SINCE 1999
SYNERGY
A Modern Day Gurukul

SYNERGY INSTITUTE OF ENGINEERING & TECHNOLOGY
Department of Computer Science & Engineering
Academic Session 2023-24
LECTURE NOTE

Name of Faculty : Mrs. Ipsita Panda
Name of Subject : Computer Graphics
Subject Code : RCS5D006
Subject Credit : 3
Semester : V
Year : 3rd
Course : B. TECH
Branch : COMPUTER SCIENCE & ENGINEERING

Admission Batch : 2021-25

Computer Graphics

Computer graphics remains one of the most existing and rapidly growing computer fields. Computer graphics may be defined as a pictorial representation or graphical representation of objects in a computer.

Application of Computer Graphics

Computer-Aided Design

Computer-Aided Design for engineering and architectural systems etc. Objects maybe displayed in a wireframe outline form. Multi-window environment is also favored for producing various zooming scales and views. Animations are useful for testing performance.

Presentation Graphics

To produce illustrations which summarize various kinds of data. Except 2D, 3D graphics are good tools for reporting more complex data.

Computer Art

Painting packages are available. With cordless, pressure-sensitive stylus, artists can produce electronic paintings which simulate different brush strokes, brush widths, and colors. Photorealistic techniques, morphing and animations are very useful in commercial art. For films, 24 frames per second are required. For video monitor, 30 frames per second are required.

Entertainment

Motion pictures, Music videos, and TV shows, Computer games

Education and Training

Training with computer-generated models of specialized systems such as the training of ship captains and aircraft pilots.

Visualization

For analyzing scientific, engineering, medical and business data or behavior. Converting data to visual form can help to understand mass volume of data very efficiently.

Image Processing

Image processing is to apply techniques to modify or interpret existing pictures. It is widely used in medical applications.

Graphical User Interface

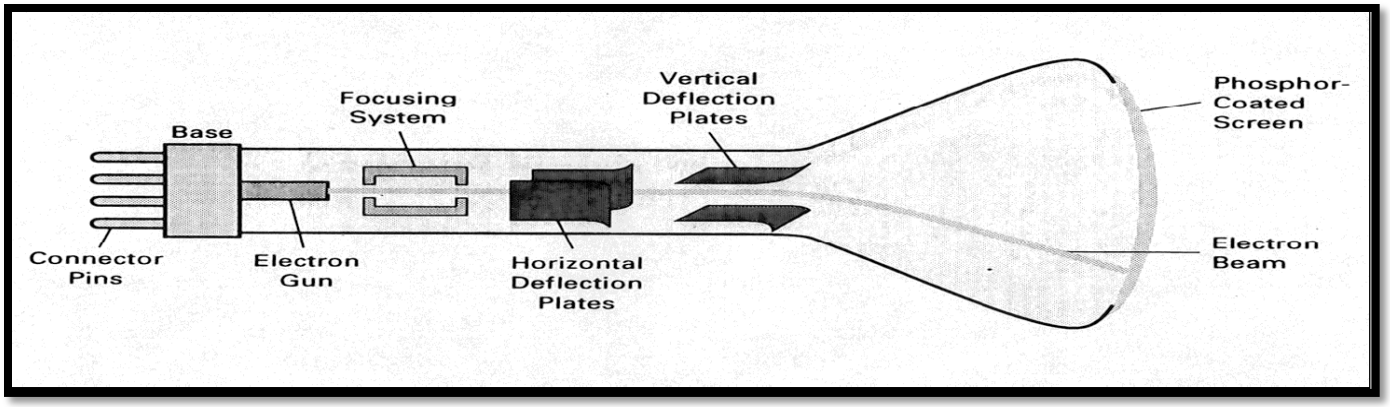
Multiple window, icons, menus allow a computer setup to be utilized more efficiently.

Overview of Graphics Systems

Cathode-Ray Tubes (CRT) –

Still the most common video display device presently electrostatic deflection of the electron beam in a CRT. An electron gun emits a beam of electrons, which passes through focusing and deflection systems and hits on the phosphor-coated screen. A beam of electrons (cathode rays), emitted by an electron gun, passes through focusing and deflection systems that direct the beam towards specified position on the phosphor-coated screen. The phosphor then emits a small spot of light at each position contacted by the electron beam. Because the light emitted by the phosphor fades very rapidly, some method is needed for maintaining the screen picture. One way to keep the phosphor glowing is to redraw the picture repeatedly by quickly directing the electron beam back over the same points. This type of display is called a **refresh CRT.**

The primary components of an electron gun in a CRT are the heated metal cathode and a control grid. Heat is supplied to the cathode by directing a current through a coil of wire, called the filament, inside the cylindrical cathode structure. This causes electrons to be "boiled off" the hot cathode surface. In the vacuum inside the CRT envelope, negatively charged electrons are then accelerated toward the phosphor coating by a high positive voltage. The accelerating voltage can be generated with a positively charged metal coating on the in side of the CRT envelope near the phosphor screen, or an accelerating anode can be used, a in fig below . Sometimes the electron gun is built to contain the accelerating anode and focusing system within the same unit. Spots of light are produced on the screen by the transfer of the CRT beam energy to the phosphor. When the electrons in the beam collide with the phosphor coating, they are stopped and there are stopped and their kinetic energy is absorbed by the phosphor. Part of the beam energy s converted by friction into heat energy, and the remainder causes electron in the phosphor atoms to move up to higher quantum-energy levels. After a short time, the "excited" phosphor electrons begin dropping back to their stable ground state, giving up their extra energy as small quantum of light energy. What we see on the screen is the combined effect of all the electrons light emissions: a glowing spot that quickly fades after all the excited phosphor electrons have returned to their ground energy level. The frequency (or color) of the light emitted by the phosphor is proportional to the energy difference between the excited quantum state and the ground state.



Different kinds of phosphor are available for use in a CRT. Besides color, a major difference between phosphors is their persistence: how long they continue to emit light (that is, have excited electrons returning to the ground state) after the CRT beam is removed. Persistence is defined as the time it takes the emitted light from the screen to decay to one-tenth of its original intensity. Lower-persistence phosphors require higher refresh rates to maintain a picture on the screen without flicker. A phosphor with low persistence is useful for animation; a high-persistence phosphor is useful for displaying highly complex, static pictures. Although some phosphor have a persistence greater than 1 second, graphics monitor are usually constructed with a persistence in the range from 10 to 60 microseconds

Raster-scan technique

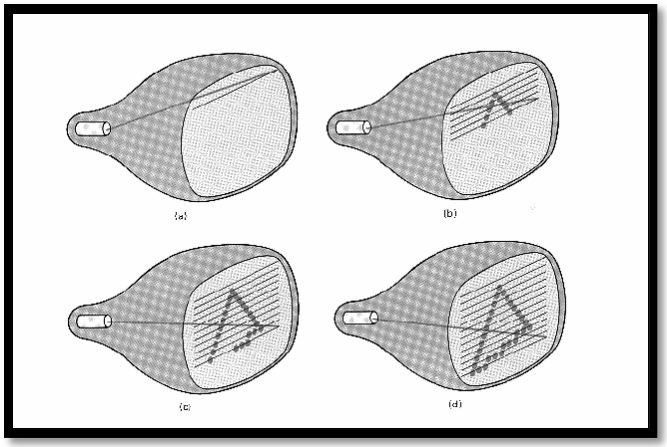
In a raster- scan system, the electron beam is swept across the screen, one row at a time from top to bottom. As the electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots. Picture definition is stored in memory area called the **refresh buffer** or **frame buffer**. This memory area holds the set of intensity values for all the screen points. Stored intensity values are then retrieved from the refresh buffer and "painted" on the screen one row (**scan line**) at a time (Fig. 4). Each screen point is referred to as a **pixel** or **pel** (shortened forms of **picture element**).

DISPLAY DEVICES

The light emitted by phosphor fades very rapidly, so it needs to redraw the picture repeatedly. There are 2 kinds of redrawing mechanisms:

– Raster-scan displays (refresh)
– Random-scan displays (vector refresh)

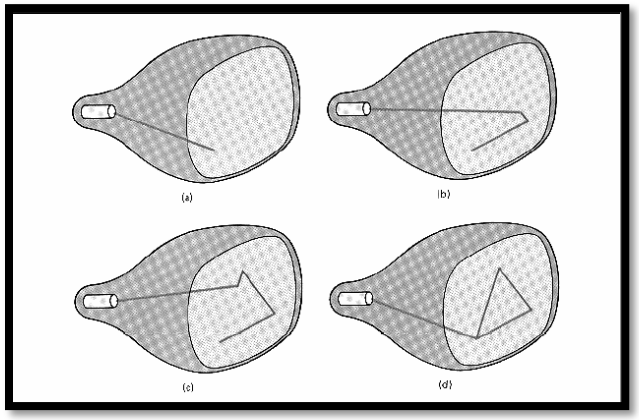
Raster-Scan and Random-Scan.



The electron beam is swept across the screen one row at a time from top to bottom. As it moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots. This scanning process is called refreshing. Each complete scanning of a screen is normally called a **frame**. The refreshing rate, called the **frame rate**, is normally 60 to 80 frames per second, or described as 60 Hz to 80 Hz. Picture definition is stored in a memory area called the **frame buffer**. This frame buffer stores the intensity values for all the screen points. Each screen point is called a **pixel** (picture element).
On black and white systems, the frame buffer storing the values of the pixels is called a **bitmap**. Each entry in the bitmap is a 1-bit data which determine the on (1) and off (0) of the intensity of the pixel. On color systems, the frame buffer storing the values of the pixels is called a **pixmap** (Though nowadays many graphics libraries name it as bitmap too). Each entry in the pixmap occupies a number of bits to represent the color of the pixel. For a true color display, the number of bits for each entry is 24 (8 bits per red/green/blue channel, each channel 28=256

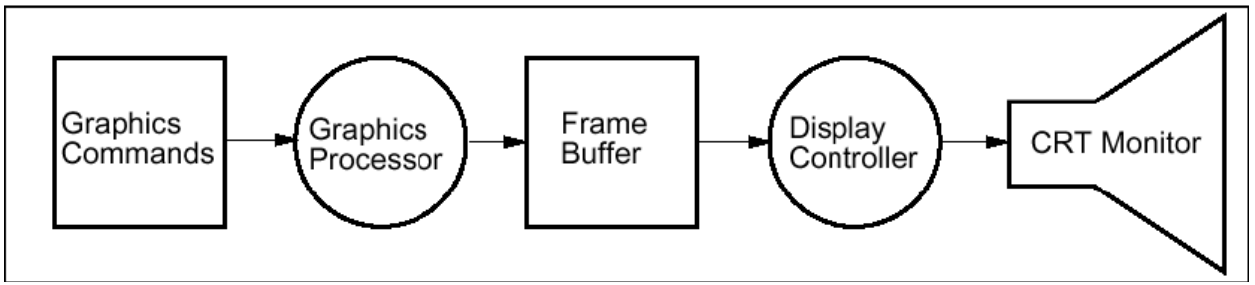
levels of intensity value, ie. 256 voltage settings for each of the red/green/blue electron guns).

Random-Scan (Vector Display)



The CRT's electron beam is directed only to the parts of the screen where a picture is to be drawn.
The picture definition is stored as a set of line-drawing commands in a refresh display file or a refresh buffer in memory. Random-scan generally have higher resolution than raster systems and can produce smooth line drawings, however it cannot display realistic shaded scenes.

Graphics Systems



Block diagram of a CRT graphics system

In this context we discuss the graphics systems of raster-scan devices. A graphics processor accepts graphics commands from the CPU and executes the graphics commands which may involve drawing into the frame buffer. The frame buffer acts as a temporary store of the image and also as a decoupler to allow the graphics processor and the display controller to operate at different speeds. The display controller reads the frame buffer line by line and generates the control signals for the screen.

Graphics commands:

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

2 kinds of graphics processors:

2D graphics processors execute commands in 2D coordinates. When objects overlap, the one being drawn will obscure objects drawn previously in the region. **BitBlt** operations (Bit Block Transfer) are usually provided for moving/copying one rectangular region of frame buffer contents to another region.

3D graphics processors execute commands in 3D coordinates. When objects overlap, it is required to determine the visibility of the objects according to the z values.

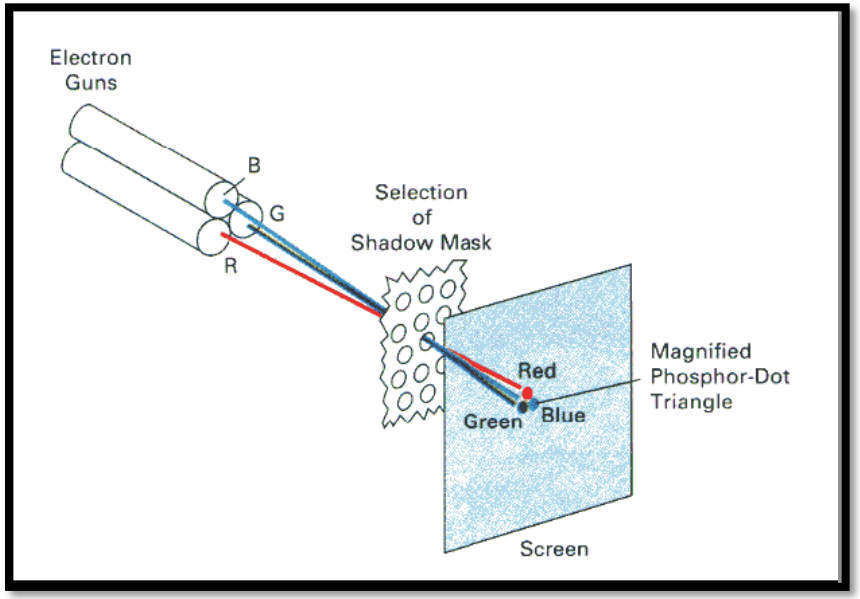
Display Controller for a raster display device reads the frame buffer and generates the control signals for the screen, ie. the signals for horizontal scanning and vertical scanning. Most display controllers include a **color map** (or video look-up table). The major function of a color map is to provide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

Color CRT monitor

The beam penetration method for displaying color pictures has been used with random-scan monitors. Two layers of phosphor, usually red and green, are coated on to theinside of the CRT screen, and the displayed color depends on how far the electron beam penetrates into the phosphor layers.

Shadow-mask

Shadow-mask methods are commonly used in raster-scan systems (including color TV) because they produce a much wider range of color than the beam penetration method. A shadow-mask CRT has three phosphor color dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. This type of CRT has three electron guns, one for each color dot, and a shadow-mask grid just behind the phosphor –coated screen. Figure 6 below illustrates the delta-delta shadow-mask method, commonly used in color CRT systems. The three electron beam are deflected and focused as a group onto the shadow mask, which contains a series of holes aligned with the phosphor-dot patterns. When the three beams pass through a hole in the shadow mask, they activate a dot triangle, which appears as a small color spot the screen the phosphor dots in the triangles are arranged so that each electron beam can activate only its corresponding color dot when it passes through the shadow mask .



Input Devices

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

Data gloves are electronic gloves for detecting fingers' movement. In some applications, a sensor is also attached to the glove to detect the hand movement as a whole in 3D space.
A **tablet** contains a stylus and a drawing surface and it is mainly used for the input of drawings. A tablet is usually more accurate than a mouse, and is commonly used for large drawings.
Scanners are used to convert drawings or pictures in hardcopy format into digital signal for computer processing.
Touch panels allow displayed objects or screen positions to be selected with the touch of a finger. In these devices a touch-sensing mechanism is fitted over the video monitor screen. Touch input can be recorded using optical, electrical, or acoustical methods.

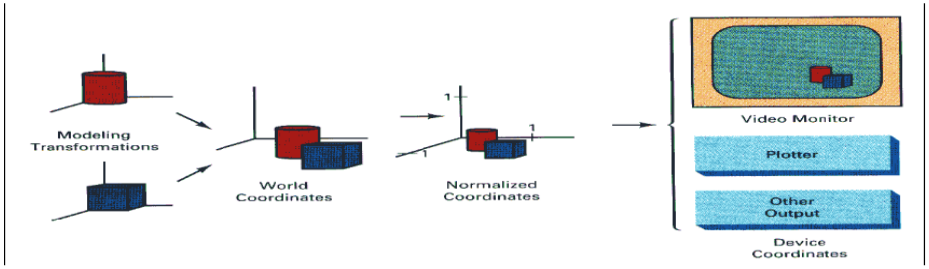
Hard-Copy Devices
Directing pictures to a printer or plotter to produce hard-copy output on 35-mm slides, overhead transparencies, or plain paper. The quality of the pictures depends on dot size and number of dots per inch (DPI).
Types of printers: line printers, laserjet, ink-jet, dot-matrix

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

Inkjet printers produce output by squirting ink in horizontal rows across a roll of paper wrapped on a drum. To produce color outputs, the 3 color pigments are shot simultaneously on a single pass along each print line on the paper.
Inkjet or pen plotters are used to generate drafting layouts and other drawings of normally larger sizes. A pen plotter has one or more pens of different colors and widths mounted on a carriage which spans a sheet of paper.

Coordinate Representations in Graphics

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



Modeling coordinates are used to construct individual object shapes.

World coordinates are computed for specifying the placement of individual objects in appropriatepositions.

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

Device coordinates are the final locations on the output devices.

Output Primitives

Drawing a Thin Line in Raster Devices

This is to compute intermediate discrete coordinates along the line path between 2 specified endpoint positions. The corresponding entry of these discrete coordinates in the frame buffer is then marked with the line color wanted.

The basic concept is:

A line can be specified in the form:

$$y = mx + c$$

Let m be between 0 to 1, then the slope of the line is between 0 and 45 degrees.

For the x-coordinate of the left end point of the line, compute the corresponding y value according to the line equation. Thus we get the left end point as (x1,y1), where y1 may not be an integer.

Calculate the distance of (x1,y1) from the center of the pixel immediately above it and call it D1

Calculate the distance of (x1,y1) from the center of the pixel immediately below it and call it D2

If D1 is smaller than D2, it means that the line is closer to the upper pixel than the lower pixel, then, we set the upper pixel to on; otherwise we set the lower pixel to on.

Then increatement x by 1 and repeat the same process until x reaches the right end point of the line.

This method assumes the width of the line to be zero.

Digital Differential Analyzer (DDA)

In computer graphics, a hardware or software implementation of a digital differential analyzer (DDA) is used for linear interpolation of variables over an interval between start and end point. DDAs are used for rasterization of lines, triangles and polygons.

The DDA method can be implemented using floating-point or integer arithmetic. The native floating-point implementation requires one addition and one rounding operation per interpolated value (e.g. coordinate x, y, depth, color component etc.) and output result. This process is only efficient when an FPU with fast add and rounding operation is available.

The fixed-point integer operation requires two additions per output cycle, and in case of fractional part overflow, one additional increment and subtraction. The probability of fractional part overflows is proportional to the ratio m of the interpolated start/end values.

DDAs are well suited for hardware implementation and can be pipelined for maximized throughput.

Where *m* represents the slope of the line and *c* is the y intercept . this slope can be expressed in DDA as

$$m = \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$$

in fact any two consecutive point(x,y) laying on this line segment should satisfy the equation.

The DDA starts by calculating the smaller of dy or dx for a unit increment of the other. A line is then sampled at unit intervals in one coordinate and corresponding integer values nearest the line path are determined for the other coordinate.

Considering a line with positive slope, if the slope is less than or equal to 1, we sample at unit x intervals (dx=1) and compute successive y values as

$$y_{k+1} = y_k + m$$

Subscript k takes integer values starting from 0, for the 1st point and increases by 1 until endpoint is reached. y value is rounded off to nearest integer to correspond to a screen pixel.

For lines with slope greater than 1, we reverse the role of x and y i.e. we sample at dy=1 and calculate consecutive x values as

$$x_{k+1} = x_k + \frac{1}{m}$$

Similar calculations are carried out to determine pixel positions along a line with negative slope. Thus, if the absolute value of the slope is less than 1, we set dx=1 if $x_{start} < x_{end}$ i.e. the starting extreme point is at the left.

Algorithm:-

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

void main()
{

   int gd = DETECT, gm = DETECT, s, dx, dy, m, x1, y1, x2, y2;
```

```
    float xi, yi, x, y;

    clrscr();

    printf("Enter the sarting point x1 & y1\n");
    scanf("%d%d", &x1, &y1);

    printf("Enter the end point x2 & y2\n");
    scanf("%d%d", &x2, &y2);

    initgraph(&gd, &gm, "");
    cleardevice();

    dx = x2 - x1;
    dy = y2 - y1;

    if (abs(dx) > abs(dy))
            s = abs(dx);
    else
            s = abs(dy);

    xi = dx / (float) s;
    yi = dy / (float) s;

    x = x1;
    y = y1;

    putpixel(x1, y1, 4);

    for (m = 0; m < s; m++) {
            x += xi;
            y += yi;
            putpixel(x, y, 4);
    }
    getch();
}
```

Bresenham's Line Drawing Algorithm

This algorithm is very efficient since it use only incremental integer calculations. Instead of calculating the non-integral values of D1 and D2 for decision of pixel location, it computes a value, p, which is defined as:
p = (D2-D1)* horizontal length of the line

if p>0, it means D1 is smaller than D2, and we can determine the pixel location accordingly

However, the computation of p is very easy:
The initial value of p is 2 * vertical height of the line - horizontal length of the line.
At succeeding x locations, if p has been smaller than 0, then, we increment p by 2 * vertical height of the line, otherwise we increment p by 2 * (vertical height of the line - horizontal length of the line)

Algorithm:-

```
void BresenhamLine(int x1, int y1, int x2, int y2)
{
int x, y, p, const1, const2;              /* initialize variables */
p=2*(y2-y1)-(x2-x1);
const1=2*(y2-y1);
const2=2*((y2-y1)-(x2-x1));
x=x1;
y=y1;
SetPixel(x,y);
while (x<xend)
{ x++;
if (p<0)
{
p=p+const1;
}
else
{ y++;
p=p+const2;
}
SetPixel(x,y);
}
```

}

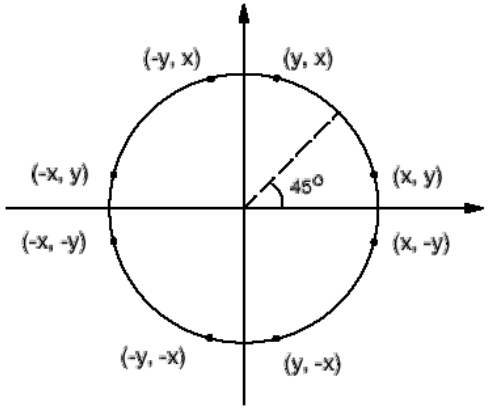
Drawing a Circle in Raster Devices (Mid Point Circle Drawing)

A circle can be specified in the form:

$$(x-xc)2 + (y-yc)2 = r2$$

where (xc,yc) is the center of the circle.

To save time in drawing a circle, we can make use of the symmetrical property of a circle which is to draw the segment of the circle between 0 and 45 degrees and repeat the segment 8 times as shown in the diagram to produce a circle. This algorithm also employs the incremental method which further improves the efficiency.



Midpoint Circle Algorithm

A circle is defined as a set of points that are all at a given distance *r* from a center positioned at. (x_c,y_c).
This is represented mathematically by the equation

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad \text{---------- (1)}$$

Using equation (1) we can calculate the value of y for each given value of x as

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \quad \text{---------- (2)}$$

Thus one could calculate different pairs by giving step increments to x and calculating the corresponding value of y. But this approach involves considerable computation at each step and also the resulting circle has its pixels sparsely plotted for areas with higher values of the slope of the curve.
Midpoint Circle Algorithm uses an alternative approach, wherein the pixel positions along the circle are determined on the basis of incremental calculations of a decision parameter.

Let

$$f(x,y) = (x - x_c)^2 + (y - y_c)^2 - r^2 \quad \text{---------- (3)}$$

Thus f(x,y)=0 represents the equation of a circle.
Further, we know from coordinate geometry, that for any point , the following holds:

1. $f(x,y) = 0 \quad \Rightarrow$ *The point lies on the circle.*
2. $f(x,y) < 0 \quad \Rightarrow$ *The point lies within the circle.*
3. $f(x,y) > 0 \quad \Rightarrow$ *The point lies outside the circle.*

In Midpoint Circle Algorithm, the decision parameter at the k^{th} step is the circle function evaluated using the coordinates of the midpoint of the two pixel centres which are the next possible pixel position to be plotted.
Let us assume that we are giving unit increments to x in the plotting process and determining the y position using this algorithm. Assuming we have just plotted the k^{th}
pixel at (X_k,Y_k), we next need to determine whether the pixel at the position (X_{k+1},Y_k) or the one at (X_{k+1} , Y_{k-1}) is closer to the circle.

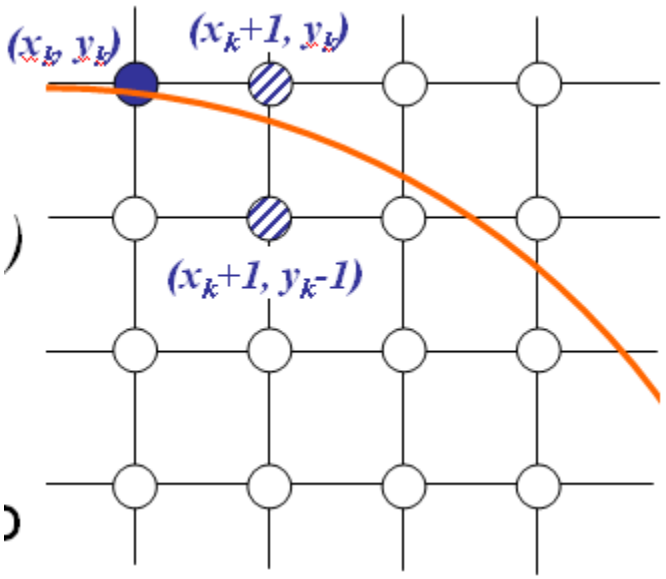
```
void MidPointCircleAlgo(int Radius,int xC,int yC)

{

   int P;

   int x,y;


void Draw(int x,int y,int xC,int yC); // Function to plots the points


P = 3 -2* Radius;

   x = 0;
```

```
    y = Radius;

    Draw(x,y,xC,yC);
while (x<=y)

    {

        x++;

        if (P<0)

        {

            P += 4 * x + 6;

        }

        else

        {

            P += 4 * (x - y) + 10;

            y--;

        }

        Draw(x,y,xC,yC);

    }
            }

void Draw(int x,int y,int xC,int yC)

{

xC=320+xC;

yC=240-yC;
```

```
    putpixel(xC + x,yC + y,1);

    putpixel(xC + x,yC - y,1);

    putpixel(xC - x,yC + y,1);

    putpixel(xC - x,yC - y,1);

    putpixel(xC + y,yC + x,1);

    putpixel(xC - y,yC + x,1);

    putpixel(xC + y,yC - x,1);

    putpixel(xC - y,yC - x,1);
}
```

Bresenham's Circle Algorithm

The midpoint circle algorithm is an algorithm used to determine the points needed for drawing a circle. The algorithm is a variant of Bresenham's line algorithm**,** and is thus sometimes known as Bresenham's circle algorithm, although not actually invented by Bresenham.
In the mid-point circle algorithm we use eight-way symmetry so only ever calculate the points for the top right eighth of a circle, and then use symmetry to get the rest of the points.
Assume that we have just plotted point (x_k, y_k). The next point is a
choice between (x_k+1, y_k) and (x_k+1, y_k-1). We would like to choose
the point that is nearest to the actual circle So how do we make this choice?



Let's re-jig the equation of the circle slightly to give us:

$f_{circ}(x,y) = x^2 + y^2 - r^2$

The equation evaluates as follows:

By evaluating this function at the midpoint between the candidate pixels we can make our decision

$$f_{circ}(x,y) \begin{cases} < 0, \text{ if } (x,y) \text{ is inside the circle boundary} \\ = 0, \text{ if } (x,y) \text{ is on the circle boundary} \\ > 0, \text{ if } (x,y) \text{ is outside the circle boundary} \end{cases}$$

Assuming we have just plotted the pixel at (x_k,y_k) so we need to choose between (x_k+1,y_k) and (x_k+1,y_k-1).

Our decision variable can be defined as:

$$P_k = f_{circ}(x_k+1, y_k-1/2)$$
$$= (x_k+1)^2 + (y_k - 1/2)^2 - r^2$$

If $p_k < 0$ the midpoint is inside the circle and and the pixel at y_k is closer to the circle. Otherwise the midpoint is outside and y_k-1 is closer.

To ensure things are as efficient as possible we can do all of our calculations incrementally First consider:

$$p_{k+1} = f_{circ}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$
$$= \left[(x_k + 1) + 1\right]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2$$

or:

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

where y_k+1 is either y_k or y_k-1 depending on the sign of p_k.
The first decision variable is given as:

$$p_0 = f_{circ}(1, r - 1/2)$$
$$= 1 + (r - 1/2)^2 - r^2$$
$$= 5/4 - r$$

Then if $p_k < 0$ then the next decision variable is given as:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

If $p_k > 0$ then the decision variable is:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_k + 1$$

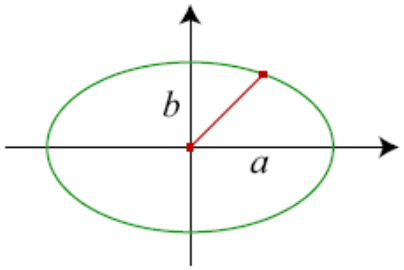
Algorithm:-

```
void PlotCirclePoints(int centerx, int centery, int x, int y)
{
 SetPixel(centerx+x,centery+y);
SetPixel(centerx-x,centery+y);
SetPixel(centerx+x,centery-y);
SetPixel(centerx-x,centery-y);
SetPixel(centerx+y,centery+x);
SetPixel(centerx-y,centery+x);
SetPixel(centerx+y,centery-x);
SetPixel(centerx-y,centery-x);
}
void BresenhamCircle(int centerx, int centery, int radius)
{
int x=0;
int y=radius;
int p=3-2*radius;
while (x<y)
{
```

```
 PlotCirclePoints(centerx,centery,x,y);
if (p<0)
{
p=p+4*x+6;
}
else
{
 p=p+4*(x-y)+10;
y=y-1;
}
x=x+1;
}
PlotCirclePoints(centerx,centery,x,y);
}
```

Ellipses

• **Implicit:** $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$ This is only for the special case where the ellipse is centered at the origin with the major and minor axes aligned with y = 0 and x = 0.



Parametric: $x(\lambda) = a\cos(2\pi\lambda),\ y(\lambda) = b\sin(2\pi\lambda)$, or in vector form

$$\bar{p}(\lambda) = \begin{bmatrix} a\cos(2\pi\lambda) \\ b\sin(2\pi\lambda) \end{bmatrix}.$$

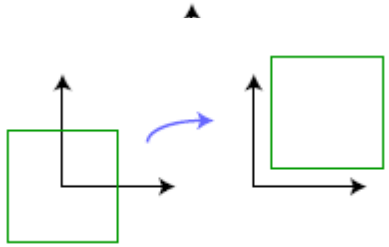
The implicit form of ellipses and circles is common because there is no explicit functional form. This is because y is a multifunction of x.

2D Transformations

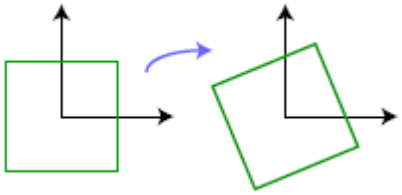
Given a point cloud, polygon, or sampled parametric curve, we can use transformations for several purposes:
1. Change coordinate frames (world, window, viewport, device, etc).
2. Compose objects of simple parts with local scale/position/orientation of one part defined with regard to other parts.
   For example, for articulated objects.
3. Use deformation to create new shapes.
4. Useful for animation.

There are three basic classes of transformations:
1. **Rigid body** - Preserves distance and angles.
• Examples: translation and rotation.
2. **Conformal** - Preserves angles.
• Examples: translation, rotation, and uniform scaling.
3. **Affine** - Preserves parallelism. Lines remain lines.
• Examples: translation, rotation, scaling, shear, and reflection.

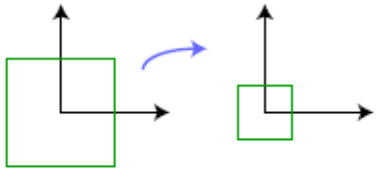
Translation by vector $\vec{t}$: $\bar{p}_1 = \bar{p}_0 + \vec{t}$.



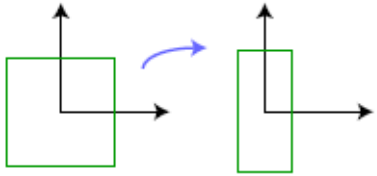
Rotation counterclockwise by θ: $\bar{p}_1 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \bar{p}_0$.



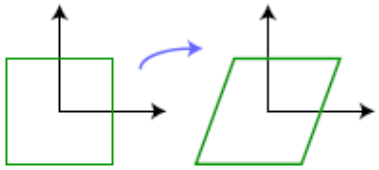
Uniform scaling by scalar a: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \bar{p}_0$.



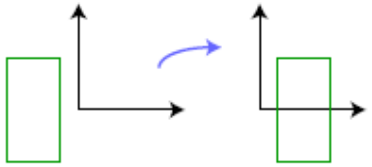
Nonuniform scaling by a and b: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \bar{p}_0$.



Shear by scalar h: $\bar{p}_1 = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \bar{p}_0$.



Reflection about the y-axis: $\bar{p}_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \bar{p}_0$.

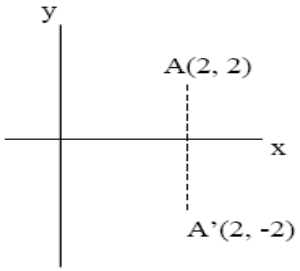


Mirroring

In modeling operations, one frequently used operation is mirroring an object. Mirroring is a convenient method used for copying an object while preserving its features. The mirror transformation is a special case of a negative scaling, as will be explained below.
Let us say, we want to mirror the point A (2, 2) about the x-axis(i.e., xz-plane), as shown in the figure.
The new location of the point, when reflected about the x-axis, will be at (2, -2). The point matrix [P*] = [2 -2] can be obtained with the matrix transformation given below.

$$[P^*] = [2 \quad 2 \quad 0 \quad 1] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= [2 \quad -2 \quad 0 \quad 1]$$



Affine Transformations

An **affine transformation** takes a point $\bar{p}$ to $\bar{q}$ according to $\bar{q} = F(\bar{p}) = A\bar{p} + \vec{t}$, a linear transformation followed by a translation.

Homogeneous Coordinates

Homogeneous coordinates are another way to represent points to simplify the way in which we express affine transformations. Normally, bookkeeping would become tedious when affine transformations of the form $A\bar{}p + \sim t$ are composed. With homogeneous coordinates, affine transformations become matrices, and composition of transformations is as simple as matrix multiplication.
In future sections of the course we exploit this in much more powerful ways.
Example: Translate the rectangle (2,2), (2,8), (10,8), (10,2) 2 units along x-axis and 3 units along y axis.
Solution: Using the matrix equation for translation, we have

$[P^*] = [P]\,[T_t]$, substituting the numbers, we get

$$[P^*] = \begin{pmatrix} 2 & 2 & 0 & 1 \\ 2 & 8 & 0 & 1 \\ 10 & 8 & 0 & 1 \\ 10 & 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 3 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 4 & 5 & 0 & 1 \\ 4 & 11 & 0 & 1 \\ 12 & 11 & 0 & 1 \\ 12 & 5 & 0 & 1 \end{pmatrix}$$

Composition of 2D Transformations

There are many situations in which the final transformation of a point is a combination of several (often many) individual transformations. For example, the position of the finger of a robot might be a function of the rotation of the robots hand, arm, and torso, as well as the position of the robot on the railroad train and the position of the train in the world, and the rotation of the planet around the sun, and . . .
Applying each transformation individually to all points in a model would take a lot of time. Instead of applying several transformations matrices to each point we want to combine the transformations to produce 1 matrix which can be applied to each point.
In the simplest case we want to apply the same type of transformation (translation, rotation, scaling) more than once.

1. translation is additive as expected
   2.scaling is multiplicative as expected
   3. rotation is additive as expected

But what if we want to combine different types of transformations?
a very common reason for doing this is to rotate a polygon about an arbitrary point (e.g. the center of the polygon) rather than around the origin.

Translate so that P1 is at the origin T(-Dx,-Dy)
Rotate R(theta)
Translate so that the point at the origin is at P1 T(Dx,Dy)

note the order of operations here is right to left:
P' = T(Dx,Dy) * R(theta) * T(-Dx,-Dy) * P
i.e.
P' = T(Dx,Dy) * { R(theta) * [T(-Dx,-Dy) * P] }
i.e.
P' = [T(Dx,Dy) * R(theta) * T(-Dx,-Dy)] * P
The matrix that results from these 3 steps can then be applied to all of the points in the polygon.
another common reason for doing this is to scale a polygon about an arbitrary point (e.g. the center of the polygon) rather than around the origin.

Translate so that P1 is at the origin
Scale
Translate so that the point at the origin is at P1

How do we determine the 'center' of the polygon?

- specifically define the center (e.h. the center of mass)
- average the location of all the vertices
- take the center of the bounding box of the polygon

Shears

A shear can be thought of as "pulling and elongating" a graphic to the right (positive x value) to the left (negative x value), up (positive y value) or down (negative y value).
For x shear, we use the matrix

$$A = \begin{bmatrix} 1 & f \\ 0 & 1 \end{bmatrix}$$

+f produces a right shear, f >0
-f produces a left shear, f < 0

For y shear, we use the matrix

$$A = \begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix}$$

+f produces a top shear, f >0
-f produces a bottom shear, f < 0

REFLECTION ABOUT THE Y AXIS

Let f: $R^2 \rightarrow R^2$ be the matrix transformation where f(**v**) =A**v** where

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

In general terms then, the matrix transformation is:

$$f(V) = Av = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix}$$

Our example triangle E is thus:

$$V_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad V_2 = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \quad V_3 = \begin{bmatrix} 5 \\ 1 \end{bmatrix}.$$

To compute the image of E reflected about the y axis by forming the products:

$$Av_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

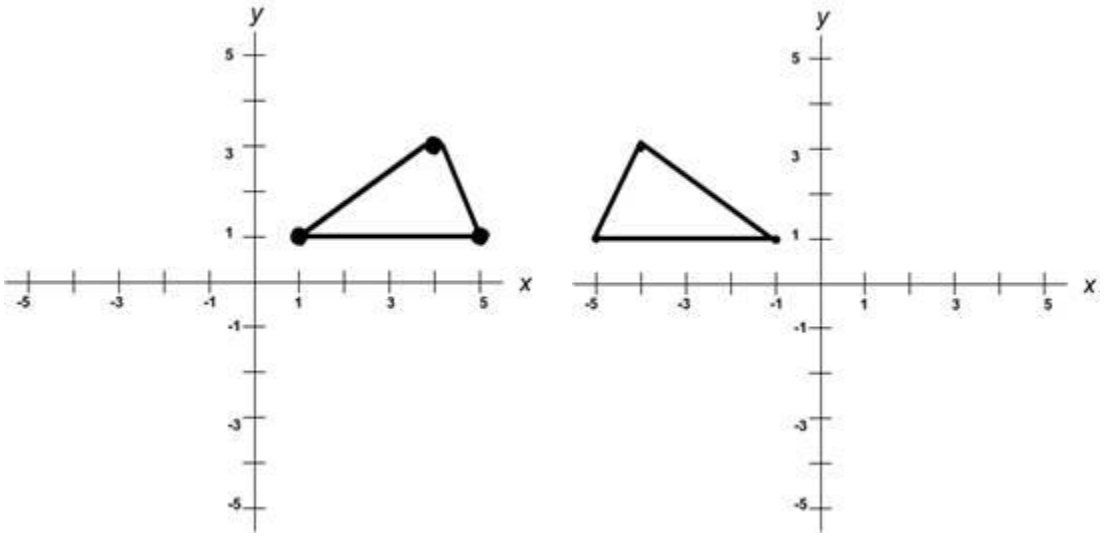
$$Av_2 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} -4 \\ 3 \end{bmatrix}$$

$$Av_3 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \end{bmatrix} = \begin{bmatrix} -5 \\ 1 \end{bmatrix}$$

E is thus transformed and reflected about the y axis with vertices

(-1, 1), (-4, 3) and (-5, 1)

and the transformation is illustrated in the graphics below:



REFLECTION ABOUT X AXIS

The general formula for reflection about the x axis is:

$$f(v) = Av = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ -y \end{bmatrix}$$

Transformation between cocordinate system

Window to Viewport

Generally user's prefer to work in world-coordinates.

1 unit can be 1 micron
1 unit can be 1 meter
1 unit can be 1 kilometer
1 unit can be 1 mile

These coordinates must then be translated to screen coordinates to be displayed in a rectangular region of the screen called the viewport
The objects are in world coordinates (with n dimensions)
The viewport is in screen coordinates (with n=2)
Want one matrix that can be applied to all points:
rectangular area of world from (Xmin,Ymin) to (Xmax,Ymax) - world-coordinate window
rectangular area of screen from (Umin,Vmin) to (Umax,Vmax) - viewport
need to rescale the world-coordinate rectangle to the screen rectangle
1. translate world-coordinate window to the origin of the world coordinate system.
2. rescale the window to the size and aspect ratio of the viewport.
3. translate the viewport to its position on the screen in the screen coordinate system.
Pscreen = M * Pworld
M = T(Umin,Vmin) * S(deltaU/deltaX, deltaV/deltaY) * T(-Xmin, -Ymin)

2 Dimensional Viewing Transformation

I

When we define an image in some world coordinate system, to display that image we must somehow map the image to the physical output device. This is a two stage process. For 3 dimensional images we must first project down to 2 dimensions, since our display device is 2 dimensional. Next, we must map the 2 D representation to the physical device. This section concerns the second part of this process, i.e. the 2D WDC to 2D physical device coordinates (PDC). We will first discuss the concept of a Window on the world (WDC), and then a Viewport (in NDC), and finally the mapping WDC to NDC to PDC.

Window

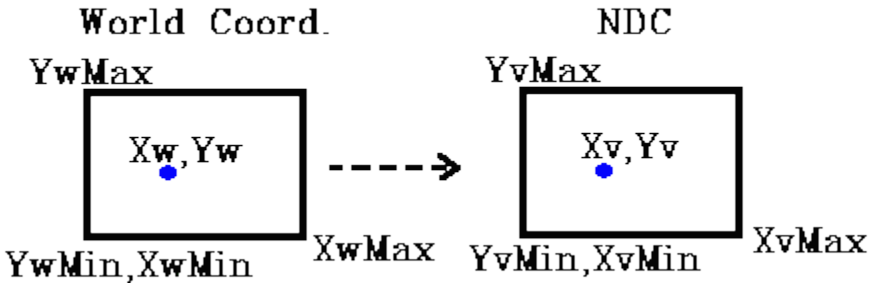
When we model an image in World Device Coordinates (WDC) we are not interested in the entire world but only a portion of it. Therefore we define the portion of interest which is a polygonal area specified in world coordinates, called the "window".

Viewport

The user may want to create images on different parts of the screen so we define a viewport in Normalized Device Coordinates (NDC). Using NDC also allows for output device independence. Later we will map from NDC to Physical Device Coordinates (PDC).

2D Window To Viewport Tranformation

We want to map a point from WDC to Ndc, as shown below:



We can see from above that to maintain relative position we must have the following relationship:

$$\frac{Xw - Xwmin}{Xwmax - Xwmin} = \frac{Xv - Xvmin}{Xvmax - Xvmin}$$

$$\frac{Yw - Ywmin}{Ywmax - Ywmin} = \frac{Yv - Yvmin}{Yvmax - Yvmin}$$

We can rewrite above as

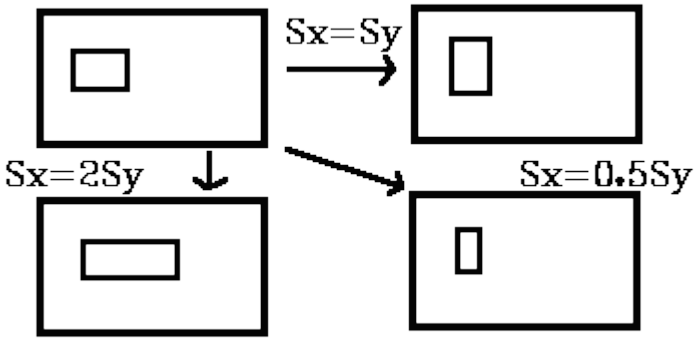
$$Xv = [(Xvmax - Xvmin)/(Xwmax - Xwmin)] * (Xw - Xwmin) + Xvmin$$
$$= Sx * (Xw - Xwmin) + Xvmin = Sx * Xw + Cx$$

where $Sx = (Xvmax - Xvmin)/(Xwmax - Xwmin)$
$Cx = Sx * (- Xwmin) + Xvmin$

Similarly for Y

$$Yv = [(Yvmax - Yvmin) / (Ywmax - Ywmin)] * (Yw - Ywmin) + Yvmin$$
$$= Sy * (Yw - Ywmin) + Yvmin = Sy * Yw + Cy$$

Note that Sx, Sy are "scaling" factors.
If Sx = Sy the objects will retain same shape, else will be distorted, as shown in the example.



Line Clipping

This section treats clipping of lines against rectangles. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that other graphic primitives can be clipped by repeated application of the line clipper.

Clipping Individual Points

Before we discuss clipping lines, let's look at the simpler problem of clipping individual points.
If the x coordinate boundaries of the clipping rectangle are Xmin and Xmax, and the y coordinate boundaries are Ymin and Ymax, then the following inequalities must be satisfied for a point at (X, Y) to be inside the clipping rectangle:-

$$Xmin < X < Xmax$$
$$\text{and } Ymin < Y < Ymax$$

If any of the four inequalities does not hold, the point is outside the clipping rectangle.

The Cohen-Sutherland Line-Clipping Algorithm

The more efficient Cohen-Sutherland Algorithm performs initial tests on a line to determine whether intersection calculations can be avoided.

Steps for Cohen-Sutherland algorithm

1. End-points pairs are check for trivial acceptance or trivial rejected using the outcode.
2. If not trivial-accepance or trivial-rejected, divided into two segments at a clip edge.
3. Iteratively clipped by testing trivial-acceptance or trivial-rejected, and divided into two segments until completely inside or trivial-rejected.

Pseudo-code of Cohen-Sutherland Algorithm

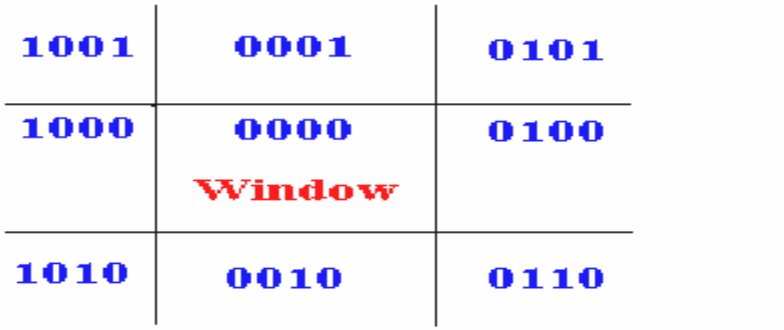
Trivial acceptance/reject test

To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions. Each region is assigned a 4-bit code deteermined by where the region lies with respect to the outside halfplanes of the clip-rectangle edges. Each bit in the outcode is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:

- Bit 1 : outside halfplane of top edge, above top edge
  $Y > Ymax$
- Bit 2 : outside halfplane of bottom edge, below bottom edge
  $Y < Ymin$
- Bit 3 : outside halfplane of right edge, to the right of right edge
  $X > Xmax$
- Bit 4 : outside halfplane of left edge, to the left of left edge
  $X < Xmin$

As you proceed around the window, extending each edge and defining an inside half-space and an outside half-space, nine regions are created - the eight "outside" regions and the one "inside"region. Each of the nine regions associated with the window is assigned a 4-bit code to identify

the region. Each bit in the code is set to either a **1**(true) or a **0**(false). If the region is to the **left** of the window, the **first** bit of the code is set to 1. If the region is to the **top** of the window, the **second** bit of the code is set to 1. If to the **right**, the **third** bit is set, and if to the **bottom**, the **fourth** bit is set. The 4 bits in the code then identify each of the nine regions as shown below.

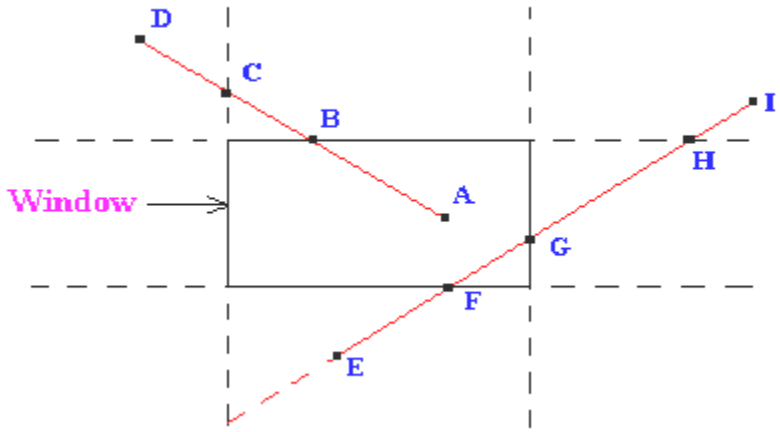


The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivally accepted or rejected. If the line cannot be trivally accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated. This process is continued until the line is accepted.
To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions. Each end point of the line segment is then assigned the code of the region in which it lies.

1. Given a line segment with endpoint $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$
2. Compute the 4-bit codes for each endpoint.
   If both codes are **0000**,(bitwise OR of the codes yields 0000) line lies completely **inside** the window: pass the endpoints to the draw routine.
   If both codes have a 1 in the same bit position (bitwise AND of the codes is **not** 0000), the line lies **outside** the window. It can be trivially rejected.
3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be **clipped** at the window edge before being passed to the drawing routine.
4. Examine one of the endpoints, say $P_1 = (x_1, y_1)$. Read P_1's 4-bit code in order: **Left**-to-**Right**, **Bottom**-to-**Top**.
5. When a set bit (1) is found, compute the **intersection** **I** of the corresponding window edge with the line from P_1 to P_2. Replace P_1 with **I** and repeat the algorithm.

Illustration of Line Clipping

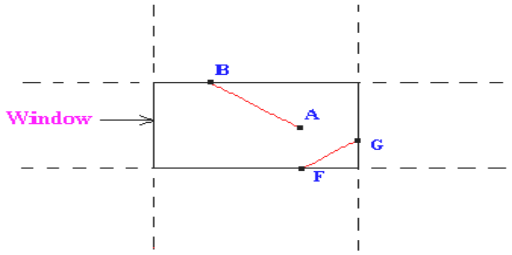
Before Clipping



1. Consider the line segment **AD**.
   Point **A** has an outcode of **0000** and point **D** has an outcode of **1001**. The logical AND of these outcodes is zero; therefore, the line cannot be trivally rejected. Also, the logical OR of the outcodes is not zero; therefore, the line cannot be trivally accepted. The algorithm then chooses **D** as the outside point (its outcode contains 1's). By our testing order, we first use the top edge to clip **AD** at **B**. The algorithm then recomputes **B**'s outcode as **0000**. With the next iteration of the algorithm, **AB** is tested and is trivially accepted and displayed.
2. Consider the line segment **EI**
   Point **E** has an outcode of **0100**, while point **I**'s outcode is **1010**. The results of the trivial tests show that the line can neither be trivally rejected or accepted. Point **E** is determined to be an outside point, so the algorithm clips the line against the bottom edge of the window. Now line **EI** has been clipped to be line **FI**. Line **FI** is tested and cannot be trivially accepted or rejected. Point **F** has an outcode of **0000**, so the algorithm chooses point **I** as an outside point since its outcode is**1010**. The line **FI** is clipped against the window's top edge, yielding a new line **FH**. Line **FH** cannot be trivally accepted or rejected. Since **H**'s outcode is **0010**, the next iteration of the algorthm clips against the window's right edge, yielding line **FG**. The next iteration of the algorithm tests **FG**, and it is trivially accepted and display.

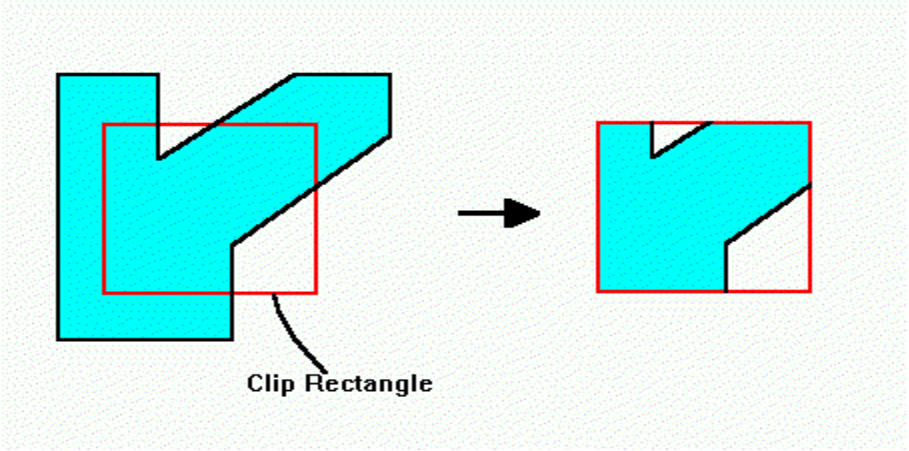
After Clipping

After clipping the segments **AD** and **EI**, the result is that only the line segment **AB** and **FG** can be seen in the window.



Clipping Polygons

An algorithm that clips a polygon must deal with many different cases. The case is particularly note worthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.
The following example illustrate a simple case of polygon clipping.



Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.
Note the difference between this strategy for a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when necessary.

Steps of Sutherland-Hodgman's polygon-clipping algorithm

- Polygons can be clipped against each edge of the window one at a time. Windows/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.
- Note that the number of vertices usually changes and will often increases.
- We are using the Divide and Conquer approach.

Here is a example of polygon clipping.

Four Cases of polygon clipping against one edge

The clip boundary determines a visible and invisible region. The edges from vertex i to vertex i+1 can be one of four types:

- Case 1 : Wholly inside visible region - save endpoint
- Case 2 : Exit visible region - save the intersection
- Case 3 : Wholly outside visible region - save nothing
- Case 4 : Enter visible region - save intersection and endpoint

Because clipping against one edge is independent of all others, it is possible to arrange the clipping stages in a pipeline. The input polygon is clipped against one edge and any points that are kept are passed on as input to the next stage of the pipeline. This way four polygons can be at different stages of the clipping process simultaneously. This is often implemented in hardware.

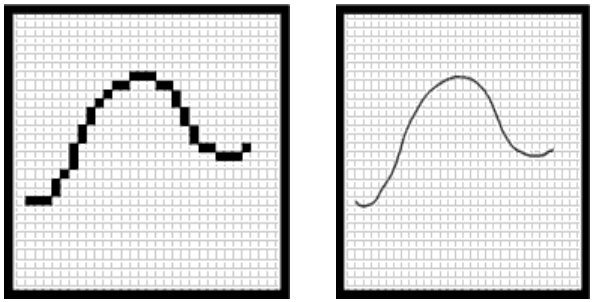
Lecture-20

Antialiasing

What anti-aliasing attempts to do is, using mathematics, fill in some of the digital system with colors that are in-between the two adjoining colors. In this case a medium gray would be between the black and the white. Some gray squares placed in the grid might help soften up the "jaggies".

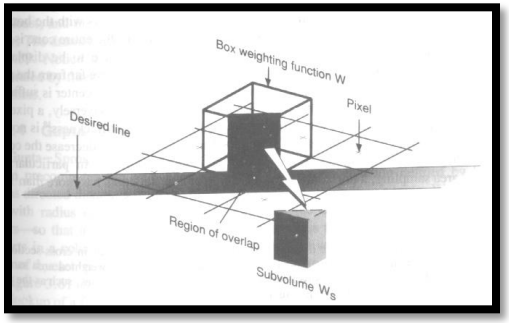
Antialiasing utilizes blending techniques to blur the edges of the lines and provide the viewer with the illusion of a smoother line.

- Two general approaches:
- **Super-sampling**
- samples at higher resolution, then filters down the resulting image
- Sometimes called post-filtering
- The prevalent form of anti-aliasing in hardware
- **Area sampling**
- sample primitives with a box (or Gaussian, or whatever) rather than spikes
- Requires primitives that have area (lines with width)



Unweighted Area Sampling

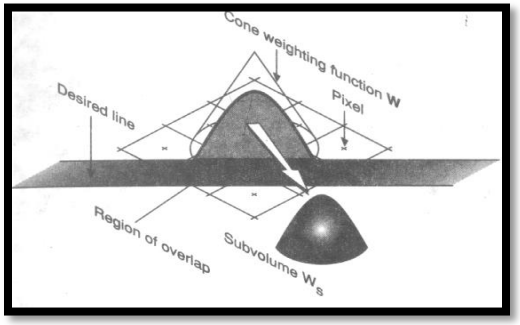
- Intensity is proportional to the amount of area covered.
- Defining the box weighting function which has volume of 1 (I_{max}).
- Determine the overlap volume Ws [0, 1].
- primitive cannot affect intensity of pixel if it does not intersect the pixel
- equal areas cause equal intensity, regardless of distance from pixel center to area
- Un-weighted sampling colors two pixels identically when the primitive cuts the same area through the two pixels
- intuitively, pixel cut through the center should be more heavily weighted than one cut along corner



Weighted Area Sampling

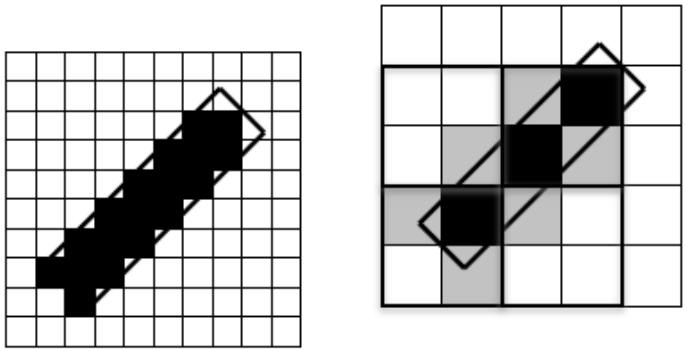
Weight the subpixel contributions according to position, giving higher weights to the central subpixels.

Weighting function, W(x,y) specifies the contribution of primitive passing through the point (x, y) from pixel center.



Super-sampling

- Sample at a higher resolution than required for display, and filter image down
- 4 to 16 samples per pixel is typical
- Samples might be on a uniform grid, or randomly positioned, or other variants
- Divide each pixel into sub-pixels.
- The number of intensities is the max number of sub-pixels selected on the line segment within a pixel.
- The intensity level for each pixel is proportional to the number of sub-pixels inside the polygon representing the line area.
- Line intensity is distributed over more pixels.



Area Sampling

Determine the percentage of area coverage for a screen pixel, then set the pixel intensity proportional to this percentage.

- Consider a line as having thickness
- Consider pixels as little squares
- **Unweighted area sampling**
- Fill pixels according to the proportion of their square covered by the line

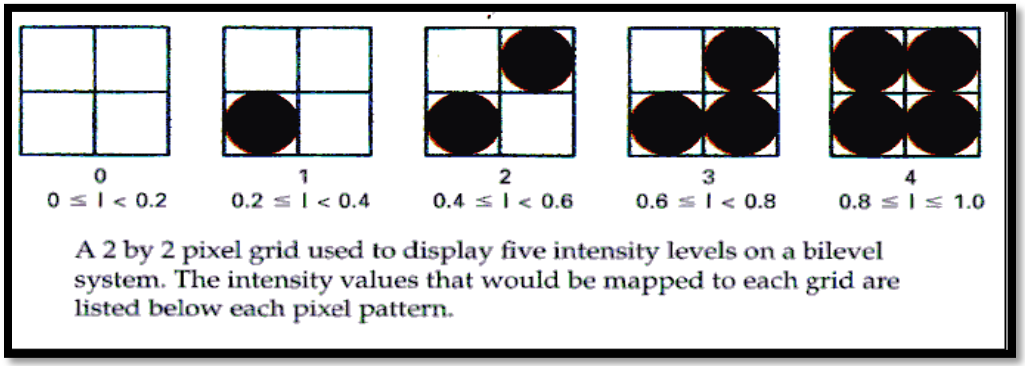
Halftone Patterns and Dithering

Halftoning is used when an output device has a limited intensity range, but we want to create an apparent increase in the number of available intensities.

Example: The following shows an original picture and the display of it in output devices of limited intensity ranges (4 colors, 8 colors, 16 colors):

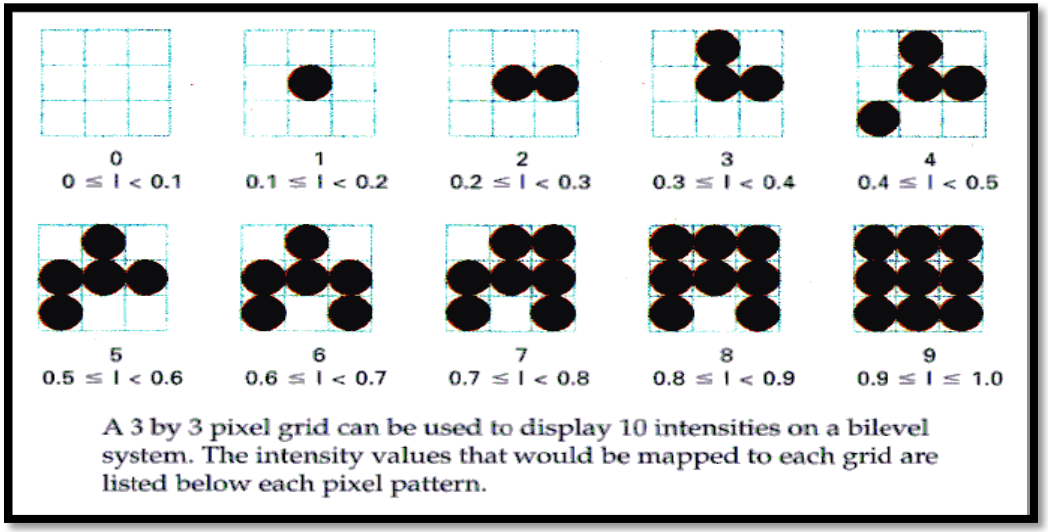
If we view a very small area from a sufficiently large viewing distance, our eyes average fine details within the small area and record only the overall intensity of the area. By halftoning, each small resolution unit is imprinted with a circle of black ink whose area is proportional to the blackness of the area in the original photograph. Graphics output devices can approximate the variable-area circles of halftone reproduction by incorporating multiple pixel positions into the display of each intensity value.

A 2 x 2 pixel grid used to display 5 intensity levels (I) on a bilevel system:



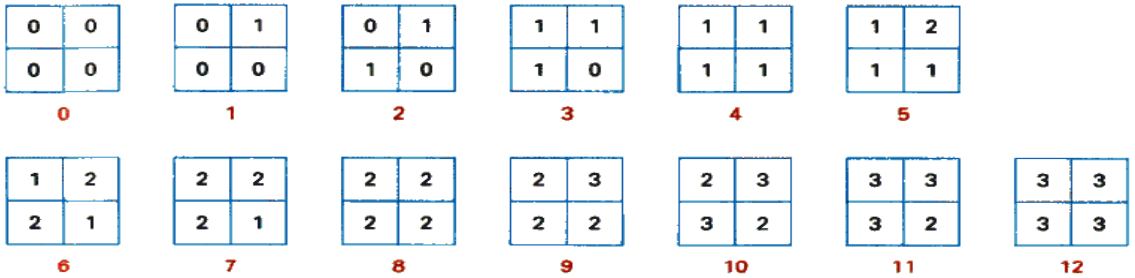
A 2 by 2 pixel grid used to display five intensity levels on a bilevel system. The intensity values that would be mapped to each grid are listed below each pixel pattern.

A 3 x 3 pixel grid used to display 10 intensities on a bilevel system:



A 3 by 3 pixel grid can be used to display 10 intensities on a bilevel system. The intensity values that would be mapped to each grid are listed below each pixel pattern.

A 2x2 pixel grid used to display 13 intensities on a 4-level system:



Dithering

The above approach, however, needs a higher resolution output device to display a picture in the same physical dimensions. So, in reality, we have to refine this approach so that it does not require for higher resolution. Dithering generally means to approximate halftone without this requirement. Interested students may find further discussion on dithering in many text books.
Below are two examples of dithering results, using 4 and 2 colors respectively.

Scan conversion of characters

Characters are polygons. However, they are used very often
Type-face, font: the overall design style for a set of characters, Courier, Helvetica, Times, etc.

Attributes of Primitives

In general, any parameter that affects the way a primitive is to be displayed is referred to as an attribute parameter. Some attribute parameters, such as color and size, determine the fundamental characteristics of a primitive. Others specify how the primitive is to be displayed under special conditions. Examples of attributes in this class include depth information for three-dimensional viewing and visibility or detectability options for interactive object-selection programs. These special-condition attributes will be considered in later chapters. Here, we consider only those attributes that control the basic display properties of primitives, without regard for* special situations. For example, lines can be dotted or dashed, fat or thin, and blue or orange. Areas might be filled with one color or with a multicolor pattern. Text can appear reading from left to right, slanted diagonally across the screen, or in vertical columns. Individual characters can be displayed in different fonts, colors, and sizes. And we can apply intensity variations at the edges of objects to smooth out the raster stairstep effect.

One way to incorporate attribute options into a graphics package is to extend the parameter list associated with each output primitive function to include the appropriate attributes. A line-drawing function, for example, could contain parameters to set color, width, and other properties, in addition to endpoint coordinates. Another approach is to maintain a system list of current attribute values. Separate functions are then included in the graphics package for setting the current values in the attribute list. To generate an output primitive, the system checks the relevant attributes and invokes the display routine for that primitive using the current attribute settings. Some packages provide users with a combination of attribute functions and attribute parameters in the output primitive commands. With the GKS and PHIGS standards, attribute settings are accomplished with separate functions that update a system attribute list.

Basic attributes of a straight line segment are its type, its width, and its color. In some graphics packages, lines can also be displayed using selected pen or brush options. In the following sections, we consider how line-drawing routines can be modified to accommodate various

attribute specifications.

Line Type

Possible selections for the line-type attribute include solid lines, dashed lines, and dotted lines. We modify a line-drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path. A dashed line could be displayed by generating an interdash spacing that is equal to the 1/4 length of the solid sections. Both the length of the dashes and the interdash spacing are often specified as user options. A dotted line can be displayed by generating very short dashes with the spacing equal to or greater than the dash size. Similar methods are used to produce other line-type variations.

To set line type attributes in a C language application program, a user invokes the function

setLinetype (lt)

where parameter it is assigned a positive integer value of 1,2,3, or 4 to generate lines that are, respectively, solid, dashed, dotted, or dash-dotted. Other values for the line-type parameter it could be used to display variations in the dotdash patterns. Once the line-type parameter has been set in a PHKS application program, all subsequent line-drawing commands produce lines with this Line type.

Line Width

Implementation of line- width options depends on the capabilities of the output device. A heavy line on a video monitor could be displayed as adjacent parallel lines, while a pen plotter might require pen changes. As with other PHIGS attributes, a line-width command is used to set the current line-width value in the attribute list. This value is then used by line-drawing algorithms to control the thickness of lines generated with subsequent output primitive commands.

We set the line-width attribute with the command:

setLinewidthScaleFactor(lw)

Line-width parameter lw is assigned a positive number to indicate the relative width of the line to be displayed. A value of 1 specifies a standard-width line. On a pen plotter, for instance, a user could set lw to a value of 0.5 to plot a line whose width is half that of the standard line. Values greater than 1 produce lines thicker than the standard.

Line Color

When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A polyline routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the setpixel procedure. The number of color choices depends on the number of bits available per pixel in the frame buffer.

We set the line color value in PHIGS with the function

setPolylineColourIndex(lc)

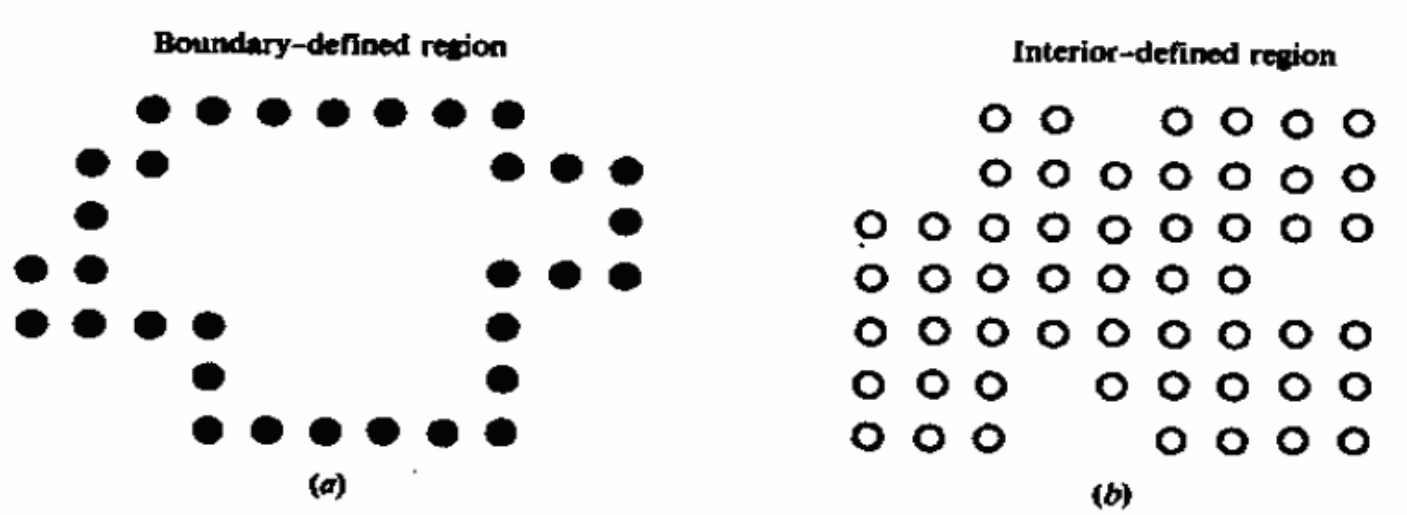
Nonnegative integer values, corresponding to allowed color choices, are assigned to the line color parameter lc. A line drawn in the background color is invisible, and a user can erase a previously displayed line by respecifying it in the background color (assuming the line does not overlap more than one background color area).

An example of the use of the various line attribute commands in an applications program is given by the following sequence of statements:

```
Setlinetype(2);
setLinewidthScaleFactor (2);
setPolylineColourIndex (5);
polyline (nl, wcpointsl);
```

```
setPolylineClourIndex (6);
polyline(n2cpoints2);
```

This program segment would display two figures, drawn with double-wide dashed lines. The first is displayed in a color corresponding to code 5, and the second in color 6.



Region Filling

Region filling is the process of "coloring in" a definite image area or region. Regions may be defined at the pixel or geometric level. At the pixel level, we describe a region either in terms of the bounding pixels that outline it or as the totality of pixels that comprise it. In the first case the region is called boundary-defined and the collections of algorithms used for filling such a region are collectively called boundary-fill algorithms. The other type of region is called an interior-defined region and the accompanying algorithms are called flood-fill algorithms. At the geometric level a region is defined or enclosed by such abstract contouring elements as connected lines and curves. For example, a polygonal region, or a filled polygon, is defined by a closed polyline, which is a polyline (i.e., a series of sequentially connected lines) that has the end of file last line connected to file beginning of file first line.

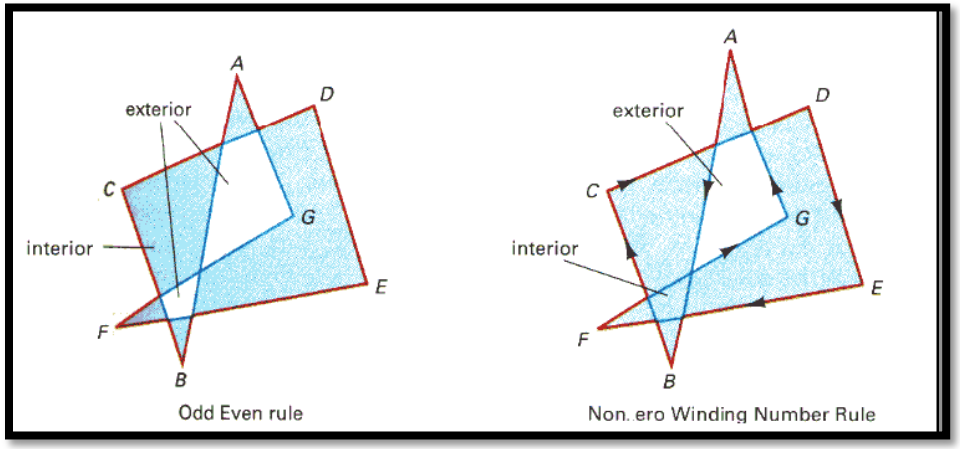
4-Connected vs. 8-Connected

An interesting point here is that, while a geometrically defined contour clearly separates the interior of a region from the exterior, ambiguity may arise when an outline consists of discrete pixels in the image space. There are two ways in which pixels are considered connected to each other to form a "continuous" boundary. One method is called 4-connected, where a pixel may have up to four neighbors the other is called 8-connectecd where a pixel may have up to eight neighbors Using the 4-connected approach, the pixels in do not define a region since several pixels such as A and B are not connected. However using the 8-connected definition we identify a triangular region.

We can further apply the concept of connected pixels to decide if a region is connected to another region. For example, using the 8-

connected approach, we do not have an enclosed region in fig since "interior" pixel C is connected to "exterior" pixel D. On the other hand, if we use the 4-connected definition we have a triangular region since no interior pixel is connected to the outside.
Note that it is not a mere coincidence that the figure is a boundary-defined region when we use the 8-connected definition for the boundary pixels and the 4-connected definition for the interior pixels. In fact, using the same definition for both boundary and interior pixels would simply result in contradiction. For example, if we use the 8-connected approach we would have pixel A connected to pixel B (continuous boundary) and at the same time pixel C connected to pixel D (discontinuous boundary). On the other hand, if we use the 4-connectd definition we would have pixel A disconnected from pixel B (discontinuous boundary) and at the same time pixel C disconnected from pixel D (continuous boundary).

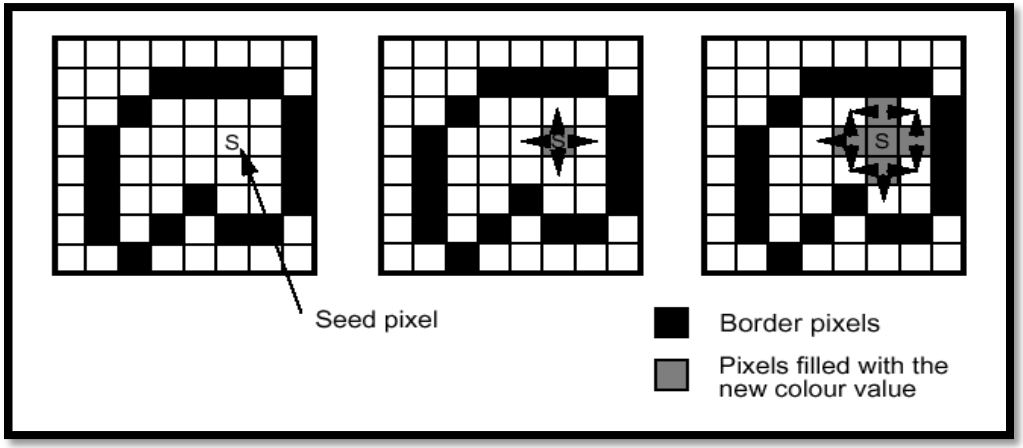
Inside-Outside Tests: The above algorithm only works for standard polygon shapes. However, for the cases which the edges of the polygon intersect, we need to identify whether a point is an interior or exterior point. Students may find interesting descriptions of 2 methods to solve this problem in many text books: odd-even rule and nonzero winding number rule.



Boundary-Fill Algorithm

- This algorithm starts at a point inside a region and paint the interior outward towards the boundary.
- This is a simple method but not efficient: 1. It is recursive method which may occupy a large stack size in the main memory.

```
void   BoundaryFill(int x, int y, COLOR fill, COLOR boundary)
{
COLOR current;
current=GetPixel(x,y);
if (current<>boundary) and (current<>fill) then
{
SetPixel(x,y,fill);
BoundaryFill(x+1, y, fill, boundary);
BoundaryFill(x-1, y, fill, boundary);
BoundaryFill(x, y+1, fill, boundary);
BoundaryFill(x, y-1,fill, boundary);
}
}
```



More efficient methods fill horizontal pixel spands across scan lines, instead of proceeding to neighboring points.

Flood-Fill Algorithm

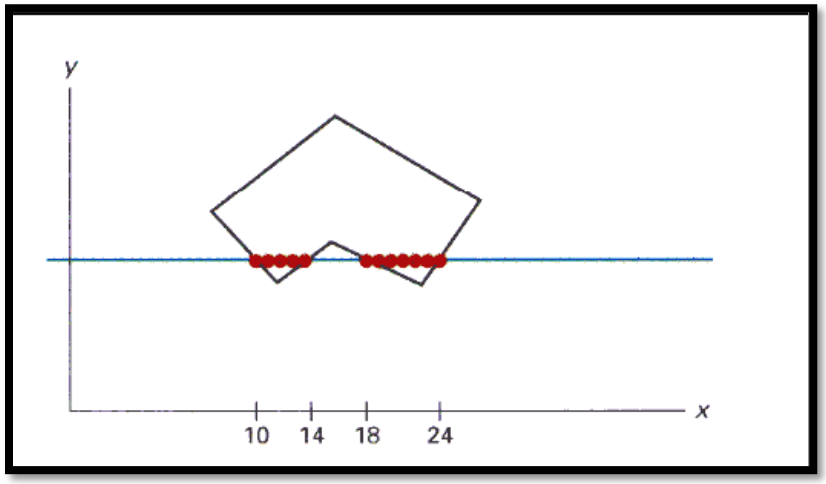
- Flood-Fill is similar to Boundary-Fill. The difference is that Flood-Fill is to fill an area which is not defined by a single boundary color.

```
void BoundaryFill(int x, int y, COLOR fill, COLOR old_color)
{
if (GetPixel(x,y)== old_color)
{
 SetPixel(x,y,fill);
```

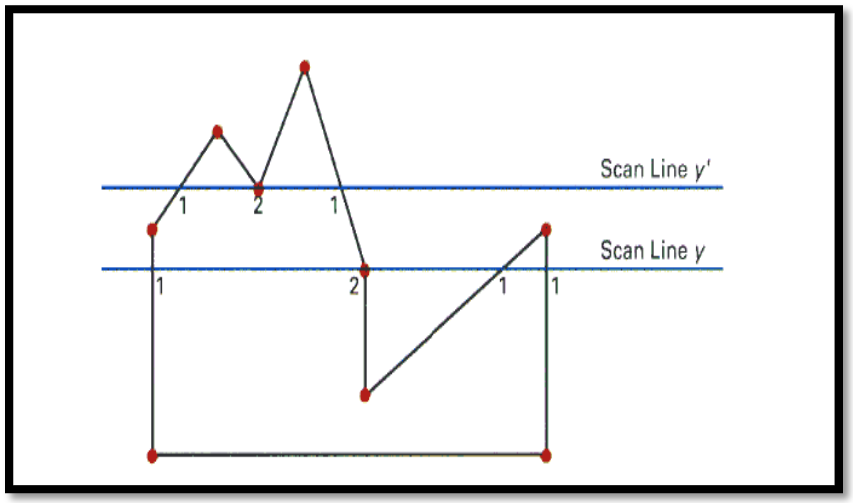
```
BoundaryFill(x+1,y,fill,boundary);
BoundaryFill(x-1,y,fill,boundary);
BoundaryFill(x,y+1,fill,boundary);
BoundaryFill(x,y-1,fill,boundary);
}
}
```

Scan-Line Polygon Fill Algorithm

Basic idea: For each scan line crossing a polygon, this algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are shorted from left to right.
Then, we fill the pixels between each intersection pair.



Some scan-line intersection at polygon vertices require special handling. A scan line passing through a vertex as intersecting the polygon twice. In this case we may or may not add 2 points to the list of intersections, instead of adding 1 points. This decision depends on whether the 2 edges at both sides of the vertex are both above, both below, or one is above and one is below the scan line. Only for the case if both are above or both are below the scan line, then we will add 2 points.



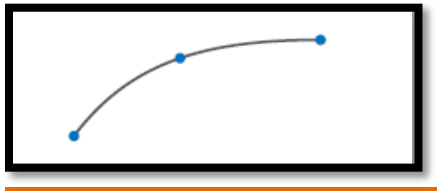
Interpolation

Interpolation Basics

Goal: We would like to be able to define curves in a way that meets the following criteria:

1. Interaction should be natural and intuitive.
2. Smoothness should be controllable.
3. Analytic derivatives should exist and be easy to compute.
4. Representation should be compact.

Interpolation is when a curve passes through a set of "control points."



Approximation is when a curve approximates but doesn't necessarily contain its control points.

Extrapolation is extending a curve beyond the domain of its control points.

Parametric Curves and Surfaces

Parametric Curves

Designing Curves

- We don't want only polygons.
- Curves are used for design. Users require a simple set of controls to allow them to edit and design curves easily.
- Curves should have infinite resolution, so we can zoom in and still see a smooth curve.
- We want to have a compact representation.

Parametric functions are of the form x (t) = f(t) and y(t) = g(t) in two dimensions. This can be extended for arbitrary dimensions. They can be used to model curves that are *not* functions of any axis in the plane.

Curves can be defined as polynomials, for example x(t) = 5t10 + 4t9 + 3t8 + ... However, coefficients are not intuitive editing parameters, and these curves are difficult to control. Hence, we will consider more intuitive parameterizations.

Bézier curve

A Bézier curve is a parametric curve frequently used in computer graphics and related fields. Generalizations of Bézier curves to higher dimensions are called Bézier surfaces, of which the Bézier triangle is a special case.

Linear Bézier curves

Given points P_0 and P_1, a linear Bézier curve is simply a straight line between those two points. The curve is given by

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , \ t \in [0,1]$$

and is equivalent to linear interpolation.

Quadratic Bézier curves

A quadratic Bézier curve is the path traced by the function B(*t*), given points P_0, P_1, and P_2,

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \ , \ t \in [0,1],$$

which can be interpreted as the linear interpolant of corresponding points on the linear Bézier curves from P_0 to P_1 and from P_1 to P_2 respectively. Rearranging the preceding equation yields:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \ , \ t \in [0,1].$$

The derivative of the Bézier curve with respect to *t* is

$$\mathbf{B}'(t) = 2(1-t)(\mathbf{P}_1 - \mathbf{P}_0) + 2t(\mathbf{P}_2 - \mathbf{P}_1)\,.$$

From which it can be concluded that the tangents to the curve at P_0 and P_2 intersect at P_1. As *t* increases from 0 to 1, the curve departs from P_0 in the direction of P_1, then bends to arrive at P_2 from the direction of P_1.

The second derivative of the Bézier curve with respect to *t* is

$$\mathbf{B}''(t) = 2(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0)\,.$$

A quadratic Bézier curve is also a parabolic segment. As a parabola is a conic section, some sources refer to quadratic Béziers as "conic arcs".[2]

Cubic Bézier curves

Four points P_0, P_1, P_2 and P_3 in the plane or in higher-dimensional space define a cubic Bézier curve. The curve starts at P_0 going toward P_1 and arrives at P_3 coming from the direction of P_2. Usually, it will not pass through P_1 or P_2; these points are only there to provide directional information. The distance between P_0 and P_1 determines "how long" the curve moves into direction P_2 before turning towards P_3.

Writing $B_{P_i,P_j,P_k}(t)$ for the quadratic Bézier curve defined by points P_i, P_j, and P_k, the cubic Bézier curve can be defined as a linear combination of two quadratic Bézier curves:

$$\mathbf{B}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2}(t) + t\mathbf{B}_{\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3}(t) \ , \ t \in [0,1].$$

The explicit form of the curve is:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \ , \ t \in [0,1].$$

For some choices of P_1 and P_2 the curve may intersect itself, or contain a cusp.

Any series of any 4 distinct points can be converted to a cubic Bézier curve that goes through all 4 points in order. Given the starting and ending point of some cubic Bézier curve, and the points along the curve corresponding to $t = 1/3$ and $t = 2/3$, the control points for the original Bézier curve can be recovered.[3]

The derivative of the cubic Bézier curve with respect to *t* is

$$\mathbf{B}'(t) = 3(1-t)^2(\mathbf{P}_1 - \mathbf{P}_0) + 6(1-t)t(\mathbf{P}_2 - \mathbf{P}_1) + 3t^2(\mathbf{P}_3 - \mathbf{P}_2)\,.$$

The second derivative of the Bézier curve with respect to t is

$$\mathbf{B}''(t) = 6(1-t)(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0) + 6t(\mathbf{P}_3 - 2\mathbf{P}_2 + \mathbf{P}_1)\,.$$

Recursive definition

A recursive definition for the Bézier curve of degree n expresses it as a point-to-point linear combination (linear interpolation) of a pair of corresponding points in two Bézier curves of degree $n - 1$.

Let $\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}$ denote the Bézier curve determined by any selection of points P_0, P_1, ..., P_n. Then to start,

$$\mathbf{B}_{\mathbf{P}_0}(t) = \mathbf{P}_0, \text{ and}$$

$$\mathbf{B}(t) = \mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_{n-1}}(t) + t\mathbf{B}_{\mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n}(t)$$

Explicit definition

The formula can be expressed explicitly as follows:

$$\begin{aligned}\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i\mathbf{P}_i \\ &= (1-t)^n\mathbf{P}_0 + \binom{n}{1}(1-t)^{n-1}t\mathbf{P}_1 + \cdots \\ &\quad \cdots + \binom{n}{n-1}(1-t)t^{n-1}\mathbf{P}_{n-1} + t^n\mathbf{P}_n, \quad t \in [0,1]\end{aligned}$$

Where $\binom{n}{i}$ are the binomial coefficients.

For example, for $n = 5$:

$$\begin{aligned}\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3\mathbf{P}_4\mathbf{P}_5}(t) = \mathbf{B}(t) &= (1-t)^5\mathbf{P}_0 + 5t(1-t)^4\mathbf{P}_1 + 10t^2(1-t)^3\mathbf{P}_2 \\ &\quad + 10t^3(1-t)^2\mathbf{P}_3 + 5t^4(1-t)\mathbf{P}_4 + t^5\mathbf{P}_5, \quad t \in [0,1]\end{aligned}$$

Terminology

Some terminology is associated with these parametric curves. We have

$$\mathbf{B}(t) = \sum_{i=0}^{n} b_{i,n}(t)\mathbf{P}_i, \quad t \in [0,1]$$

Where the polynomials

$$b_{i,n}(t) = \binom{n}{i}t^i(1-t)^{n-i}, \quad i = 0, \ldots, n$$

are known as Bernstein basis polynomials of degree n.

Note that $t^0 = 1$, $(1-t)^0 = 1$, and that the binomial coefficient, $\binom{n}{i}$, also expressed as ${}^n\mathbf{C}_i$ or $\mathbf{C}_i^{\,n}$ is:

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

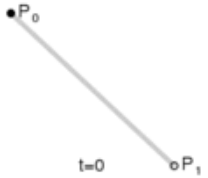
The points P_i are called *control points* for the Bézier curve. The polygon formed by connecting the Bézier points with lines, starting with P_0 and finishing with P_n, is called the *Bézier polygon* (or *control polygon*). The convex hull of the Bézier polygon contains the Bézier curve.

Properties

- The curve begins at P_0 and ends at P_n; this is the so-called *endpoint interpolation* property.
- The curve is a straight line if and only if all the control points are collinear.
- The start (end) of the curve is tangent to the first (last) section of the Bézier polygon.
- A curve can be split at any point into two subcurves, or into arbitrarily many subcurves, each of which is also a Bézier curve.
- Some curves that seem simple, such as the circle, cannot be described exactly by a Bézier or piecewise Bézier curve; though a four-piece cubic Bézier curve can approximate a circle (see Bézier spline), with a maximum radial error of less than one part in a thousand, when each inner control point (or offline point) is the distance $\frac{4\left(\sqrt{2}-1\right)}{3}$ horizontally or vertically from an outer control point on a unit circle. More generally, an *n*-piece cubic Bézier curve can approximate a circle, when each inner control point is the distance $\frac{4}{3}\tan(t/4)$ from an outer control point on a unit circle, where *t* is 360/*n* degrees, and $n > 2$.
- The curve at a fixed offset from a given Bézier curve, often called an *offset curve* (lying "parallel" to the original curve, like the offset between rails in a railroad track), cannot be exactly formed by a Bézier curve (except in some trivial cases). However, there are heuristic methods that usually give an adequate approximation for practical purposes. (For example: [1])
- Every quadratic Bézier curve is also a cubic Bézier curve, and more generally, every degree *n* Bézier curve is also a degree *m* curve for any $m > n$. In detail, a degree *n* curve with control points $P_0, \ldots, P_n$ is equivalent (including the parametrization) to the degree $n + 1$ curve with control points $P'_0, \ldots, P'_{n+1}$, where $\mathbf{P}'_k = \frac{k}{n+1}\mathbf{P}_{k-1} + \left(1 - \frac{k}{n+1}\right)\mathbf{P}_k$.
- Bézier curves follow the Variation diminishing property.

Constructing Bézier curves

Linear curves



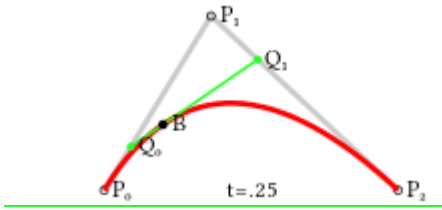
Animation of a linear Bézier curve, *t* in [0,1]

The *t* in the function for a linear Bézier curve can be thought of as describing how far B(*t*) is from P_0 to P_1. For example when *t=0.25*, B(*t*) is one quarter of the way from point P_0 to P_1. As *t* varies from 0 to 1, B(*t*) describes a straight line from P_0 to P_1.

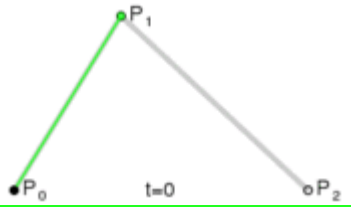
Quadratic curves

For quadratic Bézier curves one can construct intermediate points Q_0 and Q_1 such that as *t* varies from 0 to 1:

- Point Q_0 varies from P_0 to P_1 and describes a linear Bézier curve.
- Point Q_1 varies from P_1 to P_2 and describes a linear Bézier curve.
- Point B(*t*) varies from Q_0 to Q_1 and describes a quadratic Bézier curve.



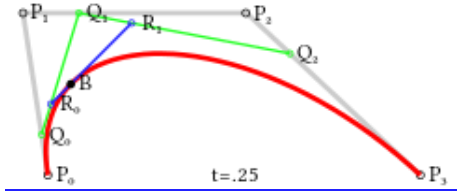
Construction of a quadratic Bézier curve



Animation of a quadratic Bézier curve, *t* in [0,1]

Higher-order curves

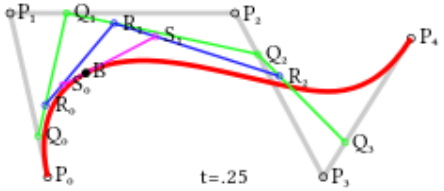
For higher-order curves one needs correspondingly more intermediate points. For cubic curves one can construct intermediate points Q_0, Q_1, and Q_2 that describe linear Bézier curves, and points R_0 & R_1 that describe quadratic Bézier curves:



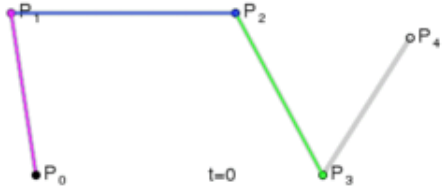
Construction of a cubic Bézier curve

Animation of a cubic Bézier curve, *t* in [0,1]

For fourth-order curves one can construct intermediate points Q_0, Q_1, Q_2 & Q_3 that describe linear Bézier curves, points R_0, R_1 & R_2 that describe quadratic Bézier curves, and points S_0 & S_1 that describe cubic Bézier curves:

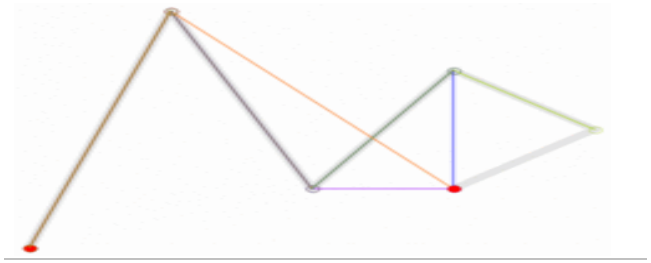


Construction of a quartic Bézier curve



Animation of a quartic Bézier curve, *t* in [0,1]

For fifth-order curves, one can construct similar intermediate points.



Animation of a fifth order Bézier curve, *t* in [0,1]

These representations rest on the process used in De Casteljau's algorithm to calculate Bézier curves.

B-spline

In the mathematical subfield of numerical analysis, a B-spline, or Basis spline, is a spline function that has minimal support with respect to a given degree, smoothness, and domain partition. Any spline function of given degree can be expressed as a linear combination of B-splines of that degree. Cardinal B-splines have knots that are equidistant from each other. B-splines can be used for curve-fitting and numerical differentiation of experimental data.

A B-spline is a piecewise polynomial function of degree k in a variable x. It is defined over a range $t_0 \le x \le t_m$, m = k+1. The points where $x = t_j$ are known as knots or break-points. The number of internal knots is equal to the degree of the polynomial. The knots must be in ascending order. The number of knots is the minimum for the degree of the B-spline, which has a non-zero value only in the range between the first and last knot. Each piece of the function is a polynomial of degree k between and including adjacent knots.

For any given set of knots, the B-spline is unique, hence the name, B being short for Basis. The usefulness of B-splines lies in the fact that any spline function of degree k on a given set of knots can be expressed as a linear combination of B-splines.

$$S_{k,t}(x) = \sum_i \alpha_i B_{i,k}(x)$$

This follows from the fact that all pieces have the same continuity properties, within their individual range of support, at the knots.

Expressions for the polynomial pieces can be derived by means of a recursion formula

$$B_{i,1}(x) := \begin{cases} 1 & \text{if} \quad t_i \le x < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,k}(x) := \frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(x) + \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(x).$$

That is, $B_{j,1}(x)$ is piecewise constant one or zero indicating which knot span x is in (zero if knot span j is repeated). The recursion equation is in two parts:

$$\frac{x - t_i}{t_{i+k-1} - t_i}$$

Ramps from zero to one as x goes from t_i to t_{i+k-1} and

$$\frac{t_{i+k} - x}{t_{i+k} - t_{i+1}}$$

Ramps from one to zero as x goes from t_{i+1} to t_{i+k}. The corresponding Bs are zero outside those respective ranges. For example, $B_{i,2}(x)$ is a triangular function that is zero below $x = t_i$, ramps to one at $x = t_{i+1}$ and back to zero at and beyond $x = t_{i+2}$. However, because B-spline basis functions have local support, B-splines are typically computed by algorithms that do not need to evaluate basis functions where they are zero, such as de Boor's algorithm.

This relation leads directly to the FORTRAN-coded algorithm BSPLV which generates values of the B-splines of order *k* at *x*. The following scheme illustrates how each piece of degree *k* is a linear combination of the pieces of B-splines of degree *k*-1 to its left.

Derivative expressions

The derivative of a B-spline of degree *k* is simply a function of B-splines of degree *k*-1.

$$\frac{dB_{i,k}(x)}{dx} = (k-1)\left(\frac{-B_{i+1,k-1}(x)}{t_{i+k} - t_{i+1}} + \frac{B_{i.k-1}(x)}{t_{i+k-1} - t_i}\right)$$

This implies that

$$\frac{d}{dx}\sum_i \alpha_i B_{i,k} = \sum_{i=r-k+2}^{s-1} (k-1)\frac{\alpha_i - \alpha_{i-1}}{t_{i+k-1} - t_i} B_{i,k-1} \; on[t_r.t_s]$$

Which shows that there is a simple relationship between the derivative of a spline function and the B-splines of degree one less.

Fractal Geometry

The word "fractal" often has different connotations for laypeople than for mathematicians, where the layperson is more likely to be familiar with fractal art than a mathematical conception. The mathematical concept is difficult to define formally even for mathematicians, but key features can be understood with little mathematical background.

Fractals' properties

Two of the most important properties of fractals are self-similarity and non-integer dimension.

What does self-similarity mean? If you look carefully at a fern leaf, you will notice that every little leaf - part of the bigger one - has the same shape as the whole fern leaf. You can say that the fern leaf is self-similar. The same is with fractals: you can magnify them many times and after every step you will see the same shape, which is characteristic of that particular fractal.

The non-integer dimension is more difficult to explain. Classical geometry deals with objects of integer dimensions: zero dimensional points, one dimensional lines and curves, two dimensional plane figures such as squares and circles, and three dimensional solids such as cubes and spheres. However, many natural phenomena are better described using a dimension between two whole numbers. So while a straight line has a dimension of one, a fractal curve will have a dimension between one and two, depending on how much space it takes up as it twists and curves. The more the flat fractal fills a plane, the closer it approaches two dimensions. Likewise, a "hilly fractal scene" will reach a dimension somewhere between two and three. So a fractal landscape made up of a large hill covered with tiny mounds would be close to the second dimension, while a rough surface composed of many medium-sized hills would be close to the third dimension.

There are a lot of different types of fractals. In this paper I will present two of the most popular types: complex number fractals and Iterated Function System (IFS) fractals.

Mandelbrot set

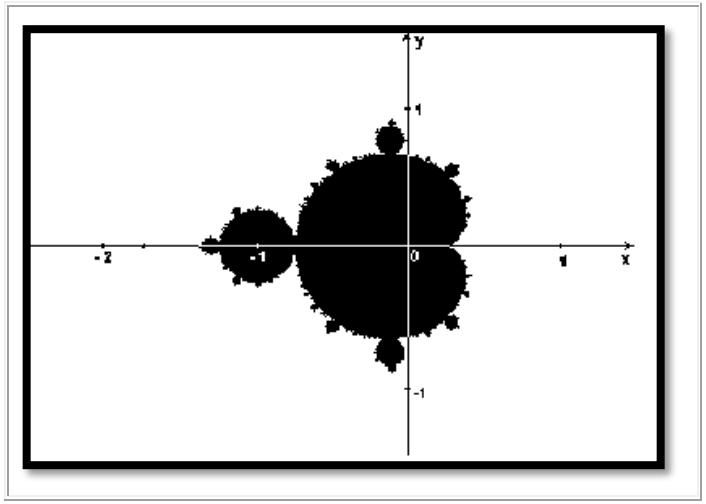
The Mandelbrot set is the set of points on a complex plain. To build the Mandelbrot set, we have to use an algorithm based on the recursive formula:

$$Z_n = Z_{n-1}^2 + C,$$

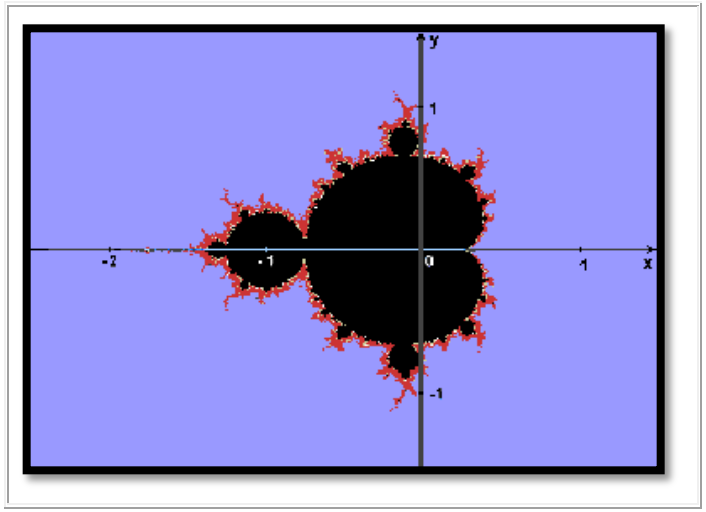
separating the points of the complex plane into two categories:

- points inside the Mandelbrot set,
- points outside the Mandelbrot set.

The image below shows a portion of the complex plane. The points of the Mandelbrot set have been colored black.



It is also possible to assign a color to the points outside the Mandelbrot set. Their colors depend on how many iterations have been required to determine that they are outside the Mandelbrot set.



How is the Mandelbrot set created?

To create the Mandelbrot set we have to pick a point (*C*) on the complex plane. The complex number corresponding with this point has the form: $C = a + b \cdot i$

After calculating the value of previous expression:

$$Z_1 = Z_0^2 + C$$

using zero as the value of Z_0, we obtain *C* as the result. The next step consists of assigning the result to Z_1 and repeating the calculation: now

the result is the complex number $C_2 + C$. Then we have to assign the value to Z_2 and repeat the process again and again.
This process can be represented as the "migration" of the initial point *C* across the plane. What happens to the point when we repeatedly iterate the function? Will it remain near to the origin or will it go away from it, increasing its distance from the origin without limit? In the first case, we say that *C* belongs to the Mandelbrot set (it is one of the black points in the image); otherwise, we say that it goes to infinity and we assign a color to *C* depending on the speed at which the point "escapes" from the origin.
We can take a look at the algorithm from a different point of view. Let us imagine that all the points on the plane are attracted by both: infinity and the Mandelbrot set. That makes it easy to understand why:

- points far from the Mandelbrot set rapidly move towards infinity,
- points close to the Mandelbrot set slowly escape to infinity,
- points inside the Mandelbrot set never escape to infinity.

Julia sets

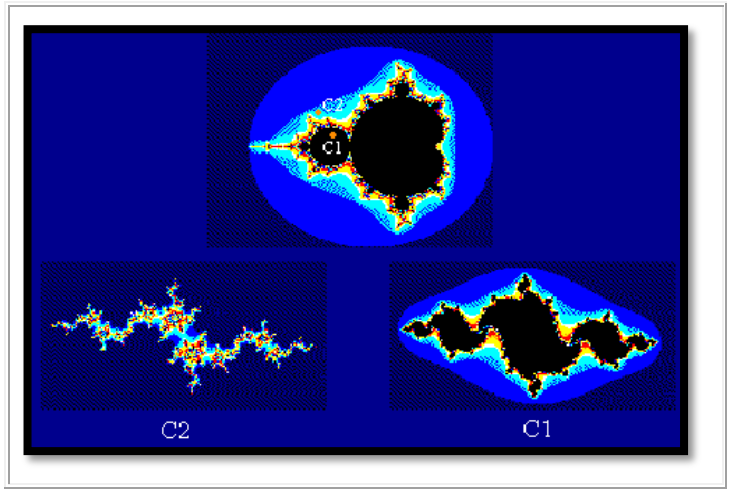
Julia sets are strictly connected with the Mandelbrot set. The iterative function that is used to produce them is the same as for the Mandelbrot set. The only difference is the way this formula is used. In order to draw a picture of the Mandelbrot set, we iterate the formula for each point *C* of the complex plane, always starting with $Z_0 = 0$. If we want to make a picture of a Julia set, *C* must be constant during the whole generation process, while the value of Z_0 varies. The value of *C* determines the shape of the Julia set; in other words, each point of the complex plane is associated with a particular Julia set.

How is a Julia set created?

We have to pick a point *C*) on the complex plane. The following algorithm determines whether or not a point on complex plane *Z*) belongs to the Julia set associated with *C*, and determines the color that should be assigned to it. To see if *Z* belongs to the set, we have to iterate the function $Z_1 = Z_0^2 + C$ using $Z_0 = Z$. What happens to the initial point Z when the formula is iterated? Will it remain near to the origin or will it go away from it, increasing its distance from the origin without limit? In the first case, it belongs to the Julia set; otherwise it goes to infinity and we assign a color to *Z* depending on the speed the point "escapes" from the origin. To produce an image of the whole Julia set associated with C, we must repeat this process for all the points *Z* whose coordinates are included in this range:

$$-2 < a < 2; \; -1{,}5 < y < 1{,}5$$

The most important relationship between Julia sets and Mandelbrot set is that while the Mandelbrot set is connected (it is a single piece), a Julia set is connected only if it is associated with a point inside the Mandelbrot set. For example: the Julia set associated with C_1 is connected; the Julia set associated with C_1 is not connected (see picture below).



Iterated Function System Fractals

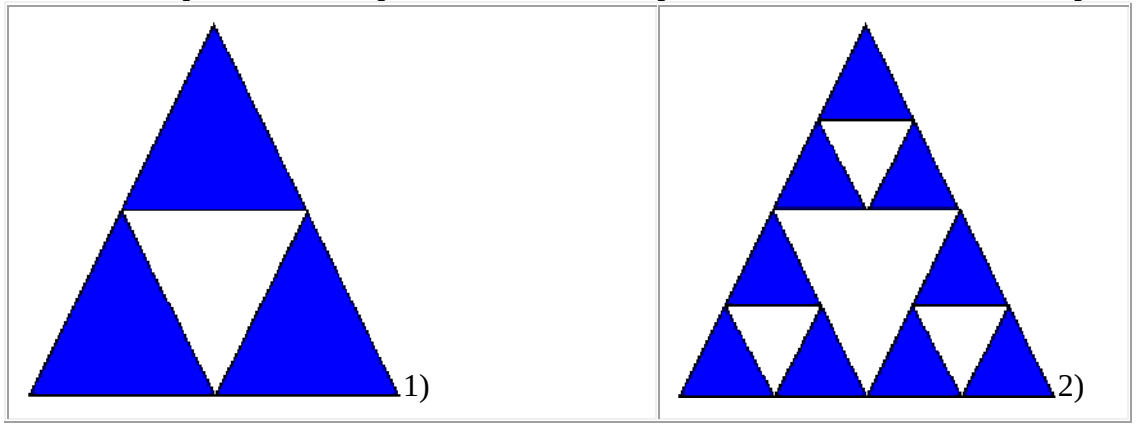
Iterated Function System (IFS) fractals are created on the basis of simple plane transformations: scaling, dislocation and the plane axes rotation. Creating an IFS fractal consists of following steps:

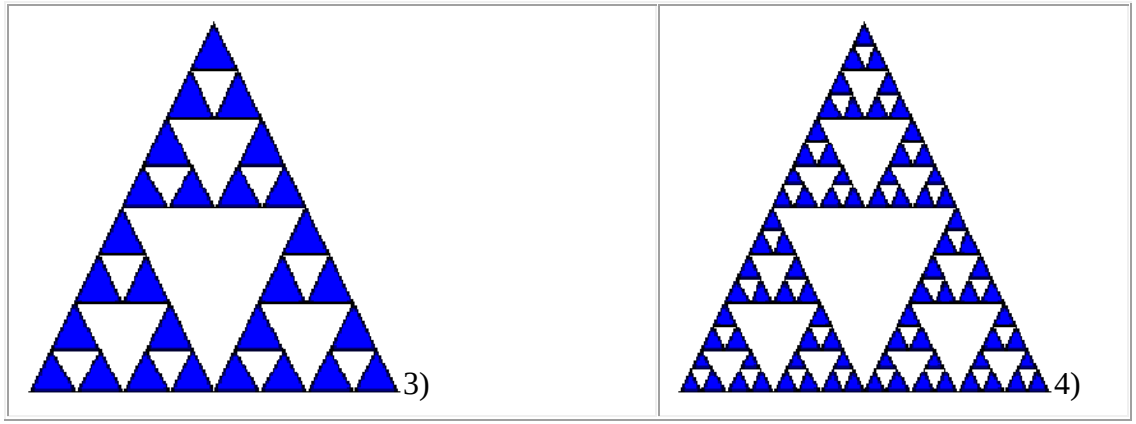
1. defining a set of plane transformations,
2. drawing an initial pattern on the plane (any pattern),
3. transforming the initial pattern using the transformations defined in first step,
4. transforming the new picture (combination of initial and transformed patterns) using the same set of transformations,
5. repeating the fourth step as many times as possible (in theory, this procedure can be repeated an infinite number of times).

The most famous ISF fractals are the Sierpinski Triangle and the Koch Snowflake.

Sierpinski Triangle

This is the fractal we can get by taking the midpoints of each side of an equilateral triangle and connecting them. The iterations should be repeated an infinite number of times. The pictures below present four initial steps of the construction of the Sierpinski Triangle:





Using this fractal as an example, we can prove that the fractal dimension is not an integer.
First of all we have to find out how the "size" of an object behaves when its linear dimension increases. In one dimension we can consider a line segment. If the linear dimension of the line segment is doubled, then the length (characteristic size) of the line has doubled also. In two dimensions, if the linear dimensions of a square for example is doubled then the characteristic size, the area, increases by a factor of *4*. In three dimensions, if the linear dimension of a box is doubled then the volume increases by a factor of *8*.
This relationship between dimension *D* , linear scaling *L* and the result of size increasing *S* can be generalized and written as:

$$S = L \cdot D$$

Rearranging of this formula gives an expression for dimension depending on how the size changes as a function of linear scaling:

$$D = \frac{log(S)}{log(L)}$$

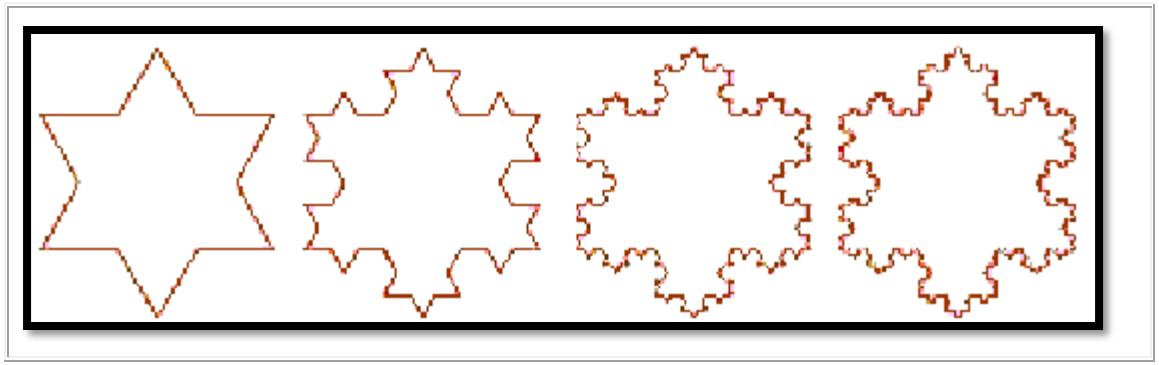
In the examples above the value of *D* is an integer □ *1*, *2*, or *3* □ depending on the dimension of the geometry. This relationship holds for all Euclidean shapes. How about fractals?
Looking at the picture of the first step in building the Sierpinski Triangle, we can notice that if the linear dimension of the basis triangle (*L*) is doubled, then the area of whole fractal (blue triangles) increases by a factor of three (*S*).

Using the pattern given above, we can calculate a dimension for the Sierpinski Triangle:

$$D = \frac{log(3)}{log(2)} = 1{,}585$$

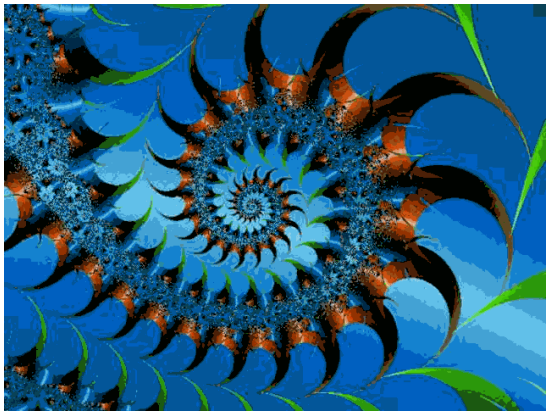
The result of this calculation proves the non-integer fractal dimension.
Koch Snowflake
To construct the Koch Snowflake, we have to begin with an equilateral triangle with sides of length, for example, *1*. In the middle of each side, we will add a new triangle one-third the size; and repeat this process for an infinite number of iterations. The length of the boundary is $3 \cdot \frac{4}{3} \cdot \frac{4}{3} \cdot \frac{4}{3} \cdot \cdots$ -infinity. However, the area remains less than the area of a circle drawn around the original triangle. That means that an infinitely long line surrounds a finite area. The end construction of a Koch Snowflake resembles the coastline of a shore.

Four steps of Koch Snowflake construction:



Another IFS fractals:

Fern leaf



Spiral

Fractals applications

Fractal geometry has permeated many area of science, such as astrophysics, biological sciences, and has become one of the most important techniques in computer graphics.

Fractals in astrophysics

Nobody really knows how many stars actually glitter in our skies, but have you ever wondered how they were formed and ultimately found their home in the Universe? Astrophysicists believe that the key to this problem is the fractal nature of interstellar gas. Fractal distributions are hierarchical, like smoke trails or billowy clouds in the sky. Turbulence shapes both the clouds in the sky and the clouds in space, giving them an irregular but repetitive pattern that would be impossible to describe without the help of fractal geometry.

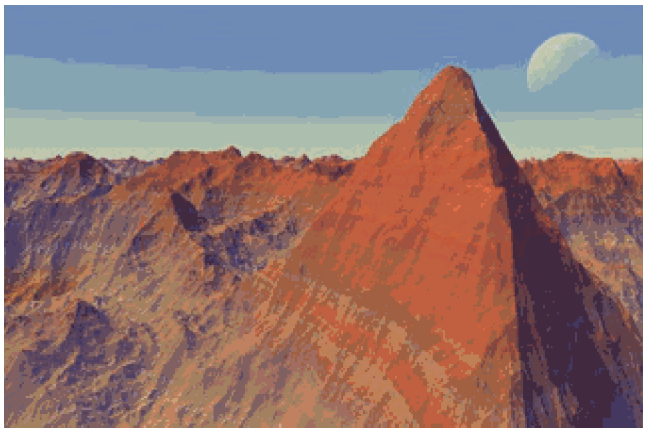
Fractals in the Biological Sciences

Biologists have traditionally modeled nature using Euclidean representations of natural objects or series. They represented heartbeats as sine waves, conifer trees as cones, animal habitats as simple areas, and cell membranes as curves or simple surfaces. However, scientists have come

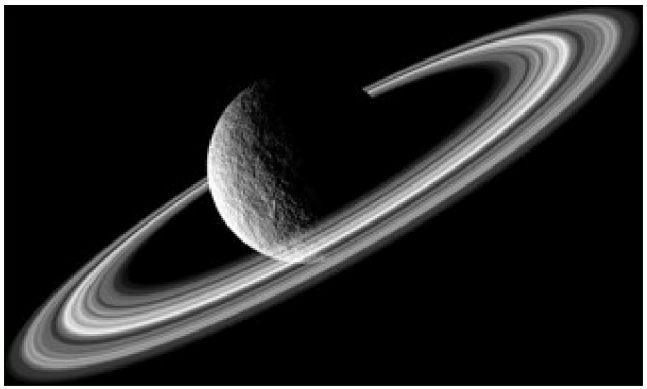
to recognize that many natural constructs are better characterized using fractal geometry. Biological systems and processes are typically characterized by many levels of substructure, with the same general pattern repeated in an ever-decreasing cascade.
Scientists discovered that the basic architecture of a chromosome is tree-like; every chromosome consists of many 'mini-chromosomes', and therefore can be treated as fractal. For a human chromosome, for example, a fractal dimension D equals 2,34 (between the plane and the space dimension).
Self-similarity has been found also in DNA sequences. In the opinion of some biologists fractal properties of DNA can be used to resolve evolutionary relationships in animals.
Perhaps in the future biologists will use the fractal geometry to create comprehensive models of the patterns and processes observed in nature.

Fractals in computer graphics

The biggest use of fractals in everyday live is in computer science. Many image compression schemes use fractal algorithms to compress computer graphics files to less than a quarter of their original size.
Computer graphic artists use many fractal forms to create textured landscapes and other intricate models.
It is possible to create all sorts of realistic "fractal forgeries" images of natural scenes, such as lunar landscapes, mountain ranges and coastlines. We can see them in many special effects in Hollywood movies and also in television advertisements. The "Genesis effect" in the film "Star Trek II - The Wrath of Khan" was created using fractal landscape algorithms, and in "Return of the Jedi" fractals were used to create the geography of a moon, and to draw the outline of the dreaded "Death Star". But fractal signals can also be used to model natural objects, allowing us to define mathematically our environment with a higher accuracy than ever before.



A fractal landscape



A fractal planet

Fractal Dimension

A **fractal dimension** is a ratio providing a statistical index of complexity comparing how detail in a pattern (strictly speaking, a fractal pattern) changes with the scale at which it is measured. It has also been characterized as a measure of the space-filling capacity of a pattern that tells how a fractal scales differently from the space it is embedded in; a fractal dimension does not have to be an integer.

Now we see an alternative way to specify the dimension of a self-similar object: The dimension is simply the exponent of the number of self-similar pieces with magnification factor **N** into which the figure may be broken.

So what is the dimension of the Sierpinski triangle? How do we find the exponent in this case? For this, we need logarithms. Note that, for the square, we have **N^2** self-similar pieces, each with magnification factor **N**. So we can write

$$\begin{aligned} \text{dimension} &= \frac{\log\ (\text{number of self-similar pieces})}{\log\ (\text{magnification factor})} \\ &= \frac{\log N^2}{\log N} \\ &= \frac{2\log N}{\log N} = 2 \end{aligned}$$

Similarly, the dimension of a cube is

$$\begin{aligned}\text{dimension} &= \frac{\log\ (\text{number of self-similar pieces})}{\log\ (\text{magnification factor})} \\ &= \frac{\log N^3}{\log N} \\ &= \frac{3\log N}{\log N} \\ &= 3\end{aligned}$$

Thus, we take as the *definition* of the fractal dimension of a self-similar object

$$\text{fractal dimension} = \frac{\log\ (\text{number of self-similar pieces})}{\log\ (\text{magnification factor})}.$$

Now we can compute the dimension of **S**. For the Sierpinski triangle consists of 3 self-similar pieces, each with magnification factor 2. So the fractal dimension is

$$\frac{\log\ (\text{number of self-similar pieces})}{\log\ (\text{magnification factor})} = \frac{\log 3}{\log 2} \approx 1.58$$

so the dimension of **S** is somewhere between 1 and 2, just as our ``eye" is telling us.

But wait a moment, **S** also consists of 9 self-similar pieces with magnification factor 4. No problem -- we have

$$\begin{aligned}\text{fractal dimension} &= \frac{\log 9}{\log 4} = \frac{\log 3^2}{\log 2^2} = \frac{2\log 3}{2\log 2} \\ &= \frac{\log 3}{\log 2} \approx 1.58\end{aligned}$$

as before. Similarly, **S** breaks into **3^N** self-similar pieces with magnification factors **2^N**, so we again have

$$\text{fractal dimension} = \frac{\log 3^N}{\log 2^N} = \frac{N\log 3}{N\log 2} = \frac{\log 3}{\log 2}$$

Fractal dimension is a measure of how "complicated" a self-similar figure is. In a rough sense, it measures "how many points" lie in a given set. A plane is "larger" than a line, while **S** sits somewhere in between these two sets.

Three Dimensional Modeling Transformations

Methods for geometric transforamtions and object modelling in 3D are extended from 2D methods by including the considerations for the z coordinate.
Basic geometric transformations are: Translation, Rotation, Scalin

Basic Transformations

Translation

We translate a 3D point by adding translation distances, tx, ty, and tz, to the original coordinate position (x,y,z):
x' = x + tx, y' = y + ty, z' = z + tz
Alternatively, translation can also be specified by the transformation matrix in the following formula:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Exercise: translate a triangle with vertices at original coordinates (10,25,5), (5,10,5), (20,10,10) by
tx=15, ty=5,tz=5. For verification, roughly plot the x and y values of the original and resultant triangles, and imagine the locations of z values.

Scaling With Respect to the Origin

We scale a 3D object with respect to the origin by setting the scaling factors sx, sy and sz, which are multiplied to the original vertex coordinate positions (x,y,z):
x' = x * sx, y' = y * sy, z' = z * sz

Alternatively, this scaling can also be specified by the transformation matrix in the following formula:

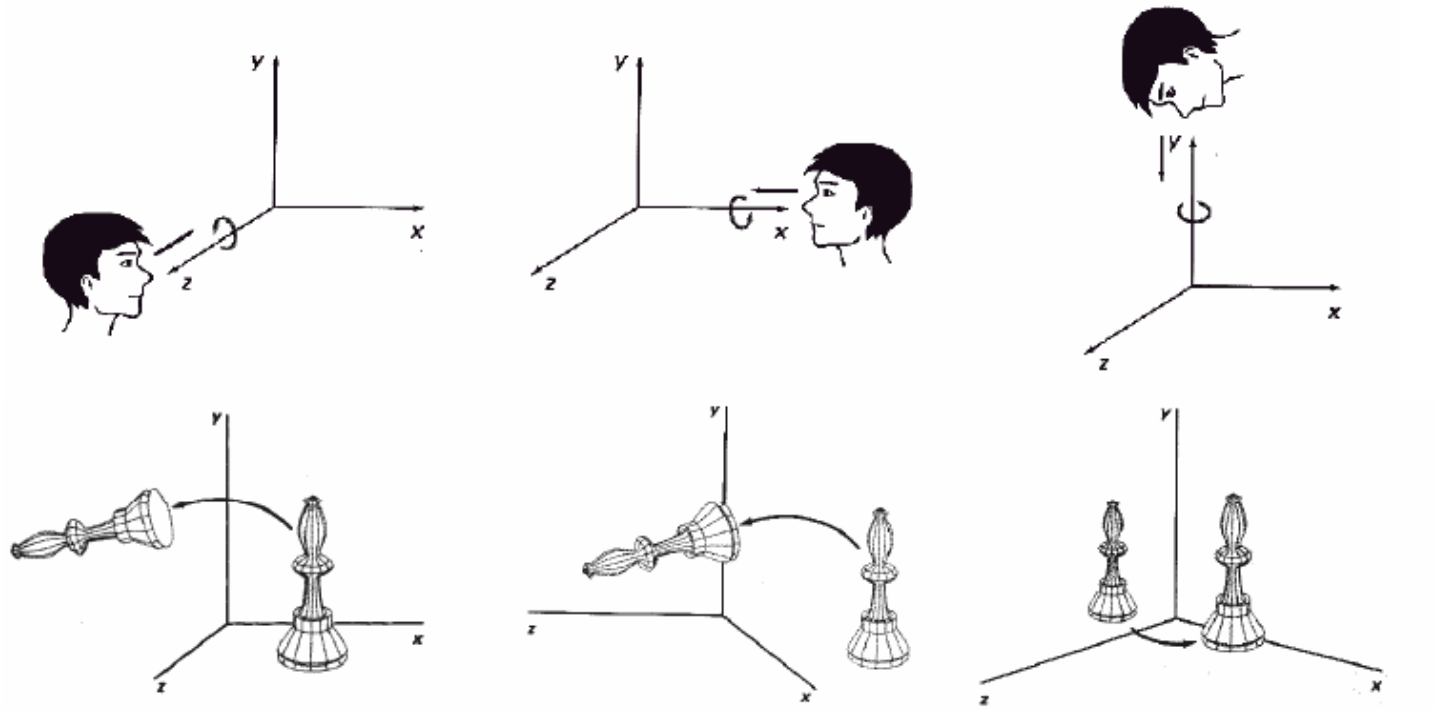
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Exercise: Scale a triangle with vertices at original coordinates (10,25, 5), (5,10,5), (20,10,10) by

sx=1.5, sy=2, and sz=0.5 with respect to the origin. For verification, roughly plot the x and y values of the original and resultant triangles, and imagine the locations of z values.

Coordinate-Axes Rotations

A 3D rotation can be specified around any line in space. The easiest rotation axes to handle are the coordinate axes.



Z-axis rotation:

$$x' = x\cos\theta - y\sin\theta,$$
$$y' = x\sin\theta + y\cos\theta, \text{ and}$$
$$z' = z$$

or:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

X-axis rotation:

$$y' = y \cos ? - z \sin ?,$$
$$z' = y \sin ? + z \cos ?, \text{ and}$$
$$x' = x$$

or:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

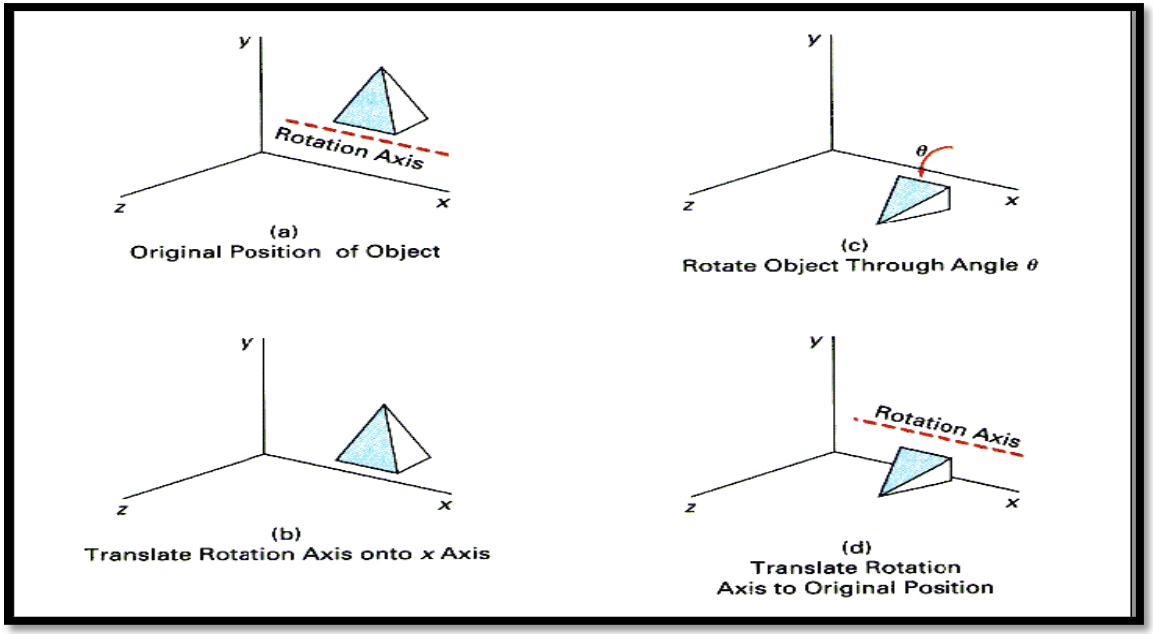
Y-axis rotation:

$$z' = z \cos ? - x \sin ?,$$
$$x' = z \sin ? + x \cos ?, \text{ and}$$
$$y' = y$$

or:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D **Rotations About an Axis Which is Parallel to an Axis**



Step 1:-

Translate the object so that the rotation axis coincides with the parallel coordinate axis.

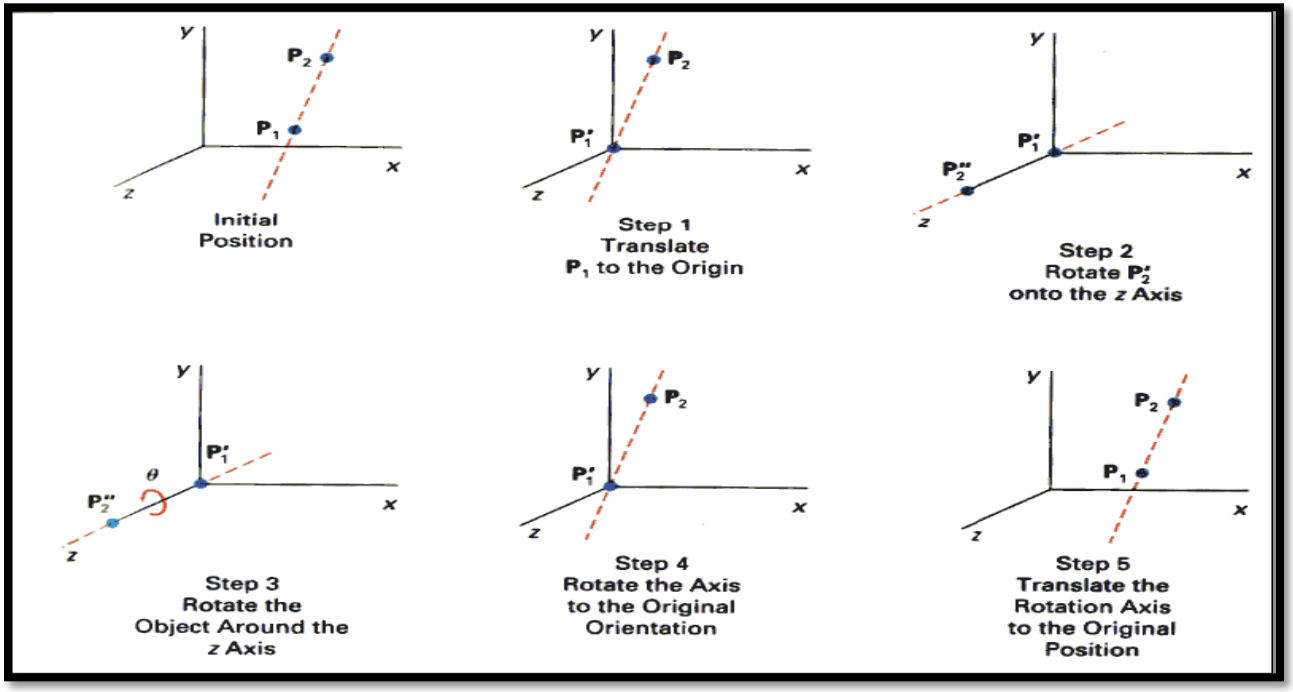
Step 2:-

Perform the specified rotation about that axis.

Step 3:-

Translate the object so that the rotation axis is moved back to its original position.

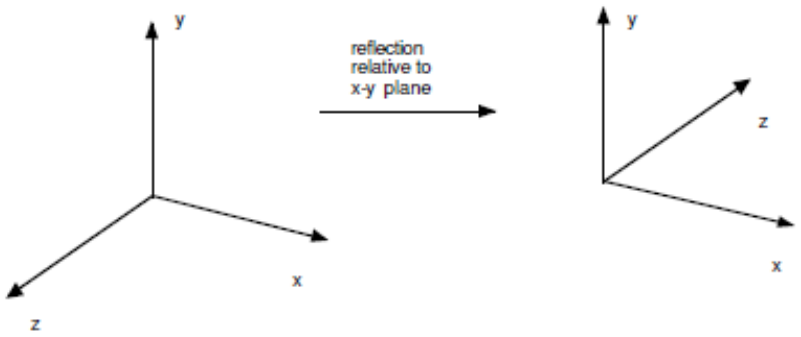
General 3D Rotations



Step 1:-
Translate the object so that the rotation axis passes through the coordinate origin.
Step 2:-
Rotate the object so that the axis of rotation coincides with one of the coordinate axes.
Step 3:-
Perform the specified rotation about that coordinate axis.
Step 4:-
Rotate the object so that the rotation axis is brought back to its original orientation.
Step 5:-
Translate the object so that the rotation axis is brought back to its original position.

Reflection

Reflection can be performed relative to a reflection axis or relative to a plane
• Reflection relative to a line is effectively a 180 deg rotation about that line
• Reflection relative to a plane is like a rotation in 4D - it is like a conversion between left-handed and right-handed coordinate systems.



Shear

Shear transformations are used for modifying shapes and for perspective transformations

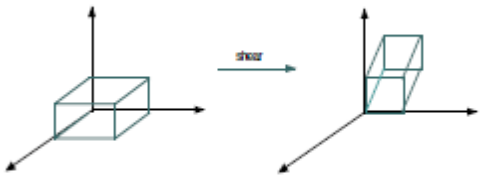
• Similar to shearing in 2D for x and y-axis shears; can also have z-axis shear.

• A z-axis shearing matrix

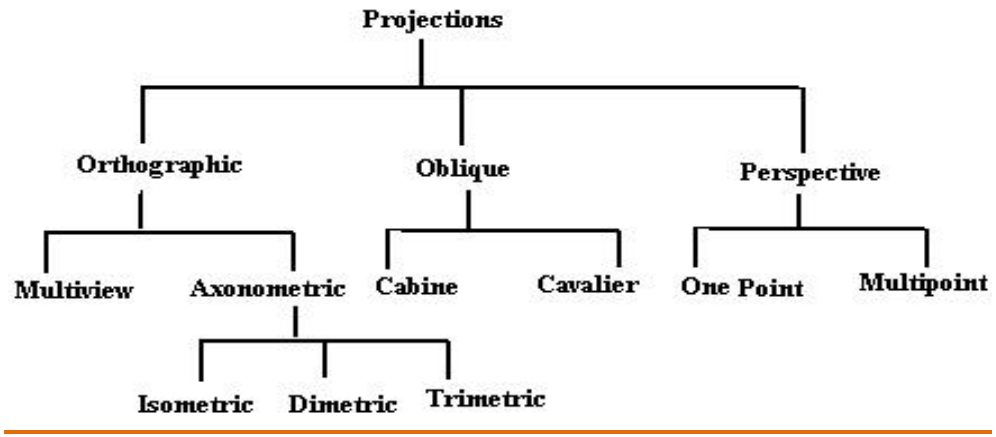
Example: shzx = shzy = 1 and

zref = (0, 0, 0)

$$\begin{bmatrix} 1 & 0 & sh_{zx} & -sh_{zx} \cdot z_{ref} \\ 0 & 1 & sh_{zy} & -sh_{zy} \cdot z_{ref} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$



Projections

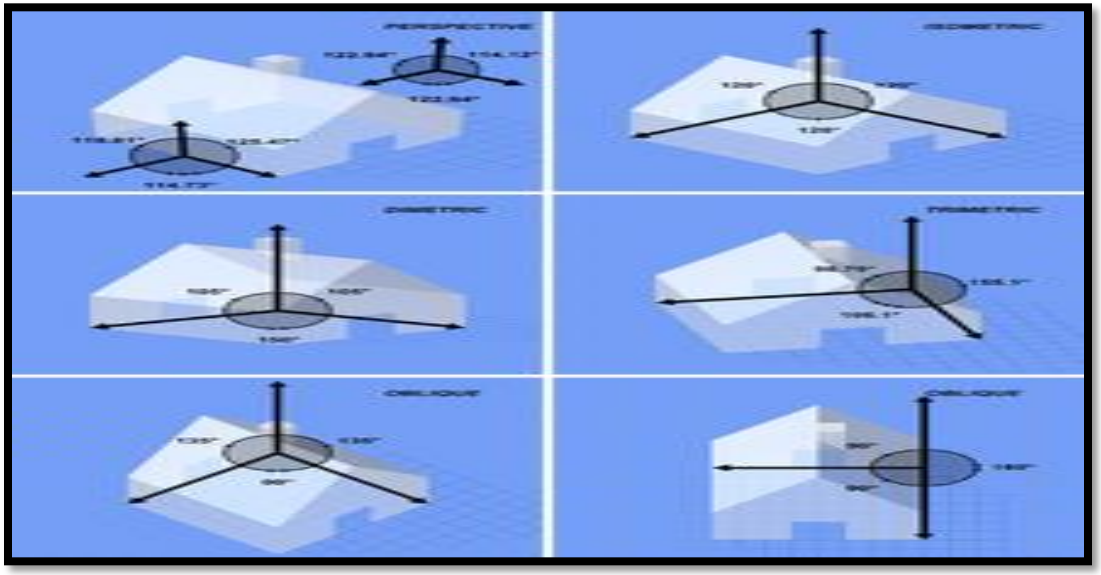


Parallel projections

Parallel projections have lines of projection that are parallel both in reality and in the projection plane. Parallel projection corresponds to a perspective projection with an infinite focal length (the distance from the image plane to the projection point), or "zoom". Within **parallel projection** there is an ancillary category known as "pictorials". Pictorials show an image of an object as viewed from a skew direction in order to reveal all three directions (axes) of space in one picture. Because pictorial projections innately contain this distortion, in the rote, drawing instrument for pictorials, some liberties may be taken for economy of effort and best effect.

Axonometric projection

Axonometric projection is a type of orthographic projection where the plane or axis of the object depicted is not parallel to the projection plane, such that multiple sides of an object are visible in the same image.[5] It is further subdivided into three groups: isometric, dimetric and trimetric projection, depending on the exact angle at which the view deviates from the orthogonal. A typical characteristic of axonometric pictorials is that one axis of space is usually displayed as vertical.



Comparison of several types of graphical projection.

1. **Isometric projection**

In **isometric pictorials** (for protocols see isometric projection), the most common form of axonometric projection,[3] the direction of viewing is such that the three axes of space appear equally foreshortened, of which the displayed angles among them and also the scale of foreshortening are universally known. However in creating a final, isometric instrument drawing, in most cases a full-size scale, i.e., without using a foreshortening factor, is employed to good effect because the resultant distortion is difficult to perceive.

2. **Dimetric projection**

In **dimetric pictorials** (for protocols see dimetric projection), the direction of viewing is such that two of the three axes of space appear equally foreshortened, of which the attendant scale and angles of presentation are determined according to the angle of viewing; the scale of the third direction (vertical) is determined separately. Approximations are common in dimetric drawings.

3. **Trimetric projection**

In **trimetric pictorials** (for protocols see trimetric projection), the direction of viewing is such that all of the three axes of space appear unequally foreshortened. The scale along each of the three axes and the angles among them are determined separately as dictated by the angle of viewing. Approximations in trimetric drawings are common,[clarification needed] and trimetric perspective is seldom used.[4]

Oblique projection

In oblique projections the parallel projection rays are not perpendicular to the viewing plane as with orthographic projection, but strike the projection plane at an angle other than ninety degrees.[1] In both orthographic and oblique projection, parallel lines in space appear parallel on the projected image. Because of its simplicity, oblique projection is used exclusively for pictorial purposes rather than for formal, working drawings. In an oblique pictorial drawing, the displayed angles among the axes as well as the foreshortening factors (scale) are arbitrary. The distortion created thereby is usually attenuated by aligning one plane of the imaged object to be parallel with the plane of projection thereby creating a true shape, full-size image of the chosen plane. Special types of oblique projections are cavalier projection and cabinet projection.[2]

Limitations

Objects drawn with parallel projection do not appear larger or smaller as they extend closer to or away from the viewer. While advantageous for architectural drawings, where measurements must be taken directly from the image, the result is a perceived distortion, since unlike perspective projection, this is not how our eyes or photography normally work. It also can easily result in situations where depth and altitude are difficult to gauge, as is shown in the illustration to the right.

In this isometric drawing, the blue sphere is two units higher than the red one. However, this difference in elevation is not apparent if one covers the right half of the picture, as the boxes (which serve as clues suggesting height) are then obscured.

Perspective Projection

Perspective in the graphic arts, such as drawing, is an approximate representation, on a flat surface (such as paper), of an image as it is seen by the eye. The two most characteristic features of perspective are that objects are drawn:

- Smaller as their distance from the observer increases.
- Foreshortened: the size of an object's dimensions along the line of sight are relatively shorter than dimensions across the line of sight
- Perspective drawings have a horizon line, which is often implied. This line, directly opposite the viewer's eye, represents objects infinitely far away. They have shrunk, in the distance, to the infinitesimal thickness of a line. It is analogous to (and named after) the Earth's horizon.
- Any perspective representation of a scene that includes parallel lines has one or more vanishing points in a perspective drawing. A one-point perspective drawing means that the drawing has a single vanishing point, usually (though not necessarily) directly opposite the viewer's eye and usually (though not necessarily) on the horizon line. All lines parallel with the viewer's line of sight recede to the horizon towards this vanishing point. This is the standard "receding railroad tracks" phenomenon. A two-point drawing would have lines parallel to two different angles. Any number of vanishing points are possible in a drawing, one for each set of parallel lines that are at an angle relative to the plane of the drawing.
- Perspectives consisting of many parallel lines are observed most often when drawing architecture (architecture frequently uses lines parallel to the x, y, and z axes). Because it is rare to have a scene consisting solely of lines parallel to the three Cartesian axes (x, y, and z), it is rare to see perspectives in practice with only one, two, or three vanishing points; even a simple house frequently has a peaked roof which results in a minimum of six sets of parallel lines, in turn corresponding to up to six vanishing points.
- In contrast, natural scenes often do not have any sets of parallel lines and thus no vanishing points.

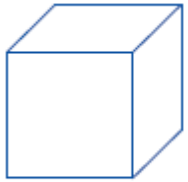
Orthographic projections:

When the observer is at a finite distance from the object, the visual rays or the projectors converge to the eye. But if the observer is imagined at an infinite distance from the transparent plane or the plane projection, the projectors will be parallel and will be perpendicular to the plane of projection as shown in fig (3.1). This projection is called orthographic projection.

Backface Removal

Consider a closed polyhedral object. Because it is closed, far side of the object will always be invisible, blocked by the near side. This observation can be used to accelerate rendering, by removing **back-faces.**

Example:
For this simple view of a cube, we have three backfacing polygons, the left side, back, and bottom:



Only the near faces are visible.

We can determine if a face is back-facing as follows. Suppose we compute a normals ~n for a mesh face, with the normal chosen so that it points outside the object For a surface point $\bar{}p$ on a planar patch and eye point $\bar{}e$, if $(\bar{}p - \bar{}e) \cdot \sim n > 0$, then the angle between the view direction and normal is less than 90◦, so the surface normal points away from $\bar{}e$. The result will be the same no matter which face point $\bar{}p$ we use. Patch and eye point $\bar{}e$, if $(\bar{}p - \bar{}e) \cdot \sim n > 0$, then the angle between the view direction and normal is less than 90◦, so the surface normal points away from $\bar{}e$. The result will be the same no matter which face point $\bar{}p$ we use.

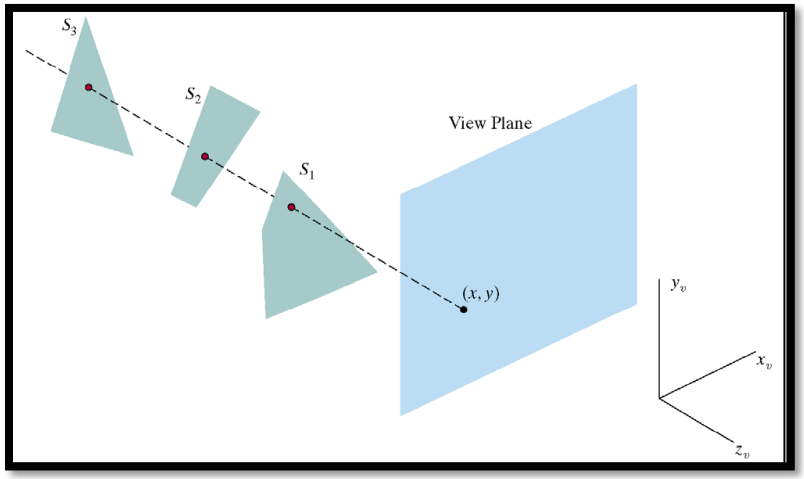
Backface removal is a “quick reject” used to accelerate rendering. It must still be used together with another visibility method. The other methods are more expensive, and removing backfaces just reduces the number of faces that must be considered by a more expensive method.

The Depth Buffer

Normally when rendering, we compute an image buffer I(i,j) that stores the color of the object that projects to pixel (i, j). The depth d of a pixel is the distance from the eye point to the object.
The **depth buffer** is an array zbuf(i, j) which stores, for each pixel (i, j), the depth of the nearest point drawn so far. It is initialized by setting all depth buffer values to infinite depth:

zbuf(i,j)= ∞.
To draw color c at pixel (i, j) with depth d:
if d < zbuf(i, j) then
putpixel(i, j, c)
zbuf(i, j) = d
end

When drawing a pixel, if the new pixel's depth is greater than the current value of the depth buffer at that pixel, then there must be some object blocking the new pixel, and it is not drawn.



Advantages

• Simple and accurate
• Independent of order of polygons drawn

Disadvantages

• Memory required for depth buffer
• Wasted computation on drawing distant points that are drawn over with closer points that occupy the same pixel

To represent the depth at each pixel, we can use pseudodepth, which is available after the homogeneous perspective transformation.1 Then the depth buffer should be initialized to 1, since the pseudodepth values are between −1 and 1. Pseudo depth gives a number of numerical advantages over true depth.

A-Buffer Algorithm

In computer graphics, **A-buffer**, also known as **anti-aliased**, **area-averaged** or **accumulation buffer**, is a general hidden surface mechanism suited to medium scale virtual memory computers. It resolves visibility among an arbitrary collection of opaque, transparent, and intersecting objects. Using an easy to compute Fourier window (box filter), it increases the effective image resolution many times over the Z-buffer, with a moderate increase in cost.

The A-buffer method is a descendant of the well known Z-buffer, which provides good quality results in moderate time.

Carpenter's A-buffer algorithm [CARP84] addresses this problem by approximating Catmull's per-pixel object-precision area sampling with per-pixel image-precision operation performed on a sub-pixel grid. Polygons are first processed in scan-line order by clipping them to each square pixel they cover. This result in list of clipped polygon fragments corresponding to each square pixel. Each fragment have 4 by 8 bit mask of parts of the pixel it covers.

The bit-mask for a fragment is computed by xoring together masks representing each of the fragment's edges. When all polygons intersecting a pixel have been processed, the area-weighted average of the colors of the pixel's visible surfaces is obtained by selecting fragments in depth-sorted order and using their bit masks to clip those of farther fragments.

The bit masks can be manipulated efficiently with Boolean operations. For example, two fragment bit masks can be added together to determine the overlap between them. The A-buffer algorithm saves only a small amount of additional information with each fragment. For example. It includes the fragment's z extent, but no information about which part of the fragment is associated with these z values. Thus, the algorithm must make assumptions about the sub-pixel geometry in cases in which fragment bit masks overlap in z.

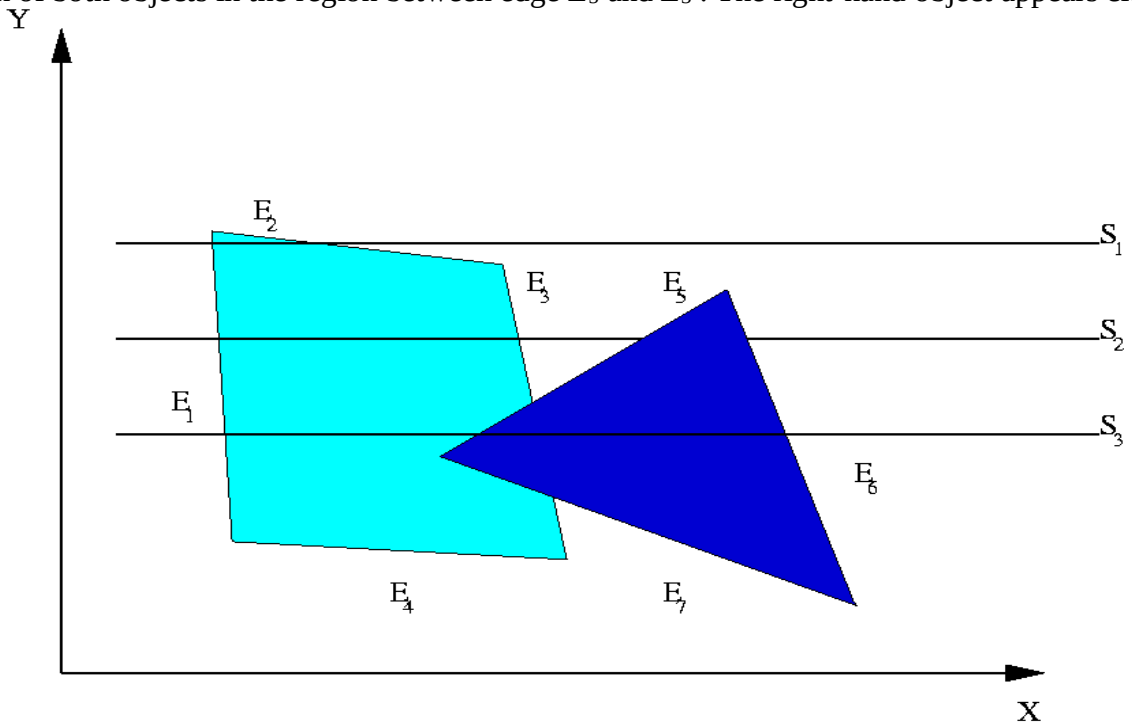
Scanline Algorithm

The scan-line algorithm is another image-space algorithm. It processes the image one scan-line at a time rather than one pixel at a time. By using area coherence of the polygon, the processing efficiency is improved over the pixel oriented method.

Using an active edge table, the scan-line algorithm keeps track of where the projection beam is at any given time during the scan-line sweep. When it enters the projection of a polygon, an IN flag goes on, and the beam switches from the background colour to the colour of the polygon. After the beam leaves the polygon's edge, the colour switches back to background colour. To this point, no depth information need be calculated at all. However, when the scan-line beam finds itself in two or more polygons, it becomes necessary to perform a z-depth sort and select the colour of the nearest polygon as the painting colour.

Accurate bookkeeping is very important for the scan-line algorithm. We assume the scene is defined by at least a polygon table containing the (A, B, C, D) coefficients of the plane of each polygon, intensity/colour information, and pointers to an edge table specifying the bounding lines of the polygon. The edge table contains the coordinates of the two end points, pointers to the polygon table to indicate which polygons the edge bounds, and the inverse slope of the x-y projection of the line for use with scan-line algorithms. In addition to these two standard data structures, the scan-line algorithm requires an active edge list that keeps track of which edges a given scan line intersects during its sweep. The active edge list should be sorted in order of increasing x at the point of intersection with the scan line. The active edge list is dynamic, growing and shrinking as the scan line progresses down the screen.

In Figure scan-line S_1 must deal only with the left-hand object. S_2 must plot both objects, but there is no depth conflict. S_3 must resolve the relative z-depth of both objects in the region between edge E_5 and E_3 . The right-hand object appears closer.



The active edge list for scan line S_1 contains edges E_1 and E_2 . From the left edge of the viewport to edge E_1 , the beam paints the background colour. At edge E_1 , the IN flag goes up for the left-hand polygon, and the beam switches to its colour until it crosses edge E_2 , at which point the IN flag goes down and the colour returns to background.

For scan-line S_2 , the active edge list contains E_1 , E_3 , E_5 , and E_6 . The IN flag goes up and down twice in sequence during this scan. Each time it goes up pointers identify the appropriate polygon and look up the colour to use in painting the polygon.

For scan line S_3 , the active edge list contains the same edges as for S_2 , but the order is altered, namely E_1 , E_5 , E_3 , E_6 . Now the question of relative z-depth first appears. The IN flag goes up once when we cross E_1 and again when we cross E_5 , indicating that the projector is piercing two polygons. Now the coefficients of each plane and the (x,y) of the E_5 edge are used to compute the depth of both planes. In the example

shown the z-depth of the right-hand plane was smaller, indicating it is closer to the screen. Therefore the painting colour switches to the right-hand polygon colour which it keeps until edge E_6 .

Note that the technique is readily extended to three or more overlapping polygons and that the relative depths of overlapping polygons must be calculated only when the IN flag goes up for a new polygon. Since this occurrence is far less frequent than the number of pixels per scan line, the scan-line algorithm is more computationally efficient than the z-buffer algorithm.

The scan-line hidden surface removal algorithm can be summarized as:

1. Establish the necessary data structures.

    a. Polygon table with coefficients, colour, and edge pointers.
    b. Edge table with line end points, inverse slope, and polygon pointers.

    c. Active edge list, sorted in order of increasing x.
    d. An IN flag for each polygon.

1. Repeat for all scan lines:

    c. Update active edge list by sorting edge table against scan line y value.
    d. Scan across, using background colour, until an IN flag goes on.
    e. When 1 polygon flag is on for surface P , enter intensity (colour) into refresh buffer.
    f. When 2 or more surface flags are on, do depth sort and use intensity I_n for surface n with minimum z-depth.
    g. Use coherence of planes to repeat for next scan line.

    The scan-line algorithm for hidden surface removal is well designed to take advantage of the area coherence of polygons. As long as the active edge list remains constant from one scan to the next, the relative structure and orientation of the polygons painted during that scan does not change. This means that we can "remember" the relative position of overlapping polygons and need not recompute the z-depth when two or more IN flags go on. By taking advantage of this coherence we save a great deal of computation.

Painter's Algorithm

The **painter's algorithm** is an alternative to depth buffering to attempt to ensure that the closest points to a viewer occlude points behind them. The idea is to draw the most distant patches of a surface first, allowing nearer surfaces to be drawn over them.
In the heedless painter's algorithm, we first sort faces according to depth of the vertex furthest from the viewer. Then faces are rendered from furthest to nearest. There are problems with this approach, however. In some cases, a face that occludes part of another face can still have its furthest vertex further from the viewer than any vertex of the face it occludes.
In this situation, the faces will be rendered out of order. Also, polygons cannot intersect at all as they can when depth buffering is used instead. One solution is to split triangles, but doing this correctly is very complex and slow. Painter's algorithm is rarely used directly in practice; however, a data-structure called BSP trees can be used to make painter's algorithm much more appealing.

The idea behind the Painter's algorithm is to draw polygons far away from the eye first, followed by drawing those that are close to the eye. Hidden surfaces will be written over in the image as the surfaces that obscure them are drawn.

The concept is to map the objects of our scene from the world model to the screen somewhat like an artist creating an oil painting. First she paints the entire canvas with a background colour. Next, she adds the more distant objects such as mountains, fields, and trees. Finally, she creates the foreground with "near" objects to complete the painting. Our approach will be identical. First we sort the polygons according to their z-depth and then paint them to the screen, starting with the far faces and finishing with the near faces.

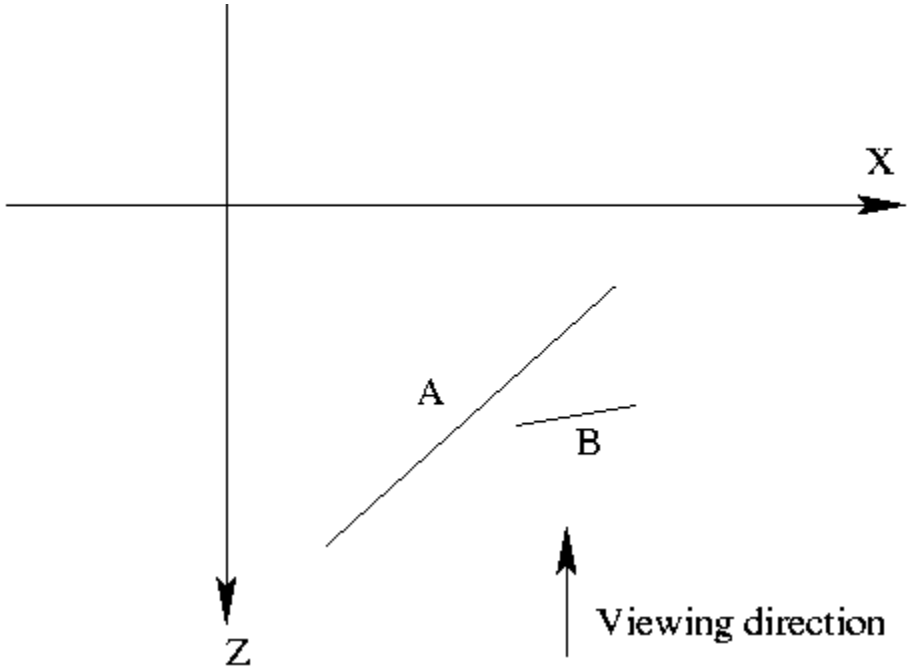
The algorithm initially sorts the faces in the object into back to front order. The faces are then scan converted in this order onto the screen. Thus a face near the front will obscure a face at the back by overwriting it at any points where their projections overlap. This accomplishes hidden-surface removal without any complex intersection calculations between the two projected faces.

The algorithm is a hybrid algorithm in that it sorts in object space and does the final rendering in image space.

The basic algorithm :

1. Sort all polygons in ascending order of maximum z-values.
2. Resolve any ambiguities in this ordering.
3. Scan convert each polygon in the order generated by steps (1) and (2).

The necessity for step (2) can be seen in the simple case shown in Figure 9.2.



B precedes A in order of maximum z but A should precede B in writing order.

At step (2) the ordering produced by (1) must be confirmed. This is done by making more precise comparisons between polygons whose z-extents overlap.

Assume that polygon P is currently at the head of the sorted list, before scan converting it to the frame-buffer it is tested against each polygon Q whose z-extent overlaps that of P. The following tests of increasing complexity are then carried out:

1. If the x-extents of P and Q do not overlap then the polygons do not overlap, hence their ordering is immaterial.
2. If the y-extents of P and Q do not overlap then the polygons do not overlap, hence their ordering is immaterial.
3. If P is wholly on the far away side of Q then P is written before Q.
4. If Q is wholly on the viewing side of P then P is written before Q.
5. If the projections of P and Q do not overlap then the order P and Q in the list is immaterial.

If any of these tests are satisfied then the ordering of P and Q need not be changed. However if all five tests fail then it is assumed that P obscures Q and they are interchanged in the list. To avoid looping in the case where P and Q inter-penetrate Q must be marked as having been moved in the list.

When polygon Q which has been marked fails all tests again when tested against P then it must be split into two polygons and each of these polygons treated separately. Q is split in the plane of P.

Often the last test will not be done because it can be very complex for general polygons.

Light Intensities

Values of intensity calculated by an illumination model must be converted to one of the allowable intensity levels for the particular graphics system in use.

We have 2 issues to consider:
1. Human perceive relative light intensities on a logarithmic scale.

Eg. We perceive the difference between intensities 0.20 and 0.22 to be the same as the difference between 0.80 and 0.88. Therefore, to display successive intensity levels with equal perceived differences of brightness, the intensity levels on the monitor should be spaced so that the ratio of successive intensities is constant:

$$I1/I0 = I2/I1 = I3/I2 = \ldots = \text{a constant}.$$

2. The intensities produced by display devices are not linear with the electron-gun voltage.
This is solved by applying a gamma correction for video lookup correction:
Voltage for intensity Ik is computed as:

$$Vk = (Ik / a)1 / ?$$

Where ***a*** is a constant and ? is an adjustment factor controlled by the user.

For example, the NTSC signal standard is =2.2.

Basic Lighting and Reflection

Up to this point, we have considered only the geometry of how objects are transformed and projected to images. We now discuss the *shading* of objects: how the appearance of objects depends, among other things, on the lighting that illuminates the scene, and on the interaction of light with the objects in the scene. Some of the basic qualitative properties of lighting and object reflectance that we need to be able to model include:

Light source - There are different types of sources of light, such as point sources (e.g., a small light at a distance), extended sources (e.g., the sky on a cloudy day), and secondary reflections (e.g., light that bounces from one surface to another).

Reflectance - Different objects reflect light in different ways. For example, diffuse surfaces appear the same when viewed from different directions, whereas a mirror looks very different from different points of view.
In this chapter, we will develop simplified model of lighting that is easy to implement and fast to compute, and used in many real-time systems such as OpenGL. This model will be an approximation and does not fully capture all of the effects we observe in the real world. In later chapters, we will discuss more sophisticated and realistic models.

Simple Reflection Models

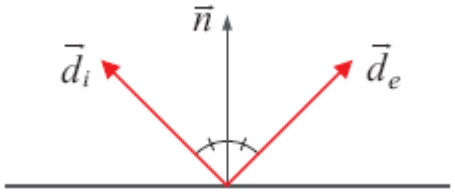
Diffuse Reflection

We begin with the diffuse reflectance model. A diffuse surface is one that appears similarly bright from all viewing directions. That is, the emitted light appears independent of the viewing location. Let ¯p be a point on a diffuse surface with normal ~n, light by a point light source in direction ~s from the surface. The reflected intensity of light is given by:

$$L_d(\bar{p}) \;=\; r_d\, I\, \max(0, \vec{s} \cdot \vec{n})$$

Perfect Specular Reflection

For pure specular (mirror) surfaces, the incident light from each incident direction ~di is reflected toward a unique emittant direction ~de. The emittant direction lies in the same plane as the incident direction ~di and the surface normal ~n, and the angle between ~n and ~de is equal to that between ~n and ~di. One can show that the emitting direction is given by ~de = 2(~n · ~di)~n − ~di. (The derivation was



covered in class). In perfect specular reflection, the light emitted in direction ~de can be computed by reflecting ~de across the normal (as 2(~n · ~de)~n − ~de), and determining the incoming light in this direction. (Again, all vectors are required to be normalized in these equations).

General Specular Reflection

Many materials exhibit a significant specular component in their reflectance. But few are perfect mirrors. First, most specular surfaces do not reflect all light, and that is easily handled by introducing a scalar constant to attenuate intensity. Second, most specular surfaces exhibit some form of *off-axis specular reflection*. That is, many polished and shiny surfaces (like plastics and metals) emit light in the perfect mirror direction and in some nearby directions as well. These off-axis specularities look a little blurred. Good examples are *highlights* on plastics and metals. More precisely, the light from a distant point source in the direction of ~s is reflected into a range of directions about the perfect mirror directions ~m = 2(~n · ~s)~n−~s. One common model for this is the following:

$$L_s(\vec{d}_e) = r_s I \max(0, \vec{m} \cdot \vec{d}_e)^{\alpha},$$

Where rs is called the specular reflection coefficient I is the incident power from the point source, and $\alpha \geq 0$ is a constant that determines the

width of the specular highlights. As α increases, the effective width of the specular reflection decreases. In the limit as α increases, this becomes a mirror.

Ambient Illumination

The diffuse and specular shading models are easy to compute, but often appear artificial. The biggest issue is the point light source assumption, the most obvious consequence of which is that any surface normal pointing away from the light source (i.e., for which ~s · ~n < 0) will have a radiance of zero. A better approximation to the light source is a uniform *ambient* term plus a point light source. This is a still a remarkably crude model, but it's much better than the point source by itself. Ambient illumintation is modeled simply by:

$$L_a(\bar{p}) = r_a \, I_a$$

Where ra is often called the ambient reflection coefficient, and Ia denotes the integral of the uniform illuminant.

Polygon Rendering Methods

A freeform surface can be approximated by polyhedral.

Rendering:

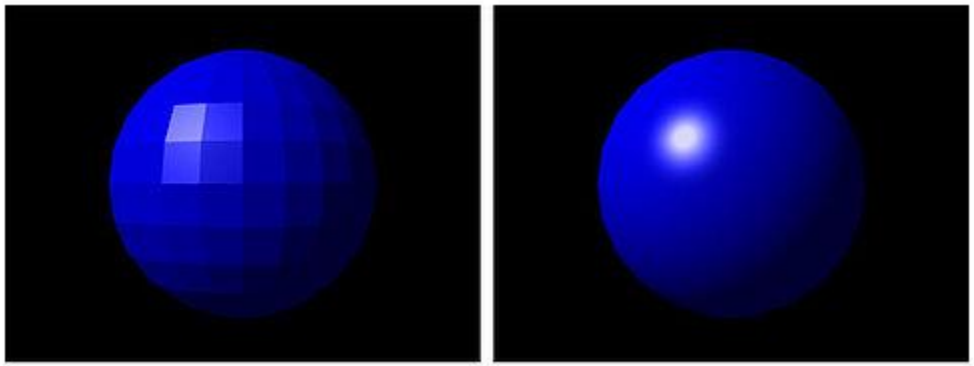
Calculate the illumination at each surface point by applying the illumination model at each surface point is computationally expensive.

Flat Shading

A single intensity is calculated for each surface polygon

• Fast and simple method

• Gives reasonable result only if all of the following assumptions are valid:



The object is a polyhedron

– Light source is far away from the surface so that

N•L is constant over each polygon
– Viewing position is far away from the surface so that V•R is constant over each polygon.

Phong Shading

Phong shading refers to an interpolation technique for surface shading in 3D computer graphics. It is also called Phong interpolation or normal-vector interpolation shading. Specifically, it interpolates surface normals across rasterized polygons and computes pixel colors based on the interpolated normals and a reflection model. *Phong shading* may also refer to the specific combination of Phong interpolation and the Phong reflection model.

Phong shading improves upon Gouraud shading and provides a better approximation of the shading of a smooth surface. Phong shading assumes a smoothly varying surface normal vector. The Phong interpolation method works better than Gouraud shading when applied to a reflection model that has small specular highlights such as the Phong reflection model.

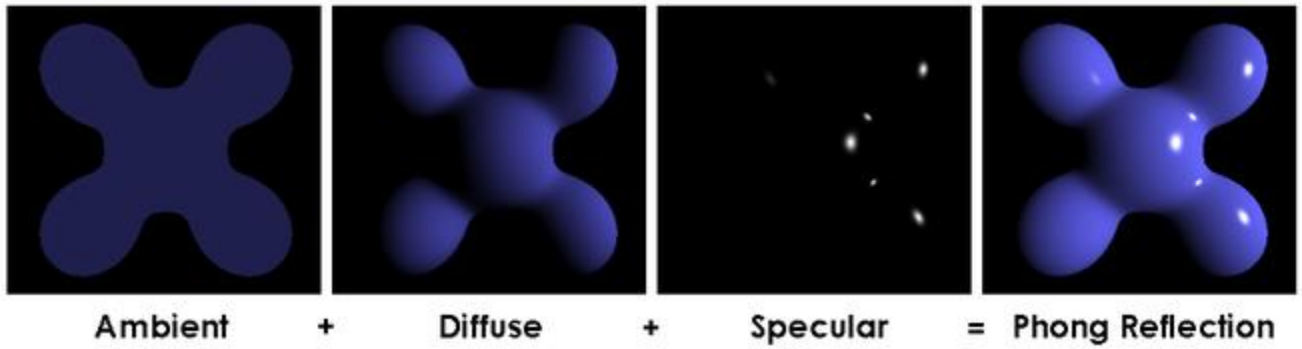
The most serious problem with Gouraud shading occurs when specular highlights are found in the middle of a large polygon. Since these specular highlights are absent from the polygon's vertices and Gouraud shading interpolates based on the vertex colors, the specular highlight will be missing from the polygon's interior. This problem is fixed by Phong shading.

Unlike Gouraud shading, which interpolates colors across polygons, in Phong shading a normal vector is linearly interpolated across the surface of the polygon from the polygon's vertex normals. The surface normal is interpolated and normalized at each pixel and then used in a reflection model, e.g. the Phong reflection model, to obtain the final pixel color. Phong shading is more computationally expensive than Gouraud shading since the reflection model must be computed at each pixel instead of at each vertex.

In modern graphics hardware, variants of this algorithm are implemented using pixel or fragment shaders.

Phong reflection model

Phong shading may also refer to the specific combination of Phong interpolation and the Phong reflection model, which is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Bui Tuong Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The reflection model also includes an *ambient* term to account for the small amount of light that is scattered about the entire scene.



Visual illustration of the Phong equation: here the light is white, the ambient and diffuse colors are both blue, and the specular color is white, reflecting a small part of the light hitting the surface, but only in very narrow highlights. The intensity of the diffuse component varies with the direction of the surface, and the ambient component is uniform (independent of direction).

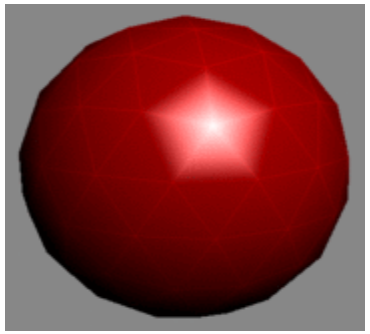
Gouraud Shading

Gouraud shading works as follows: An estimate to the surface normal of each vertex in a polygonal 3D model is either specified for each vertex or found by averaging the surface normals of the polygons that meet at each vertex. Using these estimates, lighting computations based on a reflection model, e.g. the Phong reflection model, are then performed to produce colour intensities at the vertices. For each screen pixel that is covered by the polygonal mesh, colour intensities can then be interpolated from the colour values calculated at the vertices.

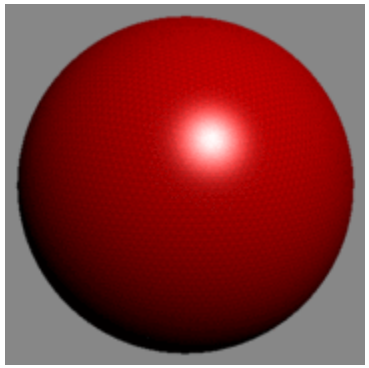
• Renders the polygon surface by linearly interpolating intensity values across the surface

Gouraud Shading Algorithm:

1. Determine the normal at each polygon vertex
2. Apply an illumination model to each vertex to calculate the vertex intensity
3. Linearly interpolate the vertex intensities over the surface polygon

- 

  Gouraud-shaded sphere - note the poor behaviour of the specular highlight.

- 

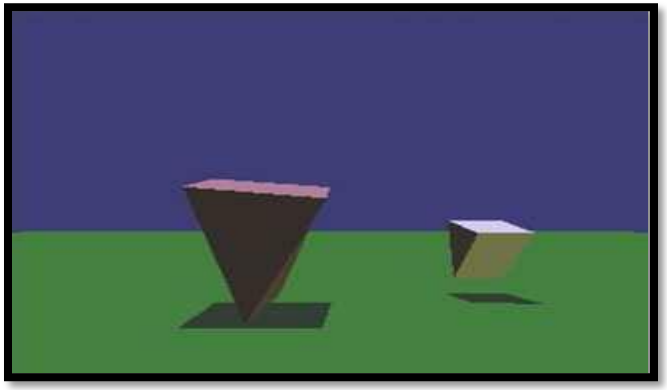
  The same sphere rendered with a very high polygon count.

The normal Nv of a vertex is an average of all neighboring normals.

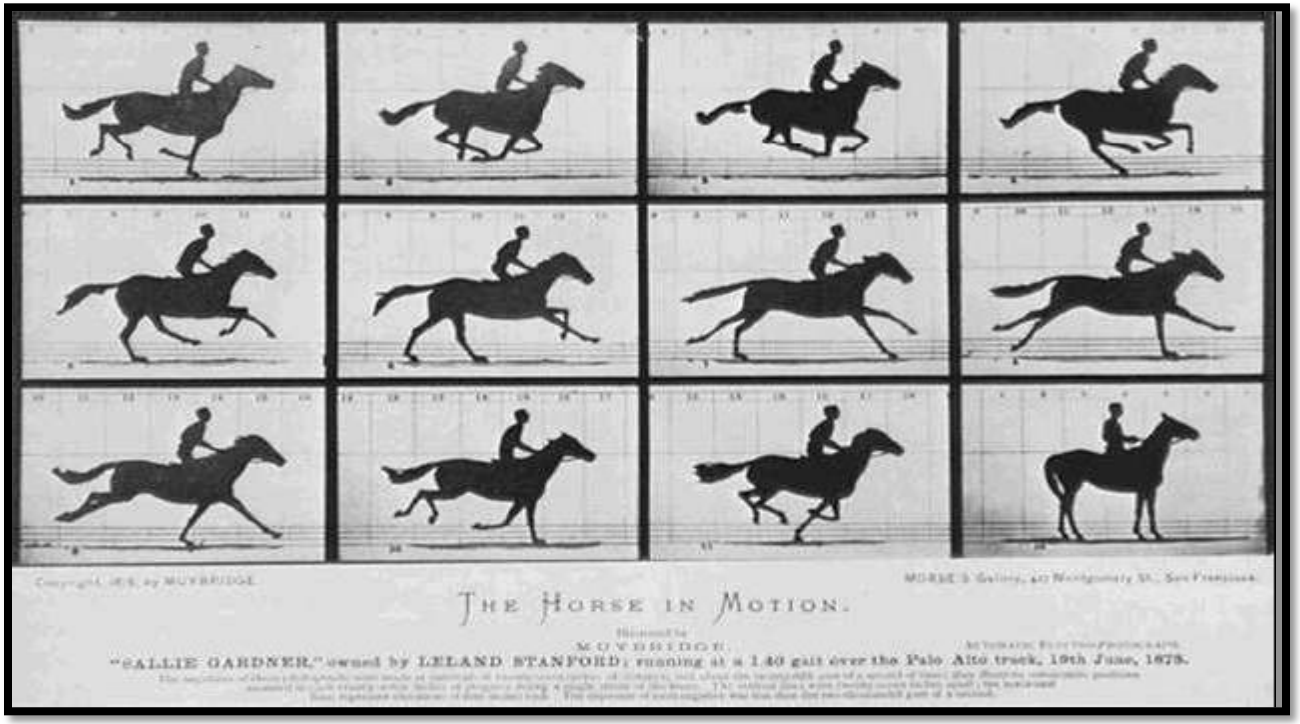
Animation

Overview

Motion can bring the simplest of characters to life. Even simple polygonal shapes can convey a number of human qualities when animated: identity, character, gender, mood, intention, emotion, and so on.
The basic principle of animation is the same for all media – displaying a rapid sequence of images which are slightly different than each other creating the illusion of movement before the eye.
However, this is how regular, 2D animation is created (or rather, was created until a few years ago). Nowadays, for both creating 2D as well as 3D animation, computer software from several vendors such as Adobe, Xara, Strata and Corel are available.



A movie is a sequence of frames of still images. For video, the frame rate is typically 24 frames per second. For film, this is 30 frames per second.



In general, animation may be achieved by specifying a model with n parameters that identify degrees of freedom that an animator may be interested in such as:-

• Polygon vertices,
• spline control,
• Joint angles,
• Muscle contraction,
• Camera parameters, or
• Color.

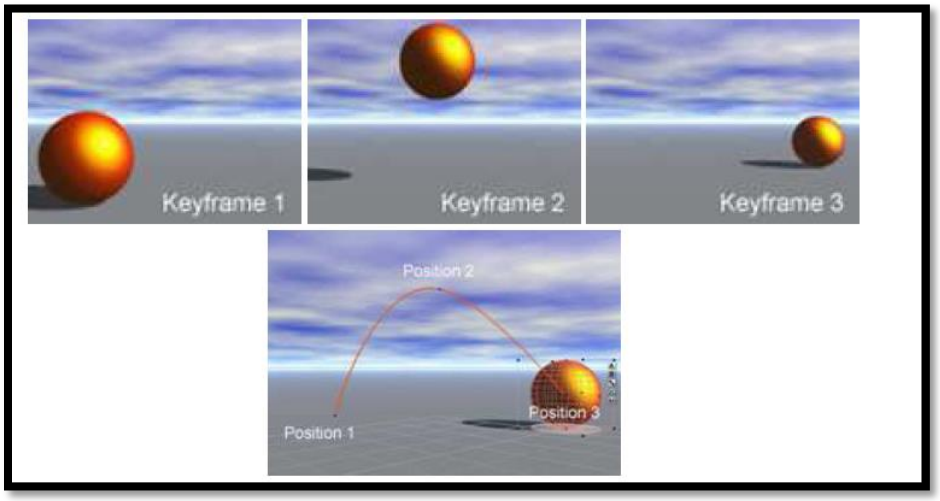
With n parameters, this results in a vector ~q in n-dimensional state space. Parameters may be varied to generate animation. A model's motion is a trajectory through its state space or a set of motion curves for each parameter over time, i.e. ~q(t), where t is the time of the current frame. Every animation technique reduces to specifying the state space trajectory.
The basic animation algorithm is then: for t=t1 to tend: render (~q(t)).
Modeling and animation are loosely coupled. Modeling describes control values and their actions. Animation describes how to vary the control values. There are a number of animation techniques, including the following:

• User driven animation

– Keyframing
– Motion capture
• Procedural animation
– Physical simulation
– Particle systems
– Crowd behaviors
• Data-driven animation

Keyframing

Keyframing is an animation technique where motion curves are interpolated through states at times, (~q1, ..., ~qT), called keyframes, specified by a user.



• Pros:
– Very expressive
– Animator has complete control over all motion parameters
• Cons:
– Very labor intensive
– Difficult to create convincing physical realism
• Uses:
– Potentially everything except complex physical phenomena such as smoke, water, or fire.

Concepts and Storyboarding

Storyboarding is the process of creating a visual representation of the actual screenplay of the animation's story. It basically consists of a series of illustrations or images presented in a sequence for pre-visualizing an animation. It is usually an intricate and tedious process as it is supposed to visually convey the actual story of the animation.

Modeling

3D modeling is usually done using specialized computer software and involves developing a mathematical model of any surface of a three dimensional object. This is actually done using 2D images of the object using a process called 3D rendering. Models may be created automatically or manually; the latter is usually done by an artist and is similar to sculpting. An example of a computerized 3D model of a human head is shown next. (Source: http://pabs.us/graphictutorials/wp-content)

Layout

To render objects on the media being used, they must first be placed within a scene, a process known as layout. In this process, the physical and spatial interrelations between the objects contained in a scene are first decided. Next, several techniques such as motion capturing and keyframing are used to capture their movement and deformation over time.

Just as in modeling, layout may also involve physical movement of the objects in the scene, similar to sculpting. An example of the use of layout is shown below in a promotional poster for the film 'Shrek'.

Virtual reality

Virtual reality (**VR**), sometimes referred to as **immersive multimedia**, is a computer-simulated environment that can simulate physical presence in places in the real world or imagined worlds. Virtual reality can recreate sensory experiences, including virtual taste, sight, smell, sound, touch, etc.

Most current virtual reality environments are primarily visual experiences, displayed either on a computer screen or through special stereoscopic displays, but some simulations include additional sensory information, such as sound through speakers or headphones. Some advanced, haptic systems now include tactile information, generally known as force feedback in medical, gaming and military applications. Furthermore, virtual reality covers remote communication environments which provide virtual presence of users with the concepts of telepresence and telexistence or a virtual artifact (VA) either through the use of standard input devices such as a keyboard and mouse, or through multimodal devices such as a wired glove, the Polhemus, and omnidirectional treadmills. The simulated environment can be similar to the real world in order to create a lifelike experience—for example, in simulations for pilot or combat training—or it can differ significantly from reality, such as in VR games. In practice, it is currently very difficult to create a high-fidelity virtual reality experience, because of technical limitations on processing power, image resolution, and communication bandwidth. However, the technology's proponents hope that such limitations will be overcome as processor, imaging, and data communication technologies become more powerful and cost-effective over time.

Virtual reality is often used to describe a wide variety of applications commonly associated with immersive, highly visual, 3D environments. The development of CAD software, graphics hardware acceleration, head-mounted displays, datagloves, and miniaturization have helped popularize the notion. In the book The Metaphysics of Virtual Reality by Michael R. Heim, seven different concepts of virtual reality are identified: simulation, interaction, artificiality, immersion, telepresence, full-body immersion, and network communication. People often identify VR with head mounted displays and data suits.

The possibility exists to have films and television programmes which are watched with a head-mounted display and computer control of the image so that the viewer appears to be inside the scene with the action going all round. The computer presents the view which corresponds to the direction the viewer is facing, through a system of head-tracking. This would give the viewers the feeling that they are actually going to the scene in person instead of looking at pictures on a screen. The term "virtual space" has been suggested as more specific for this technology, which is described in detail in the article Virtual Space - the movies of the future.

The term "artificial reality", coined by Myron Krueger, has been in use since the 1970s; however, the origin of the term "virtual reality" can be traced back to the French playwright, poet, actor, and director Antonin Artaud. In his seminal book The Theatre and Its Double (1938), Artaud described theatre as "la réalité virtuelle", a virtual reality in which, in Erik Davis's words, "characters, objects, and images take on the phantasmagoric force of alchemy's visionary internal dramas". Artaud claimed that the "perpetual allusion to the materials and the principle of the theater found in almost all alchemical books should be understood as the expression of an identity existing between the world in which the characters, images, and in a general way all that constitutes the virtual reality of the theater develops, and the purely fictitious and illusory world in which the symbols of alchemy are evolved".

The term has also been used in The Judas Mandala, a 1982 science-fiction novel by Damien Broderick, where the context of use is somewhat different from that defined above. The earliest use cited by the Oxford English Dictionary is in a 1987 article titled "Virtual reality", but the article is not about VR technology. The concept of virtual reality was popularized in mass media by movies such as Brainstorm and The Lawnmower Man. The VR research boom of the 1990s was accompanied by the non-fiction book Virtual Reality (1991) by Howard Rheingold. The book served to demystify the subject, making it more accessible to less technical researchers and enthusiasts.

Multimedia: from Wagner to Virtual Reality, edited by Randall Packer and Ken Jordan and first published in 2001, explores the term and its history from an avant-garde perspective. Philosophical implications of the concept of VR are discussed in books including Philip Zhai's Get Real: A Philosophical Adventure in Virtual Reality (1998) and Digital Sensations: Space, Identity and Embodiment in Virtual Reality (1999), written by Ken Hillis.

APPLICATIONS OF VIRTUAL REALITY

Virtual reality offers the potential for highly interactive interfaces to any computer-based simulation.Therefore VR interfaces may be applied to a limitless range of applications.

Existing applications of virtual reality involve such diverse fields as medicine, architecture, space science, visualization, and entertainment.

---------- **X X X X** ----------