array : ", out_array)
```

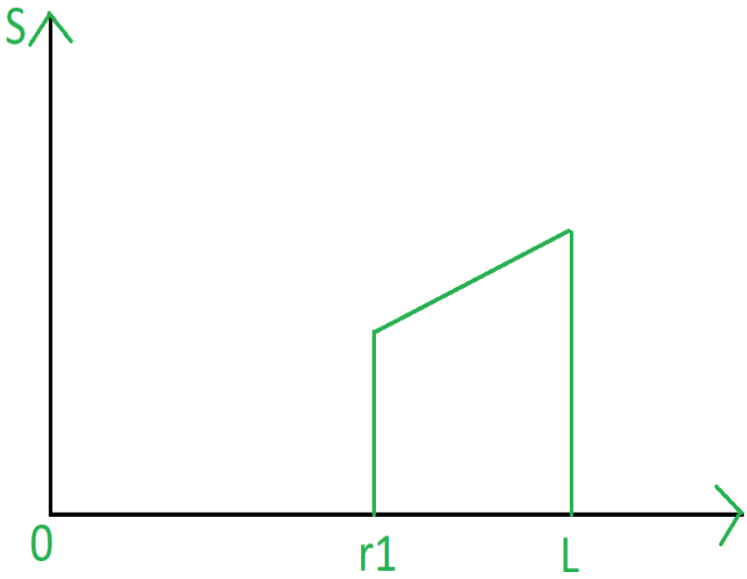
Output:

Input array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Output array: [3 4 3 4 5 6 7 8 9 9]

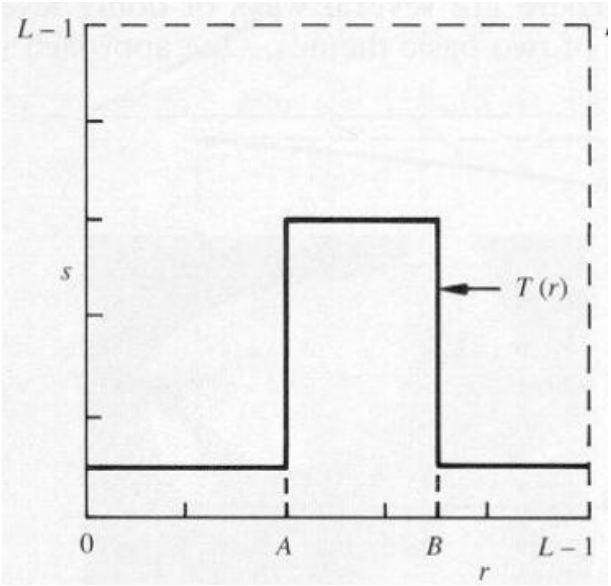
Thresholding:

Another special case of contrast stretching is thresholding where l=m=t. It is also used for noise reduction. It preserves the grey levels beyond r1.



Grey level slicing:

Focuses on enhancing a specific range of grey level in an image. The intervals are pre-defined and pixels falling in that range are manipulated. This can be used to brighten the desired range of grey level while preserving the background quality in the range.



The code below implements the piece-wise linear transformation by gray-level slicing.

For this, we consider a low contrast gray-scale image and increase gray-levels to maximum in a specified range. Another use-case is when we do not restore values above the range to default.

Before Gray-level slicing

After Gray-Level Slicing

Highlighting a specific range of grey level in an image.

Case-I:

- To display a high value for all grey levels in the range of interest.
- To display a low value for all grey levels.

Case-II:

- Brighten the desired range of grey level.
- Preserve the background quality in the range.

Bit Extraction:

An 8-bit image can be represented in the form of bit plane. Each plane represents one bit of all pixel values. Bit plane 7 contains the most significant bit (MSB) and bit plane 0 contains least significant bit (LSB). The 4 MSB planes contains most of visually significant data. This technique is useful for image compression and steganography.

14.3 REFERENCE

1] Introduction to Computer Graphics: A Practical Learning Approach By Fabio Ganovelli, Massimiliano Corsini, Sumanta Pattanaik, Marco Di Benedetto

2] Computer Graphics Principles and Practice in C: Principles & Practice in C Paperback – 1 January 2002 by Andries van Dam; F. Hughes John; James D. Foley; Steven K. Feiner (Author)

14.4 UNIT END EXERCISE

- Write a code to perform contrast stretching.

*****

15

IMAGE ENHANCEMENT TRANSFORMATION

Unit Structure

15.0 Objectives
15.1 Introduction
15.2 Summary
15.3 References
15.4 Unit End Exercises

15.0 OBJECTIVES

Introduction to Histogram Equalization for Digital Image Enhancement:

Histogram equalization improves image contrast

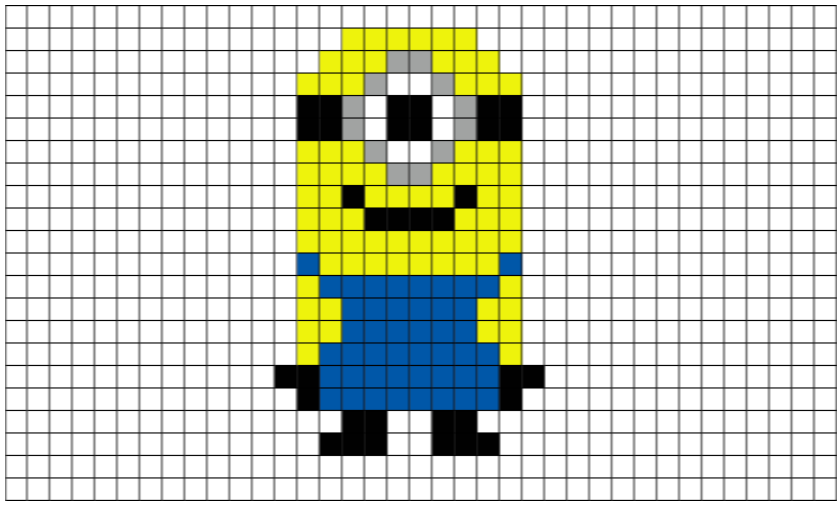
Histogram equalization is a simple technique to enhance a digital image. It is a standard tool in digital image processing and computer vision applications. It is especially effective in improving the visual quality of grayscale images. For example, histogram equalization has been applied extensively in medical imaging to improve the contrast of X-ray and MRI images. Such improvement enables more accurate medical diagnosis.

15.1 INTRODUCTION

Digital image:

Let's say you wish to draw a yellow, one-eyed creature dressed in blue overall called minion. You will draw using only coloured dots. You started drawing by making tiny dots with coloured markers on a piece of

paper. The coloured dots are arranged to form the image of a minion which you wish to visualize. The resulting image probably looks something like the picture below. This is analogous to how images are rendered by the computer screen.



This image of a minion is composed by tiny, coloured dots analogous to pixels

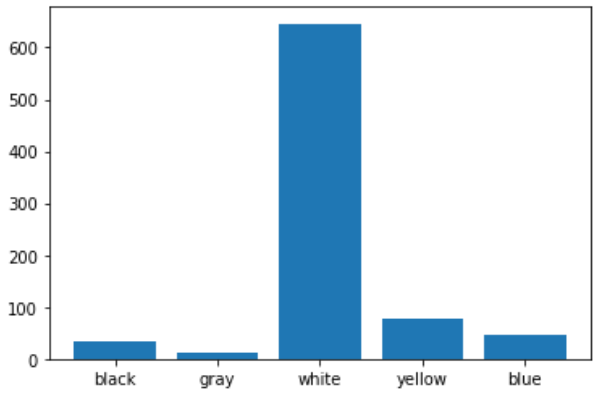
Digital images are composed of discrete elements known as pixels. Each pixel is represented by intensity value(s) and occupies a unique position in the 2D image plane. In grayscale images, each pixel is represented by a single intensity value ranging from 0 to 255. Pixels with minimum intensity value appear as black pixels. As the intensity value increases, the pixel gets lighter in various shades of grey. Pixels having maximum intensity value appear as white pixels. Colour images are slightly more complex. Each pixel is represented by three intensity values which represents the intensity of the colour components: red, green and blue. Varying the proportion and intensity of these primary colours enables the representation of various colours in the pixels.

What is a histogram?:

A histogram represents the frequency distribution of data. It is usually visualized as a bar plot. I will use the minion image above to illustrate these points. The dimension of the image is 22x37 pixels, giving a total of 814 pixels. Let's say we live in a world where there are only five possible colours: black, gray, white, yellow and blue. To obtain the histogram, we can count the number of pixels for each possible colour:

- black — 34 pixels
- gray — 12 pixels
- white — 646 pixels
- yellow — 76 pixels
- blue — 46 pixels

As a common terminology, each possible value is referred to as a bin, and the count is referred to as frequency. Hence, we just derived the colour histogram of the image with five bins. The bar plot is shown in the figure below. Here, the x-axis is the bins and the y-axis is the frequencies. In histogram equalization, we are interested in the intensity histogram of the image. That means for each possible pixel intensity value (0 to 255), we count the number of pixels having the corresponding value.



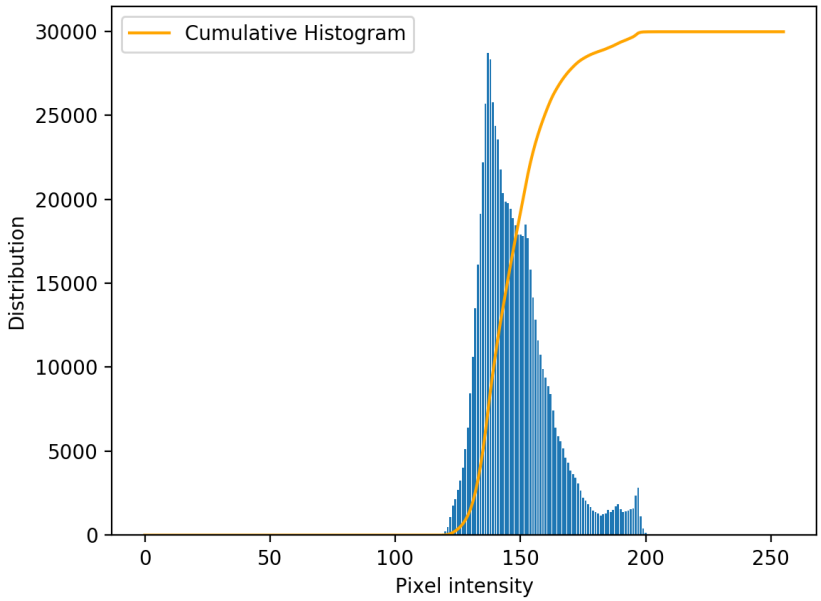
Histogram is typically visualized as bar plot like this

Histograms of low contrast images:

To understand the role of histogram equalization, we will look at two cases of low contrast images. For each image, the histogram (blue plot) and cumulative histogram (orange plot) are shown. Cumulative histogram is simply a running sum of the histogram frequency from the first to the last bin. The cumulative histogram is scaled down, so that it can be plotted in the same scale as the histogram.

Case 1: Image appears too bright:

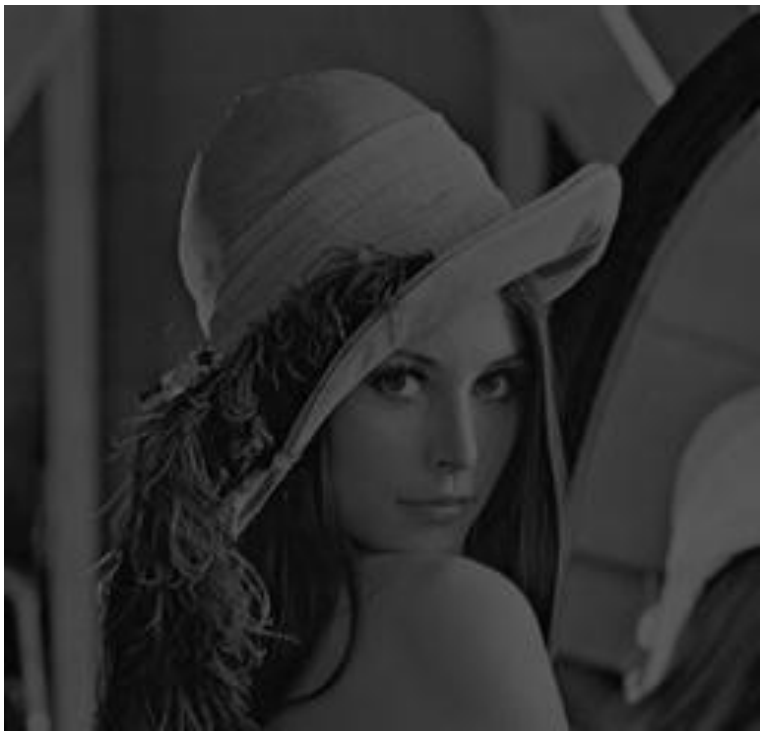
The image below looks bright yet faded. The explanation can be found in the histogram. The pixel intensities are concentrated in the upper region of the range, approximately between 125 and 200. In this region, the pixels are lightly shaded hence the bright appearance. Due to the narrow intensity range, the pixels are very similar in shades, resulting in faded appearance. In this narrow region, the cumulative histogram is increasing with a steep slope while being flat elsewhere.

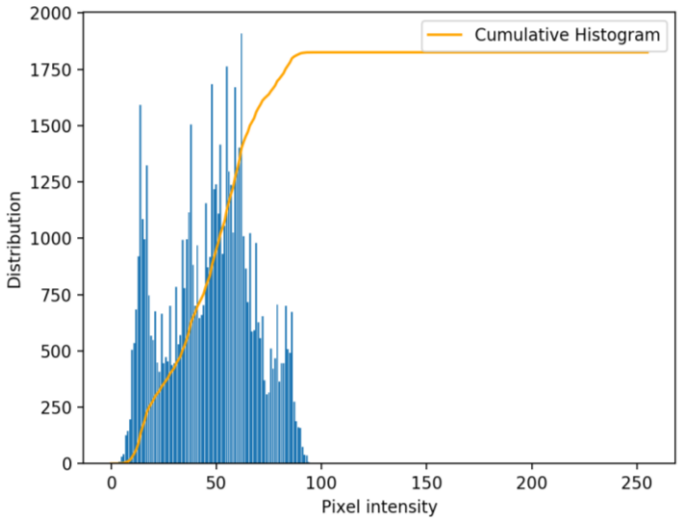


This image appears faded. Its pixel intensities are concentrated around the high intensity region of between 125 and 200.

Case 2: Image appears too dark:

The image below looks rather dark and gloomy. Looking at the histogram, the intensities are concentrated in the lower region of the range, approximately between 5 to 95. In this region, the pixels are darker in shades. It can also be observed that the cumulative histogram is increasing sharply in this region and flat elsewhere.





This image appears dark. Its pixel intensities are concentrated around the lower intensity region between 5 and 95.

In both cases, histogram analysis revealed that in images with low-contrast:

- the pixel intensities are concentrated in a narrow region resulting in pixels with similar shades, giving the image a faded appearance, and
- the cumulative histogram increases with a steep slope within a narrow region and flat elsewhere.

Histogram equalization:

The contrast of an image is enhanced when various shades in the image becomes more distinct. We can do so by darkening the shades of the darker pixels and vice versa. This is equivalent to widening the range of pixel intensities. To have a good contrast, the following histogram characteristics are desirable:

- the pixel intensities are uniformly distributed across the full range of values (each intensity value is equally probable), and
- the cumulative histogram is increasing linearly across the full intensity range.

Histogram equalization modifies the distribution of pixel intensities to achieve these characteristics.

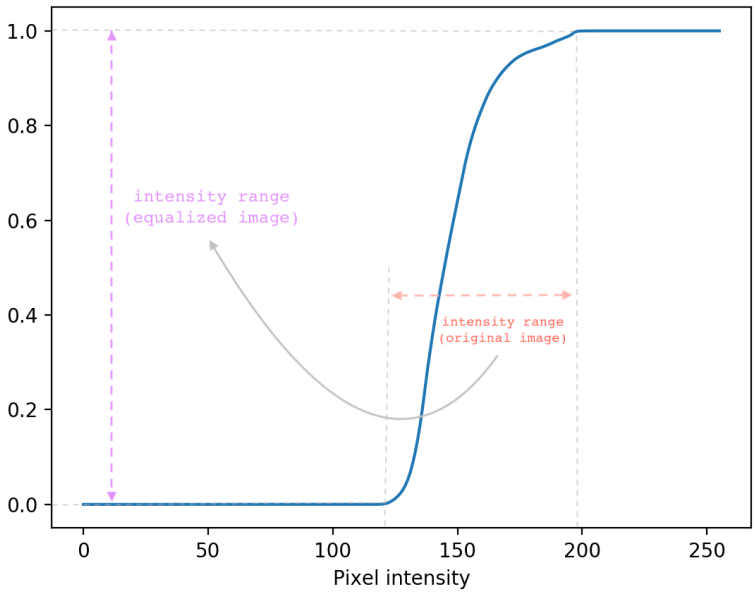
15.2 SUMMARY

The core algorithm:

Step 1: Calculate normalized cumulative histogram

First, we calculate the normalized histogram of the image. Normalization is performed by dividing the frequency of each bin by the total number of pixels in the image. As a result, the maximum value of the cumulative

histogram is 1. The following figure shows the normalized cumulative histogram of the same low contrast image presented as Case 1 in Section 3.



Normalized cumulative histogram is used as the transformation function in histogram equalization. It maps the narrow pixel intensity range to the full range.

Step 2: Derive intensity-mapping lookup table:

Next, we derive a lookup table which maps the pixel intensities to achieve an equalized histogram characteristics. Recall that the equalized cumulative histogram is linearly increasing across the full range of intensity. For each discrete intensity level i, the mapped pixel value is calculated from the normalized cumulative histogram according to:

mapped_pixel_value(i) = (L-1)*normalized_cumulative_histogram(i)

where L = 256 for a typical 8-bit unsigned integer representation of pixel intensity.

As an intuition into how the mapping works, let's refer to the normalized cumulative histogram shown in the figure above. The minimum pixel intensity value of 125 is transformed to 0.0. The maximum pixel intensity value of 200 is transformed to 1.0. All the values in between are mapped accordingly between these two values. Once multiplied by the maximum possible intensity value (255), the resulting pixel intensities are now distributed across the full intensity range.

Step 3: Transform pixel intensity of the original image with the lookup table:

Once the lookup table is derived, intensity of all pixels in the image are mapped to the new values. The result is an equalized image.

Python implementation:

Histogram equalization is available as standard operation in various image processing libraries, such as openCV and Pillow. However, we will implement this operation from scratch. We will need two Python libraries: NumPy for numerical calculation and Pillow for image I/O. The easiest way to install these libraries is via Python package installer pip . Enter the following commands on your terminal and you are set!

pip install numpy

pip install pillow

The full code is show below, followed by detailed explanation of the equalization process. To equalize your own image, simply edit the img_filename and save_filename accordingly.

Image I/O:

To read from and write to image files, we will use Pillow library. It reads image files as Imageobject. These objects can be converted easily to NumPy array, and viceversa. The required I/O operations are coded as follows. For simplicity, let the image filename be input_image.jpg residing in the same directory as as the Python script.

import numpy as np

from PIL import Imageimg_filename = 'input_image.jpg'

save_filename = 'output_image.jpg'#load file as pillow Image

img = Image.open(img_filename)# convert to grayscale

imgray = img.convert(mode='L')#convert to NumPy array

img_array = np.asarray(imgray)

#PERFORM HISTOGRAM EQUALIZATION AND ASSIGN OUTPUT TO eq_img_array

#convert NumPy array to pillow Image and write to file

eq_img = Image.fromarray(eq_img_array, mode='L')

eq_img.save(save_filename)

Histogram Equalization:

The main algorithm can be implemented in only several lines of code. In this example, the intensity-mapping lookup table is implemented as 1D list where the index represents the original image pixel intensity. The element at each index is the corresponding transformed value. Finally, there are various ways to perform the pixel intensity mapping. I used list comprehension by flattening and reshaping the 2D image array before and after the mapping.

"""

STEP 1: Normalized cumulative histogram

"""#flatten image array and calculate histogram via binning

```
histogram_array = np.bincount(img_array.flatten(), minlength=256)#normalize
num_pixels = np.sum(histogram_array)
histogram_array = histogram_array/num_pixels#cumulative histogram
chistogram_array = np.cumsum(histogram_array)
"""
```

STEP 2: Pixel mapping lookup table

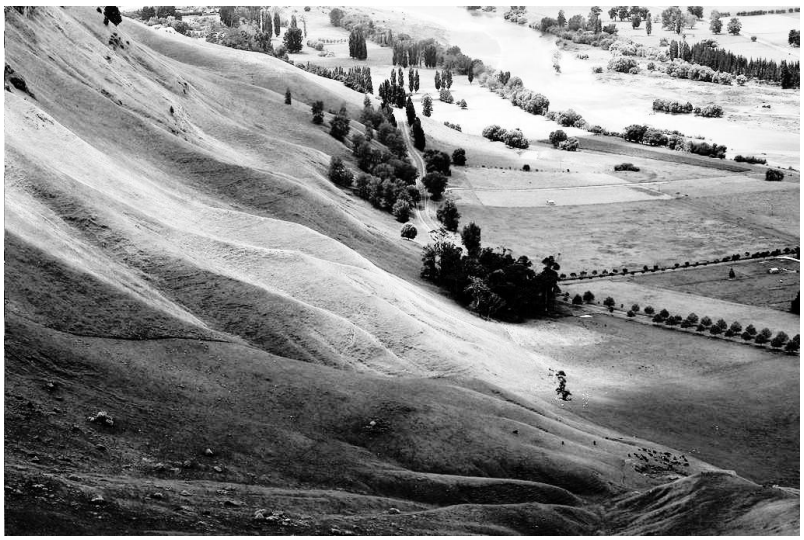
```
"""
transform_map = np.floor(255 * chistogram_array).astype(np.uint8)
"""
```

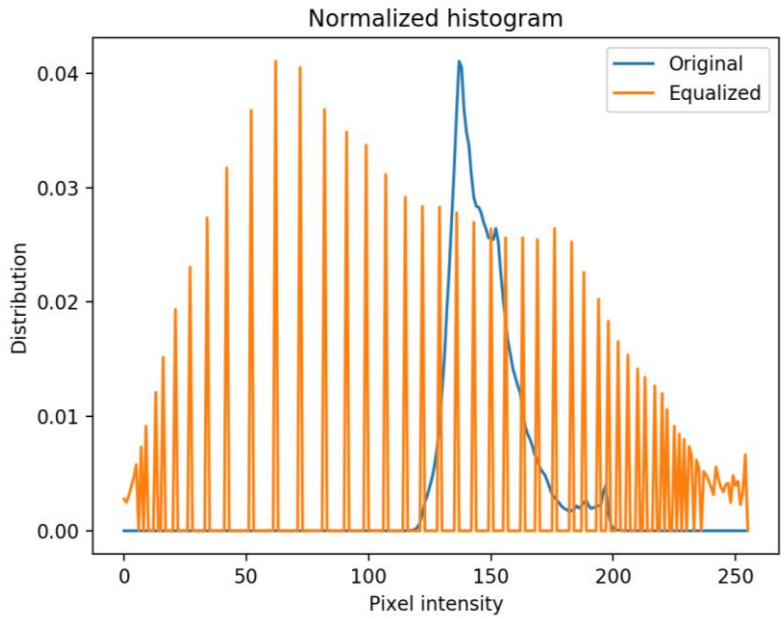
STEP 3: Transformation

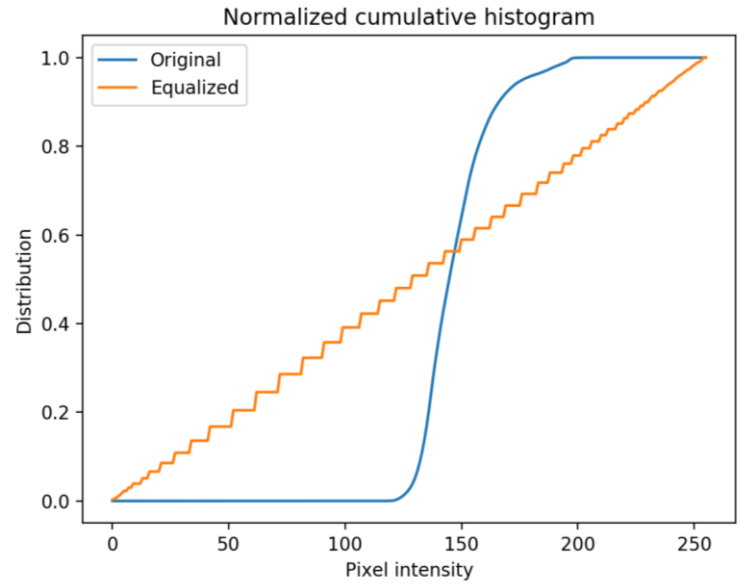
```
"""# flatten image array into 1D list
img_list = list(img_array.flatten())# transform pixel values to equalize
eq_img_list = [transform_map[p] for p in img_list]# reshape and write back into img_array
eq_img_array = np.reshape(np.asarray(eq_img_list), img_array.shape)
```

Let's look at the histogram equalization output for the two images presented in Section 3. For each result, the upper two images show the original and equalized images. Improvement in contrast is clearly observed. The lower two images show the histogram and cumulative histogram, comparing original and equalized images. After histogram equalization, the pixel intensities are distributed across the whole intensity range. The cumulative histograms are increasing linearly as expected, while exhibiting staircase pattern. This is expected as the pixel intensities of the original image were stretched into a wider range. This creates gaps of bins with zero frequency between adjacent non-zero bins, appearing as flat line in the cumulative histogram.

Case 1: Unequalized_Hawkes_Bay_NZ.jpg:



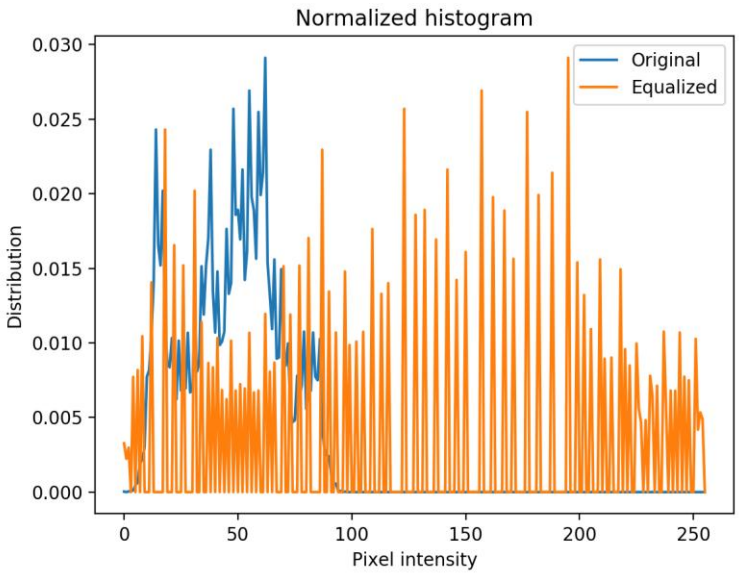

Normalized histogram
Original
Equalized
Distribution
0.04
0.03
0.02
0.01
0.00
0
50
100
150
200
250
Pixel intensity

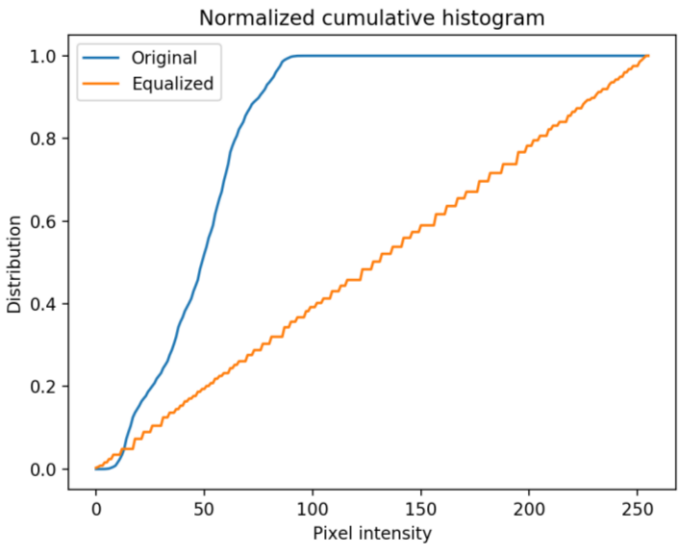

Normalized cumulative histogram
Original
Equalized
Distribution
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
Pixel intensity

Case 1: Image, histograms and cumulative histograms before and after equalization.

Case 2: lena_dark.png

Case 2: lena_dark.png





Case 1: Image, histograms and cumulative histograms before and after equalization.

15.3 REFERENCE

1] Introduction to Computer Graphics: A Practical Learning Approach By Fabio Ganovelli, Massimiliano Corsini, Sumanta Pattanaik, Marco Di Benedetto

2] Computer Graphics Principles and Practice in C: Principles & Practice in C Paperback – 1 January 2002 by Andries van Dam; F. Hughes John; James D. Foley; Steven K. Feiner (Author)

15.4 UNIT END EXERCISE

- Write a python code to perform Histogram equilization.

*****