Introduction

Computer graphics started with the display of data on hardcopy plotters and cathode ray tube (CRT). It has grown to include the creation, storage, and manipulation of models and images of objects. Today’s computers are capable of generating lifelike images that are virtually indistinguishable from the images captured in photographs. Computer graphics are not constrained by the boundaries of realism; sometimes they go far into the realm of the imaginary, as demonstrated by the computerized special effects you see in the movies. For example, fractal images – computer-generated pictures processing infinite complexity – allow computers to simulate objects in nature, from plants to mountain ranges, while ray-tracing techniques enable computers to render pictures with realistic lighting effects.

Representative Uses of Computer Graphics

- User interfaces: GUI, etc.
- Business, science and technology: histograms, bar and pie charts, etc.
- Office automation and electronic publishing: text, tables, graphs, hypermedia systems, etc.
- Computer-aided design (CAD): structures of building, automobile bodies, etc.
- Simulation and animation for scientific visualization and entertainment: flight simulation, games, movies, virtual reality, etc.
- Art and commerce: terminals in public places such as museums, etc.
- Cartography: map making

Related Field

Computer graphics concerns the pictorial synthesis of real or imaginary objects from their computer-based model, whereas the related field of image processing treats the converse process: the analysis of scenes, or the reconstruction of models of 2D or 3D objects from their pictures. Image processing has the sub-areas image enhancement (deals with improving image quality by eliminating noise), pattern detection and recognition (e.g. OCR technology), and scene analysis and computer vision (allow scientists to recognize and reconstruct a 3D model of scene from several 2D images).

Although both computer graphics and image processing deal with computer processing of pictures, they have until recently been quite separate disciplines, however, the overlap between the two is growing.
Input Technology

- Keyboard
- Mouse
- Trackball and Spaceball
- Joysticks
- Data Glove
- Digitizers
- Scanner
How Scanners Work

1. A page is placed face down on the scanner and a bright light attached to the scan head illuminates it. Dark areas of the page absorb more light than light areas, so lighter areas reflect more light.

7. The information gathered by the scanner is transmitted to the computer, where it is assembled into a graphical image. The transfer is normally accomplished with a serial port connection, which moves individual bits of data, one at a time, at a rate of several thousand bits per second.
• Touch Panel
• Light Pen
• Voice System

Output Technology
• Film recorder
• Videotape
• Printer
• Plotter
• Cathod Ray Tube (CRT)
- Vector displays

A typical vector system consists of a display controller, a display memory, and a CRT. The display memory stores the display list or display program; for plotting points, lines, and characters. The commands for plotting points, lines, and characters are interpreted by the display controller and sends digital and point coordinates to a vector generator that converts the digital coordinate values to analog voltages for writing on the CRT’s phosphor coating.

... 
MOVE 10, 15
LINE 400, 300
CHAR Lucy
...

- Raster displays

The raster graphics is based on television technology. Raster displays store display primitives (such as lines, characters, and solidly shaded or patterned area) in a video buffer in terms of their component pixels. A pixel is determined by the bits representing it in the video buffer as shown in the following:

How images are displayed on the screen?

1. The starting point for an image displayed on the computer screen is the video buffer or video RAM.
2. A pixel is determined by the bits representing it in the video buffer. By changing the 1s and 0s that represent a pixel in the video buffer, a
program running on the computer can change the image or color of an image.

3. The inside of the monitor screen is coated with tiny specks of phosphorescent matter called phosphors. Each pixel is composed of three phosphors: one red, one green, and one blue. A phosphor glows when impacted by an electron beam, and stays illuminated for a short period of time (typically a few thousandth of a second) after the electron beam is removed. Various combinations of red, green, and blue phosphor intensities produce various hues and intensities of color.

4. A special device called a digital-to-analog converter (DAC) onboard the video adapter converts the bits in the video buffer to voltage levels for the monitor’s three electron guns. On palette-based video adapters, the DAC acquires color information from the palette rather than directly from the bits in the video buffer.

5. To illuminate one pixel, the video adapter uses the voltage levels computed by the DAC to drive the output from the monitor’s three electron guns. Each gun shoots an electron beam at the screen. The strength of the electron beams, and hence the intensity of the color associated with each gun, is determined by the voltage levels. If a pixel’s red component is 255 (the highest value that can be represented with an 8-bit number), the red gun is turned to its highest intensity for that pixel. If the green and blue components are 0, no blue or green colors will be introduced and the pixel will appear on the screen bright red. If, on the other hand, equal amounts of red and blue are introduced and green is again withheld, the pixel will appear magenta.

6. To help target the guns on a particular phosphor, the electron beams pass through holes in a shadow mask on their way to the screen. The spacing of the holes determines the spacing of the pixels on the screen, which is commonly referred to as the monitor’s dot pitch. A typical monitor has a dot pitch of about .30 millimeters, meaning individual pixels on the screen appear .30 millimeters (slightly more than 1/100 of an inch) apart.
7. The result of all this is an image on the computer screen. From the standpoint of a graphics program running on the computer, the challenge is to poke the right bit values into the video buffer. The video hardware takes care of the rest. Determining what values to write to the video buffer is what the science of computer graphics is all about.

8. To paint an entire screen containing hundreds of thousands of pixels, electromagnets inside the monitor deflect the paths of the electron beams so that they sweep across the screen left to right, top to bottom, illuminating every pixel. On noninterlaced systems, the electron beams travel left to right along the first (top) row of pixels to illuminate that row, or scan line or raster line. Then they jump to the leftmost pixel in the second row and illuminate that row, followed by the third row, the fourth row, and so on. This happens so quickly that the entire screen is painted in about 1/60 of a second. Because the screen is fully refreshed 60 times per second, we say that it has a refresh rate of 60 cycles per second, or 60 Hertz (Hz).

9. On interlaced monitors, odd-numbered scan lines are illuminated in one cycle and even-numbered scan lines in the next. One full screen refresh requires two complete cycles, reducing the monitor’s effective refresh rate from 60 screen refreshes per second to 30. Interlaced monitors cost less, but often flicker due to the lower refresh rate.

Pixels are arranged on the computer screen in rows and columns. The screen’s resolution is determined by the number of pixels in each row and column. Each pixel’s color can be set
independently, but the number of colors that can be displayed on the screen at once may be limited by the graphics hardware that you’re using. At one end of the spectrum, you have monochrome systems that display only two colors. At the opposite end, you have true-color systems capable of displaying more than 16.7 million colors. Most of the video adapters used in today’s personal computers are capable of displaying some number of colors in between. The maximum number of colors that can be displayed simultaneously is determined by the number of data bits set aside for each pixel in a video buffer. On true-color systems, each pixel is represented by 24 bits of color information: eight for the pixel’s red component, eight for the green component, and eight for the blue component.

- **R**
- **G**
- **B**

**Video RAM**

**Screen**

Advantage: capable of displaying more than 16.7 million colors

Disadvantage: very large video RAM

On palette-based video adapters, the bits representing a pixel in the video buffer do not specify the pixel’s color directly. The value stored in the palette entry table, in turn, determines the color of the pixel. On a 256-color system, which uses only eight bits per pixel, if each palette entry holds 24 bits of color information, the video hardware can still display any color it cares to. The catch is that it can only display 256 different colors at a time.

For example the 16-color system:
**Example** Standard video modes for PC-compatibles

<table>
<thead>
<tr>
<th>Resolution</th>
<th>CGA</th>
<th>4 colors</th>
<th>16 colors</th>
<th>256 colors</th>
</tr>
</thead>
<tbody>
<tr>
<td>320x200</td>
<td>CGA</td>
<td>EGA lo-res</td>
<td>VGA/MCGA</td>
<td></td>
</tr>
<tr>
<td>640x350</td>
<td>EGA mono</td>
<td>Older EGAs</td>
<td>EGA hi-res</td>
<td></td>
</tr>
<tr>
<td>720x348</td>
<td>Hercules</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>VGA mono</td>
<td>VGA hi-res</td>
<td>VESA 101h</td>
<td></td>
</tr>
<tr>
<td>800x600</td>
<td>SVGA mono</td>
<td>VESA 102h</td>
<td>VESA 103h</td>
<td></td>
</tr>
<tr>
<td>1024x768</td>
<td>SVGA+ mono</td>
<td>VESA 104h</td>
<td>VESA 105h</td>
<td></td>
</tr>
<tr>
<td>1280x1024</td>
<td>SVGA+ mono</td>
<td>VESA 106h</td>
<td>VESA 107h</td>
<td></td>
</tr>
</tbody>
</table>

**Note!** VESA (Video Electronics Standards Association)

The monitor may be a digital or analog monochrome, RGBI or RrGgBb digital color, or an analog RGB color monitor. Monochrome monitors - used with the MDA, Hercules, or EGA – are less expensive than color monitors. RGBI monitors – used with the CGA or EGA – accept digital color inputs, allowing the display of 16 unique colors. RrGgBb monitors also accept digital color inputs and are used with the EGA. Analog RGB monitors must be used with the VGA. Analog RGB monitors can display an unlimited number of colors.
Display Frame Buffer Organization

Graphics display memories (otherwise known as frame buffers) can be essentially divided into two classifications. One is a packed-pixel organization and the other is a bit-planar organization. A bit-planar memory map effectively represents each pixel in the image with one memory bit in contigu-
ous memory. If multiple bits-per-pixel are needed, multiple memory maps are employed. Packed-pixel organization has meaning only in cases where more than one bit-per-pixel is required. Each pixel is composed of contiguous bits in memory. Figure 2.46 illustrates the difference in memory organization between packed-pixel and bit-planar display memory for an implementation using four bits-per-pixel.

Probably the most commonly used display mode for VGA controllers is 640x480 resolution (or higher) with 16 colors. This requires four bits per pixel, which are organized in memory as four planar bitmaps as established by the EGA display controller. Figure 2.47 illustrates how these four bits-per-pixel were used in the EGA controller to select colors on the digital CRT. For each pixel, one bit from each bit plane was used to form a four-bit pixel value, which was then used as an address into a 16-location color palette lookup table RAM memory. The output of this memory was a six-bit value divided into three groups of two bits each. Each pair of bits drove one of the three electron guns in the CRT, so for each gun four different intensity levels could be selected.

The VGA improved on the scheme by switching to an analog CRT and adding another stage of color lookup tables to allow a wider selection of colors and finer control. Figure 2.48 illustrates how this was done. The six bits from the EGA color palette RAM are combined with two bits from a color select register to form an eight-bit value used as an address into three color lookup tables with 256 locations each. Each of the three tables produces a six-bit value that is used to drive a digital-to-analog converter that generates an analog value to drive the electron gun in the analog CRT. This means that 18 bits of resolution are available for each color displayed on the CRT, so each of the 16 colors displayed can be any of 262,144 (256K) different colors.

In practice, however, the lookup tables are programmed to produce the same colors that were produced in the digital monitor of the EGA in almost all cases. For motion video applications, however, this display mode is not really useful anyway because there are not enough distinct colors to render a realistic looking natural image. The real advantage of the VGA for motion video applications is its ability to display 256 colors on the screen simultaneously. Unfortunately, a standard

Figure 2.47 EGA digital display output data flow.
VGA can do this only at a resolution of $320 \times 200$ pixels, which is not supported by popular user interfaces such as Microsoft Windows. To display 256 or more colors at resolutions higher than this requires a Super VGA. To display 256 colors, memory is organized in a packed-pixel configuration using eight bits-per-pixel. These eight bits directly drive the color lookup tables for conversion into analog format. Figure 2.49 illustrates this.

For the $320 \times 200$ display mode, the graphics display memory starts at address A0000(hex) in the PC address space, and is 64,000 bytes long. The first byte of this region corresponds to the upper-left pixel on the display, and the first 320 bytes correspond to the first pixel row on the display. This arrangement makes address calculation for pixel drawing considerably simpler than it is for $640 \times 480$ or other modes. Assuming a $(0,0)$ coordinate at the upper-left corner of the screen, any $(x,y)$ coordinate can be calculated as $A0000(hex) + y \times 320 + x$. To illustrate the default color arrangement for 256-color mode, a program named PAL_EDIT is provided. This allows the user to select any of the 256 colors provided by the hardware palette and modify the RGB components to produce a different color. Any color beyond the first sixteen can be modified without affecting the appearance of the user interface screen. If any of the colors that are used in the user interface screen...
Figure 2.49  VGA analog display data flow for 256 color mode.

(from the first sixteen) are modified, the result will be immediately apparent in the appearance of the display. The following is a source code listing for the main program segment:
A range of 256 colors is enough to produce a recognizable video image. However, this arrangement presents certain difficulties for the processing of color information. The color lookup tables effectively isolate the color information stored in the pixel map from the actual colors displayed on the screen. A better graphics organization for natural images and motion video is to use at least 15 bits-per-pixel. Each pixel can then be divided into three fields to directly drive the analog-to-digital converters for the red, green, and blue values going to the CRT. This is sometimes referred to as direct color mode. Figure 2.50 illustrates how memory is organized in this case. This arrangement allows either 32,768 (32K) or 65,536 (64K) colors to be displayed on the screen simultaneously and produces quite good color resolution for motion video, although it is considered by some to be inadequate for high-resolution color still images.

The highest color resolution generally used in PC graphics adapters uses 24 bits-per-pixel. This is generally referred to as true color. The memory organization scheme is essentially the same as that for 15 or 16 bits-per-pixel except that 8 bits each are used for red, green, and blue.

![Figure 2.50 Direct color display data flow for 15-bit color or higher.](image)

**Software Portability and Graphics Standards**

We have moved from low-level, device-dependent packages supplied by manufacturers for their particular display devices to higher-level, device-independent packages. The main purpose of using a device-independent package in conjunction with a high-level programming language is to promote application program portability. The first graphics specification to be officially standardized was GKS (the Graphical Kernel System). The others packages are, PHIGS (Programmer's Hierarchical Interactive Graphics System), OpenGL, etc.
Achromatic and Colored Light

Achromatic Light

Achromatic light is what we see on a black-and-white television set or display monitor. Quantity of light is the only attribute of achromatic light. Quantity of light can be discussed in the physics sense of energy.

Color Representation

Visible light is a form of electromagnetic radiation, part of the same spectrum as radio waves, microwaves, X-rays, and gamma rays. The portion of this spectrum visible to the human eye is a very narrow band from approximately 400 to 700 nM in wavelength.

Note! \[ \text{wavelength (in meters)} = \frac{\text{velocity of light (in meters per second)}}{\text{frequency (in hertz)}} \]

where velocity of light = 3*10^8

CIE Standard

The most commonly used standard for the measurement and specification of color is that defined by the International Commission on Illumination in 1931. This is referred to as the CIE
standard. CIE stands for the French phrase *Commission Internationale de l’Eclairage*. The CIE standard allows a color to be specified as a numeric triple (X,Y,Z).

The basis of the standard, adopted in 1931, is color-matching experiments where a user controls or weights three primary light sources to match a sample light source. In other words the weights in:

\[ C = rR + gG + bB \]

Are determined experimentally.

The result of such experiments can be summarized by color-matching functions. These are shown in the following figure:

To avoid negative weights the CIE devised a standard of three primaries X, Y, and Z, which when additively mixed will produce all perceivable colors using positive weights. The three corresponding matching functions are shown in the following figure:
Note!

\[ X = k \int_\lambda P(\lambda) x(\lambda) d\lambda \]
\[ Y = k \int_\lambda P(\lambda) y(\lambda) d\lambda \]
\[ Z = k \int_\lambda P(\lambda) z(\lambda) d\lambda \]

Where \( k \) is a constant and \( P(\lambda) \) is the spectral energy distribution – the variation of the energy of light sensation with wavelength.

CIE xyY

An alternative way of specifying the \((X,Y,Z)\) triple is \((x,y,Y)\) where \((x,y)\) are known as chromaticity coordinates:

\[ x = X/(X+Y+Z) \quad \text{and} \quad y = Y/(X+Y+Z). \]

Given \((x,y,Y)\), the transformation to the corresponding \((X,Y,Z)\) is \( X = (x/y)Y \), \( Y = Y \), \( Z = ((1-x-y)/y)Y \).

Plotting \((x,y)\) for all visible colors yields a two-dimensional \((x,y)\) space known as the CIE chromaticity diagram. It is used to define color gamuts, or color ranges, that show the effect of adding colors together as shown in the following figure:
Any two colors, say I and J, can be added to produce any color along their connecting line by varying the relative amounts of the two colors being added. A third color, say K, can be used with various mixtures of I and J produce the gamut of all colors in triangle IJK, again by varying relative amount. Because no triangle whose vertices are within the visible area can completely cover the visible area, the shape of the diagram shows why visible red, green, and blue cannot be additively mixed to match all colors.

The RGB Color Model

It is important to note that the chromaticity diagram represents all theoretically visible colors, not necessarily all colors that can be reproduced using currently available technology. For example, the phosphors used in the construction of cathode ray tubes produce colors that lie somewhere inside the boundaries of the CIE diagram. To find the coordinates for a particular phosphor in the chromaticity diagram the spectral distribution of the phosphor is measured. This is then multiplied by each of the CIE spectral distributions and used to calculate the coordinates used on the chromaticity diagram.
The red, green, and blue (RGB) color model used in color CRT monitors and color raster graphics employs a Cartesian coordinate system. The RGB primaries are additive primaries; that is, the individual contributions of each primary are added together to yield the result. The subset of interest is the unit cube shown below:

- Blue = (0, 0, 1)
- Cyan = (0, 1, 1)
- Magenta = (1, 0, 1)
- White = (1, 1, 1)
- Black = (0, 0, 0)
- Green = (0, 1, 0)
- Red = (1, 0, 0)
- Yellow = (1, 1, 0)

The main diagonal of the cube, with equal amounts of each primary, represents the gray levels; black is (0, 0, 0); white is (1, 1, 1).

The CMY Color Model

Cyan, magenta, and yellow are the complements of red, green, and blue, respectively. When used as filters to subtract color from white light, they are called subtractive primaries. The subset of the Cartesian coordinate system for the CMY model is the same as that for RGB except that
white (full light) instead of black (no light) is at the origin. Colors are specified by what is removed or subtracted from white light, rather than by what is added to blackness.

A knowledge of CMY is important when dealing with hardcopy devices that deposit colored pigments onto paper, such as electrostatic and ink-jet plotters. When a surface is coated with cyan ink, no red light is reflected from the surface. Cyan subtracts red from the reflected white light, which is itself the sum of red, green, and blue. Hence, in terms of the additive primaries, cyan is white minus red, that is, blue plus green. Similarly, magenta absorbs green, so it is red plus blue; yellow absorbs blue, so it is red plus green. A surface coated with cyan and yellow absorbs red and blue, leaving only green to be reflected from illuminating white light. A cyan, yellow, and magenta surface absorbs red, green, and blue, and therefore is black. The relations are represented by the following equation:

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

The unit column vector is the RGB representation for white and the CMY representation for black. The conversion from RGB to CMY is then

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

This conversion is relevant for use on ink-jet and xerographic color printers.

The CMYK Color Model

CMYK uses black (abbreviated as K) as a fourth color. It is used in the four-color printing process of printing presses and some hard-copy devices. Given a CMY specification, black is used in place of equal amounts of C, M, and Y, according to the relations:

\[
K := \min (C, M, Y) \\
C := C - K \\
M := M - K \\
Y := Y - K
\]

The YIQ Color Model

The YIQ model is used in U.S. commercial color television broadcasting (NTSC – National Television Standards Committee). The Y component is luminance, and is defined to be the same as
the CIE Y primary. I and Q are *chromatics* (the combination of hue and saturation). Y component is used for downward compatibility with back-and-white television. The RGB-to-YIQ mapping is defined as follows:

\[
\begin{bmatrix}
Y \\
I \\
Q
\end{bmatrix} = \begin{bmatrix}
0.299 & 0.587 & 0.114 \\
0.596 & -0.275 & -0.321 \\
0.212 & -0.528 & 0.311
\end{bmatrix}
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

**Note**

PAL is used in Germany standard broadcasting. The Y component is luminance. U and V are hue and saturation respectively. **Hue** (or tint) is the element that distinguishes between colors (green, red, blue, yellow, etc.). **Saturation** is the amount of color such that 100 percent saturation means there is no white component in the color. Less than 100 percent saturation tends to produce lighter and lighter colors. The RGB-to-YUV mapping is defined as follows:

\[
Y = 0.299R + 0.587G + 0.114B \\
U = 0.463B - 0.147R - 0.289G \\
V = 0.615R - 0.515G - 0.100B
\]

The I and Q in NTSC components are actually derived from the U and V components in the YUV system. I and Q are defined as: I = V\cos33 - U\sin33 and Q = V\sin33 + U\cos33.

SECAM is used in France. The Y component is luminance. Db and Dr are hue and saturation respectively. The RGB-to-YDbDr mapping is defined as follows:

\[
Y = 0.299R + 0.587G + 0.114B \\
Db = 1.333B - 0.450R - 0.883G \\
Dr = 0.217B - 0.133R - 0.116G
\]

For **CCIR 601**, The Y component is luminance. Cr and Cb are hue and saturation respectively. The RGB-to-YCrCb mapping is defined as follows:

\[
Y = \frac{77R}{256} + \frac{150G}{256} + \frac{29B}{256} \\
Cr = \frac{(131R}{256} - \frac{110G}{256} - \frac{21B}{256}) + 128 \\
Cb = \frac{(131B}{256} - \frac{44R}{256} - \frac{87G}{256}) + 128
\]

**Color Quantization**

This was invented to enable acceptable display of images on displays (screens) with low chromatic resolution. For example, Have image at 24 bits per pixel but screen (VGA) has 256 separate colors. Assume that 256 colors are implemented by color lookup table.
VGA can display $2^{18}$ different colors, but only 256 colors at a time.

The process of finding small set of representative colors for a larger set is called "quantization."

**Note!** Quantization can be used for compression, i.e., quantize to 256 colors, by storing lookup table and 8 bits index per pixel. Reduces storage requirement for picture by about 66%.

Lookup tables are also used in some high chromatic resolution displays. For example, raster technologies model one/256 24 bits/pixel:

Purpose of lookup table in this case is to allow for fast global color changes in a displayed image. For example, to change displayed image to photographic negative by changing lookup tables:

This can be accomplished much faster than updating every pixel.
Now, however, assume high resolution image, lower resolution display; try to determine best way to load lookup table.

A. Obvious: Popularity Algorithms:

Examine all pixels, construct histogram of pixel color frequencies.

![Histogram of pixel color frequencies](image)

In practice, only store data for non-zero pixel counts. Choose 256 colors with highest counts; load these into lookup table. Can be loaded into lookup table in any order. When displaying a pixel, if pixel color is in lookup table, write table index into video memory, else find closest color in lookup table, and write that index into video memory.

To measure distances between colors:

- **Euclidean Distance in RGB space**
  
  desired pixel color: \((R_1, G_1, B_1)\)
  
  lookup table entry : \((R_2, G_2, B_2)\)
  
  \[
  d = \sqrt{(R_2-R_1)^2 + (G_2-G_1)^2 + (B_2-B_1)^2}
  \]
  
  In practice, don’t take square root
  
  \[
  d = (R_2-R_1)^2 + (G_2-G_1)^2 + (B_2-B_1)^2
  \]
  
  Square function to ensure that each term is positive.

- **Taxicab metric or Manhattan metric**
  
  \[
  d = |R_2-R_1| + |G_2-G_1| + |B_2-B_1|
  \]
  
  In practice, not all subtractions are always necessary. If, for example, \(|R_i - R_1|\) is already larger than smallest distance found, don’t need other two subtractions.

**Note!** The popularity algorithm tends to work best on images with a small number of colors. One problem with the algorithm is that small highlights may not be colored correctly. Highlights generally only cover a few pixels, so their colors usually do not have enough representation in the image to be chosen by the popularity algorithm. To help alleviate this problem, you may force the corners of the color cube (white, red, green, blue, cyan, magenta, yellow, and black) to be selected as the first eight palette entries. Since most highlights are white, this helps reduce unwanted artifacts in the resulting image.
B. Median Cut Algorithms:

Examine all pixels in 24-bit image, distribute them in 3D RGB space.

Pixel have tendency to cluster. Find rectangular extent of these points: i.e.

find min R, max R
min G, max G
min B, max B

Gives subvolume surrounding all pixel colors.

Find longest axis of these subvolume:

if (maxR – minR > maxB – minB) and (maxR – minR > maxG – minG ) then it’s the R axis
else …

Subdivide along this axis into two smaller subvolume. Repeat procedure with smaller volumes until there are exactly 256 volumes.

How to subdivide?

1. (Easy) Divide at middle of volume along chosen axis
2. (Better) Find point on chosen axis so that the #pixels color in each subvolume is about the same.

Use average in that component of pixel colors in the volume.
Sub divide until \#subvolumes = \# lookup table entries. (Don’t subdivide volume with only one pixel color), load lookup table with a representative from each small volume (e.g., average of all pixel colors inside volume). Pixel with color inside volume is displayed with representative color instead.

**Warning!** Don’t use recursion to process subdivided volumes! Use queue data structure by subdivided into two objects; place each one on end of queue; pull object from front of queue to process; repeat ...

---

**How Median-Cut Color Quantization Works**

1. A common application for median-cut color quantization is choosing 256 colors to represent a true-color image containing several thousand colors. As an aid in visualizing how the median-cut method works, we’ll represent color space with a cube. Each axis corresponds to one of the three primary colors and is numbered from 0 to 255, with higher numbers representing higher color intensities. Colors in the image can be plotted inside the cube like points on a three-dimensional graph.

2. The first step is to shave off all ends of the cube that do not contain pixels. If no pixel has a red value less than 8 or greater than 250, for example, then the parts of the cube from R=0 to R=7 and R=251 to R=255 can be discarded.

3. The second step is to cut the resulting box in two at the median point of its longest side. If the box’s longest side parallels the B axis, the computer picks the median blue value from all the blue values represented in the box (for example, the 50,000th entry in a sorted list of 100,000 blue values) and makes the cut at that point. The box is now divided into two smaller boxes containing equal numbers of pixels.
This process—trimming the empty ends from the box and performing a median cut along its longest side—is repeated for the two smaller boxes. The original cube is now partitioned into four boxes containing approximately the same number of pixels.

Medium cuts are applied repeatedly to divide the cube into 8, 16, 32, 64, 128, and finally 256 boxes. The boxes contain about the same number of pixels, and vary in size in inverse proportion to the pixel density.

With color space divided in this manner, picking palette colors is easy. Each of the 256 boxes represents a roughly equal constituency of pixel color values, and the center of each box is an optimal location for a palette color. Given the corner coordinates, simple math yields the coordinates of the center point. (Some graphics programs average all the pixel values in the box rather than compute a center point. This approach takes a little longer but produces an even better palette.) Computing R, G, and B coordinates for all 256 boxes produces 256 representative palette colors.
PART I IMAGE PROCESSING

Image Representation

What is an image?

An image is a 2D rectilinear array of pixels. Pixels are samples (not a little square) from continuous function.

Image Resolution

- Intensity resolution
  
  Each pixel has only “Depth” bits for colors/intensities.

- Spatial resolution
  
  Image has only “Width” x “Height” pixels.

- Temporal resolution
  
  Monitor refreshes images at only “Rate” Hz e.g.

<table>
<thead>
<tr>
<th>Type</th>
<th>Width x Height</th>
<th>Depth</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTSC</td>
<td>640 x 480</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>Workstation</td>
<td>1280 x 1024</td>
<td>24</td>
<td>75</td>
</tr>
<tr>
<td>Film</td>
<td>3000 x 2000</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>Laser Printer</td>
<td>6600 x 5100</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Image Compressions

There are two basics.

1. Lossless Compression. Compression factor is usually not vary dramatic.

![Diagram of lossless compression]

2. Lossy Compression. Very high compression factors are possible.

![Diagram of lossy compression]

approximately same as original
The higher the compression factor, the more detail is lost (usually), and more CPU time is needed (usually) to reconstruct the image. Aside, many compression techniques can be implemented in hardware.

Remark! There is no algorithm that can always compress any image! If it is, then you can compress again and again until it is zero.

Lossless Compression Techniques

1. Run Length Encoding (RLE)

Very popular. In most pictures, there is the idea of “scan-line coherence”: very often the color remains the same from one pixel to the next. Several consecutive pixels with the same color can be encoded into a “run”.

\[
\begin{array}{cccc}
\text{Length} & R & G & B \\
\end{array}
\]

Represents linear run of N+1 pixels (current pixel and N following pixels), all of same color.

Work best when large areas have same color. The best case is when screen has the same color. The worst case is when every consecutive pixels have the different color. This will insert length field of 0 for each pixels. The size of “compressed” image increases instead of decreases. The maximum size of run length is determined by number of bits in length field:

Typically: 8 bits : 256
16 bits : 65536

The alternative is intermix runs encoded pixels with runs of ordinary pixels.

\[
\begin{array}{ccc}
0 & R & G & B \\
1 & N & & \\
\end{array}
\]

\[
\begin{array}{c}
\text{run length} \\
0 = N+1 normal pixels follow \\
1 = represent length of run of pixel of the same color. \\
\end{array}
\]

Note! One bad bit can damage the rest of the image!

Encoder should try each method and determine the best mix of these encodings (minimize size of compressed image).
2. Huffman Coding

Count all pixel values:

<table>
<thead>
<tr>
<th>value</th>
<th>frequency</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, 0)</td>
<td>1500</td>
<td>5%</td>
</tr>
<tr>
<td>(30, 0, 0)</td>
<td>3000</td>
<td>10%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let each value be represented by a node, labeled by percent.

5%       10%   ...

(0, 0, ) (30, 0,

Sort according to percent in increasing order (from small to large).

p1  p2  p3  p4   ...

Pick 2 nodes with lowest percentages (p1, p2 first time). Combine these into a tree:

 p1 + p2  label root of tree by sum of child percentages

0

1

p1  p2

Label left child branch with 0, right child branch with 1.

Repeat (using the isolated nodes and root node of tree) until there is only one tree. Leaf nodes contain pixel color information.

100%

0 1

0 1 0 1

0 1

0 1

Encode every pixel color by bitstream: collection of 0’s and 1’s on edges as tree is traversal from root to pixel color. Pixels with most popular (i.e. highest percentage) colors will have shortest bit strings.

For example:

<table>
<thead>
<tr>
<th>colors</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>binary code</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>
Compress picture:
- save tree (or code book of bitstream and pixel values)
- record bitstream for each pixel

Decompress picture:
- rebuild tree
- use bitstream for each pixel value to traverse tree and find pixel's true color

In practice, may need to record size of original image.

3. LZW

Compression Algorithm (Encoder)
initialize string table;
if ((K = next character) == EOF) stop;
p = K;
while ((K = next character) != EOF)
{
    if (pK in string table )
    {
        p = pK;
    }
    else
    {
        output code for p;
        add pK to string table;
        p = K;
    }
}
output code for p;

UnCompression Algorithm (Decoder)

initialize string table;
if ((code = next code) == EOF) stop;
K = c = table[code];
output c;
p = c;
while ((code = next code) != EOF)
{
    if (code not defined) /* no table[code] yet */
    {
        save pK in the next entry of table;
        c = table[code];
        output c;
        K is the first character of c i.e. if c = c_m, c_{m-1}, ..., c_2, c_1 then K = c_m;
    }
    else
    {
        c = table[code];
        output c;
        K is the first character of c i.e. if c = c_m, c_{m-1}, ..., c_2, c_1 then K = c_m;
        save pK in the next entry of table;
    }
    p = c;
}

Note! p, c are strings
K is a character
pK is p concatenate with K

Example
Assume alphabet of input is {a,b,...,z}.
input: ababba
a) Encoder

initialize string table;

<table>
<thead>
<tr>
<th>String table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

K = a
p = K = a

K = b
pK = ab is not in string table
output code for p ==> 0
add ab to string table

<table>
<thead>
<tr>
<th>String table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
</tbody>
</table>

p = K = b

K = a
pK = ba is not in string table
output code for p ==> 1
add ba to string table

<table>
<thead>
<tr>
<th>String table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
</tbody>
</table>
p = K = a

K = b
pK = ab is in string table
p = pK = ab

K = b
pK = abb is not in string table
output code for p => 26
add abb to string table

String table
0 a
1 b
...
25 z
26 ab
27 ba
28 abb

p = K = b

K = a
pK = ba is in string table
p = pK = ba

K = EOF
output code for p => 27

encoding code : 0 1 26 27

b) Decoder
initialize string table;

String table
0 a
1 b
...
25 z
code = 0
K = c = table[0] = a
output c ==> a
p = c = a

code = 1
c = table[1] = b
output c ==> b
K = b
save pK = ab in the next entry of table

String table
0  a
1  b
...  
25  z
26  ab

p = c = b

code = 26
c = table[26] = ab
output c ==> ab
K = a
save pK = ba in the next entry of table

String table
0  a
1  b
...  
25  z
26  ab
27  ba

p = c = ab

code = 27
c = table[27] = ba
output c ==> ba
K = b

save pK = abb in the next entry of table

String table
0  a
1  b
...
25  z
26  ab
27  ba
28  abb

p = c = ba

c = EOF

decoding text: ababba

Note!
สำหรับแฟ้มแบบ .gif นั้น output code เป็นแบบ variable length มีขนาดตั้งแต่ < code size > + 1 บิตถึงมากที่สุด 12 บิต เมื่อ < code size > เป็น color depth อาจมีค่าเป็น 2, 3, .... 8
การเรียงของบิตของ output code จะเรียงจากขวามาซ้ายตามลำดับ เช่น output code ที่มีขนาด 5 บิต
แทนด้วย aaaaa, bbbbb, ... ตามลำดับ จะเก็บเป็นไบต์ ๆ ดังนี้

ไบต์ที่ค่า
0  bbbaaaa
1  dcccccb
2  eeeeddd
3  gffffe
4  hhhhggg
5  ...

เมื่อจาก output code มีขนาดจำนวน ดังนั้น ขนาดของตารางจะถูกจำกัดด้วย ดังนั้นเมื่อตารางเต็มจะต้องลบข้อมูลเก่าแล้วเริ่มต้นใหม่ ซึ่งต้องมี code ที่จะหยุดการทำงานให้ decoder คือ clear code ซึ่งมีค่าเท่ากับ 2< code size > และ eof code มีค่าเท่ากับ clear code + 1 ดังนั้น output code ลัดไป (free code) จึงมีค่าเท่ากับ clear code + 2
**Lossy Compression Techniques**

1. **Quantization by shifting**

   - Original: 8 bits 8 bits 8 bits
   - Right shift 3 bits

   Leaves 15 bits per pixel. Result is still visually adequate (most of the time).

2. **Look Up Tables**

   **Example:**
   
   - 24 bits
   - R G B
   
   **Pixels**
   
   (8 bit index into 256 entries table)

   Load 256 most representative colors into table

**Quantization**

Artifacts due to limited intensity resolution. For example, Uniform Quantization

\[ P(x,y) = \text{trunc}(I(x,y)+0.5) \]

Image with decreasing bits per pixel (notice contouring):

- 8 4 2 1 bit(s)
Reducing Effects of Quantization

- Halftoning
  Many devices are bilevel – they produce just two intensity levels. How can we expand the range of available intensities? The answer lies in the spatial integration that our eyes perform. If we view a very small area from a sufficiently large viewing distance, our eyes average fine detail within the small area and record only the overall intensity of the area. Classical halftoning uses dots of varying size to represent intensities. Area of dots proportional to intensity in image. The following figure shows 2x2 patterns to represent intensity.

- Dithering
  Distribute errors among pixels.
  - Random dither
    Randomize quantization errors (errors appear as noise)
    \[ P(x,y) = \text{trunc}(I(x,y)+\text{noise}(x,y)) \]
    , where noise is random in \([-0.5,0.5]\)
  - Ordered dither
    Pseudo-random quantization errors. Use matrix to store pattern of thresholds:
    \[
    i = x \mod n \quad \text{and} \quad j = y \mod n \quad \text{where} \quad i = 0, 1, \ldots, n-1, j = 0, 1, \ldots, n-1
    \]
    if \((i(x,y)) > D_n(i,j))\)
    \[ P(x,y) = I(x,y); \quad \text{// intensity on} \]
    \[ \text{// else} \quad \text{intensity off} \]

To obtain \(n^2+1\) intensity levels (include 0), we set up an \(n\) by \(n\) dither matrix \(D_n\), whose elements are distinct positive integers in the range 0 to \(n^2 - 1\).

Bayer's ordered dither matrices

\[
D_n = \begin{bmatrix}
4D_{n^2} + D_2(1,1)U_{n^2} \\
4D_{n^2} + D_2(2,1)U_{n^2}
\end{bmatrix}
\]

\[
D_2 = \begin{bmatrix}
3 & 1 \\
0 & 2
\end{bmatrix}
\]

\[
D_3 = \begin{bmatrix}
8 & 3 & 7 \\
5 & 1 & 2 \\
4 & 9 & 6
\end{bmatrix}
\]

\[
D_4 = \begin{bmatrix}
15 & 7 & 13 & 5 \\
3 & 11 & 1 & 9 \\
12 & 4 & 14 & 6 \\
0 & 8 & 2 & 10
\end{bmatrix}
\]

\[U_{n^2}\] is the “unity” matrix (all elements are 1).
- Error Diffusion Dither (Floyd-Steinberg dither)

Spread quantization error neighbor pixels, error dispersed to pixels right
and below.

\[
\begin{array}{c|c|c|c}
\text{column j} & \text{row i} & \text{row i+1} \\
\hline
7/16 & 3/16 & 5/16 & 1/16 \\
\end{array}
\]

Note! \(7/16 + 3/16 + 5/16 + 1/16 = 1\)

Pixel operations

Interpolation and extrapolation between two images offers a general, unifying approach to
many common point and area image processing operations. Brightness, contrast, saturation, tint,
and sharpness can all be controlled with one formula, separately or simultaneously. In several cases,
there are also performance benefits.

Linear interpolation is often used to blend two images. Blend fractions (alpha) and \((1 - \alpha)\) are used in a weighted average of each component of each pixel:

\[
\text{out} = (1 - \alpha) \times \text{in0} + \alpha \times \text{in1}
\]

Typically alpha is a number in the range \(0.0\) to \(1.0\). This is commonly used to linearly interpolate two images. What is less often considered is that alpha may range beyond the interval \(0.0\) to \(1.0\). Values above one subtract a portion of \(\text{in0}\) while scaling \(\text{in1}\). Values below \(0.0\) have the opposite effect.

Extrapolation is particularly useful if a degenerate version of the image is used as the image
to get "away from." Extrapolating away from a black-and-white image increases saturation.
Extrapolating away from a blurred image increases sharpness. The interpolation/extrapolation
formula offers one-parameter control, making display of a series of images, each differing in
brightness, contrast, sharpness, color, or saturation, particularly easy to compute, and inviting
hardware acceleration. [Image Processing By Interpolation and Extrapolation by Paul Haeberli and
Douglas Voorhies, \(\text{http://www.sgi.com/grafica/interp/index.html}\)]
Adjusting Brightness

- Simply scale pixel components
  - Must clamp to range (e.g., 0 to 255)

To control image brightness, we use pure black as the degenerate (zero alpha) image. Interpolation darkens the image, and extrapolation brightens it. In both cases, brighter pixels are affected more.

Note! \((r,g,b)\) of \(i_0 = (0,0,0)\), alpha is brightness factor

Adjusting Contrast

- Compute mean luminance \(\mathcal{L}\) for all pixels
  - \(\text{luminance} = 0.30*r + 0.59*g + 0.11*b\)

- Scale deviation from \(\mathcal{L}\) for each pixel component
  - Must clamp to range (e.g., 0 to 255)

Contrast can be controlled using a constant gray image with the average image luminance. Interpolation reduces contrast and extrapolation boosts it. Negative alpha generates inverted images with varying contrast. In all cases, the average image luminance is constant.

\[
\text{float avg} = 0.0f; \\
\text{for} (\text{int x}=0; x < \text{size.width}; x++) \\
\text{for} (\text{int y}=0; y < \text{size.height}; y++) \\
\text{avg} += (0.30f*(old_img[y][x][0]\&0xff) \\
+ 0.59f*(old_img[y][x][1]\&0xff) \\
+ 0.11f*(old_img[y][x][2]\&0xff)); \\
\text{avg} /= (\text{size.width}\*\text{size.height});
\]

Note! \((r,g,b)\) of \(i_0 = (\text{avg,avg,avg})\), alpha is contrast factor
Adjusting Saturation

To alter saturation, pixel components must move towards or away from the pixel's luminance value. By using a black-and-white image as the degenerate version, saturation can be decreased using interpolation, and increased using extrapolation. This avoids computationally more expensive conversions to and from HSV space. Repeated update in an interactive application is especially fast, since the luminance of each pixel need not be recomputed. Negative alpha preserves luminance but inverts the hue of the input image.

```java
img = new byte[size.height][size.width][3];
for (int x=0; x < size.width; x++)
    for (int y=0; y < size.height; y++)
    {
        float t;
        float l = (0.30f*(old_img[y][x][0]&0xff)
                +0.59f*(old_img[y][x][1]&0xff)
                +0.11f*(old_img[y][x][2]&0xff));
        t=(1-alpha)*l+alpha*(old_img[y][x][0]&0xff);
        if (t>255) t=255;
        else if (t<0) t = 0;
        img[y][x][2] = (byte)t; // blue
    }
```

**Note**! alpha is saturate factor

Aliasing and Antialiasing

To understand aliasing, we have to introduce some basic concepts from the field of signal processing. A **signal** is a function that conveys information. Signals are often thought of as functions of time, but can equally well be functions of other variables. Since we can think of images as intensity variations over space, we will refer to signals in the **spatial domain** (as functions of spatial coordinates, rather than in the temporal domain (as functions of time). Although images are 2D functions of two independent spatial variables (x,y), for convenience our examples will often use the 1D case of a signal spatial variable x. This case can be thought of as an infinitesimally thin slice through the images, representing intensity along a single horizontal line.
Signals can be classified by whether or not they have values at all points in the spatial domain. A **continuous signal** is defined at a continuum of positions in space; a **discrete signal** is defined at a set of discrete points in space. Before scan conversion, the projection of our 3D objects onto the view plane may be treated as a continuous 2D signal whose value at each infinitesimal point in the plane indicates the intensity at that point. In contrast, the array of pixel values in the graphic’s frame buffer is a discrete 2D signal whose value is defined only at the positions in the array. Converting a continuous signal to a finite array of values may result in a loss of information. The process of selecting a finite set of values from a signal is known as **sampling**, and the selected values are called **samples**. Once we have selected these samples, we must then display them using a process, known as **reconstruction**, that attempts to recreate the original continuous signal from the samples. The array of pixels in the frame buffer is reconstructed by the graphics system’s display hardware, which converts these discrete intensity values to continuous, analog voltages that are applied to the CRT’s electron gun.
For example, if we consider a single sinusoidal function of $x$, it is easily seen that the relationship between the sampling frequency and the function is as shown below:

\[ R(x) \]

Representative of the sampling of a sine wave. (a) Sampling interval is less than one-half the period of the sine wave. (b) Sampling interval is equal to one-half the period of the sine wave. (c) Sampling interval is greater than one-half the period of the sine wave. (d) Sampling interval is much greater than one-half the period of the sine wave.
If the sampling frequency is greater than twice the frequency of the sinusoid (a) then no information is lost. If the sampling frequency is equal to twice the sine wave frequency (b) then the samples can coincide with the sine wave zero crossing as shown, and no information can be recovered from the concerning the sine wave. When the frequency is less than twice that of the sine wave (c and d) then the information contained in the samples implies sine waves (shown by the dotted line) at lower frequencies than the function being sampled. These lower frequencies are known as aliases.

**Aliasing**
Artifacts due to under-sampling or poor reconstruction.

**Spatial Aliasing**
Artifacts due to limited spatial resolution.

หมายเหตุ samples are shown as yellow dots. Left and right objects are sampled, but the others are not.
• Temporal Aliasing
  Artifacts due to limited temporal resolution, e.g. strobing or flickering

Antialiasing
• Sample at higher rate
  - Not always possible
  - Doesn’t solve problem

• Oversampling or Super Sampling
  Compute several intensity for each physical pixel. For example, pretend that each pixel is actually several smaller pixels. Compute color at each subpixels and display average color of subpixels.

Example: resolution of real display is 256x256

```
   256

   256
```

Compute image at resolution of 512x512 instead.

```
  512

  512
```

Compute color of 4 subpixels, average and write into physical pixel.

• Pre-filter to form bandlimited signal
  - Form bandlimited function (low-pass filter)
  - Trades alias for blurring
Sampling Theory

Sampling theory provides an elegant mathematical framework to describe the relationship between a continuous signal and its samples. So far, we have considered signals in the spatial domain. A signal may also be considered in the frequency domain; that is, we may represent it as a sum of sine waves, possibly offset from each other (the offset is called phase shift), and having different frequencies and amplitudes. Each sine wave represents a component of the signal’s frequency spectrum. We sum these components in the spatial domain by summing their values at each point in space.

A signal in the spatial domain is the sum of phase-shifted sines. Each component is shown with its effect on the signal shown at its right. (a) Approximation of a square wave. (b) Approximation of intensity plot of Mandril.
Determining which sine waves must be used to represent a particular signal is the central topic of **Fourier analysis**. Starting from an original signal, \( f(x) \), we can generate a different function, the Fourier transform of \( f \), called \( F(u) \), whose argument \( u \) represents frequency. The value \( F(u) \), for each frequency \( u \), tells how much (i.e., the amplitude) of the frequency \( u \) appears in the original signal \( f(x) \).

- **Spatial domain:**
  - Function: \( f(x) \)
  - Filtering: convolution

- **Frequency domain:**
  - Function: \( F(u) \)
  - Filtering: multiplication

Any signal can be written as a sum of periodic functions.

**Spectral Analysis**
• Fourier transform:

\[ F(u) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi xu} \, dx \]

• Inverse Fourier transform:

\[ f(x) = \int_{-\infty}^{\infty} F(u) e^{+i2\pi xu} \, du \]

**Convolution**

• Convolution of two functions (= filtering):

\[ g(x) = f(x) \otimes h(x) = \int_{-\infty}^{\infty} f(\lambda) h(x - \lambda) \, d\lambda \]

• Convolution theorem
  - Convolution in frequency domain is same as multiplication in spatial domain, and vice-versa
Ideal Low-Pass Filter

- Frequency domain

\[ \text{Sinc}(x) = \frac{\sin(\pi x)}{\pi x} \]

- Spatial domain

Finite low-pass filters
- Point sampling (bad)
- Triangle filter
- Gaussian filter

Triangle Filter

- Convolution with triangle filter

Input

Output
Gaussian Filter

- Convolution with Gaussian filter

Filtering

กําหนด kernel เพื่อแทน spatial frequency ของ filter (หรือ masking) เป็น matrix ขนาด n*m สมมุติว่า คือ matrix K จากนั้นใช้สูตร

\[
p = n-1 \quad q = m-1
\]

\[
\text{newpixel}(i,j) = \sum \left[ \sum K(p,q) \cdot \text{oldpixel}(i+p-(n-1)/2,j+q-(m-1)/2) \right]
\]

\[
p = 0 \quad q = 0
\]

สำหรับ

\[
i = 0, \ldots, x
\]

\[
j = 0, \ldots, y
\]

หมายเหตุ ในกรณีที่ \(i+p-(n-1)/2\) หรือ \(j+q-(m-1)/2\) มีค่าน้อยกว่า 0 กําหนดให้ \(\text{newpixel}(i,j) = \text{oldpixel}(i,j)\)

เช่น ต้องการทําภาพให้คมชัด (sharpening) กําหนด matrix K ดังนี้

\[
\begin{array}{ccc}
0.0 & -1.0 & 0.0 \\
-1.0 & 5.0 & -1.0 \\
0.0 & -1.0 & 0.0 \\
\end{array}
\]

ภาพเดิม ภาพใหม่

| 14 14 14 15 | 14 14 14 15 |
| 16 18 12 10 | 16 20 14 10 |
| 20 28 32 16 | 20 58 91 16 |
| 12 12 13 17 | 12 12 13 17 |

หมายเหตุ ถ้าค่า pixel ของภาพใหม่มีค่าลบต้องหา absolute ทำให้เป็นค่าบวก
**Adjust Blurriness**

- Convolve with a filter whose entries sum to one
  - Each pixel becomes a weighted average of its neighbors

\[
\begin{bmatrix}
\frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\
\frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\
\frac{1}{16} & \frac{1}{16} & \frac{1}{16}
\end{bmatrix}
\]

ใช้ lowpass filter (eliminate high-frequency components in the Fourier domain while leaving low-frequencies untouched) Filter has to have all positive coefficients. จะเกิด smoothing หรือ blurring.

**Edge Detection**

- Convolve with a filter that finds differences between neighbor pixels

\[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & +8 & -1 \\
-1 & -1 & -1
\end{bmatrix}
\]

ใช้ highpass filter (eliminate low-frequency components) Filter has to choose a positive value in the center location with negative coefficients in the rest of the mask and the sum is zero.
How Watercoloring Works

1. A watercolor filter transforms an image so that it looks as if it had been painted with watercolors. Above is a digital image restored from a photograph.

2. The first step in applying a watercolor filter is to smooth the colors in the image. One way to do this is to perform a median averaging process on every pixel. The color value of each pixel and its 24 neighbors are placed in a list and sorted from lowest to highest. The median (thirteenth) color value in the list is then assigned to the target pixel.
After smoothing the colors, the computer processes each pixel in the image with a sharpening kernel to make the edges stand out more.

The resulting image resembles a watercolor painting. This is just one example of the ways in which unrelated image processing techniques can be combined to produce unusual visual effects.
Guassian Function

In one dimension, the Gaussian function is the probability function of the normal distribution,

\[ f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \]

sometimes also called the frequency curve.

In two dimensions, the circular Gaussian function is the distribution function for uncorrelated variables \( X \) and \( Y \) having a bivariate normal distribution and equal standard deviation \( \sigma = \sigma_x = \sigma_y \),

\[ f(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{((x-\mu_x)^2 + (y-\mu_y)^2)}{2\sigma^2}}. \]

The corresponding elliptical Gaussian function corresponding to \( \sigma_x \neq \sigma_y \) is given by

\[ f(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{((x-\mu_x)^2/\sigma_x^2 + (y-\mu_y)^2/\sigma_y^2)}{2}}. \]
The idea of Gaussian smoothing is to use this 2-D distribution as a `point-spread' function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we can perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point. Figure 3 shows a suitable integer-valued convolution kernel that approximates a Gaussian with a $\sigma$ of 1.0.

\[
\begin{array}{ccccc}
1 & 4 & 7 & 4 & 1 \\
4 & 16 & 26 & 16 & 4 \\
7 & 26 & 41 & 26 & 7 \\
4 & 16 & 26 & 16 & 4 \\
1 & 4 & 7 & 4 & 1 \\
\end{array}
\]

Figure 3 Discrete approximation to Gaussian function with $\sigma=1.0$

Once a suitable kernel has been calculated, then the Gaussian smoothing can be performed using standard convolution methods. The convolution can in fact be performed fairly quickly since the equation for the 2-D isotropic Gaussian shown above is separable into $x$ and $y$ components. Thus the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the $x$ direction, and then convolving with another 1-D Gaussian in the $y$ direction. (The Gaussian is in fact the only completely circularly symmetric operator which can be decomposed in such a way.) Figure 4 shows the 1-D $x$ component kernel that would be used to produce the full kernel shown in Figure 3 (after scaling by 273 and rounding). The $y$ component is exactly the same but is oriented vertically.

\[
0.006 \quad 0.061 \quad 0.242 \quad 0.383 \quad 0.242 \quad 0.061 \quad 0.006
\]

Figure 4 One of the pair of 1-D convolution kernels used to calculate the full kernel shown in Figure 3 more quickly.
Image Warping
Do mapping and resampling.

Mapping

- Define transformation
  - Describe the destination \((x, y)\) for every location \((u, v)\)
    in the source (or vice-versa, if invertible)

Example Mappings

- Scale by factor:
  - \(x = \text{factor} \times u\)
  - \(y = \text{factor} \times v\)
Example Mappings

- Rotate by $\Theta$ degrees:
  - $x = u \cos \Theta - v \sin \Theta$
  - $y = u \sin \Theta + v \cos \Theta$

- Shear in X by factor:
  - $x = u + \text{factor} \cdot v$
  - $y = v$

- Shear in Y by factor:
  - $x = u$
  - $y = v + \text{factor} \cdot u$
Other Mappings

- Any function of $u$ and $v$:
  - $x = f_x(u, v)$
  - $y = f_y(u, v)$

**Image Warping Implementation I**

- Forward mapping:
  ```
  for (int u = 0; u < umax; u++) {
    for (int v = 0; v < vmax; v++) {
      float x = f_x(u, v);
      float y = f_y(u, v);
      dst(x, y) = src(u, v);
    }
  }
  ```

- Iterate over source image

---

Department of Computer Science, Prince of Songkla University
Image Warping Implementation II

- Reverse mapping:
  
  ```
  for (int x = 0; x < xmax; x++) {
    for (int y = 0; y < ymax; y++) {
      float u = f_x^{-1}(x, y);
      float v = f_y^{-1}(x, y);
      dst(x, y) = src(u, v);
    }
  }
  ```

- Iterate over destination image
  - Must resample source
  - May oversample, but much simpler!
Resampling

- Evaluate source image at arbitrary \((u,v)\)

\((u,v)\) does not usually have integer coordinates

Point Sampling

- Take value at closest pixel:
  - \(\text{int } u = \text{trunc}(u+0.5)\);
  - \(\text{int } v = \text{trunc}(v+0.5)\);
  - \(\text{dst}(x,y) = \text{src}(u,v)\);

This method is simple, but it causes aliasing

Triangle Filtering

- Convolve with triangle filter
Triangle Filtering

- Bilinearly interpolate four closest pixels
  - $a = \text{linear interpolation of } \text{src}(u_1, v_1) \text{ and } \text{src}(u_2, v_2)$
  - $b = \text{linear interpolation of } \text{src}(u_1, v_1) \text{ and } \text{src}(u_2, v_1)$
  - $\text{dst}(x, y) = \text{linear interpolation of } "a" \text{ and } "b"$

Gaussian Filtering

- Convolve with Gaussian filter

Width of Gaussian kernel affects bluriness
Gaussian Filtering

- Compute weighted sum of pixel neighborhood:
  - Weights are normalized values of Gaussian function


Filtering Methods Comparison

- Trade-offs
  - Aliasing versus blurring
  - Computation speed

Point  Bilinear  Gaussian
Image Warping Implementation

- Reverse mapping:
  ```
  for (int x = 0; x < xmax; x++) {
    for (int y = 0; y < ymax; y++) {
      float u = f_x^{-1}(x, y);
      float v = f_y^{-1}(x, y);
      dst(x, y) = resample_src(u, v, w);
    }
  }
  ```

  ![Source image](image1)
  ![Destination image](image2)

Image Warping Implementation

- Reverse mapping:
  ```
  for (int x = 0; x < xmax; x++) {
    for (int y = 0; y < ymax; y++) {
      float u = f_x^{-1}(x, y);
      float v = f_y^{-1}(x, y);
      dst(x, y) = resample_src(u, v, w);
    }
  }
  ```

  ![Source image](image1)
  ![Destination image](image2)
Example: Scale

- Scale (src, dst, sx, sy):
  
  ```c
  float w = max(1.0/sx,1.0/sy);
  for (int x = 0; x < xmax; x++) {
    for (int y = 0; y < ymax; y++) {
      float u = x / sx;
      float v = y / sy;
      dst(x,y) = resample_src(u,v,w);
    }
  }
  ```

Example: Rotate

- Rotate (src, dst, theta):
  
  ```c
  for (int x = 0; x < xmax; x++) {
    for (int y = 0; y < ymax; y++) {
      float u = x*cos(-Θ) - y*sin(-Θ);
      float y = x*sin(-Θ) + y*cos(-Θ);
      dst(x,y) = resample_src(u,v,w);
    }
  }
  ```

\[ x = u \cos Θ - v \sin Θ \]
\[ y = u \sin Θ + v \cos Θ \]
Example: Fun

- Swirl (src, dst, theta):

```java
for (int x = 0; x < xmax; x++) {
    for (int y = 0; y < ymax; y++) {
        float u = rot(dist(x,xcenter)*theta);
        float v = rot(dist(y,ycenter)*theta);
        dst(x,y) = resample_src(u,v,w);
    }
}
```

Forward Mapping V.S. Inverse Mapping

Forward mapping transferring the intensity levels from the input image to the output image, pixel by pixel. If an input pixel maps to position between four output pixels, then its intensity level is divided among the four output pixels (or filtering) according to the interpolation rule.

An alternative, and more effective, implementation is achieved by the inverse mapping. In this case, the output pixels are mapped back into the input image, one at a time, to establish their intensity levels. If an output pixel falls between four input pixels, its intensity level is determined by intensity level interpolation (or filtering) of neighbor input pixels.
The forward mapping is somewhat wasteful, since many input pixels might map to positions outside the border of output image. Furthermore, each output pixel might be addressed several times, with many input pixels contributing to its final intensity level value. The inverse mapping, however, generates the output image pixel by pixel, line by line. The intensity level of each pixel is uniquely determined by one interpolation (or filtering) step between pixel neighborhood. Nevertheless, the reverse mapping approach is the more practical for general use.

Combining Images

- Image Compositing
  - Blue-Screen mattes
  - Alpha Channel
- Image Morphing
  - Combines warping and cross-dissolving
  - Feature-Based Warping

Image Compositing

Blue-Screen Matting
  Composite foreground and background images.
a) Create background image.

b) Create foreground image with blue background.

c) Insert non-blue foreground pixels into background
Alpha Channel
Encodes pixel coverage information
\( \alpha = 0 \): no coverage (or transparent)
\( \alpha = 1 \): full coverage (or opaque)
\( 0 < \alpha < 1 \): partial coverage (or semi-transparent)
Semi-Transparent Objects

- Suppose we put A over B over background G
  
  - How much of B is blocked by A?
    \[ \alpha_A \]
  
  - How much of B shows through A
    \[ (1 - \alpha_A) \]
  
  - How much of G shows through both A and B?
    \[ (1 - \alpha_A)(1 - \alpha_B) \]

- How do we combine 2 partially covered pixels?
  - 3 possible colors (0, A, B)
  - 4 regions (0, A, B, AB)

Composition Algebra

- 12 reasonable combinations

Diagram showing combinations and regions.
Image Morphing

Morphing is a technique that allows one object to transform gradually into another. Suppose we have two images from which we wish to create a sequence of movie frames. That sequence is to depict the transformation of the object in the first scene into the object in the second scene. An example would be transforming the face of the human into the face of a tiger. In a dissolve, the first image gradually fades out as the second fades in. During a dissolve points on the object are incrementally warped from their initial position to their final position. At each step in sequence, both the initial and final images are warped so that their control points map to positions intermediate between their initial and final positions. This produces two sequences in which the marked features move gradually from their initial to their final positions. A dissolve between these two sequences completes the morph operation.

Example: C = A Over B

- For colors that are not premultiplied:
  - \( C = \alpha_A A + (1-\alpha_A) \alpha_B B \)
  - \( \alpha = \alpha_A + (1-\alpha_A) \alpha_B \)

- For colors that are premultiplied:
  - \( C' = A' + (1-\alpha_A) B' \)
  - \( \alpha = \alpha_A + (1-\alpha_A) \alpha_B \)
Cross-Dissolving

- **Blend images with “over” operator**
  - alpha of bottom image is 1.0
  - alpha of top image varies from 0.0 to 1.0

\[ \text{blend}(i,j) = (1-t) \text{ src}(i,j) + t \text{ dst}(i,j) \quad (0 \leq t \leq 1) \]

- **The warping step is the hard one**
  - Aim to align features in images

How specify mapping for the warp?
Feature-Based Warping

- Beier & Neeley use pairs of lines to specify warp
  - Given p in dst image, where is p' in source image?

![Mapping Diagram]

\[ u \text{ is a fraction} \]

\[ v \text{ is a length (in pixels)} \]

Warping with Multiple Line Pairs

- Use weighted combination of points defined by each pair of corresponding lines

![Mapping Diagram]

\[ p' \text{ is a weighted average} \]
Part II Modeling

• How do we ...

  ◦ Represent 3D objects in a computer?

  ◦ Construct such representations quickly and/or automatically with a computer?

  ◦ Manipulate 3D objects with a computer?

Different methods for different object representations

3D Objects

How can this object be represented in a computer?
3D Objects

This one?

3D Objects

How about this one?
3D Objects

3D Object Representations

- Raw data
  - Point cloud
  - Range image
  - Polygon soup

- Surfaces
  - Mesh
  - Subdivision
  - Parametric
  - Implicit

- Solids
  - Voxels
  - BSP tree
  - CSG
  - Sweep

- High-level structures
  - Scene graph
  - Skeleton
  - Application specific
Point Cloud

- Unstructured set of 3D point samples
  - Acquired from range finder, computer vision, etc

Range Image

- Set of 3D points mapping to pixels of depth image
  - Acquired from range scanner
Polygon Soup

• Unstructured set of polygons
  - Created with interactive modeling systems?

Mesh

• Connected set of polygons
**Subdivision Surface**

- Coarse mesh & subdivision rule
  - Define smooth surface as limit of sequence of refinements

**Parametric Surface**

- Tensor product spline patches
  - Careful constraints to maintain continuity
Implicit Surface

- Points satisfying: $F(x,y,z) = 0$

![Polygonal Model](image1)  ![Implicit Model](image2)

Voxels

- Uniform grid of volumetric samples
  - Acquired from CAT, MRI, etc.

![Voxel Grid](image3)  ![Voxel Image](image4)

---

Bill Lorensen
SIGGRAPH '89
Course #4 Notes

FxDPH Figure 12.26
Stanford Graphics Laboratory
BSP Tree

- Binary space partition with solid cells labeled
  - Constructed from polygonal representations

CSG

- Hierarchy of boolean set operations (union, difference, intersect) applied to simple shapes
Sweep

- Solid swept by curve along trajectory

Scene Graph

- Union of objects at leaf nodes
Skeleton

- Graph of curves with radii

Application Specific
Model Construction

- Interactive modeling tools
  - CAD programs
  - Subdivision surface editors :)

- Scanning tools
  - CAT, MRI, laser, magnetic, robotic arm, etc.

- Computer vision
  - Stereo, motion, etc.

- Procedural generation
  - Sweeps, fractals, grammars

Interactive Modeling Tools

- User constructs objects with drawing program
  - Menu commands, direct manipulation, etc.
  - CSG, parametric surfaces, quadrics, etc.

Scanning tools

- Acquire geometry of objects with active sensors
  - CAT/MRI
  - Laser range scanner
  - Magnetic sensor
  - Robotic arm
  - etc.
• Acquire geometry of objects with active sensors
  ◦ CAT/MRI
  ◦ Laser range scanner
  ◦ Magnetic sensor
  ◦ Robotic arm
  ◦ etc.

• Acquire geometry of objects with active sensors
  ◦ CAT/MRI
  ◦ Laser range scanner
  ◦ Magnetic sensor
  ◦ Robotic arm
  ◦ etc.

Computer Vision
• Infer 3D geometry from images
  ◦ Stereo
  ◦ Motion
  ◦ Constraints
  ◦ etc.
Procedural Modeling

- Goal:
  - Describe 3D models algorithmically

- Best for models resulting from ...
  - Repeating processes
  - Self-similar processes
  - Random processes

- Advantages:
  - Automatic generation
  - Concise representation
  - Parameterized classes of models

Procedural Modeling

- Sweeps
- Fractals
- Grammars

Example: Seashells

- Create 3D polygonal surface models of seashells

"Modeling Seashells,"
Deborah Fowler, Hans Meinhardt,
and Przemysław Prusinkiewicz,
Computer Graphics (SIGGRAPH 92),
• Sweep generating curve around helico-spiral axis

Helico-spiral definition:
$$\Theta_{t+1} = \Theta_t + \Delta \Theta$$
$$r_{t+1} = r_t \lambda$$
$$z_{t+1} = z_t / \lambda$$

• Connect adjacent points to form polygonal mesh

• Generate different shells by varying parameters

Different helico-spirals
Fractals

- Defining property:
  - Self-similar with infinite resolution

Mandelbrot Set

Deterministic Fractal Generation

- General procedure:
  - Initiator: start with a shape
  - Generator: replace subparts with scaled copy of original

Apply generator repeatedly
**Statistical Fractal Generation**

- **General procedure:**
  - **Initiator:** start with a shape
  - **Generator:** replace subparts with a self-similar random pattern

![Fractal Generation Diagram]

**Random Midpoint Displacement**

- **Example: terrain**

![Terrain Fractal Diagram]

**Grammars**

- **Useful for creating plants**

  \[
  \text{Tree} \rightarrow \text{Branch Tree} \quad | \quad \text{Leaf}
  \]

  \[
  \text{Branch} \rightarrow \text{Cylinder} \quad | \quad [\text{Tree}]
  \]

![Grammar Diagram]
• Useful for creating plants

![Image of fractal plant]

In this section, we will discuss the use of polygon mesh.

### Polygons

We wish to be able to represent a surface. Our basic surface primitive is a polygon (a many-sided figure). A polygon may be represented as a number of line segments connected end to end to form a closed figure. Alternatively, it may be represented as the points where the sides of the polygon are connected. The line segments which make up the polygon boundary are called *sides* or *edges*. The endpoints of the sides are called the polygon *vertices*. The simplest polygon is the triangle, having three sides and three vertex points.

![Image of triangle]

Polygon can have any number of vertices (at least 3). In practice, 3 or 4 is the most common, 6 is usually highest. We can divide polygons into two classes: *convex* and *concave*. A convex polygon is a polygon such that for any two points inside the polygon, all points on the line segment connecting them are also inside the polygon. A concave polygon is one which is not convex. A triangle is always convex.

![Images of convex and concave polygons]

**convex polygons**  
**concave polygons**
Polygon Representation

A. Direct Vertex List

In vector list, first point of a closed object is usually repeated, for example

```
0  0  0
0 10  0
10  0  0
0  0  0
```

In direct vertex list, first point is usually not repeated,

```
0  0  0
0 10  0
10  0  0
```

Disadvantage: For example

```
A   B   D   E
B   C   E   F
E   F   H   I
D  E  G  H
```

with 4-sided polygons, every vertex is included in 4 vertex lists! (wasted 300% of memory).

B. Indirect Vertex List
Disadvantage: For example

Pair occurs in 2 polygon vertex lists. Data occupies twice as much as it should.
This is a popular way as try and reduce memory requirements.
Disadvantage: Takes more to lookup geometry.

Representing Objects by Polygons (Polygon Meshes)

We assume that
- all polygon's vertices lie in one plane
- edges don't cross, e.g.

This is the classic representational form in three-dimensional graphics. An object is represented by a mesh of polygon facets. For example,
Part II | Rendering

Camera Models
The most common model is pin-hole camera.

Camera Parameters

- Position
  - Eye position \((px, py, pz)\)

- Orientation
  - View direction \((dx, dy, dz)\)
  - Up direction \((ux, uy, uz)\)

- Aperature
  - Field of view \((xfov, yfov)\)

- Film plane
  - “Look at” point
  - View plane normal
Moving the camera

View Frustum
The Rendering Pipeline

- **Local space**
  - Modeling and Positioning of component parts of object

- **World space**
  - Specification of lighting and surface attributes of object

- **Eye space**
  - Back-face culling

- **Screen space**
  - Clipping against view frustum
  - Rendering: hidden surface removal, scan conversion, shading

- Modeling Transformation
- Viewing Transformation
- Screen Transformation
3D Coordinates

Right-handed and left-handed three-dimensional coordinate systems are shown in the following figure.

The difference between the two systems is the sense of the z axis. Rotating your fingers around the z axis, from the positive x axis to the positive y axis, gives a different z direction for your thumb depending on which system is used. Right-handed systems are the standard mathematical convention although left-handed systems have been, and still are, used in the special context of viewing systems in computer graphics. When display 3D coordinate need to convert to 2D coordinate.

Modeling Transformations

Geometric Transformations, 2D

Assume 2D vector line drawing, array of vector endpoint i.e. vector list.

Transformations: Scaling, Rotation, Translation

Scaling

To scale vector, scale vector endpoints and draw line between new endpoints.

General case: \((x, y) \rightarrow (S_x x, S_y y)\)

- \(S_x = S_y\) \(\Rightarrow\) no distortion
- \(S_x \neq S_y\) \(\Rightarrow\) distortion
- \(S = 1\) \(\Rightarrow\) no effect
- \(S > 1\) \(\Rightarrow\) enlarge
- \(S < 0\) \(\Rightarrow\) reflection
- \(S = 0\) \(\Rightarrow\) large coordinating (object shrink to point \((0,0)\))
- \(0 < S < 1\) \(\Rightarrow\) shrink

Note! If the scale factors were greater than 1, the image would be both larger and further from the origin. But if the scale factors were less than 1, the image would be both smaller and closer to the origin.
**Question?** Can undo a scaling transformation?

**Answer:** Scaling by $(S_x, S_y)$ can be undone by scaling by $(1/S_x, 1/S_y)$. This is the inverse of original scaling.

Matrix representation scaling: represent a point in 2D by a column vector (matrix with 2 rows 1 columns i.e. 2x1 matrix) \[
\begin{bmatrix}
x \\
y
\end{bmatrix}
\]. Scaling is represented by a matrix size 2x2 \[
\begin{bmatrix}
S_x & 0 \\
0 & S_y
\end{bmatrix}
\].

To apply this scaling to point multiply scaling matrix times points matrix.

\[
\begin{bmatrix}
S_x & 0 \\
0 & S_y
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix} = \begin{bmatrix}
S_x x \\
S_y y
\end{bmatrix}
\]

**Note!** Matrix multiplication is not commutative.
We caution reader that some graphics textbooks use a convention of premultiplying matrixes by row vectors, rather than postmultiplying by column vectors. Matrixes must be transposed to go from one convention to the other, just as the row and column vectors are transposed:

\[(P \cdot M)^T = M^T \cdot P^T\]

**Rotation**

Points can be rotated through an angle \(\theta\) about the origin. Rotation about an arbitrary point is discussed in section Compound Transformations.

For example, shows the rotation of the house by \(\theta = 45^\circ\).

To prepare for the discussion of rotations we shall review some basic trigonometry. Suppose we have a point \(P_1 = (x_1, y_1)\) and we rotate it about the origin by an angle \(\theta\) to get a new position \(P_2 = (x_2, y_2)\). We wish to find a transformation which will change \((x_1, y_1)\) into \((x_2, y_2)\). But, before we can check any transformation to see if it is correct, we must first know what \((x_2, y_2)\) should be in terms of \((x_1, y_1)\) and \(\theta\). To determine this we shall need the trigonometric functions sine and cosine (abbreviated \(\sin\) and \(\cos\)). We can define \(\sin\) and \(\cos\) for an angle \(\theta\) in the following manner. Let us draw a line segment from the origin at the angle \(\theta\) counterclockwise from the x axis, and suppose that the line segment we have draw has length \(L\).

The line segment will then have endpoints \((0, 0)\) and \((x, y)\) and length \(L = (x^2 + y^2)^{1/2}\).

Then, the ratio of the height of the \((x, y)\) endpoint above the x axis (the y-coordinate value) and the length of the segment will be the sine of the angle

\[\sin \theta = \frac{y}{(x^2 + y^2)^{1/2}}\]
and the ratio of the distance to the right of the y axis (the x-coordinate value) and the length of the segment will be the cosine of the angle
\[
\cos \theta = \frac{x}{(x^2 + y^2)^{1/2}}
\]
Note that if we draw a segment with length \(L = 1\), then
\[
\sin \theta = y \quad \text{and} \quad \cos \theta = x
\]
To determine the form for the rotation transformation matrix, consider the point \((1, 0)\). If we rotate this point counterclockwise by an angle \(\theta\), it becomes \((\cos \theta, \sin \theta)\)

\[
\begin{pmatrix}
\cos \theta & \sin \theta \\
-c\sin \theta & c\cos \theta
\end{pmatrix} =
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix} =
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\]

If we rotate the point \((0, 1)\) counterclockwise by an angle \(\theta\), it becomes \((-\sin \theta, \cos \theta)\),

\[
\begin{pmatrix}
-s\sin \theta & \cos \theta \\
-c\sin \theta & c\cos \theta
\end{pmatrix} =
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix} =
\begin{pmatrix}
a & b \\
c & d
\end{pmatrix}
\]
We can see the values of \(a\), \(b\), \(c\), and \(d\) needed to form the rotation matrix.

OR

\[
\begin{pmatrix}
r\sin(\theta+\phi) \\
r\cos(\theta+\phi)
\end{pmatrix} =
\begin{pmatrix}
r\sin \theta \cos \phi + r\cos \theta \sin \phi \\
r\cos \theta \cos \phi - r\sin \theta \sin \phi
\end{pmatrix}
\]
\[
\begin{pmatrix}
r \cos(\theta+\phi) \\
r \sin(\theta+\phi)
\end{pmatrix} =
\begin{pmatrix}
r \cos \theta \cos \phi - r \sin \theta \sin \phi \\
r \sin \theta \cos \phi + r \cos \theta \sin \phi
\end{pmatrix}
\]

Department of Computer Science, Prince of Songkla University
\[ x' = r\cos(\theta + \phi) = r\cos(\theta)\cos(\phi) - r\sin(\phi)\sin(\theta) \]
\[ y' = r\sin(\theta + \phi) = r\sin(\theta)\cos(\phi) + r\cos(\theta)\sin(\phi) \]
but \( x = r\cos(\phi) \) and \( y = r\sin(\phi) \)
so,
\[ x' = x\cos(\theta) - y\sin(\theta) \]
\[ y' = x\sin(\theta) + y\cos(\theta) \]

The transformation matrix for a counterclockwise rotation of \( \theta \) about the origin is
\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\]

Apply row points vector times this matrix.
\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} (x \cos \theta - y \sin \theta) \\ (x \sin \theta + y \cos \theta) \end{bmatrix}
\]

However, we use column vector, so the rotation matrix should be,
\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\]

Apply this matrix times points matrix,
\[
\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{bmatrix}
\]

The sign of an angle determines the direction of rotation. We have defined the rotation matrix so that a positive angle will rotate the image in a counterclockwise direction with respect to the axes. In order to rotate in the clockwise direction we use a negative angle, so the rotation matrix in row vector form for an angle \( \theta \) clockwise would be
\[
\begin{bmatrix}
\cos (-\theta) & \sin (-\theta) \\
-\sin (-\theta) & \cos (-\theta)
\end{bmatrix}
\]
and in the column vector form would be,

\[
\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta 
\end{bmatrix}^T
\]

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\]

**Translation**

\((x, y) \rightarrow (x + D_x, y + D_y)\)

eample

<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However, cannot represent as 2x2 matrix. No way to add a constant to a coordinate using 2x2 matrix.

**Homogeneous Coordinate Transformation Matrices**

Represent every points in 2D by a 3D point.

\[(x, y) \rightarrow (x, y, 1)\]

2D cartesian coordinate Homogeneous Coordinate

Given Homogeneous Coordinate there can find 2D point it represent.

\[(x, y) \rightarrow (x, y)\]

\[(x, y, w) \rightarrow (x/w, y/w, w/w) \text{ where } w \neq 0\]

Now translation matrix is 3x3

\[
\begin{bmatrix}
1 & 0 & D_x \\
0 & 1 & D_y \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
= 
\begin{bmatrix}
x + D_x \\
y + D_y \\
1
\end{bmatrix}
\]
If points are expressed in homogeneous coordinates, all three transformations can be treated as multiplication. Redo other transformations:

**Scaling**

\[
\begin{bmatrix}
S_x & 0 & 0 \\
0 & S_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

**Rotation**

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

**Inverse Transformations**

We have seen how to use transformations to map each point \((x, y)\) into a new point \((x', y')\). Sometimes, however, we are faced with the problem of undoing the effect of a transformation; given the transformed point \((x', y')\), we must find the original point \((x, y)\). It can be done by matrix inversion.

The inverse of a matrix is another matrix such that when the two are multiplied together, the identity matrix results.

If the inverse of matrix \(T\) is \(T^{-1}\), then \(TT^{-1} = T^{-1}T = I\).

To see that this is what we need, consider equation \(TP_1 = P_2\). If we multiply both sides of this equation by the inverse of transformation matrix \(T\), we get

\[
T^{-1}TP_1 = T^{-1}P_2
\]

\[
P_1 = T^{-1}P_2
\]

This shows that the inverse of \(T\) transforms \(P_2\) back into \(P_1\).

**Note!** How to find inverse of a matrix, see appendix.

The inverse of translation matrix is

\[
\begin{bmatrix}
1 & 0 & -D_x \\
0 & 1 & -D_y \\
0 & 0 & 1
\end{bmatrix}
\]

The inverse of scaling matrix is

\[
\begin{bmatrix}
1/S_x & 0 & 0 \\
0 & 1/S_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
The inverse of rotation matrix is
\[
\begin{bmatrix}
\cos \theta & \sin \theta & 0 \\
-\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

**Compound Transformations**

**Example 1**, consider the rotation of an object about some arbitrary point P1. Because we know how to rotate only about the origin, we convert our original (difficult) problem into three separate (easy) problems. Thus, to rotate about P1, we need a sequence of three fundamental transformations:

1. Translate such that P1 is at the origin
2. Rotate
3. Translate such that the point at the origin returns to P1.

**General Idea:**
- use transformation to get situation into a simple form
- perform desired operation
- use inverse to get situation back into original form

In the example 1:
1. Translate so that P is at (0, 0) i.e. translate by (-p, -q)
2. Rotate by \( \theta \) around origin
3. Translate so that P is as original position i.e. translate by (p, q)

To use matrix to do:
1. \[ M_1 = \begin{bmatrix} 1 & 0 & -p \\ 0 & 1 & -q \\ 0 & 0 & 1 \end{bmatrix} \]

2. \[ M_2 = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

3. \[ M_3 = \begin{bmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & 1 \end{bmatrix} \]

To transform vector endpoints:

\[
\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow M_3 M_2 M_1 \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
\]

In practice premultiply transformation matrices \( M_t = M_3 M_2 M_1 \) then use \( M_t \) to transform points

\[
\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \rightarrow M_t \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
\]

**Example 2**, suppose we wish to scale, rotate the house with P1 as the center and position to P2 as shown below,
Example 3 Reflect image through x axis

This is done by scaling Y coordinate by –1:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

Example 4 Reflect through some other line, easy case: line is parallel to a coordinate axis.

1. translate so that line l is along x axis

\[
M_1 = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & -k \\
0 & 0 & 1 \\
\end{bmatrix}
\]

2. Reflection

\[
M_2 = \begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]
3. Inverse of $M_1$

$$M_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix}$$

**Example 5** Harder case, line is not parallel to coordinate

1. $M_1$: translate such that $l$ passes through origin (must know a point on $l$)
2. $M_2$: rotate so that $l$ is along x axis, i.e. rotate by $\theta$ (must find $\cos \theta$, $\sin \theta$, this can be toward from knowing another point in line).
3. $M_3$: reflect through x axis
4. $M_4 = M_2^{-1}$: rotate back from $M_2$ by $-\theta$
5. $M_5 = M_1^{-1}$: translate back

Therefore $M_T = M_5M_4M_3M_2M_1$

**Shear Transformations**

The shear transformation cause the image to slant. The y shear preserves all x-coordinate values but shifts the y value. The amount of change in the y value depends upon the x position.

![Y shear matrix](image)

Y shear matrix

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Similarly can have X shear,

X shear matrix

$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
Geometric Transformations, 3D

Just as 2D transformations can be represented by 3x3 matrices using homogeneous coordinates, so 3D transformations can be represented by 4x4 matrices. Thus, instead of representing a point as (x, y, z), we represent it as (x, y, z, w) or (x/w, y/w, z/w, 1) where w not equal to 0.

The 3D coordinate system used in this text is right-handed. By convention, positive rotations in a right-handed system are such that, when looking from a positive axis toward the origin, a 90° counterclockwise rotation will transform one positive axis into the other. This table follows from this convention:

<table>
<thead>
<tr>
<th>If axis of rotation is</th>
<th>Direction of positive rotation is</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y to z</td>
</tr>
<tr>
<td>y</td>
<td>z to x</td>
</tr>
<tr>
<td>z</td>
<td>x to y</td>
</tr>
</tbody>
</table>

right-handed coordinate system

The reader is warned that not all graphics texts follow this convention. We use a right-handed system here because it is the standard mathematical convention, even though it is convenient in 3D graphics to think of a left-handed system superimposed on the face of a display, since a left-handed system gives the natural interpretation that larger z values are further from the viewer. Notice that, in a left-handed system, positive rotations are clockwise when looking from a positive axis toward the origin.

A. Scaling

\[
\begin{bmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

inverse is obvious
B. Translation

\[
\begin{bmatrix}
1 & 0 & 0 & D_x \\
0 & 1 & 0 & D_y \\
0 & 0 & 1 & D_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

inverse is very obvious

\[
\begin{bmatrix}
1 & 0 & 0 & -D_x \\
0 & 1 & 0 & -D_y \\
0 & 0 & 1 & -D_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

C. Rotation

I. The 2D rotation is just a 3D rotation about the z axis, which is

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

หมายเหตุ

\[(0, 0) \rightarrow (1, 0) \quad \text{on the } x \text{ axis,}
\]

so,

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
a \\
b \\
c \\
d
\end{bmatrix}
= \begin{bmatrix}
a \\
b
\end{bmatrix}
\]
If we rotate the point \((0, 1)\) counterclockwise by an angle \(\theta\), it becomes \((- \sin \theta, \cos \theta)\),

![Diagram showing rotation of point](image)

so,

\[
\begin{bmatrix}
-\sin \theta & \cos \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
= \begin{bmatrix} 0 & 1 \\ c & d \end{bmatrix}
\begin{bmatrix} a & b \\ c & d \end{bmatrix}
= \begin{bmatrix} a & b \\ c & d \end{bmatrix}
\]

We can see the values of \(a, b, c,\) and \(d\) needed to form the rotation matrix. The transformation matrix for a counterclockwise rotation of \(\theta\) about the origin is

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\begin{bmatrix} x \\ y \end{bmatrix}
= \begin{bmatrix} a & b \\ c & d \end{bmatrix}
\begin{bmatrix} x \\ y \end{bmatrix}
\]

II. The \textit{x-axis} rotation matrix is

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

หมายเหตุ

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
= \begin{bmatrix} 0 & 1 \\ c & d \end{bmatrix}
\begin{bmatrix} a & b \\ c & d \end{bmatrix}
= \begin{bmatrix} a & b \\ c & d \end{bmatrix}
\]

Department of Computer Science, Prince of Songkla University
If we rotate the point \((0, 1)\) counterclockwise by an angle \(\theta\), it becomes \((- \sin \theta, \cos \theta)\),

\[
\begin{pmatrix}
- \sin \theta \\
\cos \theta
\end{pmatrix}
\]

so,

\[
\begin{bmatrix}
-a & b \\
c & d
\end{bmatrix}
\]

We can see the values of \(a, b, c,\) and \(d\) needed to form the rotation matrix. The transformation matrix for a counterclockwise rotation of \(\theta\) about the origin is

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
- \sin \theta & \cos \theta
\end{bmatrix}
\]

III. The \textit{y-axis} rotation matrix is

\[
\begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
- \sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

If we rotate the point \((0, 1)\) counterclockwise by an angle \(\theta\), it becomes \((- \sin \theta, \cos \theta)\),
so, \[
\begin{bmatrix}
-\sin \theta & \cos \theta \\
\end{bmatrix}
= \begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
a & b \\
c & d
\end{bmatrix}
= \begin{bmatrix}
c & d
\end{bmatrix}
\]

We can see the values of a, b, c, and d needed to form the rotation matrix. The transformation matrix for a counterclockwise rotation of \( \theta \) about the origin is

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\]

Note! Rotation about x-axis, y-axis, and z-axis called "pitch", "yaw", and "roll" respectively, and inverse is just changed sign of \( \theta \).

D. Shear transformations

xy-shear (shearing about z axis)

\[
\begin{bmatrix}
1 & 0 & Sh_x & 0 \\
0 & 1 & Sh_y & 0
\end{bmatrix}
\]

\[
SH_{xy}(Sh_x,Sh_y) =
\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

xz-shear (shearing about y axis)
$$\begin{bmatrix}
1 & Sh_x & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & Sh_y & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}$$

\( SH_{xz}(Sh_x,Sh_z) = \)

yz-shear (shearing about x axis)

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

E. Structure-deforming transformations

Barr (“Global and local deformations of solid primitives”, Computer Graphics, 18(3), 21-30, 1984) uses a formula definition for the transformations:

\[ X = F_x(x) \]

\[ Y = F_y(y) \]

\[ Z = F_z(z) \]

where \((x,y,z)\) is a vertex in an undeformed solid and \((X,Y,Z)\) is the deformed vertex.

a. Tapering is developed from scaling by choosing a tapering axis and differentially scale the other two components. For example, to taper an object along its Z axis:

\[ X = rx \]

\[ Y = ry \]

\[ Z = z \]

where \(r = f(z)\) is a linear or non-linear tapering function.

b. Twisting can be developed as a differential rotation just as tapering is a differential scaling. To twist an object about its z axis we apply:

\[ X = x\cos\theta - y\sin\theta \]

\[ Y = x\sin\theta + y\cos\theta \]

\[ Z = z \]

where \(\theta = f(z)\).
c. Bending along an axis is a composite transformation comprising a bent region and a region outside the bent region where the deformation is a rotation and a translation. Barr defines a bend region along the Y axis as:

\( y_{\text{min}} \leq y \leq y_{\text{max}} \)

the radius of curvature of the bend is \( \frac{1}{k} \) and the center of the bend is at \( y = y_0 \). The bending angle is:

\( \theta = k(y' - y_0) \)

where:

\[
\begin{align*}
  y' &= \begin{cases} 
    y_{\text{min}} & y \leq y_{\text{min}} \\
    y & y_{\text{min}} < y < y_{\text{max}} \\
    y_{\text{max}} & y \geq y_{\text{min}} 
  \end{cases}
\end{align*}
\]

The deforming transformation is given by:

\[
X = x \\
Y = \begin{cases} 
  -\sin(\theta - \frac{1}{k}) + y_0 & y_{\text{min}} \leq y \leq y_{\text{max}} \\
  -\sin(\theta - \frac{1}{k}) + y_0 + \cos(\theta)(y - y_{\text{min}}) & y < y_{\text{min}} \\
  -\sin(\theta - \frac{1}{k}) + y_0 + \cos(\theta)(y_{\text{max}} - y_{\text{min}}) & y > y_{\text{max}} 
\end{cases}
\]

\[
Z = \begin{cases} 
  -\cos(\theta - \frac{1}{k}) + y_0 & y_{\text{min}} \leq y \leq y_{\text{max}} \\
  -\cos(\theta - \frac{1}{k}) + \frac{1}{k} + \sin(\theta)(y - y_{\text{min}}) & y < y_{\text{min}} \\
  -\cos(\theta - \frac{1}{k}) + \frac{1}{k} + \sin(\theta)(y_{\text{max}} - y_{\text{min}}) & y > y_{\text{max}} 
\end{cases}
\]
Original object

Tapering

Twisting

Bending
Compound Transformations, 3D

Example 1 Reflection through a coordinate plane

Reflection through $XY$ plane

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

or reflection through $XZ$ plane

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

or $YZ$ plane

\[
\begin{bmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Example 2 กำหนดให้ระบบที่มีจุด $P = (14, 45, -52)$ อยู่บนระนาบ ขนานกับระนาบ $YZ$ จงเขียน matrix ต่าง ๆ ที่จำเป็นในการทำ reflection ของจุดดังกล่าว

1. translate $P$ ให้อยู่บนระนาบ $YZ$

\[
M_1 = \begin{bmatrix}
1 & 0 & 0 & -14 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
2. reflection \( P \) ที่ระนาบ \( YZ \)

\[
M_2 = \begin{bmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

3. translate \( P \) กลับ

\[
M_3 = M_1^{-1} = \begin{bmatrix}
1 & 0 & 0 & 14 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[M_1 = M_3 M_2 M_1\]

The Window-To-Viewport Transformation

Window is a rectangular region in world coordinates that part of data which is to be displayed. Viewport is a rectangular region on device in which data is to be displayed.

The window in world coordinates and the viewport in device coordinates determine the mapping that is applied to all the output primitives in world coordinates.

The effect of drawing output primitives with two viewports. Output primitives specifying the house were first drawn with viewport 1, the viewport was changed to viewport 2, and then the application program again called the graphics package to draw the output primitives.
Given a window and viewport, what is the transformation matrix that maps the window from world coordinates into the viewport in device coordinates? This matrix can be developed as a three-step transformation composition as shown below:

Maximum range of screen coordinates \((D_{u_{\text{max}}} D_{v_{\text{max}}})\)

\[
\begin{align*}
\text{Window in world Coordinates} & \quad \text{Window translated to origin} & \quad \text{Window scaled to size of viewport translated by \((u_{\text{min}}, v_{\text{min}})\) to final position} \\
(x_{\text{max}}, y_{\text{max}}) & \quad (x_{\text{min}}, y_{\text{min}}) & \quad (u_{\text{max}}, v_{\text{max}}) & \quad (u_{\text{min}}, v_{\text{min}}) \\
\end{align*}
\]

\[
M_1 = \begin{bmatrix} 1 & 0 & -x_{\text{min}} \\ 0 & 1 & -y_{\text{min}} \\ 0 & 0 & 1 \end{bmatrix}
\]

\[
M_2 = \begin{bmatrix} u_{\text{max}} - u_{\text{min}} & 0 & 0 \\ x_{\text{max}} - x_{\text{min}} & v_{\text{max}} - v_{\text{min}} & 0 \\ 0 & y_{\text{max}} - y_{\text{min}} & 1 \end{bmatrix}
\]

\[
M_3 = \begin{bmatrix} 1 & 0 & u_{\text{min}} \\ 0 & 1 & v_{\text{min}} \\ 0 & 0 & 1 \end{bmatrix}
\]

Therefore, the overall matrix \(M_{wv} = M_3 M_2 M_1\). Display a point \([x, y, 1]\) in world coordinates on viewport in device coordinates at \(M_{wv} [x, y, 1]^T\) or

\[
\begin{bmatrix} (x - x_{\text{min}}) \cdot (u_{\text{max}} - u_{\text{min}}) + u_{\text{min}} \\ (y - y_{\text{min}}) \cdot (v_{\text{max}} - v_{\text{min}}) + v_{\text{min}} \end{bmatrix}, \quad 1\] or \([u,v,1]\).
**Viewing Transformation**

- Mapping from world to camera coordinates
  - Eye position maps to origin
  - Right vector maps to X axis
  - Up vector maps to Y axis
  - Back vector maps to Z axis

**Finding the viewing transformation**

- We have the camera (in world coordinates)
- We want $T$ taking objects from world to camera
  \[ p^c = T \cdot p^w \]
- Trick: find $T^{-1}$ taking objects in camera to world
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w'
  \end{bmatrix} = \begin{bmatrix}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  w
  \end{bmatrix}
  \]

- Trick: map from camera coordinates to world
  - Origin maps to eye position
  - Z axis maps to Back vector
  - Y axis maps to Up vector
  - X axis maps to Right vector
    \[
    \begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w'
    \end{bmatrix} = \begin{bmatrix}
    R_x & U_x & B_x & E_x \\
    R_y & U_y & B_y & E_y \\
    R_z & U_z & B_z & E_z \\
    R_w & U_w & B_w & E_w
    \end{bmatrix}
    \begin{bmatrix}
    x \\
    y \\
    z \\
    w
    \end{bmatrix}
    \]

- This matrix is $T^{-1}$ so we invert it to get $T$ ... easy!
Screen Transformation

The fundamental transformation that take us into screen space is the projection (projection transformation), which takes a point in the scene and projects it onto a view plane.

Parametric Representation of Lines

We can specify a point (a position in a plane) with an ordered pair of numbers $(x, y)$. Two points will specify a line. Lines are described by equations such that if a point $(x, y)$ satisfies the equations, then the point is on the line. If the two points used to specify a line are $(x_1, y_1)$ and $(x_2, y_2)$, then an equation for the line is given by

$$
y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1) + y_1
$$

A little more algebra solving for $y$ gives

$$
y = y_2 - y_1 \cdot \frac{x - x_1}{x_2 - x_1} + y_1
$$

or

$$
y = m \cdot x + b
$$

where

$$
m = \frac{y_2 - y_1}{x_2 - x_1}
$$

$$
b = y_1 - m \cdot x_1
$$

This is called the slope-intercept form of the line. There is one more useful form of the line equation called the parametric form because the $x$ and $y$ values on the line are given in terms of a parameter $t$.

This may be expressed by the equation

$$
x(t) = x_1 + t \cdot (x_2 - x_1)
$$

When $t$ is 0, $x$ is $x_1$. As $t$ increase to 1, $x$ moves uniformly to $x_2$. But for a line segment, we must have the $y$ coordinate moving from $y_1$ to $y_2$ at the same time as $x$ changes.

$$
y(t) = y_1 + t \cdot (y_2 - y_1)
$$

The two equations together describe a straight line.

In 3D we use three equations together to describe a straight line:

$$
P_2 = (x_2, y_2, z_2)
$$

$$
P_1 = (x_1, y_1, z_1)
$$

$$
x(t) = x_1 + t \cdot (x_2 - x_1)
$$

$$
y(t) = y_1 + t \cdot (y_2 - y_1)
$$

$$
z(t) = z_1 + t \cdot (z_2 - z_1)
$$
Note that with this form, we can generate the point on the line segment by letting $t$ sweep from 0 to 1; also, given the parameter value for a point on the line, we can easily test to see if the point lies within the segment boundaries.

**Projection Transformations**

I. Parallel Projection

Perhaps the simplest way is to discard the $z$ coordinate. This is a special case of a method known as parallel projection. A parallel projection is formed by extending parallel lines from each vertex on the object until they intersect the plane of the screen. The point of intersection is the projection of the vertex. We connect the projected vertices by line segments which correspond to connections on the original object. Fortunately, the projection of a line is itself a line, so only line endpoints need actually to be projected.

Our special case of discarding the $z$ coordinates is the case where the screen, or viewing surface, is parallel to the $xy$ plane, and the lines of projection are parallel to the $z$ axis.

In a general parallel projection, we may select any direction for the lines of projection (so long as they do not run parallel to the plane of the viewing surface). Suppose that the direction of projection is given by the vector $[x_p, y_p, z_p]$ and that the image is to be projected onto the $xy$ plane. If we have a point on the object at $(x_1, y_1, z_1)$, we wish to determine where the projected point $(x_2, y_2)$ will lie.

Let us begin by writing the equations for a line passing through the point $(x, y, z)$ and in the direction of projection. This is easy to do using the parameter form

- $x(t) = x_1 + t \cdot x_p$
- $y(t) = y_1 + t \cdot y_p$
- $z(t) = z_1 + t \cdot z_p$

\[
\begin{align*}
(x_2, y_2, z_2 = 0) \\
(x_p, y_p, z_p) \\
(x_1, y_1, z_1)
\end{align*}
\]
Now, we ask where does this line intersect the xy plane? That is, what are the x and y values when z is 0? If z is 0, the third equation tells us that the parameter $t$ is

$$t = -\frac{z_1}{z_p}$$

Substituting this into the first two equations gives

$$x_2 = x_1 - z_1 \cdot \frac{x_p}{z_p}$$
$$y_2 = y_1 - z_1 \cdot \frac{y_p}{z_p}$$

This projection formula is in fact a transformation which may be written in homogeneous coordinates matrix form as below:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_p/z_p & 0 \\ 0 & 1 & -y_p/z_p & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

This transformation always gives 0 for $z_2$, the z position of the view plane, but it is often useful to maintain the z information. We will find it useful as part of the hidden-surface removal process. A transformation that includes determining a z-coordinate value $z_2$ (which turns out to be the same as $z_1$) is as follows:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_p/z_p & 0 \\ 0 & 1 & -y_p/z_p & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

II Perspective Projection

An alternative projection procedure is a perspective projection. In a perspective projection, the further away an object is from the viewer, the smaller it appears. In a perspective projection, the lines of projection are not parallel. Instead, they all converge at a single point called the **center of projection**. It is the intersection of these converging lines with the plane of the screen that determine the projected image. The projection gives the image which would be seen if the viewer’s eye were located at the center of projection. The lines of projection would correspond to the paths of the light rays coming from the object to the eye.
If the center of projection is at \((x_c, y_c, z_c)\) and the point on the object is \((x_1, y_1, z_1)\), then the projection ray will be the line containing these points and will be given by

\[
x(t) = x_c + t \cdot (x_1 - x_c) \\
y(t) = y_c + t \cdot (y_1 - y_c) \\
z(t) = z_c + t \cdot (z_1 - z_c)
\]

The projected point \((x_2, y_2)\) will be the point where this line intersects the xy plane. The third equation tells us that \(t\), for this intersection point \((z = 0)\) is

\[
t = -z_c/(z_1 - z_c)
\]

Substituting into the first two equations gives

\[
x_2 = x_c - z_c \cdot (x_1 - x_c)/(z_1 - z_c) \\
y_2 = y_c - z_c \cdot (y_1 - y_c)/(z_1 - z_c)
\]

With a little algebra, we can rewrite this as

\[
x_2 = (x_c z_1 - x_1 z_c) / (z_1 - z_c) \\
y_2 = (y_c z_1 - y_1 z_c) / (z_1 - z_c)
\]

This projection can be put into the form of a transformation matrix if we take full advantage of the properties of homogeneous coordinates. The form of the homogeneous coordinates matrix for column vector is

\[
\begin{bmatrix}
-z_c/(z_1 - z_c) & 0 & x_c/(z_1 - z_c) & 0 \\
0 & -z_c/(z_1 - z_c) & y_c/(z_1 - z_c) & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

เนื่องจาก matrix ข้างต้น มีค่า \(z_1\) ติดอยู่ด้วย เราต้องการ matrix ที่ไม่มีค่า \(z_1\) สามารถเขียนใหม่ดังนี้
The above transformation maps points on an object to points on the view plane. We shall, however, find it useful to use a modified version of this transformation. We want a form which yields the same $x$ and $y$ values so that we can still display the image, but we would like to compute a $z$ value different from zero. We would like to find $z$ values such that we can preserve the depth relationship between objects, even after they are transformed. If object $A$ lies in front of object $B$, we want the perspective transformation of object $A$ to lie in front of the perspective transformation of object $B$. A form of the perspective transformation which meets these requirements is as follows:
หมายเหตุ พิจารณากรณีที่จุด center of projection ของ perspective projection อยู่ที่ (0,0,0) และ view plane อยู่บนระนาบ XY ห่างจากแกน Z เป็นระยะทาง

\[
\begin{bmatrix}
-z_c & 0 & x_c & 0 \\
0 & -z_c & y_c & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & -z_c
\end{bmatrix}
\]

\[
\begin{align*}
\text{center of projection (0, 0, 0)} & \\
\text{RHS coordinate} & \\
(x_2, y_2, z_2=d) & \\
x(t) &= t \cdot x_1 \quad y(t) = t \cdot y_1 \\
\text{เมื่อ } z_2 = z(t) = d \text{ จะได้ } t = d / z_1 \\
\text{ตั้งบัน } x_2 = d \cdot x_1 / z_1 \text{ และ } y_2 = d \cdot y_1 / z_1 \\
\text{เขียนเป็น matrix ได้ ดังนี้} & \\
\end{align*}
\]

\[
\begin{bmatrix}
d / z_1 & 0 & 0 & 0 \\
0 & d / z_1 & 0 & 0 \\
0 & 0 & d / z_1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

เนื่องจาก matrix ข้างต้น มีค่า $z_1$ ติดอยู่ด้วย เราต้องการ matrix ที่ไม่มีค่า $z_1$ สามารถเขียนใหม่ดังนี้

\[
\begin{bmatrix}
d & 0 & 0 & 0 \\
0 & d & 0 & 0 \\
0 & 0 & d & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]
แทน

\[
\begin{bmatrix}
  d & 0 & 0 & 0 \\
  0 & d & 0 & 0 \\
  0 & 0 & d & 0 \\
  0 & 0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  y_1 \\
  z_1 \\
  1 \\
\end{bmatrix}
= \\
\begin{bmatrix}
  x_2/d \cdot z_2 \\
  y_2/d \cdot z_2 \\
  z_2 \\
  1 \\
\end{bmatrix}
\]

เราวดการให้ $z_2$ เป็น $d$ ไม่ใช่ $z_1 \cdot d$ ดังนั้นหา matrix ซ้ายด้านด้วย $d$ จะได้

\[
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 1/d & 0 \\
\end{bmatrix}
\]

ตรวจสอบได้ง่าย

\[
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 1/d & 0 \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  y_1 \\
  z_1 \\
  1 \\
\end{bmatrix}
= \\
\begin{bmatrix}
  x_1 \\
  y_1 \\
  z_1 \\
  z_2/d \\
\end{bmatrix}
\]
We take $w=1$ for all $w$.

\[
\begin{bmatrix}
dx/z_1 \\
dy/z_1 \\
d \\
1
\end{bmatrix}
\]

Clipping

The technique for not showing the part of drawing outside the window is called *clipping*.

In clipping we examine each line of the display to determine whether or not it is completely inside the window, lies completely outside the window, or crosses a window boundary. If it is inside, the line is displayed; if it is outside, nothing is drawn. If it crosses the boundary, we must determine the point of intersection and draw only the portion which lies inside.

Different graphic elements may require different clipping techniques. A character, for example, may be either entirely included or omitted depending on whether or not its center lies within the window. This technique will not work for lines, and some methods used for lines will not work for polygons.
Clipping Lines

A. Cohen-Sutherland Line-Clipping (or Outcode)

A popular method for clipping lines is the Cohen-Sutherland Outcode algorithm. The algorithm makes clever use of bit operations (outcodes) to perform this test efficiently.

Each region has 4 bit code

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP</td>
<td>BOTTOM</td>
<td>RIGHT</td>
<td>LEFT</td>
</tr>
</tbody>
</table>

Given vector to draw: \( P_0 \rightarrow P_1 \), compute 4 bits code \( c_0 \) & \( c_1 \) (see CompOutCode in algorithm).

First case (trivial accept): If \( c_0 \) and \( c_1 \) are both 0000 then can draw vector (completely inside).

Second case (trivial reject): Let \( c = c_1 \text{ AND } c_2 \). If \( c \neq 0000 \), then vector is entirely invisible (completely outside). So don’t draw.

Third case: Subdivide vector at a window boundary, decide which part might be visible; reprocess that part. To subdivide at a boundary: either \( P_0 \) or \( P_1 \) is off-window (test outcode \( \neq 0000 \) ). Assume that it’s \( P_0 \) (if not change \( P_0 \) and \( P_1 \) and proceed). Examine outcode for \( P_0 \), if \( P_0 \) lies on what region, divide line according to that region. For example, TOP region divide line at top of clip rectangle.

Before,

\[
\begin{array}{c}
\text{\textbf{P}_0} \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
\end{array}
\quad \text{or} \quad
\begin{array}{c}
\text{\textbf{P}_0} \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
\end{array}
\quad \text{or} \quad
\begin{array}{c}
\text{\textbf{P}_0} \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
\end{array}
\]

After divide by line at top of clip rectangle i.e. \( y_{\text{max}} \)

After,
For BOTTOM divide by ymin, RIGHT divide by xmax, and LEFT divide by xmin.

Repeat loop by go to first case.

How to calculate the new point at the clipping boundary? In this case, we can use the formula from slope-intercept form. Recall that

\[ y = \frac{(y_1 - y_0)(x - x_0) + y_0}{x_1 - x_0} \text{ or } y = y_0 + \text{slope} \cdot (x - x_0) \text{ where } \text{slope} = \frac{y_1 - y_0}{x_1 - x_0} \]

or

\[ x = \frac{(x_1 - x_0)(y - y_0) + x_0}{y_1 - y_0} \text{ or } x = x_0 + \frac{1}{\text{slope}} \cdot (y - y_0). \]

We don’t have any problem with infinity slope because we clip with the boundary, xmax, xmin, ymax, and ymin. For example,

Because \( P_0 \) is on the TOP region,

\[ x := x_0 + (x_1 - x_0) \cdot \frac{\text{ymax} - y_0}{y_1 - y_0}; \]

\[ y := \text{ymax}. \]

Note! \( y_1 - y_0 \neq 0 \)

For other regions, see algorithm.

Note!

1. It’s possible for a vector to be subdivided once, e.g.

2. It’s possible for a vector to be subdivided twice, e.g.
3. It's possible for a vector to be subdivided three times, e.g.

![Diagram showing vector subdivision three times]

4. It's possible for a vector to be subdivided four times (the most), e.g.

![Diagram showing vector subdivision four times]

Algorithm:

```plaintext
procedure CohenSutherlandLineClipAndDraw (x0, y0, x1, y1, xmin, xmax, ymin, ymax : real; value : integer);
{ Cohen-Sutherland clipping algorithm for line P0 = (x0, y0) to P1 = (x1, y1) and clip rectangle with diagonal from (xmin, ymin) to (xmax, ymax). }

type
  edge = (LEFT, RIGHT, BOTTOM, TOP);
  outcode = set of edge;

var
  accept, done : boolean;
  outcode0, outcode1, outcodeOut : outcode;
  { Outcodes for P0, P1, and whichever point lies outside the clip rectangle}
  x, y : real;

procedure CompOutCode (x, y : real; var code : outcode);
{Compute outcode for the point (x, y)}

begin
  code := [];
  if y > ymax then code := [TOP]
  else if y < ymin then code := [BOTTOM];
  if x > xmax then code := code + [RIGHT]
  else if x < xmin then code := code + [LEFT]
  end;

begin
  accept := false; done := false;
```
CompOutCode \((x_0, y_0, \text{outcode}_0)\); CompOutCode \((x_1, y_1, \text{outcode}_1)\);

repeat

if (\text{outcode}_0 = [] ) and (\text{outcode}_1 = []) then { Trivial accept and exit}

\[
\text{begin accept := true; done := true end}
\]

else if (\text{outcode}_0 * \text{outcode}_1) <> [] then

\[
\text{done := true } \quad \text{ ( Logical intersection is true, so trivial reject and exit.)}
\]

else

\{
Failed both tests, so calculate the line segment to clip:
from an outside point to an intersection with clip edge. 
\}

\[
\text{begin}
\]

\{
At least one endpoint is outside the clip rectangle; pick it. 
\}

if \text{outcode}_0 <> [] then

\[
\text{outcodeOut := outcode}_0 \text{ else outcodeOut := outcode}_1;
\]

\{
Now find intersection point;
use formulas \( y = y_0 + \text{slope} \times (x-x_0) \), \( x = x_0 + (1/\text{slope}) \times (y-y_0) \).
\}

if \text{TOP in outcodeOut then}

\[
\text{begin } \quad \text{ ( Divide line at top of clip rectangle )} \\
\text{x := x}_0 + (x_1 - x_0) \times (\text{ymax} - y_0) / (y_1 - y_0); \\
\text{y := ymax}
\]

\[
\text{end}
\]

else if \text{BOTTOM in outcodeOut then}

\[
\text{begin } \quad \text{ ( Divide line at bottom of clip rectangle )} \\
\text{x := x}_0 + (x_1 - x_0) \times (\text{ymin} - y_0) / (y_1 - y_0); \\
\text{y := ymin}
\]

\[
\text{end}
\]

else if \text{RIGHT in outcodeOut then}

\[
\text{begin } \quad \text{ ( Divide line at right edge of clip rectangle )} \\
\text{y := y}_0 + (y_1 - y_0) \times (\text{xmax} - x_0) / (x_1 - x_0); \\
\text{x := xmax}
\]

\[
\text{end}
\]

else if \text{LEFT in outcodeOut then}

\[
\text{begin } \quad \text{ ( Divide line at left edge of clip rectangle )} \\
\text{y := y}_0 + (y_1 - y_0) \times (\text{xmin} - x_0) / (x_1 - x_0); \\
\text{x := xmin}
\]

\[
\text{end}
\]

\{
Now we move outside point to intersection point to clip,
\}
and get ready for next pass. } 

if (outcodeOut = outcode0) then

begin

\( x_0 := x; y_0 := y; \) CompOutCode \((x_0, y_0, \text{outcode0})\)

end

else

begin

\( x_1 := x; y_1 := y; \) CompOutCode \((x_1, y_1, \text{outcode1})\)

end

end \{ \text{Subdivide} \}

until done;

if accept then Line \((x_0, y_0, x_1, y_1, \text{value})\) \{ Version for real coordinates \}

end; \{ \text{CohenSutherlandLineClipAndDraw} \}

Advantages:
- Simple to program and understanding
- Fast trivial accept and reject
- Economical memory usage

Disadvantages:
- Finding intersection point might need to do 4 times intersection calculations per vector
- Somewhat awkward to build in hardware

B. **Midpoint Subdivision Line-Clipping**

Same as Cohen-Sutherland, but third case: Find point on vector halfway between \(P_1\) and \(P_2\).

\[ \begin{array}{c}
\text{P}_1 \\
\text{P}_2 \\
\text{P}_3 \\
\end{array} \]

Recursively process \(\overline{P_1P_3}\) and \(\overline{P_3P_2}\). Example above \(\overline{P_1P_3}\) will be immediate trivial reject, \(\overline{P_3P_2}\) will be further subdivide. Number of subdivisions is limited by number of bits in coordinate system resolution. To limit number of subdivisions: if \(P_1P_2\) are very closed, then draw vector and done.
Algorithm:
Clip(P1, P2) {
    If (P1 is closed to P2) then return
    c1 = code(P1);
    c2 = code(P2);
    If (c1 == 0 and c2 == 0) then
draw vector
    else if (c1 AND c2) != 0) then return
    else {
        P3.x = (P1.x + P2.x)/2
        P3.y = (P1.y + P2.y)/2
        Clip(P1, P3);
        Clip(P3, P2);
    }
}

Note!
- Optimization: divide by 2 can be replaced by shift-right 1 i.e. P3.x = (P1.x + P2.x) / 2
  can be replaced by P3.x = (P1.x + P2.x) >> 1
- Be careful! To test,
  If (P1 is closed to P2) then return
  If do this way:
  If (P1.x == P2.x) and (P1.y == P2.y) then return
  Can cause an infinite loop. For example P1.x = P2.x = 1 , P1.y = 1 and P2.y = 2;
  P3 is computed as P3.x = (1+1)/2 = 1
  P3.y = (1+2)/2 = 1
  Then recursive call P1P3, since P1 = P3 ten return.
  Now recursive call P3P2, P3 = (1,1) and P2 = (1,2) => infinite loop.
  Correction! Don’t test P1 and P2 for exact equality instead:
  If (abs(P1.x-P2.x) <= 1) and (abs(P1.y-P2.y) <=1) then return

C.
Liang–Barsky Line Clipping

Faster line clippers have been developed that are based on analysis of the parametric equation of a line segment, which we can write in the form

\[ x = x_1 + u \Delta x \]
\[ y = y_1 + u \Delta y, \quad 0 \leq u \leq 1 \] (6-3)

where \( \Delta x = x_2 - x_1 \) and \( \Delta y = y_2 - y_1 \). Using these parametric equations, Cyrus and Beck developed an algorithm that is generally more efficient than the Cohen–Sutherland algorithm. Later, Liang and Barsky independently devised an even faster parametric line-clipping algorithm. Following the Liang–Barsky approach, we first write the point-clipping conditions 6-5 in the parametric form:

\[ x_{min} \leq x_1 + u \Delta x \leq x_{max} \] (6-7a)
\[ y_{min} \leq y_1 + u \Delta y \leq y_{max} \] (6-7b)

Each of these four inequalities can be expressed as

\[ u \rho_k = q_k, \quad k = 1, 2, 3, 4 \] (6-13)
where parameters $p$ and $q$ are defined as

\[
\begin{align*}
p_1 &= -\Delta x, & q_1 &= x_1 - x_{\text{min}} \\
p_2 &= \Delta x, & q_2 &= x_{\text{max}} - x_1 \\
p_3 &= -\Delta y, & q_3 &= y_1 - y_{\text{min}} \\
p_4 &= \Delta y, & q_4 &= y_{\text{max}} - y_1
\end{align*}
\] (6-12)

Any line that is parallel to one of the clipping boundaries has $p_k = 0$ for the value of $k$ corresponding to that boundary ($k = 1, 2, 3, \text{ and } 4$ correspond to the left, right, top, and bottom boundaries, respectively). If, for that value of $k$, we also find $q_k < 0$, then the line is completely outside the boundary and can be eliminated from further consideration. If $q_k > 0$, the line is inside the parallel clipping boundary.

When $p_k < 0$, the infinite extension of the line proceeds from the outside to the inside of the infinite extension of this particular clipping boundary. If $p_k > 0$, the line proceeds from the inside to the outside. For a nonzero value of $p_k$, we can calculate the value of $u$ that corresponds to the point where the infinitely extended line intersects the extension of boundary $k$ as

\[
u = \frac{q_k}{p_k}
\] (6-13)

For each line, we can calculate values for parameters $u_1$ and $u_2$ that define the part of the line that lies within the clip rectangle. The value of $u_1$ is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ($p < 0$). For these edges, we calculate $r = q_k / p_k$. The value of $u_1$ is taken as the largest of the set consisting of 0 and the various values of $r$. Conversely, the value of $u_2$ is determined by examining the boundaries for which the line proceeds from inside to outside ($p > 0$). A value of $r$ is calculated for each of these boundaries, and the value of $u_2$ is the minimum of the set consisting of 1 and the calculated $r$ values. If $u_1 > u_2$, the line is completely outside the clip window and can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter $u$.

This algorithm is presented in the following procedure. Line intersection parameters are initialized to the values $u_0 = 0$ and $u_2 = 1$. For each clipping boundary, the appropriate values for $p$ and $q$ are calculated and used by the function clipTest to determine whether the line can be rejected or whether the intersection parameters are to be adjusted. When $p < 0$, the parameter $r$ is used to update $u_1$; when $p > 0$, parameter $r$ is used to update $u_2$. If updating $u_1$ or $u_2$ results in $u_1 > u_2$, we reject the line. Otherwise, we update the appropriate $u$ parameter only if the new value results in a shortening of the line. When $p = 0$ and $q < 0$, we can discard the line since it is parallel to and outside of this boundary. If the line has not been rejected after all four values of $p$ and $q$ have been tested, the endpoints of the clipped line are determined from values of $u_1$ and $u_2$.

```c
#include "graphics.h"
define ROUND(a) ((int)(a+0.5))
int clipTest (float p, float q, float * u1, float * u2)  
```
In general, the Liang–Barsky algorithm is more efficient than the Cohen-Sutherland algorithm, since intersection calculations are reduced. Each update of parameters \( u_i \) and \( u_j \) requires only one division; and window intersections of the line are computed only once, when the final values of \( u_i \) and \( u_j \) have been computed. In contrast, the Cohen-Sutherland algorithm can repeatedly calculate intersections along a line path, even though the line may be completely outside the clip window. And, each intersection calculation requires both a division and a multiplication. Both the Cohen-Sutherland and the Liang–Barsky algorithms can be extended to three-dimensional clipping (Chapter 12).
Rotated Windows

So far, we've assumed window boundaries are parallel to coordinate axes:

\[
\begin{array}{c}
\text{y} \\
\hline
\text{x}
\end{array}
\]

What if window boundaries are rotated?

\[
\begin{array}{c}
\text{y} \\
\hline
\text{x}
\end{array}
\]

One solution: Rotate all vector by inverse of window rotation (transforming data into simple case).

Disadvantage: Does unnecessary rotation of vector outside window.

Clip to Window v.s. Clip to Viewport

**Recommendation:** Clip to window then do window-to-viewport transformation on vectors that will be draw. Doing operation in reverse order, window-to-viewport then clip to viewport, may transform vector that are later discarded.

Miscellaneous Question

- What about window/viewport that are not rectangular? Can be done but rarely is actually done!
- Can, in fact, implement clipping to arbitrary convex/nonconvex polygon. Sutherland-Hodgman algorithm will do this.

Clipping Polygons

An algorithm that clips a polygon must deal with many different cases, as shown in the following figure:

( a ) multiple components
(b) simple convex case

(c) concave case with many exterior edges

Clipped polygon may have more edges or fewer edges from original.

The Sutherland-Hodgman Polygon-Clipping Algorithm

Four clip edges, each defining one boundary of the clip rectangle:

(a) Clip rectangle

(b) right clip boundary

(c) bottom clip boundary

(d) left clip boundary

(e) top clip boundary

The algorithm accepts a series of polygon vertices $v_1, v_2, ..., v_n$. The algorithm clips against a single clip edge and outputs another series of vertices defining the clipped polygon. In a second pass, the partially clipped polygon is then clipped against the second clip edge, and so on. The algorithm moves around the polygon from $v_n$ to $v_1$ and then back to $v_n$, at each step (each edge) examining the relationship between successive vertices (edge) and the clip edge. At each step, zero, one, or two vertices are added to the output list of vertices that defines the clipped polygon.

Four possible cases must be analyzed, as shown below:
Let's consider the polygon edge from vertex $s$ to vertex $p$. In case 1, when the polygon edge is completely inside the clip boundary, vertex $p$ is added to the output list. In case 2, the intersection point $i$ is output as a vertex because the edge intersects the boundary. In case 3, both vertices are outside the boundary, so there is no output. In case 4, the intersection point $i$ and $p$ are both added to the output list.

Procedure SutherlandHodgmanPolygonClip() accepts an array inVertexArray of vertices and creates another array outVertexArray of vertices. To keep the code simple, we show no error checking on array bounds, and we use the procedure Output() to place a vertex into outVertexArray. The procedure Intersect() calculates the intersection of the polygon edge from vertex $s$ to vertex $p$ with clip Boundary, which is defined by two vertices on the clip polygon's boundary. The function Inside() returns true if the vertex is on the inside of the clip boundary, where “inside” is defined as “to the left of the clip boundary when one looks from the first vertex to the second vertex of the clip boundary”. This sense corresponds to a clockwise enumeration of edges.
To calculate whether a point lies outside a clip boundary, see text book. For the simple case of an upright clip rectangle, we need only test the sign of the horizontal or vertical distance to its boundary. For example: right boundary,

![Diagram showing horizontal distances inside and outside a boundary]

Algorithm:

```pascal
type
  vertex = point;  { point holds real x, y }
  edge = array [1..2] of vertex;
  vertexArray = array [1..MAX] of vertex;  {MAX is a declared constant }

procedure SutherlandHodgmanPolygonClip(
  inVertexArray : vertexArray;  {Input vertex array }
  var
    outVertexArray : vertexArray;  {Output vertex array }
  inLength : integer;  {Number of entries in inVertexArray }
  var
    outLength : integer;  {Number of entries in outVertexArray }
    clipBoundary : edge);  {Edge of clip polygon }

var
  s, p,  { Start, end point of current polygon edge }
  i : vertex;  {Intersection point with a clip boundary }
  j : integer;  {Vertex loop counter }

Procedure Output(
  newVertex : vertex;
  var
    outLength : integer; vari outVertexArray : vertexArray);  {Add newVertex to outVertexArray and then updates outLength }

begin
  ...
end;

function Inside (testVertex : vertex; clipBoundary : edge) : boolean;
{Checks whether the vertex lies inside the clip edge or not }

begin
  ...
end;
```

Department of Computer Science, Prince of Songkla University
procedure Intersect (first, second : vertex; clipBoundary : edge; var intersectPt : vertex)
( Clips polygon edge (first, second) against clipBoundary, outputs the new point )
begin
...
end;

begin
outLength := 0;
s := inVertexArray [inLength];
(Start with the last vertex in inVertexArray)
for j := 1 to inLength do
begin
  p := inVertexArray [j];
  {Now s and p corresponding to the vertices in 4 cases }
  if Inside(p, clipBoundary) then
    if Inside(s, clipBoundary) then
      Output(p, outLength, outVertexArray) { case 1 }
    else
      begin
        Intersect(s, p, clipBoundary, i);
        Output(i, outLength, outVertexArray);
        Output(p, outLength, outVertexArray);
      end
  else
    { cases 2 and 3 }
    if Inside(s, clipBoundary) then
      begin
        Intersect(s, p, clipBoundary, i);
        Output(i, outLength, outVertexArray);
      end;
    { No action for case 3 }
  s := p
  { Advance to next pair of vertices }
end { for }
end. { SutherlandHodgmanPolygonClip }

EXAMPLE:

\[ \begin{array}{c}
\textbf{a} \\
\textbf{2} \\
\textbf{b} \\
\textbf{3}
\end{array} \]
inVertexArray is
1
2
3

clipBoundary is top, left and bottom => outVertexArray = inVertexArray

clipBoundary is right
s = 3 p = 1 corresponding to case 4 => output

    b

1

s = 1 p = 2 corresponding to case 2 => output

    a

s = 2 p = 3 corresponding to case 3 no output

outVertexArray is

    b

1

    a

EXAMPLE: ใช้วิธีการของ Sutherland-Hodgman-Clipping แสดงวิธีการ clipping polygon ดังล่าง

ให้ clip ตามลำดับดังนี้ left, top, right, bottom เมื่อ inVertexArray คือ

1
2
3
4

ให้แสดง outVertexArray ของการ clip แต่ละครั้ง โดยให้คำนวณจุดตัดของแต่ละขอบกับขอบolygon ที่อยู่ตรงในภาพ

Department of Computer Science, Prince of Songkla University
Solution

1. Clip left

\[ S = 4 \quad P = 1 \Rightarrow \text{output } a \]
\[ S = 1 \quad P = 2 \Rightarrow \text{output } b, 2 \]
\[ S = 2 \quad P = 3 \Rightarrow \text{output } 3 \]
\[ S = 3 \quad P = 4 \Rightarrow \text{output } 4 \]

OutVertexArray ที่มี

\[ a \]
\[ b \]
\[ 2 \]
\[ 3 \]
\[ 4 \]

2. Clip top

\[ S = 4 \quad P = a \Rightarrow \text{output } a \]
\[ S = a \quad P = b \Rightarrow \text{output } b \]
\[ S = b \quad P = 2 \Rightarrow \text{output } 2, c \]
\[ S = 2 \quad P = 3 \Rightarrow \text{output } d, 3 \]
\[ S = 3 \quad P = 4 \Rightarrow \text{output } 4 \]

OutVertexArray ที่มี

\[ a \]
\[ b \]
\[ c \]
\[ d \]
\[ 3 \]
\[ 4 \]
3. Clip right

1. $S = 4 \quad P = a \Rightarrow \text{output } a$
2. $S = a \quad P = b \Rightarrow \text{output } b$
3. $S = b \quad P = c \Rightarrow \text{output } c$
4. $S = c \quad P = d \Rightarrow \text{output } d$
5. $S = d \quad P = 3 \Rightarrow \text{output } e$
6. $S = 3 \quad P = 4 \Rightarrow \text{output } f, 4$

OutVertexArray ถ้า

a
b
c
d
e
f
4

4. Clip bottom

1. $S = 4 \quad P = a \Rightarrow \text{output } g, a$
2. $S = a \quad P = b \Rightarrow \text{output } b$
3. $S = b \quad P = c \Rightarrow \text{output } c$
4. $S = c \quad P = d \Rightarrow \text{output } d$
5. $S = d \quad P = e \Rightarrow \text{output } e$
6. $S = e \quad P = f \Rightarrow \text{output } h$
OutVertexArray คือ $g$\n\a\n\b\n\c\n\d\n\e\nh

Polygon Scan Conversion

We need to generate pairs of segment end points and fill in horizontally between them. This is usually achieved by constructing an edge list for each polygon. This can be done by using an array of linked lists. Initially, all the elements are set to nil. Then each edge of the polygon is rasterized, and the x (and z) coordinate is inserted into the linked list corresponding to that value of y. Each of the linked lists is then sorted in order of increasing x. Filling in of the polygon is then achieved, for each scan line, by taking successive pairs of x values and filling in between them (because a polygon has to be closed, there will be an even number of elements in the linked list). Note that this method is powerful enough to cope with concave polygons with holes.
Note If the object contains only convex polygons then the linked lists have only two x coordinates. There is no sort required. We can compare between two x coordinates, which one is less than.
3-10

PIXEL ADDRESSING AND OBJECT GEOMETRY

So far we have assumed that all input positions were given in terms of scan line number and pixel position number across the scan line. As we saw in Chapter 2, there are, in general, several coordinate references associated with the specification and generation of a picture. Object descriptions are given in a world reference frame, chosen to suit a particular application, and input world coordinates are ultimately converted to screen display positions. World description objects are given in terms of precise coordinate positions, which are inherently small mathematical points. Pixel coordinates, however, reference finite screen areas. If we want to preserve the specified geometry of world objects, we need to compensate for the mapping of mathematical input points to finite pixel areas. One way to do this is simply to adjust the dimensions of displayed objects to account for the amount of overlap of pixel areas with the object boundaries. Another approach is to map world coordinates onto screen positions between pixels, so that we align object boundaries with pixel boundaries instead of pixel centers.

Screen Grid Coordinates

An alternative to addressing display positions in terms of pixel centers is to reference screen coordinates with respect to the grid of horizontal and vertical pixel boundary lines spaced one unit apart (Fig. 3-28). A screen coordinate position is then the pair of integer values identifying a grid intersection position between two pixels. For example, the mathematical line path for a polyline with screen endpoints (0, 0), (5, 2), and (1, 4) is shown in Fig. 3-29.

With the coordinate origin at the lower left of the screen, each pixel area can be referenced by the integer grid coordinates of its lower left corner. Figure 3-30 illustrates this convention for an 8 by 8 section of a raster, with a single illuminated pixel at screen coordinate position (4, 5). In general, we identify the area occupied by a pixel with screen coordinates (x, y) as the unit square with diagonally opposite corners at (x, y) and (x + 1, y + 1). This pixel-addressing scheme has several advantages. It avoids half-integer pixel boundaries, it facilitates precise object representations, and it simplifies the processing involved in many scan-conversion algorithms and in other raster procedures.

The algorithms for line drawing and curve generation discussed in the preceding sections are still valid when applied to input positions expressed as screen grid coordinates. Decision parameters in these algorithms are now simply a measure of screen grid separation differences, rather than separation differences from pixel centers.

Maintaining Geometric Properties of Displayed Objects

When we convert geometric descriptions of objects into pixel representations, we transform mathematical points and lines into finite screen areas. If we are to maintain the original geometric measurements specified by the input coordinates...
for an object, we need to account for the finite size of pixels when we transform the object definition to a screen display.

Figure 3-31 shows the line plotted in the Bresenham line-algorithm example of Section 3-2. Interpreting the line endpoints (20, 10) and (30, 18) as precise grid crossing positions, we see that the line should not extend past screen grid position (30, 18). If we were to plot the pixel with screen coordinates (30, 18), as in the example given in Section 3-2, we would display a line that spans 11 horizontal units and 9 vertical units. For the mathematical line, however, $\Delta x = 10$ and $\Delta y = 8$. If we are addressing pixels by their center positions, we can adjust the length of the displayed line by omitting one of the endpoint pixels. If we think of screen coordinates as addressing pixel boundaries, as shown in Fig. 3-31, we plot a line using only those pixels that are "interior" to the line path; that is, only those pixels that are between the line endpoints. For our example, we would plot the leftmost pixel at (20, 10) and the rightmost pixel at (29, 17). This displays a line that
has the same geometric magnitudes as the mathematical line from \((20, 10)\) to \((30, 18)\).

For an enclosed area, input geometric properties are maintained by displaying the area only with those pixels that are interior to the object boundaries. The rectangle defined with the screen coordinate vertices shown in Fig. 3-32(a), for example, is larger when we display it filled with pixels up to and including the border pixel lines joining the specified vertices. As defined, the area of the rectangle is 12 units, but as displayed in Fig. 3-32(b), it has an area of 20 units. In Fig. 3-32(c), the original rectangle measurements are maintained by displaying...

**Figure 3-33**
Circle path and midpoint circle algorithm plot of a circle with radius 5 in screen coordinates.

**Figure 3-34**
Modification of the circle plot in Fig. 3-33 to maintain the specified circle diameter of 10.
The problem of exactly where the borders of a polygon lie can be solved by the following simple rasterization rules:

1. Horizontal edges are simply discarded.
2. An edge which goes from scan line $y_{\text{bottom}}$ to $y_{\text{top}}$ should generate $x$ values for scan lines $y_{\text{bottom}}$ through to $y_{\text{top}} - 1$.
3. Similarly, horizontal segments should be filled from $x_{\text{left}}$ to $x_{\text{right}} - 1$.

Example:

```
edge goes from $y_{\text{bottom}}$ to $y_{\text{top}}$ and $x_{\text{left}}$ to $x_{\text{right}}$
edge goes from $y_{\text{bottom}}$ to $y_{\text{top}} - 1$ and $x_{\text{left}}$ to $x_{\text{right}} - 1$
```

**How to compute $x$?**

Consider the following picture:

```
\[ \begin{align*}
    x(t) &= x_0 + t(x_1 - x_0) \\
    y(t) &= y_0 + t(y_1 - y_0)
\end{align*} \]

So,

\[ y_i = y(t_i) = y_0 + t_i(y_1 - y_0) \]

therefore,

\[ t_i = (y_i - y_0)/(y_1 - y_0) \]
Similarly,
\[ y_j = y(t_j) = y_0 + t_j(y_1 - y_0) \]
therefore,
\[ t_j = \frac{(y_j - y_0)}{(y_1 - y_0)} = \frac{(y_i+1 - y_0)}{(y_1 - y_0)} = t_i + \frac{1}{(y_1 - y_0)} \]
For x,
\[ x_j = x(t_j) = x_0 + t_j(x_1 - x_0) \]
\[ = x_0 + (t_i + \frac{1}{(y_1 - y_0)})(x_1 - x_0) \]
\[ = x_i + \frac{(x_1 - x_0)}{(y_1 - y_0)} \]

**Algorithm:** Assume data are vertices of polygon, \( v[1], \ldots, v[n] \)

```plaintext
for y = 0 to device_max_y
{
    edge_list_at[y] := nil;
}

vfirst :=v[1];
for i = 2 to n
{
    if (v[i-1].y == v[i].y) continue; /* discard horizontal edge */
    AddEdgeToList(v[i-1],v[i]);
}

AddEdgeToList(v[n],v[vfirst]);
for y=0 to device_max_y
{
    if (edge_list_at[y] != nil) RenderSpan(y);
}
```

where

```
AddEdgeToList(v1,v2)
{
    if (v1.y>v2.y)
    {
```
\[ v1 <- v2; /* exchange so that v1.y is less than */ \]

\[
\begin{align*}
  &dy = v2.y - v1.y; \\
  &x = v1.x; \\
  &dx = (v2.x - v1.x)/dy; \\
  &\text{for } y=v1.y \text{ to } v2.y-1 \\
  &\{ \\
  &\quad e = \text{new edge\_node}; \\
  &\quad e.x = x; \\
  &\quad \text{put } e \text{ in } \text{edge\_list\_at}[y] \text{ in ascending order of } x; \\
  &\quad x = x + dx; \\
  &\} \\
\end{align*}
\]

and

\[
\begin{align*}
  &\text{RenderSpan}(y) \\
  &\{ \\
  &\quad \text{edge\_node1} = \text{edge\_list\_at}[y]; \\
  &\quad \text{while (edge\_node1 != nil)} \\
  &\quad \{ \\
  &\quad \quad \text{edge\_node2} = \text{edge\_node1}.next; \\
  &\quad \quad x1 = \text{edge\_node1}.x; \\
  &\quad \quad x2 = \text{edge\_node2}.x; \\
  &\quad \quad \text{for } x = x1 \text{ to } x2-1 \\
  &\quad \quad \{ \\
  &\quad \quad \quad \text{putpixel}(x,y,\text{color}); \\
  &\quad \quad \} \\
  &\quad \quad \text{edge\_node1} = \text{edge\_node2}.next; \\
  &\quad \} \\
\end{align*}
\]

Hidden Surfaces Removal

We described how to obtain different views of a scene. We developed perspective projections which made constructed objects look more realistic. But for a realistic scene, we should
only draw those lines and polygons which could actually be seen, not those which would be hidden by other objects. What is hidden and what is visible depend upon the point of view. So far, we have learned how to model and project three-dimensional objects, but all parts of the objects are always displayed. This gives our drawings a transparent quality. Such figures are called wire-frame drawings because they look as if they are wire outlines of the supposedly solid objects. Then the rendering of polygon comes along. In this chapter we consider the problem of hidden surfaces removal. There exist many solutions to this problem.

**Back-Face Removal or Back-Face Culling**

There is a simple test which we can perform which will eliminate most of the faces which cannot be seen. This test identifies surfaces which face away from the viewer. They are the surfaces which make up the back of the object. They cannot be visible because the bulk of the object is in the way. This does not completely solve the hidden surface problem because we can still have the front face of an object obscured by a second object or by another part of itself. But the test can remove roughly half of the surfaces from consideration and thus simplify the problem.

How can we tell which face is front or back? Support that we make the rule that all solid objects are to be constructed out of polygons in such a way that only the light surfaces are open to the air; the dark faces meet the material inside the object. This means that when we look at an object face from the outside, it will appear to be drawn counterclockwise. So, order the endpoints of polygon edges so that they go counterclockwise when viewed from outside of object. For example,

![Diagram](image)

- **Front**: a.d.c.b
- **Side**: c.d.e.f
- **Side**: b.g.h.a
- **Back**: g.f.e.h

If a polygon is visible, the light surface should face toward us and the dark surface should face away from us. Since a cross product can be formed which gives the direction of the light face, this vector should point toward us. So, if the normal vector points toward the viewer, the face is visible (a front face), otherwise, the face is hidden (a back face) and should be removed.
How can we tell whether or not a vector points toward the viewer? Compute \( \mathbf{E} \), a vector in direction of viewer:

\[
\mathbf{E} = \mathbf{N} \theta
\]

\[\theta < 90^\circ\]  \hspace{1cm}  \[\theta > 90^\circ\]

front face  \hspace{1cm}  back face

\[
\mathbf{E} \cdot \mathbf{N} = |\mathbf{E}| |\mathbf{N}| \cos(\theta), \text{ where } |\mathbf{E}| = \sqrt{E_x^2 + E_y^2 + E_z^2}
\]

Or \( \mathbf{E} \cdot \mathbf{N} = E_xN_x + E_yN_y + E_zN_z \)

Because \( \cos(\theta) \) is positive if the vectors are pointing somewhat in the same direction and negative if they point away from each other. Try to avoid computation of \( \theta \). Instead using \( \mathbf{E} \cdot \mathbf{N} \) (the second formula), a back-facing polygon may be identified by the negative dot product, \( \mathbf{E} \cdot \mathbf{N} \), (Strictly speaking, the dot product is positive for a front-facing polygon; a zero dot product indicates a polygon being viewed on edge.) For parallel projection along z-axis on XY-plane, \( \mathbf{E} = (0,0,1) \), i.e. unit vector along z-axis. Therefore \( \mathbf{E} \cdot \mathbf{N} = N_z \).

For perspective projection, \( \mathbf{E} \) is a vector from object vertex to center of projection, \( \mathbf{E} = (x_c-x, y_c-y, z_c-z) \)
Note! - This shouldn’t be used if scene contains reflection. Because the hidden surfaces may be a font face on reflection.

General hidden surface removal algorithm

Painter’s Algorithm

Since front polygon has minimum z value that is larger than that at back polygon.

**Algorithm**: Sort all polygons so that minimum z value is increasing.

For example above, far polygon is draw first:

Advantage: Easy work.
Disadvantage:
- sorting is require
- Every part of every polygon is rendered whether or not it is visible.
- Big Problem! Polygon may be cyclically overlap in z. For example,

In these cases, it can not use painter's algorithm. It will be necessary to split one or more objects to make a linear order possible (see binary space-partitioning trees in text book.)

Environments whose objects each exist in a plane of constant z, such as window management, are said to be 2 ½ D and can be correctly handled with the painter's algorithm. Also if there is no depth ambiguities because each depth is thought of as corresponding to a different plane of constant z, we can use painter's algorithm, for example,

Z-buffer Algorithms

Avoid sorting.
Avoid subdivision.
Still draw invisible parts of polygons.

Render each polygon (in any order) into frame buffer memory, as discussed in chapter 5, except keep a z-value for every pixel.
Z-values for pixels can be computed from polygon description (see below). When pixel is a candidate for illumination (i.e., we know pixel coordinates and color/intensity that should be displayed): compute z-value from polygon information; compare this value with value in z-buffer for this pixel. If new z-value is closer to viewer than value in z-buffer, then illuminate pixel and replace value in z-buffer. If new z-value is further from viewer than value in z-buffer, then do nothing. If two values are equal then use oldvalue or newvalue or better method would be to average colors of polygons (see "anti aliasing" in later chapter).

Disadvantage:
- May be illuminating pixel multiple times.
- Use lot of memory. For example low resolution 320x200 = 64000 pixels. If z-buffer entry is an integer (2 bytes), this requires 128000 bytes. On PC with segment memory architecture, such as Intel 80xxx, this can be a problem. For example int zbuf[200][320]; won't compile. First solution,

  ```
  int *zbuf;
  ... 
  zbuf = malloc(128000);
  ```

won't run. There are sometimes library functions to allocate large amounts of memory for huge model (see halloc() of MicroSoft or Turbo C). Another solution,

```
typedef int vector[320];
vector *zbuf[200];
int i;
...
for i = 0 to 199
  zbuf[i] = (vector *) malloc (sizeof (vector));
```

Note! If there are some problems, see what model are you using. If it is tiny or small, change to large model).

Z-buffer can be kept and maintained by hardware. For example Siligon Graphics Iris, model 4D 210-GTX has 16 bit per pixel z-buffer in hardware.

Both disadvantages will be solved by scan line algorithm in the next section.

Note! The following algorithm assume that minimum z value is closer to view.

Algorithm:
```c
for y := 0 to device_max_y
  {
  
```
for x := 0 to device_max_x
{
    z_buffer_at[x][y] = device_max_z;
}
}

for each polygon
{
    for each pixel in polygon's projection
    {
        currz := polygon's z-value at pixel coordinates (x, y);
        if (currz <= z_buffer_at[x][y])
        {
            putpixel(x,y,color);
            z_buffer_at[x][y] = currz;
        }
    }
}

How to compute z?

I. On edge

Consider the following picture:

Parametric equation of line,

\[ x(t) = x_0 + t(x_1-x_0) \]
\[ y(t) = y_0 + t(y_1-y_0) \]
\[ z(t) = z_0 + t(z_1-z_0) \]

So,
\[ y_i = y(t_i) = y_0 + t_i(y_1-y_0) \]

therefore,
\[ t_i = \frac{(y_i-y_0)}{(y_1-y_0)} \]

Similarly,
\[ y_j = y(t_j) = y_0 + t_j(y_1-y_0) \]

therefore,
\[ t_j = \frac{(y_j-y_0)}{(y_1-y_0)} \]
\[ = \frac{(y_i+1-y_0)}{(y_1-y_0)} \]
\[ = t_i + \frac{1}{(y_1-y_0)} \]

For x, see polygon scan conversion
Similarly,
\[ z_j = z(t_j) = z_0 + t_j(z_1-z_0) \]
\[ = z_0 + (t_i + \frac{1}{(y_1-y_0)})(z_1-z_0) \]
\[ = z_i + (z_1-z_0)/(y_1-y_0) \]

II. On segment
Consider the following picture:

Similarly in case I,
\[ z_j = z(t_j) = z_0 + t_j(z_1-z_0) \]
\[ = z_0 + (t_i + 1/(x_1-x_0))(z_1-z_0) \]
\[ = z_i + (z_1-z_0)/(x_1-x_0) \]

The following algorithm is combined polygon rendering (in chapter 5) and hidden surface removing (in this chapter).

**Algorithm:** Assume data are vertices of polygon, \( v[1], ..., v[n] \)

\[
\text{for y = 0 to device_max_y }
\]

}
edge_list_at[y] := nil;
}
vfirst = v[1];

for i = 2 to n
{
    if (v[i-1].y == v[i].y) continue; /* discard horizontal edge */
    AddEdgeToList(v[i-1], v[i]);
}
AddEdgeToList(v[n], v[vfirst]);
for y=0 to device_max_y
{
    if (edge_list_at[y] != nil) RenderSpan(y);
}

where

AddEdgeToList(v1,v2)
{
    if (v1.y>v2.y)
    {
        v1<->v2; /* exchange so that v1.y is less than */
    }
    dy = v2.y-v1.y;
    x = v1.x;
    dx = (v2.x-v1.x)/dy;
    z = v1.z; /* adding here */
    dz = (v2.z-v1.z)/dy; /* adding here */
    for y=v1.y to v2.y-1
    {
        e = new edge_node;
        e.x = x;
        e.z = z; /* adding here */
        put e in edge_list_at[y] in ascending order of x;
        x = x+dx;
    }
}
\[
z = z + dz; /* adding here */
\]

and

\[
\text{RenderSpan}(y) \\
\{ \\
\quad \text{edge}_\text{node}1 = \text{edge\_list\_at}[y]; \\
\quad \text{while (edge}_\text{node}1 \neq \text{nil}) \\
\quad \{ \\
\quad \quad \text{edge}_\text{node}2 = \text{edge}_\text{node}1.next; \\
\quad \quad x1 = \text{edge}_\text{node}1.x; \\
\quad \quad x2 = \text{edge}_\text{node}2.x; \\
\quad \quad \text{if } (x1 \neq x2) \quad /* adding here */ \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad dx = x2-x1; \quad /* adding here */ \\
\quad \quad \quad \quad currz = \text{edge}_\text{node}1.z; \quad /* adding here */ \\
\quad \quad \quad \quad dz = (\text{edge}_\text{node}2.z - currz)/dx; \quad /* adding here */ \\
\quad \quad \quad \quad \text{for } x = x1 \text{ to } x2-1 \\
\quad \quad \quad \quad \{ \\
\quad \quad \quad \quad \quad \text{if } (\text{currz} \leq \text{z\_buffer\_at}[x][y]); \quad /* adding here */ \\
\quad \quad \quad \quad \quad \{ \\
\quad \quad \quad \quad \quad \quad \text{putpixel}(x,y,\text{color}); \\
\quad \quad \quad \quad \quad \quad \text{z\_buffer\_at}[x][y] = \text{currz}; \quad /* adding here */ \\
\quad \quad \quad \quad \quad \} \\
\quad \quad \quad \quad \quad \text{currz} = \text{currz} + dz; \quad /* adding here */ \\
\quad \quad \quad \quad \} \\
\quad \quad \} \\
\quad \text{edge}_\text{node}1 = \text{edge}_\text{node}2.next; \\
\quad \} \\
\}
\]
Scan-Line Algorithms

Compute illumination values of pixels in scan-line order: data is processed pixel by pixel rather than polygon by polygon.

Algorithm maintains data structures to describe the above idea; as we progress pixel-to-pixel and scan-line-to-scan-line, data structures are updated. Because one pixel is usually very much like the next. This mean that progressing from one pixel to next (or one scan-line to next) should be simple calculation. Therefore update of data structures from pixel-to-pixel and scan-line-to-scan-line is usually an incremental update.

Basic Organization:

Precompute all polygons
Yscan Loop /* for each scan line do */
begin
Calculate data structures needed for new scanplane
Xscan Loop /* for each pixel on scan line do */
begin
Update data structures for next step in X direction;
Determine which polygon is visible at pixel;
Draws into frame buffer as required
end
end
In Yscan Loop, maintain a list of "active curves" intersections of current scan plane ($Y = Y_{scan}$) and polygons. When Yscan drops to next scan line

- Add curve to list if Yscan has just dropped below maximum $y$ for a polygon
- Remove curve from list when Yscan drops below minimum $y$.

**Polygon:** $P_1, P_2, P_3, P_4$

**Edges:** $P_1P_2, P_2P_3, P_3P_4, P_4P_1$

**Intersection curve:** $Q_1Q_2$

**Active edges are:** $P_1P_2, P_2P_3$

**Algorithms:**

1. Precompute all polygons maxy and miny for edges, sort according to decreasing maxy.
2. **In Yscan Loop**
   - Create entry in active edge list when Yscan drops below maxy
   - Delete entry when Yscan drops below miny

Edges become active in pairs (for convex polygons) unless an edge is horizontal and coincides with scan line. Ignore such edges.
In active edge list, maintain "edge descriptor block".

<table>
<thead>
<tr>
<th>x</th>
<th>current x and z coordinates for intersection of scan ray and polygon (initialization is shown below)</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>How many scan lines this edge intersects (initially:</td>
</tr>
<tr>
<td>count</td>
<td></td>
</tr>
<tr>
<td>Dx</td>
<td>Will explain later</td>
</tr>
<tr>
<td>Dz</td>
<td></td>
</tr>
</tbody>
</table>

Initialize x and z in edge descriptor.

Each time through Yscan Loop, counts in all edge descriptor blocks are decremented. If a count is set to 0, block is removed.

When Yscan drops to next line. How do we update x and z in edge blocks?

Could conceivably redo above computation for new Yscan. Bad idea! uses too much time.

Overall: When Yscan drops by one:
- Decrement count in edge descriptor blocks
- Discard blocks with count ≤ 0
- Update x and z for all edges still active by $D_x$ and $D_z$ respectively, where
\[ D_x = \frac{x_2 - x_1}{y_2 - y_1}, \quad D_z = \frac{z_2 - z_1}{y_2 - y_1} \]

then edge descriptor block is:

<table>
<thead>
<tr>
<th>x</th>
<th>z</th>
<th>count</th>
<th>( D_x )</th>
<th>( D_z )</th>
</tr>
</thead>
</table>

update of active edges by \( x = x - D_x \) and \( z = z - D_z \)

- Check to see if any edges should become active. If so, create edge block, initialize the fields.
- Fall into Xscan Loop for this scanline.

**Xscan Loop**

Keep and maintain list of active segments, derived from active edge list.

Get, from Yscan Loop, active edges, arranged in pairs. Similar to Yscan Loop, segment becomes active when \( x \) passes \( \text{minx} \) on segment. Create segment descriptor block.

<table>
<thead>
<tr>
<th>z</th>
<th>count</th>
<th>( D_z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>current ( z )-value at current Yscan and ( x ) values</td>
<td>how many more loops segment will exist (initially: (</td>
<td>\text{maxx} - \text{minx} + 1</td>
</tr>
</tbody>
</table>

Initial \( z \)-value in segment block is copied from edge descriptor block. When moving across polygon (i.e. passing through Xscan Loop). Update \( z \) in segment block with \( D_x \), where \( D_x = \frac{z_2 - z_1}{x_2 - x_1} \)

and keep \( D_z \) in segment block.

Xscan proceeds as follows:
- decrement count in all segment blocks
- delete any block with count \( \leq 0 \)
- update z-value in all blocks
- check active list to see if new blocks should be created
- among all segment blocks, find one with largest z-value, illuminate pixel with color of corresponding polygon

For example, more than one active segment

Area Subdivision Algorithms

Given area on screen. If area contains simple image then draw the image else (area has complicated images) subdivide into several smaller areas. Recursively process small areas.

Simple area contains either 0 or 1 polygon. Complicate area contains more than 1 polygon.

Polygon can be related to a given area in one of 4 ways:
Algorithms:

Given an area:

1. All polygon disjoint? If so, return.
2. Is there a surrounding whose minimum z is greater than all z-coordinates of intersecting and contained? If so, threat that surrounding as only one polygon in the area and return
3. If there is only one polygon in the area, clip edge to area and draw
4. Subdivide into 4 smaller areas, readjust intersecting, surrounding, contained and disjoint list for smaller areas, process each smaller area recursively

Note! Infinite recursion is possible: Stopping condition for recursions. If area is smaller than screen resolution, don't process it recursively, but illuminate the spot on the screen. The depth of all relevant polygons is computed at the center of this pixel-sizes, the polygon with the closest z coordinate define the color.

Fact:

- Surrounding of parent will be surrounding of child. For example,

- Disjoint of parent will be disjoint of child. For example,
- Intersecting of parent may be intersecting, surrounding, or disjoint of child. For example,

- Contained of parent may be contained or intersecting or disjoint of child. For example,

To make this efficient, one idea is to make rearrangement of polygon lists efficient.

Clever techniques use "triangle table".
Examine polygons from new end of disjoint up to new end of surrounding. If polygon is found that is disjoint from child, exchange with the first polygon past end of disjoints, increment new end of disjoint to be passed with child.

Symmetrically, search from new end of surrounding down to new end of disjoint, looking for polygon that surrounds. If found, exchange with polygon immediately in front of surrounding list; decrement new end of surrounding. When child area is processed and next child area needs to be processed, get back previous state of polygon lists by resetting new end of disjoint to old end of disjoint and old end of surrounding to old end of surrounding.

Remark!
- Rearrangement of polygons in 4 respective lists for child (i.e. subdivided areas) proceeds quickly. The triangle table idea was first presented by Jim Blinn.
- To make this (area subdivision) efficient, another idea is to optimize the subdivision. Instead of regular subdivision try to subdivide along polygon edges. This will result in fewer subdivisions, but need to maintain arbitrary shaped areas. This is the "Weiler-Atherton" algorithm.

Illumination and Shading
In this section, we discuss how to shade surfaces based on the position, orientation, and characteristics of the surfaces and the light sources illuminating them. We develop a number of different illumination models that express the factors determining a surface's color at a given point. Illumination models are also frequently called lighting models or shading models. Here, however, we reserve the term shading model for the broader framework in which an illumination model fits. The shading model determines when the illumination model is applied and what arguments it will receive. For example, some shading models invoke an illumination model for every pixel in the image, whereas others invoke an illumination model for only some pixels, and shade the remaining pixels by interpolation. Graphics researchers have often approximated the underlying rules of optics and thermal radiation, either to simplify computation or because more accurate models were not known in the graphics community. Consequently, many of the illumination and shading models traditionally used in computer graphics include a multitude of kludges, "hacks," and simplifications that have no firm grounding in theory, but that work well in practice.

We first develop illumination models for monochromatic surfaces and lights, and then show how the computations can be generalized to handle the color systems.
ILLUMINATION MODELS

The simplest illumination model possible is that each object is displayed using an intensity intrinsic to it. We can think of this model, which has no external light source, as describing a rather unrealistic world of nonreflective, self-luminous objects. An illumination model can be expressed by an illumination equation in variables associated with the point on the object being shaded. The illumination equation that expresses this simple model is

\[ I = k_l \]

where \( I \) is the resulting intensity and the coefficient \( k_l \) is the object’s intrinsic intensity. Since this illumination equation contains no terms that depend on the position of the point being shaded, we can evaluate it once for each object.

1. Ambient Light

Now imagine, instead of self-luminosity, that there is a diffuse, nondirectional source of light. This is known as ambient light. If we assume that ambient light impinges equally on all surfaces from all directions, then our illumination equation becomes

\[ I = I_a k_a \]

\( I_a \) is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object’s surface is determined by \( k_a \), the ambient-reflection coefficient, which ranges from 0 to 1. The ambient-reflection coefficient is a material property. The ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials.

2. Diffuse Reflection

Now consider illuminating an object by a point light source, whose rays emanate uniformly in all directions from a single point. The object’s brightness varies from one part to another, depending on the direction of and distance to the light source.

Lambert’s Law:

\[ I = I_a k_a \]

\( I_a \) is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object’s surface is determined by \( k_a \), the ambient-reflection coefficient, which ranges from 0 to 1. The ambient-reflection coefficient is a material property. The ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials.

Lambert’s Law:

\[ I = I_a k_a \]

\( I_a \) is the intensity of the ambient light, assumed to be constant for all objects. The amount of ambient light reflected from an object’s surface is determined by \( k_a \), the ambient-reflection coefficient, which ranges from 0 to 1. The ambient-reflection coefficient is a material property. The ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials.
Illumination Model:

\[ I = I_a \cdot k_a + I_p \cdot k_d \cdot \cos \theta \]

Where \( I_a \) is the point light source’s intensity and \( k_d \) is a diffuse-reflection coefficient, constant between 0 and 1 and varies from one material to another.

In practice, \( \cos \theta \) is computed by \( \hat{N} \cdot \hat{L} \). This assume \( \hat{N} \) and \( \hat{L} \) are unit vectors

\[ \hat{N} \cdot \hat{L} = |\hat{N}| \cdot |\hat{L}| \cdot \cos \theta \]

\[ \cos \theta = \frac{\hat{N} \cdot \hat{L}}{|\hat{N}| \cdot |\hat{L}|} \]

If \( \hat{N} \) and \( \hat{L} \) are unit vectors, then \( |\hat{N}| \) and \( |\hat{L}| = 1 \).

Therefore \( \cos \theta = \hat{N} \cdot \hat{L} \). If \( \hat{N} \cdot \hat{L} < 0 \), then no illumination from light source.

3. Specular Reflection

Specular reflection can be observed on any shiny surface. Shiny surfaces reflect light unequally in different directions.

\[ \hat{R} = 2 \cdot \hat{N} \cdot (\hat{N} \cdot \hat{L}) - \hat{L} \]

The projection of \( \hat{L} \) onto \( \hat{N} \) is \( |L| \cdot \cos \theta \) (see appendix). Therefore

\[ T = |L| \cdot \cos \theta (\hat{N}) \]

or

\[ T = |L| \cdot (\hat{N} \cdot \hat{L}) (\hat{N}) = \hat{N} \cdot (\hat{N} \cdot \hat{L}) \]

if \( |\hat{N}| = 1 \) then

\[ T = \hat{N} \cdot (\hat{N} \cdot \hat{L}) \]

Since \( \hat{R} = T + S \) (see appendix)
and $S = T - L$ (see appendix L + S = T or S = T - L)

Therefore $R = T + T - L$

$R = 2T - L = 2 \frac{N(N \cdot L)}{|N|^2} - L$ if $|N| = 1$ then $R = 2 \frac{N(N \cdot L)}{|N|^2} - L$

The **Phong illumination model.** Phong Bui-Tuong developed a popular illumination model for nonperfect reflectors, such as the apple. It assumes that maximum specular reflectance occurs when $\alpha$ is zero and falls off sharply as $\alpha$ increases. This rapid falloff is approximated by $\cos^n \alpha$, where $n$ is the material's specular-reflection exponent. Values of $n$ typically vary from 1 to several hundred, depending on the surface material being simulated. A value of 1 provides a broad, gentle falloff, whereas higher values simulate a sharp, focused highlight. For a perfect reflector, $n$ would be infinite. We treat a negative value of $\cos \alpha$ as zero.

![Graphs of $\cos \alpha$, $\cos^2 \alpha$, $\cos^8 \alpha$, and $\cos^{64} \alpha$](image)

Phong's model is

$$I = k_a I_a + k_d I_p \cdot N \cdot L + k_s I_p \cdot \cos^n \alpha.$$  

where $n$ controls shininess (large $n$ give more shininess surfaces) and $k_s$ is the material's specular-reflection coefficient, which ranges between 0 and 1.

If the direction of $R$ and $V$ are normalized, then $\cos \alpha = \frac{R \cdot V}{|R||V|}$. In practice, we use

$$I = k_a I_a + k_d I_p \cdot N \cdot L + k_s I_p \cdot \left(\frac{R \cdot V}{|R||V|}\right)^n.$$  

This is the most popular shading equation.

However, there are other shading models, some better than Phong e.g. Brrance-Sparrow shading. For improving the Point-Light-Source Model and Multiple Light Sources, see textbook.

**Colored lights and surfaces.** So far, we have described monochromatic lights and surfaces. Colored light and surfaces are commonly treated by writing separate equations of each component of the color model. For example, in the RGB color system:

$$I_R = k_{aR} I_{aR} + k_{dR} I_{dR} \cdot N \cdot L + k_{sR} I_{sR} \cdot \left(\frac{R \cdot V}{|R||V|}\right)^n$$

$$I_G = k_{aG} I_{aG} + k_{dG} I_{dG} \cdot N \cdot L + k_{sG} I_{sG} \cdot \left(\frac{R \cdot V}{|R||V|}\right)^n$$

$$I_B = k_{aB} I_{aB} + k_{dB} I_{dB} \cdot N \cdot L + k_{dB} I_{dB} \cdot \left(\frac{R \cdot V}{|R||V|}\right)^n$$

A simplifying assumption is made here that a three-component color model can completely model the interaction of light with objects. This assumption is wrong, we will discuss in later chapter. But it is
easy to implement and often yields acceptable pictures. In theory, the illumination equation should be evaluated continuously over the spectral range being modeled; in practice, it is evaluated for some number of discrete spectral samples.

The halfway vector

An alternative formulation of Phong's illumination model uses the halfway vector $H$, so called because its direction is halfway between directions of the light source and the viewer, $H = (L + V) / 2$ or unit vector $H = (L + V) / (|L + V|)$

![Diagram of vectors](image)

The vector $R$ is expensive to calculate and although Phong gives an efficient method for calculating $R$, it is better to use a vector $H$. The specular term then becomes a function of $(N \cdot H)$ rather than $(R \cdot N)$. If the light source and the viewpoint are considered to be infinity then $L$ and $V$ are constant over the domain of the scene, then the use of $N \cdot H$ offers a computational advantage, since $H$ is constant.

SHADING MODELS FOR POLYGONS

It should be clear that we can shade any surface by calculating the surface normal at each visible point and applying the desired illumination model at that point. Unfortunately, this brute-force shading model is expensive. In this section, we describe more efficient shading models for surfaces defined by polygons and polygon meshes.

1. Constant Shading

The simplest shading model for a polygon is constant shading, also known as faceted shading or flat shading. This approach applies an illumination model once to determine a single intensity value that is then used to shade an entire polygon. In essence, we are sampling the value of the illumination equation once for each polygon, and holding the value across the polygon to reconstruct the polygon's shade.
2. Interpolated Shading

An alternative to evaluating the illumination equation at each point on the polygon, Wylie, Romney, Evans, and Erdahl pioneered the use of interpolated shading, in which shading information is linearly interpolated across a triangle from values determined for its vertices. Gouraud generalized this technique to arbitrary polygons.

3. Polygon Mesh Shading

Suppose that we wish to approximate a curved surface by a polygon mesh. If each polygon facet in the mesh is shaded individually, it is easily distinguished from neighbors whose orientation is different, producing a "faceted" appearance. This is true if the polygons are rendered using constant shading, interpolated shading, or even per-pixel illumination calculations, because two adjacent polygons of different orientation have different intensities along their borders. The simple solution of using a finer mesh turns out to be surprisingly ineffective, because the perceived difference in shading between adjacent facets is accentuated by the Mach band effect which exaggerates the intensity change at any edge where there is a discontinuity in magnitude or slope of intensity. At the border between two facets, the dark facet looks darker and the light facet looks lighter.

![Diagram of illumination and perception](image-url)
Mach band effect is difficult to handle, impossible to eliminate entirely.

Two basic shading models for polygon meshes take advantage of the information provided by adjacent polygons to simulate a smooth surface. In order of increasing complexity (and realistic effect), they are known as Gouraud shading and Phong shading, after the researchers who developed them.

### Gouraud Shading

Gouraud shading, also called intensity interpolation shading or color interpolation shading, eliminates intensity discontinuities. Gouraud shading extends the concept of interpolated shading applied to individual polygons by interpolating polygon vertex illumination values that take into account the surface being approximated.

The Gouraud shading process requires that the normal be known for each vertex of the polygon mesh. If the vertex normals are not stored with the mesh and cannot be determined directly from the actual surface, then, Gouraud suggested, we can approximate them by averaging the surface normal of all polygon facets sharing each vertex.

Unit vector \( \mathbf{N}_v = \frac{\sum_{1 \leq i \leq n} \mathbf{N}_i}{|\sum_{1 \leq i \leq n} \mathbf{N}_i|} \)

**Note!** vector \( \mathbf{N}_v = \frac{\sum_{1 \leq i \leq n} \mathbf{N}_i}{n} \), the unit vector is \( \left( \frac{\sum_{1 \leq i \leq n} \mathbf{N}_i}{n} \right) \frac{1}{|\sum_{1 \leq i \leq n} \mathbf{N}_i|} \), which is

\[ \sum_{1 \leq i \leq n} \frac{\mathbf{N}_i}{|\sum_{1 \leq i \leq n} \mathbf{N}_i|} \]
What happens, however, if polygons have been clipped? We have to recalculate vertex normals and vertex intensities. Consider the following figure which shows a cross-section through three polygons.

Before clipping the two vertex normals are $N_1$ and $N_2$. After clipping the new vertex normal $N'_2$ can be calculated by interpolating between $N_1$ and $N_2$. Thus the object is shaded up to the clip boundary as if the clipped portion still existed.

Since normal vector is changed when scaling ($S_x \neq S_y \neq S_z$), and rotation. Normal vectors are invariant under translation and scaling ($S_x \neq S_y \neq S_z$). For rotation and scaling ($S_x \neq S_y \neq S_z$) need to be applied to normal vector too. This implies that rotation and scaling ($S_x \neq S_y \neq S_z$) are the special cases when apply transformation. To avoid these special cases, represent normal vector in homogeneous coordinates as $N = [N_x, N_y, N_z, 0]$.

1. **Translation**: $N$ is unchanged, do nothing.
2. **Rotation**: $N$ is rotated appropriately by multiply $N$ with 4x4 homogeneous rotation matrix.
3. **Scaling**: if $S_x = S_y = S_z$ then length of $N$ is changed but direction of $N$ is unchanged else

   A transformed surface normal $N'$ is $N' = \left[ N_x/S_x, N_y/S_y, N_z/S_z, 0 \right]$


Let $T = [tx, ty, tz]$ be a tangent vector perpendicular with normal vector $N = [N_x, N_y, N_z]$.

\[
T \cdot N = 0
\]

\[
T \cdot N = T^T N^T = 0
\]

Assume that the homogeneous transformation matrix is
Because $N$ is unchanged for translation then we can ignore $a_{41}$, $a_{42}$, $a_{43}$. Furthermore, we can reduce to 3x3 matrix because the 4th component always be zero ([N_x, N_y, N_z, 0]).

Let

\[
M = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]

แนวทางพิสูจน์คือ เมื่อมีการ transform ด้วย $M$ เราจะหา $T' = TM$ จากนั้นหา $N'$ ที่ตั้งฉากกับ $T'$ ให้ได้ ดังนี้

\[
T \cdot N = TM \cdot M^{-1} \cdot N \Rightarrow
\]

\[
= (TM) \cdot (N M^{-1} T)^T \Rightarrow
\]

\[
= (TM) \cdot (NM^{-1} T) \Rightarrow
\]

\[
= T' \cdot N' = 0
\]

นั่นคือ ต้อง transform $N$ ด้วย $M^{-1}$.

ในการกรณีของ scaling

\[
M = \begin{bmatrix}
S_x & 0 & 0 \\
0 & S_y & 0 \\
0 & 0 & S_z
\end{bmatrix}
\]

\[
M^{-1} = \begin{bmatrix}
1/S_x & 0 & 0 \\
0 & 1/S_y & 0 \\
0 & 0 & 1/S_z
\end{bmatrix}
\]

หรือเขียนเป็น homogeneous scaling matrix คูณกับ row vector $[N_x, N_y, N_z, 0]$ ดังนี้
The next step is to find vertex intensities by using the vertex normal with any desired illumination model. Finally, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line.

A difference equation may be used:

\[
\begin{align*}
I_a &= I_1 - (I_1 - I_2) \frac{(y_1 - y_s)}{(y_1 - y_2)} \\
I_b &= I_1 - (I_1 - I_3) \frac{(y_1 - y_s)}{(y_1 - y_3)} \\
I_g &= I_b - (I_b - I_a) \frac{(x_b - x_s)}{(x_b - x_a)}
\end{align*}
\]

OR

\[
I_a = (1-\alpha)I_1 + \alpha I_2 \quad \text{where} \quad \alpha = \frac{ys-y_1}{y_2-y_1}
\]

Note! \(0 \leq \alpha \leq 1\)

Note! \(I_{a+1} = I_a + \frac{I_b - I_1}{y_2 - y_1}\)
Phong Shading

Phong shading, also known as normal-vector interpolation shading, interpolates the surface normal vector $N$, rather than the intensity. Interpolation occurs across a polygon span on a scan line, between starting and ending normals for the span. These normals are themselves interpolated along polygon edges from vertex normals that are computed, if necessary, just as in Gouraud shading.

Even with an illumination model that does not take into account specular reflectance, the results of normal-vector interpolation are in general superior to intensity interpolation, because an approximation to the normal is used at each point. This reduces Mach-band problems in most cases, but greatly increases the cost of shading in a straightforward implementation, since the interpolated normal must be normalized every time it is used in an illumination model.

Shadow Generations

Hidden surface removal algorithms determine which surfaces can be seen from the viewpoint; shadow algorithms determine which surfaces can be “seen” from the light source. Thus, visible-surface algorithms and shadow algorithms are essentially the same. The surfaces that are visible from the light source are not in shadow; those that are not visible from the light source are in shadow. When there are multiple light sources, a surface must be classified relative to each of them.

Here, we consider shadow algorithms for point light sources. Visibility from a point light source is, like visibility from the viewpoint, all or nothing. Note that areas in the shadow of all light sources are still illuminated by the ambient light.
Scan-Line Shadow Generator

Preprocessing step: for each polygon P, examine all other polygon one-by-one.

Project each Q_i into plane that contains P. For each vertex of Q_i, construct ray with origin at light source. Find intersection of that line with plane of P. Those intersection points form a new polygon, “the shadow polygon”.

Don’t want to project Q_i “backwards” onto plane at P. To represent each line parametrically, when solving for intersection, check to see if t > 0.

Now, use scan-line algorithm:- when , at a pixel, a polygon is found to be visible, check to see if that point is inside a shadow polygon.

At I, check to see if I is contained in any shadow polygon. If so, use only ambient illumination else it is not in shadow. Don’t need to keep track of z-values for shadow polygon.

**Advantage:** Can reuse scan-line processing code.


Two Pass Z-buffer Shadow Generator

Maintain 2 z-buffers: one is the normal (viewer) z-buffer, other is a z-buffer for lights viewport. Light viewport or shadow z-buffer is filled in during a preprocessing step.
1. Transform into light coordinate system
2. Run z-buffer algorithm with light source as viewport. No need to do anything to frame buffer
3. Run z-buffer algorithm from original viewport same as ordinary z-buffer, but when algorithm decides to illuminate a pixel at (x,y,z): transform that pixel’s coordinates into light source coordinates (x’,y’,z’). If z value in z-buffer of light source (from 2.) is closer to the light than is z’, then there is something blocking the light and must be in shadow.

Example For parallel projection,

Ray Tracing

Invented in 1964 for pen-plotter. Reinvented by Turner Whitted (called recursive ray-tracing algorithm) in 1980. Became very popular rendering technique. Several special effects almost “for free” such as Perspective Projection (objects that are far away appear to be smaller). Hidden Surface Removal and Shadows are discussed in chapter 6 and 8 respectively. In this chapter we will talk about ray tracing more such as Reflections and Retractions.

Advantage: very easy to implement
Disadvantage: - consumes vast amount of CPU time!
- pictures tend to look “too clean” (these can be overcome)

Initial idea, model physically interaction of light and surfaces:
Generate photons coming from light source. Trace protons through scene. Accumulate photons at each pixel as simulation runs. Nearly impossible, too many photons must be simulated to get meaning results.

Instead, generate path of photon that would have passed through pixel. Trace that ray into scene. Accumulate surface intersections with ray.

A. Perspective Projection

Rays centered at eye position (center of projection) project far objects onto smaller screen areas.

B. Hidden Surface Removal

Idea, for every pixel, create a ray with origin at viewer’s position and direction of center of pixel. For every ray, find intersections of that ray with all polygons. Remember which polygon was closes for pixel to color of this polygon.
Advantage:
- Effective
- Easy to program
- Uses little memory
- Can be extended (see later)

Disadvantage:
This takes great amounts of CPU times! For example, 200 x 300 = 64,000 pixels = 64,000 rays. Suppose picture has 20 polygons => 1,280,000 ray-polygon intersection calculations! Fortunately, there are methods to accelerate this.

Ray representation:

\[ \text{Pixel position } P = (P_x, P_y, P_z) \]

Starting point (viewer) \( S = (S_x, S_y, S_z) \)

Direction of ray, \( D = (D_x, D_y, D_z) = (P_x - S_x, P_y - S_y, P_z - S_z) \)

Ray is represented parametrically
\[ \begin{align*}
  x(t) &= S_x + tD_x \\
  y(t) &= S_y + tD_y \\
  z(t) &= S_z + tD_z \\
\end{align*} \]

where \( t \geq 0 \)

To find intersection of ray and polygon: plane equation of polygon,
\[ A*x + B*y + C*z + D = 0 \]
substitute ray parametric equation into plane equation

\[ A^x(t) + B^y(t) + C^z(t) + D = 0 \]

\[ A^*(S_x + t*D_x) + B^*(S_y + t*D_y) + C^*(S_z + t*D_z) + D = 0 \]

\[ t^*(A^*D_x + B^*D_y + C^*D_z) + (A^*S_x + B^*S_y + C^*S_z + D) = 0 \]

\[ t = - \frac{A^*S_x + B^*S_y + C^*S_z + D}{A^*D_x + B^*D_y + C^*D_z} \]

Check, denominator zero or close to it? If the above is zero, then either ray and plane are parallel or ray lines in the plane; ignore intersection. Once ray-plane intersection is found, need to check to see if intersection point is inside the polygon.

Algorithm:

for every pixel do

begin

create ray; \( t_0 = \infty \)

for every polygon do

begin

find \( t_i \) of ray-polygon intersection

if there is an intersection then

if \( t_i < t_0 \) then

begin

close polygon = this polygon

\( t_0 = t_i \)

end

end

if hit \( (t_0 = \infty) \) put pixel with color of closest polygon

end
C. Shadow

Always consider only polygon ray intersection that is closest to ray origin.

Generate ray in direction of light source (L). Test L for intersection with all polygon. If any intersection, then in shadow. This requires almost no code development.

Algorithm:

```
for each pixel do
begin
    generate ray from eye through pixel
    for each polygon do
    begin
        find ray-polygon intersection
        remembering closest
    end
    to get shadows; when closest intersection with ray is found
    create light seeking ray, origin P (point of intersection) direction P->L
    for each polygon do
    begin
        test ray for intersection with polygon
        if intersects, then we’re in shadow, break;
        else illuminated;
    end
end
```

**Advantage:** Easy to code; almost no new code development.

**Disadvantage:** Takes along time (CPU-intensive).
D. Reflection

Calculate direction of perfect reflection; generate ray whose origin is at the point of intersection and direction is \( R \). Recursively process that ray to find closest intersection.

Reflection rays may generate more reflection rays. Be careful! Since reflection ray is processed recursively, deeper recursion is possible. To protect infinite recursion, recursion should stop if:

1. Reflection ray does not intersect any object i.e. ray hits background.
2. More than 6 or 7 levels of recursion have been generated because effect of deeper recursion have little effect.

E. Refraction

Light changes direction when it crosses the boundary between two media. This effect is called refraction. The change in direction is related to a property of materials called the index of refraction (\( \eta \)).

\[
\begin{align*}
\eta_1 & \quad \theta_1 \\
\eta_2 & \quad \theta_2
\end{align*}
\]

The exact relation is called Snell's Law:

\[
\frac{\sin \theta_1}{\sin \theta_2} = \frac{\eta_2}{\eta_1}
\]

Calculating the refraction vector. The unit vector in the direction of refraction, \( T \), can be calculated as
Where \( M \) is a unit vector perpendicular to \( N \) in the plane of the incident ray \( I \) and \( N \). We see that

\[
M = \frac{(N\cos\theta_1 - I)}{\sin\theta_1}.
\]

By substitution,

\[
T = \sin\theta_2 M - \cos\theta_2 N.
\]

If we let \( \eta_3 = \eta_1 = \sin\theta_2 \), then after rearranging terms

\[
T = \left( \eta_3 \cos\theta_1 - \cos\theta_2 \right) N - \eta_3 I.
\]

Note that \( \cos\theta_1 \) is \( N.I \), and \( \cos\theta_2 \) can be computed as

\[
\cos\theta_2 = \sqrt{1 - \sin^2\theta_2} = \sqrt{1 - \eta_3^2\sin^2\theta_1} = \sqrt{1 - \eta_3^2(1 - (N.I)^2)}
\]

Thus,

\[
T = \left[ \eta_3(N.I) - \sqrt{1 - \eta_3^2(1 - (N.I)^2)} \right]N - \eta_3 I.
\]

**Total internal reflection.** When light passes from one medium into another whose index of refraction is lower, the angle \( \theta_2 \) of the transmitted ray is greater than the angle \( \theta_1 \). If \( \theta_1 \) becomes sufficiently large, then \( \theta_2 \) exceeds 90° and the ray is reflected from the interface between the media, rather than being transmitted. This phenomenon is known as *total internal reflection*, and the smallest \( \theta_1 \) at which it occurs is called the *critical angle*. Total internal reflection occurs when the square root of \( 1 - \eta_3^2(1 - (N.I)^2) \) is imaginary.

The light ray is bent both when it enters a transparent object and when it leaves.
Combined Effects

Whitted’s recursive ray-tracing algorithm conditionally spawns reflection rays and refraction rays from the point of intersection. The shadow, reflection, and refraction rays are often called secondary rays, to distinguish them from the primary rays from the eye. If the object is specularly reflective, then a reflection ray is reflected about the surface normal in the direction of R, which may be computed as in chapter 7. If the object is transparent, then a refraction ray is sent into the object along T at an angle determined by Snell’s Law. Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays. The rays thus form a ray tree. In Whitted’s algorithm, a branch is terminated if the reflected and refracted rays fail to intersect an object, if some user-specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom-up, and each node’s intensity is computed as a function of its children’s intensities.

N = surface normal, R = reflected ray, L = shadow ray, T = transmitted ray

The halfway vector. An alternative formulation of Phong’s illumination model uses the halfway vector H, so called because its direction is halfway between the directions of the light source and the viewer.
H is also known as the direction of maximum highlights. If the surface were oriented so that its normal were in the same direction as H, the viewer would see the brightest specular highlight, since R and V would also point in the same direction. The new specular-reflection term can be expressed as \((N \cdot H)^n\), where \(H = (L+V) / |L + V|\). Note that \(\beta\), the angle between N and H, is not equal to \(\alpha\), the angle between R and V, so the same specular exponent \(n\) produces different results in the two formulations. Although using a \(\cos^n\) term allows the generation of recognizably glossy surfaces, you should remember that it is based on empirical observation, not on a theoretical model of the specular-reflection process.

We can represent Whitted’s illumination equation as

\[
I = k_a I_a + \sum k_d I_p (N \cdot L)^n + k_s I_p (N \cdot H)^n + k_t I_t
\]

Where \(I_r\) is the intensity of the reflected ray, \(k_t\) is the transmission coefficient ranging between 0 and 1, and \(I_t\) is the intensity of the refracted transmitted ray. Values for \(I_r\) and \(I_t\) are determined by recursively equation above at the closest surface that the reflected and transmitted rays intersect. To approximate attenuation with distance, Whitted multiplied the \(I\) calculated for each ray by the inverse of the distance traveled by the ray.

Pseudocode for simple recursive ray tracing without antialiasing:

select center of projection and window on view plane;
for each scan line in image do
  for each pixel in scan line do
    begin
      determine ray from center of projection through pixel;
      pixel := RT_trace(ray, 1);
    end;

  { Intersect ray with objects and compute shade at closest intersection. }
  { Depth is current depth in ray tree. }
procedure RT_trace (ray : RT_ray; depth : integer) : RT_color;
begin
  determine closest intersection of ray with an object;
  if object hit then
    begin
      compute normal at intersection;
      RT_trace := RT_shade (closest object hit, ray, intersection, normal, depth);
    end
  else
RT_trace := BACKGROUND_VALUE;

end;

{ Compute shade at point on object, tracing rays for shadows, reflection, refraction. }

procedure RT_shade

object : RT_object;     { Object intersected }
ray : RT_ray;           { Incident ray }
point : RT_point;       { Point of intersection to shade }
normal : RT_normal;     { Normal at point }
depth : integer        { Depth in ray tree }

): RT_color;

var

color : RT_color;       { Color of ray }
rRay, tRay, sRay : RT_ray; { Reflected, refracted, and shadow rays }
rColor, rColor : RT_color;  { Reflected and refracted ray colors }

begin

color := ambient term;

for each light do

begin

sRay := ray to light from point;
if dot product of normal and direction to light is positive then
compute how much light is blocked by opaque and transparent surfaces, and use to
scale diffuse and specular terms before adding them to the color;
end;

if depth < maxDepth then    { Return if depth is too deep. }

begin

if object is reflective then

begin

rRay := ray in reflection direction from point;
rColor := RT_trace (rRay, depth + 1);
scale rColor by specular coefficient and add to color;
end;

if object is transparent then

begin

tRay := ray in refraction direction from point;
if total internal reflection does not occur then
begin

void RT_trace(Ray &r, int depth)
{
    if (depth == 0) {
        RT_shade := color; { Return color of ray. }
    } else {
        // ... recursive call...
    }
}

// Scale and add the color from the trace.
scale tColor by transmission coefficient and add to color;

RT_shade := color; { Return color of ray. }
end;

Ray-Object Intersection Tests

Ray แทนด้วย จุด coordinate เริ่มต้น (x0,y0,z0) และ 媒介ของทิศทาง (v,u,w) นำมาเขียนเป็น
สมการ parametric ได้ดังนี้

\[ x(t) = x_0 + t \cdot v \]
\[ y(t) = y_0 + t \cdot u \]
\[ z(t) = z_0 + t \cdot w \]

วิธีหาจุดตัดกับวัตถุ กำหนดให้โดยการแทนค่า \( x(t), y(t), z(t) \) ในสมการของวัตถุ แล้วหาค่า \( t \) ถ้าหาค่า \( t \) ได้ และ \( t \geq 0 \) แสดงว่า ray ตัดกับวัตถุ แต่ถ้าหาค่า \( t \) ไม่ได้ (เป็น complex number) หรือ \( t < 0 \) แสดงว่าไม่ตัดกับวัตถุ \( t \)
แสดงระยะทางจากจุดเริ่มต้นของ ray ไปยังจุดตัด จุดตัดสามารถหาได้โดยการแทนค่า \( t \) ในสมการ parametric

ตัวอย่าง

- **sphere intersections**

สมการของ sphere คือ

\[ x^2 + y^2 + z^2 - r^2 = 0 \]

\((x0,y0,z0)\)

\((x1,y1,z1)\)

ในที่นี้ จุดเริ่มต้นของ ray คือ (x0,y0,z0) 媒介 = (v,u,w)

วิธีหาจุดตัด

\[ x(t) = x_0 + t \cdot v \]
\[ y(t) = y_0 + t \cdot u \]
\[ z(t) = z_0 + t \cdot w \]

แทน \( x,y,z \) ในสมการของวัตถุ

\[ (x_0 + t \cdot v)^2 + (y_0 + t \cdot u)^2 + (z_0 + t \cdot w)^2 - r^2 = 0 \]

หาค่า \( t \)

\[ (u^2+v^2+w^2)t^2 + 2(x_0v+y_0u+z_0w)t + (x_0^2+y_0^2+z_0^2-r^2) = 0 \]

สมการอยู่ในรูปแบบ \( at^2 + bt + c = 0 \) หาค่า \( t \) โดยสูตร
\[ t = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]

- **quadratic surface intersections**

สมการโดยทั่วไป
\[ Ax^2 + By^2 + Cz^2 + Dxy + Eyz + Fzx + Gx + Hy + Kz + L = 0 \]
เมื่อ \( A, B, C, D, E, \) และ \( F \) ไม่เป็น 0 พร้อมกัน

เช่น
- **Ellipsoid (รูปวงรี)**
  \[ \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \] เมื่อ \( a, b, c > 0 \)
- **Cone**
  \[ h^2x^2 - 2r^2y^2 + h^2z^2 = 0 \]
- **Hyperbolic paraloid (รูปอานมือ)**
  \[ \frac{x^2}{a^2} - \frac{y^2}{b^2} = cz \] เมื่อ \( a, b > 0 \) และ \( c \neq 0 \)

- **polygon intersections**

To find intersection of ray and polygon: plane equation of polygon,
\[ A^x x + B^y y + C^z z + D = 0 \]
substitute ray parametric equation into plane equation
\[
A^x(x(t) + t^v) + B^y(y(t) + t^u) + C^z(z(t) + t^w) + D = 0
\]
\[
t*(A^x v + B^y u + C^z w) + (A^x x_0 + B^y y_0 + C^z z_0 + D) = 0
\]
\[
t = \left( -\frac{A^x x_0 - B^y y_0 - C^z z_0 - D}{(A^v + B^u + C^w)} \right)
\]

Check, denominator zero or close to it? If the above is zero, then either ray and plane are parallel or ray lines in the plane; ignore intersection.
Once ray-plane intersection is found, need to check to see if intersection point is inside the polygon. We have to project the intersection point and polygon onto one of the primary planes, either xy, xz, or yz. If the polygon is perpendicular to one of these planes, its projection onto that plane will be a single line. To avoid this problem, and to make sure that the projection is as large as possible, we find the dominant axis of the normal vector and use the perpendicular to that axis. In general, if the surface normal is \((N_x, N_y, N_z)\), use the coordinates of the two smallest values. For example, if the surface normal is \((0,0,1)\), then use the xy plane.

To detect if intersection point is inside the polygon, (this works only for convex polygon, concave polygons need to be subdivided into convex polygons)
Using a ray tracing method with polygonal databases, we must define a fast algorithm to compute ray–polygon intersection. The following algorithm is quite similar to but faster than the barycentric approach described in Snyder and Barr (1987).

The goal of the algorithm is first to determine if a ray goes through the polygon, and then to determine the coordinates of the intersection point and parameters, to localize this point with respect to the polygon’s vertices. These parameters are used to compute the interpolated normal at this point, and can be used also to compute the entry of a texture map.

First Step: Intersecting the Embedding Plane

This step is common with the other intersection algorithms but can be presented again. A polygon is described by its vertices \( V_i \) \( (i \in \{0, \ldots, n - 1\}, n \geq 3) \). Let \( x_i, y_i, \) and \( z_i \) the coordinates of the vertex \( V_i \). The normal of the plane containing the polygon, \( \mathbf{N} \), is computed with the cross product:

\[
\mathbf{N} = \overrightarrow{V_0 V_1} \times \overrightarrow{V_0 V_2}.
\]

For each point \( P \) of the plane, the quantity \( P \cdot \mathbf{N} \) is constant. This constant value is computed by the dot product \( d = -\overrightarrow{V_0} \cdot \mathbf{N} \). The implicit
RAY TRACING

representation of the plane,
\[
\mathbf{N} \cdot \mathbf{P} + d = 0, \quad (1)
\]
is computed once, and then stored in the polygon description.
Let the parametric representation of the ray be
\[
\mathbf{r}(t) = \mathbf{O} + \mathbf{D}t. \quad (2)
\]
The evaluation of the parameter \( t \) corresponding to the intersection point
can be obtained using the equations (1) and (2):
\[
t = -\frac{d + \mathbf{N} \cdot \mathbf{O}}{\mathbf{N} \cdot \mathbf{D}}. \quad (3)
\]
This calculation requires 12 floating operations and three tests:
- If polygon and ray are parallel (\( \mathbf{N} \cdot \mathbf{D} = 0 \)), the intersection is rejected.
- If the intersection is behind the origin of the ray (\( t \leq 0 \)), the intersection is rejected.
- If a closer intersection has been already found for the ray (\( t > t_{ray} \)),
  the intersection is rejected.

Second Step: Intersecting the Polygon

A parametric resolution is now presented. This solution is based on
triangles. If a polygon has \( n \) vertices (\( n > 3 \)), it will be viewed as a set of
\( n - 2 \) triangles. For this reason, the resolution is restricted to convex
polygons. The point \( P \) (see Fig. 1) is given by
\[
\mathbf{V}_0 \mathbf{P} = \alpha \mathbf{V}_0 \mathbf{V}_1 + \beta \mathbf{V}_0 \mathbf{V}_2. \quad (4)
\]
The point \( P \) will be inside the triangle (\( \triangle \mathbf{V}_0 \mathbf{V}_1 \mathbf{V}_2 \)) if
\[
\alpha \geq 0, \beta \geq 0, \text{ and } \alpha + \beta \leq 1.
\]
Equation (4) has three components:
\[
\begin{align*}
\mathbf{x}_P - x_0 &= \alpha(x_1 - x_0) + \beta(x_2 - x_0) \\
\mathbf{y}_P - y_0 &= \alpha(y_1 - y_0) + \beta(y_2 - y_0) \\
\mathbf{z}_P - z_0 &= \alpha(z_1 - z_0) + \beta(z_2 - z_0).
\end{align*}
\quad (5)
\]
A solution exists and is unique. To reduce this system, we wish to project the polygon onto one of the primary planes, either $xy$, $xz$, or $yz$. If the polygon is perpendicular to one of these planes, its projection onto that plane will be a single line. To avoid this problem, and to make sure that the projection is as large as possible, we find the dominant axis of the normal vector and use the plane perpendicular to that axis. As in Snyder and Barr (1987), we compute the value $i_0$,

$$i_0 = \begin{cases} 
0 & \text{if } |N_x| = \max(|N_x|, |N_y|, |N_z|) \\
1 & \text{if } |N_y| = \max(|N_x|, |N_y|, |N_z|) \\
2 & \text{if } |N_z| = \max(|N_x|, |N_y|, |N_z|).
\end{cases}$$

Consider $i_1$ and $i_2$ ($i_1$ and $i_2 \in \{0, 1, 2\}$), the indices different from $i_0$. They represent the primary plane used to project the polygon. Let $(u, v)$ be the two-dimensional coordinates of a vector in this plane; the coordinates of $V_0P$, $V_0V_1$, and $V_0V_2$, projected onto that plane, are

$$u_0 = P_{i_1} - V_{0_{i_1}} \quad u_1 = V_{1_{i_1}} - V_{0_{i_1}} \quad u_2 = V_{2_{i_1}} - V_{0_{i_1}}$$

$$v_0 = P_{i_2} - V_{0_{i_2}} \quad v_1 = V_{1_{i_2}} - V_{0_{i_2}} \quad v_2 = V_{2_{i_2}} - V_{0_{i_2}}$$

Equations 5 then reduce to

$$\begin{cases} 
u_0 = \alpha \cdot u_1 + \beta \cdot u_2 \\
u_0 = \alpha \cdot v_1 + \beta \cdot v_2 \end{cases}$$
The solutions are
\[
\alpha = \frac{\det \begin{pmatrix} u_0 & u_2 \\ v_0 & v_2 \end{pmatrix}}{\det \begin{pmatrix} u_1 & u_2 \\ v_1 & v_2 \end{pmatrix}} \quad \text{and} \quad \beta = \frac{\det \begin{pmatrix} u_1 & u_0 \\ v_1 & v_0 \end{pmatrix}}{\det \begin{pmatrix} u_1 & u_2 \\ v_1 & v_2 \end{pmatrix}}.
\]

The interpolated normal from the point \( P \) may be computed by
\[
N_P = (1 - (\alpha + \beta))N_0 + \alpha N_1 + \beta N_2.
\]

Pseudo-Code for a Ray–Triangle Intersection

\[
\begin{align*}
O: \text{point;} & \quad \text{Origin of the ray} \\
D: \text{vector;} & \quad \text{Direction of the ray} \\
P: \text{point;} & \quad \text{Intersection point} \\
V: \text{array[0..2] of point;} & \quad \text{Polygon vertices}
\end{align*}
\]

\[
P \leftarrow O + Dt;
\]

\( i_1 \) and \( i_2 \) are in the polygon description.

\[
\begin{align*}
u_0 & \leftarrow P[i_1] - V[0][i_1]; \\
v_1 & \leftarrow P[i_2] - V[0][i_2]; \\
u_2 & \leftarrow V[1][i_1] - V[0][i_1]; \\
v_1 & \leftarrow V[1][i_2] - V[0][i_2]; \\
v_2 & \leftarrow V[2][i_2] - V[0][i_2];
\end{align*}
\]

\[
\begin{aligned}
\text{if } u_1 &= 0 \\
\text{then } \beta &\leftarrow u_0/u_2; \\
\text{if } 0 \leq \beta &\leq 1 \\
\text{then } \alpha &\leftarrow (v_0 - \beta * v_2)/v_1;
\end{aligned}
\]

\[
\begin{aligned}
\text{else } \beta &\leftarrow (v_0 * u_1 - u_0 * v_1)/(v_2 * u_1 - u_2 * v_1); \\
\text{if } 0 \leq \beta &\leq 1 \\
\text{then } \alpha &\leftarrow (u_0 - \beta * u_2)/u_1;
\end{aligned}
\]

The values \( \alpha \) and \( \beta \) are the interpolation parameters.

\[
\text{return } (\alpha \geq 0 \text{ and } \beta \geq 0 \text{ and } (\alpha + \beta) \leq 1)
\]

See also Efficient Generation of Sampling Jitter Using Look-up Tables (64); Fast Line–Edge Intersections on a Uniform Grid (29); Transforming Axis-Aligned Bounding Boxes (548)

See Appendix 2 for C Implementation (735)
AN EFFICIENT RAY–POLYGON INTERSECTION

Didier Badouel

/* the value of t is computed. */
* i1 and i2 come from the polygon description.
* V is the vertex table for the polygon and N the
* associated normal vectors.
*/
P[0] = ray.O[0] + ray.D[0]*t;
u0 = P[i1] - V[0][i1]; v0 = P[i2] - V[0][i2];
inter = FALSE; i = 2;
  do {
    /* The polygon is viewed as (n-2) triangles. */
    u1 = V[i-1][i1] - V[0][i1]; v1 = V[i-1][i2] - V[0][i2];
    u2 = V[i ] [i1] - V[0][i1]; v2 = V[i ] [i2] - V[0][i2];

    if (u1 == 0) {
      beta = u0/u2;
      if ((beta >= 0.)&& (beta <= 1.)) {
        alpha = (v0 - beta*v2)/v1;
        inter = ((alpha >= 0.)&& (alpha+beta) <= 1.));
      }
    } else {
      beta = (v0*u1 - u0*v1)/(v2*u1 - u2*v1);
      if ((beta >= 0.)&& (beta <= 1.)) {
        alpha = (u0 - beta*u2)/u1;
        inter = ((alpha >= 0.)&& (alpha+beta) <= 1.));
      }
    }
  } while (!inter)&&(i < poly.n);

if (inter) {
  /* Storing the intersection point. */
  /* the normal vector can be interpolated now or later. */
  if (poly.interpolate) {
    gamma = 1 - (alpha+beta);
    ray.normal[0] = gamma * N[0][0] + alpha * N[i-1][0] +
                     beta * N[i][0];
    ray.normal[1] = gamma * N[0][1] + alpha * N[i-1][1] +
                     beta * N[i][1];
    ray.normal[2] = gamma * N[0][2] + alpha * N[i-1][2] +
                     beta * N[i][2];
  }
  return (inter);
}
Or assume we are in xy plane

\[ I = (lx, ly) \]

\[ (x1, y1) \quad (x2, y2) \]

\[ f(x, y) = (x - x1)(y2 - y1) - (x2 - x1)(y - y1) \]

\[ f(lx, ly) = 0 \quad \text{if } I \text{ is on a line} \]

\[ f(lx, ly) < 0 \quad \text{iff } I \text{ is on one side} \]

\[ f(lx, ly) > 0 \quad \text{iff } I \text{ is on other side} \]

By successively considerly polygon edges and one disjoint vertex. Test to see if I is on same side of line as the vertex. If true (for all polygon edges), then I is inside polygon.

Example,

Examine cross products

\[ AB \times BC \text{ points into page} \]

\[ AB \times BI \text{ points into page} \]

\[ \{ \text{can conclude nothing} \} \]
BC \times CA \text{ points into page}
\quad \text{I is outside}
\quad \text{BC \times CI \text{ points out of page}}

To test to see if point is inside polygon repeatedly form pairs of cross products (as above). If directions are always same, then point (I) is inside polygon. To tell if directions of these vectors are same: examine sign of corresponding components.

**Surface Normal Calculations**

- For **planar** objects, the normal vector can be found by forming the cross product of two edges of the polygon.
- For **sphere** objects, the normal vector at the point of intersection is the vector from the center of the sphere to the point on the surface.
- For **quadratic surfaces**, the normal vector at a point can be found by taking the partial derivatives of the surface equation with respect to x, y, and z, and then evaluating these derivatives at the point of intersection.

**Ray Tracing Efficiency**

For a given ray, you must determine the nearest object the ray intersects (other than the one from which the ray started). This means you must search the entire list of objects to find the nearest one, (in other words, the one whose intersection yields the minimum value of the parameter t along the ray).

If you take the straightforward approach of simply casting rays and testing every object in the object list for each ray, you quickly saturate the computing requirements of even the simplest scenes. For example, a list of 10 requires on average five intersection calculations per ray. If the ray is reflected, then even more rays are cast. You could easily perform 100 intersection tests per screen pixel. With a screen resolution of 320x240, this corresponds to 7,680,000 intersection tests. For 100 objects, this increases to more than 75 million intersection tests. Even on the fastest computers, this takes a long time and is not efficient. Over the years, people have developed many techniques for improving the efficiency.
Since rays can come from any direction, traditional efficiency ploys, such as back-facing cannot be used in recursive ray tracing. Objects that would otherwise be invisible, including back faces, may be reflected from or refracted through visible surfaces.

**Bounding volumes**

Complex objects are comprised of collections of more primitive objects. Given this object definition, you can construct a bounding volume that completely contains the object it encloses. If a ray does not intersect this bounding volume, it does not intersect any of the objects in the bounding volume and those objects can be removed from further consideration for this ray. This algorithm implemented in the ray tracer is primarily based on one published by Kay and Kajiya in the SIGGRAPH proceedings on bounding methods from 1986.

**Parameterized Ray Tracing**

If the ray trees for an entire image can be preserved, then surface properties can be altered and a new image recomputed relatively quickly, at the cost of only reevaluating the trees. Sequin and Smyrl [SIGGRAPH 1989] present techniques that minimize the time and space needed to process and store ray trees.

**Note!** There are other algorithms, see text. Each pixel can do parallel processing.
Texture

One of the problems with rendering images is that objects appear unnaturally clean and smooth. Almost all surfaces in the real world are textured. To achieve a realistic appearance, you must add texture surfaces using various patterns, images, and random variations. The goal here is not only to add just about any kind of texture to an object, but also to allow for random textures that can be easily parameterized.

The idea of texture mapping (or the mapping of an image onto another surface) is conceptually simple. When a point on the surface needs to be rendered, the color for that point is determined by transforming the three-dimensional intersection point to a two-dimensional point in the texture map image. The color of the texture map at that point is input to the shading model to determine the color of the surface at the intersection point.

While texture mapping is a powerful technique, it does have its limitations. The most obvious limitation is mapping a two-dimensional image onto a three-dimensional object without excessively distorting the texture-mapped image. Flat surfaces are no problem, but surfaces with complex shapes can be difficult to map. Even for a relatively simple surface such as a sphere, the texture map must be distorted to get it to fit onto the three-dimensional surface. For spheres, this results in the compression of the texture at the poles.

An overall texture mapping consists of a surface parameterization followed by the normal geometric transformations, as shown below:
Now as we have seen the "standard" object representation and "standard" rendering regime imply algorithms driven from screen space. Interpolative shading and Z-buffer hidden surface removal imply a pixel-by-pixel ordering for each polygon. This means that we have to find a single texture value for each pixel to insert into the interpolative shading scheme. The easiest way to do this is by inverse mapping.

The use of an anti-aliasing method is mandatory with texture mapping. If we simply point sample at the center of the pixel and take the value of $T(u,v)$ at the corresponding point in texture space then grossly incorrect results will follow. One pixel is often covered by a number of texels. To avoid aliasing problems, we must consider all relevant texels. We compute a value for the pixel by summing all texels that lie within the quadrilateral, weighting each by the fraction of the texel that lies within the quadrilateral. For example, Blinn and Newell, "Texture and reflection in computer generated images", Comm. ACM, 19(10) 362-7, use a pyramid weighting function with the pyramid distorted to fit the approximating quadrilateral.

Filter kernel is approximated using a pyramid.
Using filtering techniques in a context where, for example, large texture areas are mapped into small screen areas is costly. An elegant prefiltering technique where cost does not grow in proportion to mapped texture area, is Williams’ 'mip-mapping' scheme, "Pyramidal parametrics", Computer Graphics, 17 (3), 1-11.

Texture mapping, we consider the transfer of the pattern

• Pre-calculate how the texture should look at various distances, then use the appropriate texture. This is called mipmapping

Two-part mapping

For a certain polygon, how do we map each interior point in the polygon into a point in T(u,v) space? Then, for a number of connected polygons making up an entire object, how do we map each polygon into T(u,v) space? This is a global mapping problem. The second problem tends to predominate and we will now look at a method that combines both these requirements into a single algorithm. Introduced by Bier and Sloan, "Two-part texture mapping", IEEE Computer Graphics and Applications*, 6(9), p40-53, 1986. It is a method that will map two-dimensional texture onto unconstrained polygon mesh models. The basis of the method is as follows:

(1) The first stage is a mapping from two-dimensional texture space to a simple three-dimensional intermediate surface such as a cylinder:

\[ T(u,v) \rightarrow T'(x_i,y_i,z_i) \]

This is known as the S mapping.

(2) A second stage maps the three-dimensional texture pattern onto the object surface:

\[ T'(x_i,y_i,z_i) \rightarrow O(x_w,y_w,z_w) \]

This is known as the O mapping.
Various possibilities occur for the $O$ mapping where the texture values for $O(x_w, y_w, z_w)$ are obtained from $T(x_i, y_i, z_i)$. The four $O$ mappings are shown below:

(a) reflected ray (this is in fact environment mapping)

(b) Object normal

(c) Object centroid

(d) intermediate surface normal

If the intermediate surface is a cylinder and $O$ mapping is an intermediate surface normal, we call shrink wrap. For example, inverse mapping using the shrink wrap method.
T(u,v)

\[(\theta, z) \to (u, v) = \left( \frac{\theta}{2 \pi}, \frac{z}{h} \right)\]

\[(x_w, y_w, z_w) \to (\theta, z) = (\tan^{-1}(y_w/x_w), z_w)\]

Note!
\[\theta = \tan^{-1}(y/x) = \tan^{-1}(y_w/x_w)\]
and \[Z_i = Z_w\]
Note! In the shrink wrap case we simply join the object point to the center of the cylinder and the intersection of this line with the surface of the cylinder gives us \((x, y, z)\).

Any point on the curved surface of a cylinder of radius \(r\) and height \(h\) can be represented as:
\[(\theta, z) \text{ which is } (rcos\theta, rsin\theta, hw), \quad \text{where } 0 \leq \theta \leq 2\pi, \quad 0 \leq w \leq 1 \quad \text{and } z = hw \text{ or } w = z/h.\]
We can associate texture values \((u, v)\) with a point on the cylinder by
\[(u, v) = (\theta/2\pi, w) = (\theta/2\pi, z/h), \quad \text{where } (u, v) \in [0, 1].\]

**Procedural or Solid Texturing**

Peachey [SIGGRAPH 85] and Perlin [SIGGRAPH 85] simultaneously developed the idea of **solid texturing** to solve this problem. The underlying principle of solid texturing is to create a three-dimensional texture map from which the textured object appears to be carved. This texture map either may be defined explicitly as a three-dimensional array of values (which consumes huge amounts of memory) or be defined by a procedural function. The procedural function takes an \((x, y, z)\) point and returns the surface characteristics at that point. Perlin introduced the \text{noise()} function to generate many of his procedural textures. To this day, the images he produced for his SIGGRAPH paper in 1985 are considered some of the best in computer graphics.
Wood Grain – An Example of a Solid Texture [Alan Watt]

procedure wood_grain(u,v,w:real; var r,g,b:real);
var radius, angle: real;
  grain : integer;
begin
  radius := sqrt(sqr(u)+sqr(w));
  if w = 0 then angle := pi/2
  else angle := arctan(u,w);
  { arctan evaluates arctan (u/w), but uses quadrant information to
  return a value in the range 0..2pi }
  radius := radius + 2*sin(20*angle+v/150);
  grain := round(radius) mod 60;
  if grain < 30 then
    begin
      r := r_light;
      g := g_light;
      b := b_light;
    end
  else
    begin
      r := r_dark;
      g := g_dark;
      b := b_dark;
    end;
end { wood_grain }

------------------------------------------

Marble – An Example of a Solid Texture [Alan Watt]

procedure marble(u,v,w:real; var r,g,b : real);
const width = 0.02;
var  d, dd : real;
  i : integer;
begin
  d := (u+15000)*width + 7*noise(u/100, v/200, w/200);
  dd := trunc(d) mod 17;
  if dd < 4 then
\[
i := 0.7 + 0.2 \times \text{noise}(u/70, v/50, w/50);
\]

else

\[
\text{if (dd} < 9 \text{) or (dd} \geq 12 \text{) then}
\]

begin

\[
d := \text{abs}(d - \text{trunc}(d/17) \times 17 - 10.5) \times 0.1538462;
\]

\[
i := 0.4 + 0.3 \times d + 0.2 \times \text{noise}(u/100, v/100, w/100);
\]

end

else

\[
i := 0.2 + 0.2 \times \text{noise}(u/100, v/100, w/100);
\]

\[
r := 0.9 \times i;
\]

\[
g := 0.8 \times i;
\]

\[
b := 0.6 \times i;
\]

end \{ \text{marble} \}

procedure initialize_noise;

var x, y, z, xx, yy, zz : nrange;

begin

\{ set up the noise lattice \}

for x := 0 to max\_noise do

for y := 0 to max\_noise do

for z := 0 to max\_noise do

begin

\text{noise\_table}[x,y,z] := \text{round}(\text{random}(1) \times 10000);

if x = max\_noise then xx := 0 else xx := x;

if y = max\_noise then yy := 0 else yy := y;

if z = max\_noise then zz := 0 else zz := z;

end;

end \{ initialize\_noise \};

function frac(r:real):real;

begin

frac := r - \text{trunc}(r);

end \{ frac \};

function noise(x,y,z:real):real;
var ix, iy, iz : integer;
ox, oy, oz : real;
n : integer;
n00, n01, n11 : real;
n0, n1 : real;

begin

{ offset x, y, z to ensure they are positive }
x := x + 15000; y := y + 15000; z := z + 15000;
{ find lattice coordinates and real offsets }
ix := trunc(x) mod max_noise;
iy := trunc(y) mod max_noise;
iz := trunc(z) mod max_noise;
ox := frac(x); oy:= frac(y); oz := frac(z);
{ interpolate to get noise value at (ix+ox, iy+oy, iz+oz) }
n := noise_table[ix, iy, iz];
n00 := n + ox*(noise_table[ix+1, iy, iz] - n);
n := noise_table[ix, iy, iz +1];
n01 := n + ox*(noise_table[ix+1, iy, iz+1]-n);
n := noise_table[ix, iy+1, iz];
n10 := n + ox*(noise_table[ix+1, iy+1, iz]-n);
n := noise_table[ix, iy+1, iz+1];
n11 := n + ox*(noise_table[ix+1, iy+1, iz+1]-n);
n0 := n00+oy*(n10-n00);
n1 := n01+oy*(n11-n01);
noise:=(n0+oz*(n1-n0)*0.0001;

end { noise }

[Perlin [SIGGRAPH 85] We can use Noise() to create function turbulence() which gives a reasonable visual appearance of turbulent flow. We may then use turbulence() to simulate the appearance of marble.

A suitable procedure for the simulation of turbulence using Noise() is :

function turbulence(p)
    t = 0;
    scale = 1;
while (scale > pixelsize)
{
    t+=abs(Noise(p/scale)*scale;
    scale /= 2;
}
return t;

We observe that marble consists of heterogeneous layers. The unperturbed layers alone can be modeled by a simple color filtered sine wave:

function boring_marble(point)
    x = point[1];
    return marble_color(sin(x))

where point[1] denotes the first (i.e x) component of the point vector and marble_color() has been defined as a spline function mapping scalars to color vectors. To go from this to realistic marble we need only perturb the layers:

function boring_marble(point)
    x = point[1] + turbulence(point);
    return marble_color(sin(x))

Listing in Java

import java.awt.*;

class VectorDxyz
{
    double x, y, z;
    VectorDxyz()
    VectorDxyz(double x, double y, double z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
};

public final class Marble_Texture extends Object {
    // No Constructor

/ Borrow from Alan Watt
static final int max_noise=100;
static int noise_table[][][]=new int [max_noise+1][max_noise+1][max_noise+1];

//Class Methods
public static void initialize_noise()
{
    int x,y,z,xx,yy,zz;
    boolean first_time = true;

    if (first_time)
    {
        for (x=0;x<=max_noise;x++)
            for (y=0;y<=max_noise;y++)
                for (z=0;z<=max_noise;z++)
                {
                    noise_table[x][y][z]=Math.round((float)(Math.random()*10000));
                    if (x==max_noise) xx=0; else xx=x;
                    if (y==max_noise) yy=0; else yy=y;
                    if (z==max_noise) zz=0; else zz=z;
                    noise_table[x][y][z]=noise_table[xx][yy][zz];
                }
        first_time = false;
    }
}

private static double frac(double r)
{
    return(r-(long)r);
}

public static double noise(double x, double y, double z)
{
    int ix,iy,iz,n;

    noise_table[x][y][z]=Math.round((float)(Math.random()*10000));
    if (x==max_noise) xx=0; else xx=x;
    if (y==max_noise) yy=0; else yy=y;
    if (z==max_noise) zz=0; else zz=z;
    noise_table[x][y][z]=noise_table[xx][yy][zz];

    first_time = false;
}

double ox, oy, oz, n00, n01, n10, n11, n0, n1;

x=x+15000; y=y+15000; z=z+15000;
x=(int)(x)%max_noise;
y=(int)(y)%max_noise;
iz=(int)(z)%max_noise;
ox=frac(x); oy=frac(y); oz=frac(z);
n=noise_table[ix][iy][iz];
n00=n+ox*(noise_table[ix+1][iy][iz]-n);
n=noise_table[ix][iy][iz+1];
n01=n+ox*(noise_table[ix+1][iy][iz+1]-n);
n=noise_table[ix][iy+1][iz];
n10=n+ox*(noise_table[ix+1][iy+1][iz]-n);
n=noise_table[ix][iy+1][iz+1];
n11=n+ox*(noise_table[ix+1][iy+1][iz+1]-n);

n0=n00+oy*(n10-n00);
n1=n01+oy*(n11-n01);
return (n0+oz*(n1-n0))*0.0001;
}

public static double turbulence(double u, double v, double w) {
    double t = 0;
    double scale = 4.0;  //2, 4, 8, .. // change distored here
    double pixel_size = 0.5;  //0.5, 0.25, 0.125 change # vein here

    while (scale > pixel_size) {
        t+=Math.abs(noise(u/scale,v/scale,w/scale)*scale);
        scale /= 2.0;
    }
    return t;
}

public static VectorDxyz dnoise(double u, double v, double w)
{  
double center;

double tx, ty, tz, x,y,z;

    center = noise(u, v, w);
    tx = u+0.1;
    ty = v;
    tz = w;
    x = (noise(tx,ty,tz)-center)*10.0;
    tx = u;
    ty += 0.1;
    y = (noise(tx,ty,tz)-center)*10.0;
    ty = v;
    tz += 0.1;
    z = (noise(tx,ty,tz)-center)*10.0;
    return new VectorDxyz(x,y,z);
}

public static VectorDxyz marble_color(double t)  
{
    double red, green, blue;
    // a spline function mapping scalars to color vectors.
    // x(t) = ??
    // red = x(t);
    // green = x(t;
    // blue = x(t);
    red = t;
    green = t;
    blue = t;
    return new VectorDxyz(red,green,blue);
}

public static VectorDxyz marble(double u, double v, double w)  
{
    double x = u+turbulence(u,v,w);

    // Your code here...
return marble_color(Math.sin(x));
}
// boring_marble
// public static VectorDxyz marble(double u, double v, double w)
// {
//     double x = u;
//     return marble_color(Math.sin(x));
// }

Bump Mapping

Texture mapping affects a surface’s shading, but the surface continues to appear geometrically smooth. If the texture map is a photograph of a rough surface, the surface being shaded will not look quite right, because the direction to the light source used to create the texture map is typically different from the direction to the light source illuminating the surface. Blinn [SIGGRAPH 78] developed a way to provide the appearance of modified surface geometry that avoids explicit geometrical modeling. His approach involves perturbing the surface normal before it is used in the illumination model, just as slight roughness in a surface would perturb the surface normal. This method is known as bump mapping.

Let normal vector $\mathbf{N} = \mathbf{O}_u \times \mathbf{O}_v$, we want to perturb $\mathbf{N}$ to be a new normal vector $\mathbf{N}'$ as shown below.

In implementing bump mapping, a scheme is required that perturbs the normal vector in a way that is independent of the orientation (rotation) and position (translation) of the surface. Blinn shows that a good approximation to the new normal $\mathbf{N}'$ is

$$\mathbf{N}' = \mathbf{N} + \frac{\mathbf{B}_v (\mathbf{N} \times \mathbf{O}_v) - \mathbf{B}_v (\mathbf{N} \times \mathbf{O}_w)}{|\mathbf{N}|} \quad \text{or} \quad \mathbf{N}' = \mathbf{N} + \frac{\mathbf{D}}{|\mathbf{N}|}$$
where \( B_u \) and \( B_v \) are the partial derivatives of the bump map \( B(u,v) \), a displacement function or height field but use its derivatives at the point \((u,v)\) to calculate \( D \). Bilinear interpolation can be used to derive bump map values for specified \((u,v)\) positions, and finite differences can be used to compute \( B_u \) and \( B_v \).

![Diagram of bump mapping](image)

An example of the stages involved in bump mapping is shown below:

1. **Original surface**
2. **Bump map**
3. **Lengthening or shortening \( O_u \) and \( O_v \) using \( B_u \) and \( B_v \)**
4. **Vector \( N' \)**
Another convenient primitive is the vector valued differential of the Noise() [Peachey, "Solid Texturing of Complex Surfaces", SIGGRAPH 85], defined by the instantaneous rate of change of Noise() along the x, y, and z directions, respectively. We will call this function Dnoise(). Dnoise() provides a simple way of specifying normal perturbation as follow:

\[
\begin{align*}
  f &= 1; \\
  \text{while } (f < \text{pixel_freq}) \\
  &\{ \\
    \text{Normal} = \text{Normal} + \text{Dnoise}(f \times x, f \times y, f \times z); \\
    f &= 2; \\
  &\}\end{align*}
\]

where

\[
\text{Dnoise}(\text{real } u, \text{real } v, \text{real } w) \\
\{ \\
  \text{real} \text{ center}; \\
  \text{real } tx, ty, tz, x, y, z; \\
  \text{center} = \text{noise}(u, v, w); \\
  tx = u + 0.1; \\
  ty = v; \\
  tz = w; \\
  x = (\text{noise}(tx, ty, tz)-\text{center}) \times 10; \\
  tx = u; \\
  ty += 0.1; \\
  y = (\text{noise}(tx, ty, tz)-\text{center}) \times 10; \\
  ty = v; \\
  tz += 0.1; \\
  z = (\text{noise}(tx, ty, tz)-\text{center}) \times 10; \\
  \text{return } (x, y, z); \\
\}
\]
Part IV Animation

Read chapter 16 of Donald and M. Pauline Baker, "Computer Graphics C Version".

How Double Buffering Works

1. A computer graphics application that employs double buffering maintains two separate video buffers, which we'll refer to as buffers A and B. A switch on the video adapter determines which video buffer is shown on the screen. The buffer that the switch points to is referred to as the foreground buffer, and the other as the background buffer. The switch can be toggled back and forth with software commands, effectively swapping the foreground and background buffers and quickly changing what appears on the screen. Initially, both video buffers are empty and the switch points to buffer A.

2. To begin the animation sequence, the computer draws an image into buffer B. The drawing process is hidden from view because buffer B is not displayed.
3. Toggling the switch swaps the buffers, bringing buffer B to the foreground and displaying the image that was just drawn. When the switch occurs, the new foreground image appears almost instantaneously—in the 1/60 of a second or so that it takes for the monitor to refresh the screen.

4. The second scene in the animation sequence is drawn into buffer A, which is now the background buffer. Once again, the drawing action is hidden from view because the screen shows only what's in the foreground.

5. Swapping the buffers a second time displays the second screen in the animation sequence. This process—drawing an image in the background buffer and swapping the buffers—is repeated over and over to create a moving image. No matter how long it takes for the computer to create each new screen, all the user sees is the completed images. However, the higher the frame rate (the more quickly the screen can be drawn and displayed) and the smaller the incremental difference between frames, the smoother the animation.
How Mask Animation Works

1. Mask animation is frequently used when a small, nonrectangular object is moved about the scene against a backdrop formed by a full-screen image. A pair of masks stored in main computer memory serves as a template for drawing the object. The AND mask, which stamps a silhouette of the object into a background image, outlines the object with 1s in a field of 0s. The XOR mask, which draws the object in the impression created by the AND mask, contains the pixel values depicting the object. In this example, 4 represents the color red and 6 represents yellow. Background pixels are represented by 0s.

2. At the outset, the background scene is drawn into the video buffer.

3. A rectangular block of pixels at the location where the object will first appear is copied from the video buffer to a small buffer set aside in main computer memory.
4. The AND mask is applied to the same rectangular region of the video buffer. Pixels that correspond to 1s in the AND mask remain unchanged, while pixels that correspond to 0s are set to 0. The object is now silhouetted against the background.

5. The XOR mask is applied at the same location. Pixels that correspond to 0s in the XOR mask remain unchanged; other pixels assume the color values specified in the mask. The object is now fully displayed on the screen.

6. To animate the object's movement, the computer first erases the object by copying the background rectangle back to the video buffer. Then it makes a copy of the background underlying the object's new location.

7. The object is drawn in its new position with the AND and XOR masks. If it is repeatedly erased and redrawn in this manner and the steps are performed quickly enough, the object will appear to glide across the screen.
Scan Converting Lines

- Function of two variables \( f(x, y) = ax + by + c = 0 \)
- Point-slope form \( y - y_1 = m(x - x_1) \) เป็นสมการเส้นตรงที่ผ่านจุด \((x_1, y_1)\) และมีความชัน \(m\)
- Slope-intercept form \( y = mx + B \) เป็นสมการเส้นตรงที่มีความชัน \(m\) และมีระยะตัดแกน \(Y\) ที่ \(B\)
- Two points form \( \frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \) เป็นสมการเส้นตรงที่ผ่านจุด \((x_1, y_1)\) และ \((x_2, y_2)\)

หมายเหตุ เกี่ยวกับ 2. โดยที่ \((x_1, y_1) = (0, B)\)

The Basic Incremental Algorithm (Simple Digital Differential Analyzer or Simple DDA)

The simplest strategy for scan conversion of lines is to compute the slope \(m\) as \(\nabla y/\nabla x\), to increment \(x\) by 1 starting with the leftmost point, to calculate \(y = mx + B\) for each \(x\), and to intensify the pixel at \((x, \text{Round}(y))\), where \(\text{Round}(y) = \text{Floor}(0.5 + y)\). This computation selects the closest pixel – that is, the pixel whose distance to the true line is smallest. This brute-force strategy is inefficient, however, because each iteration requires a floating-point multiply, addition and invocation of Floor.

We can eliminate the multiplication by noting that

\[
y_{i+1} = mx_{i+1} + B = m(x_{i+1} + \nabla x) + B = y_i + m\nabla x
\]

and, if \(\nabla x = 1\), then \(y_{i+1} = y_i + m\).

Thus, a unit change in \(x\) changes \(y\) by \(m\), which is the slope of the line. For all points \((x, y)\) on the line, we know that, if \(x_{i+1} = x_i + 1\), then \(y_{i+1} = y_i + m\); that is, the values of \(x\) and \(y\) are defined in terms of their previous values (see figure below).
This is what defines an incremental algorithm: At each step, we make incremental calculations based on the preceding step.

**Procedure Line:**

```pseudocode
procedure Line(   { Assumes –1 ≤ m ≤ 1, x0 < x1 } 
x0, y0,    { Left endpoint } 
x1, y1,    { Right endpoint } 
color : integer);   { value to place in line's pixels }
var
x : integer;   { x runs from x0 to x1 in unit increments. }
dy, dx, y, m : real;
begin
  dy := y1 – y0;
  dx := x1 – x0;
  m := dy/dx;
  y := y0;
  for x := x0 to x1 do
    begin
      putpixel(x,Round(y), color);   { Set pixel to color value } 
      y := y + m   { Step y by slope m } 
    end
end; { Line }
```

If \( m > 1 \), a step in \( x \) creates a step in \( y \) that is greater than 1. Thus we must reverse the roles of \( x \) and \( y \) by assigning a unit step to \( y \) and incrementing \( x \) by \( \nabla x = \nabla y/m = 1/m \).
- **Midpoint Line Algorithm (Bresenham Algorithm or Integer DDA)**

  จะพิจารณาเฉพาะกรณีเส้นตรงจากจุด \((x_0,y_0)\) ไปยัง \((x_1,y_1)\) มีความลาดชันมากกว่าหรือเท่ากับ 0 และน้อยกว่าหรือเท่ากับ 1 และ \((x_0,y_0)\)อยู่ที่มุมซ้ายล่างส่วน \((x_1,y_1)\) อยู่ที่มุมขวาด้านบน

  ![Desired Line](image)

  **Previous choices for pixel** \((x_p, y_p)\)  **Current pixel** \((x, y)\)  **Next pixel** \((x, y + \delta)\)

  **ข้อสังเกตุ**

  ถ้าจุด \((x_1,y_1)\)อยู่ในเส้น ค่า \(f(x_1,y_1)\) จะเท่ากับ 0 แต่ถ้า \((x_1,y_1)\)อยู่ข้างนอกเส้น ค่า \(f(x_1,y_1)\) จะมีค่ามากกว่า 0 แต่ถ้า \((x,y)\)อยู่ข้างล่างเส้น ค่า \(f(x,y)\) จะมีค่ามากกว่า 0

  **พิจารณา**

  จาก \(f(x,y) = dy.x-dx.y + B.dx = 0\)

  พิจารณาจุด \(M\) หรือ \((x,y+\delta)\) ซึ่งอยู่ข้างบนเส้นจะได้ว่า

  \[
  f(x,y+\delta) = dy.x-dx.(y+\delta) + B.dx = dy.x-dx.y + B.dx - dx.\delta
  \]

  \[
  = 0 - dx.\delta
  \]

  \(< 0\) เนื่องจาก \(dx\) และ \(\delta\) มีค่ามากกว่า 0

  **สรุป** ถ้า \(f(x,y+\delta) < 0\) จอสัมผัสจะอยู่เหนือเส้น หรือ เส้นจะอยู่ใต้จุด \(M\)

  **หาสูตร**

  \[
  dy = y_1 - y_0\] และ \(dx = x_1 - x_0\)

  สามารถเขียนเป็นสมการเส้นตรงในรูปความลาดชันได้ดังนี้
\[ y = \frac{dy}{dx} \times x + B \]

เมื่อ \( B \) เป็นจุดตัดที่แกน \( y \)

พิจารณาสมการเส้นตรงต่อไปนี้

\[ f(x,y) = ax+by+c = 0 \]

เขียนใหม่ให้หูเป็นรูปความลาดชัน ได้เป็น

\[ y = ax - \frac{c}{b} \]

นั่นคือ \( a = \frac{dy}{dx}, \quad b = -dx, \quad \text{และ} \quad c = B.d_x \) (Note! \( B = -c \)

\[ \frac{b}{-B} \]

ถ้าต้องการตรวจสอบว่าเส้นตรงที่ต้องการอยู่ใกล้ NE หรือ E ทำได้โดยการแทนค่าที่พุ่ม \( M \) คือ \((x_p+1, y_p+1/2)\) ใน \( f(x,y) \) อีกได้เป็นสมการว่าจุด \( M \) อยู่ข้างบนเส้น หรือเส้นอีก \( E \) นั่นเอง แต่ถ้าเป็นอย่างอื่น แสดงว่า

อยู่ข้างล่างเส้น หรือเส้นอีก \( NE \) เขียนเป็นสูตรได้ดังนี้

\[ d = f(x_p+1, y_p+1/2) = a(x_p+1) + b(y_p+1/2) + c \]

นั่นคือ ถ้า \( d > 0 \) เลือก NE ถ้า \( d < 0 \) เลือก E ถ้า \( d = 0 \) เลือกในที่ใกล้กันได้ สมมุติว่าเลือก NE

- กรณีที่เลือก E ค่า \( d \) ที่จะนำไปคำนวณ \( M \) ต้องมากกว่า \( 1 \) ได้ \( d_{\text{new}} \) ดังนี้

\[ d_{\text{new}} = f(x_p+2, y_p+1/2) = a(x_p+2) + b(y_p+1/2) + c \]

แต่

\[ d_{\text{old}} = f(x_p+1, y_p+1/2) = a(x_p+1) + b(y_p+1/2) + c \]

\[ d_{\text{new}} - d_{\text{old}} = a \quad \text{หรือ} \quad d_{\text{new}} = d_{\text{old}} + a \quad \text{หรือ} \quad d_{\text{new}} = d_{\text{old}} + dy \]

- กรณีที่เลือก NE ค่า \( x,y \) ที่จะนำไปคำนวณ \( M \) ต้องมากกว่า \( 1 \) ทั้งคู่ ได้ \( d_{\text{new}} \) ดังนี้

\[ d_{\text{new}} = f(x_p+2, y_p+3/2) = a(x_p+2) + b(y_p+3/2) + c \]

แต่

\[ d_{\text{old}} = f(x_p+1, y_p+1/2) = a(x_p+1) + b(y_p+1/2) + c \]

\[ d_{\text{new}} - d_{\text{old}} = a + b \quad \text{หรือ} \quad d_{\text{new}} = d_{\text{old}} + a + b \quad \text{หรือ} \quad d_{\text{new}} = d_{\text{old}} +(dy - dx) \]

เนื่องจากเราถูกจำกัดด้าน \((x_p,y_p)\) เราสามารถคำนวณค่า \( d \) นั่นได้ \((p=0)\) ดังนี้

\[ d_{\text{start}} = f(x_p+1, y_p+1/2) = a(x_p+1) + b(y_p+1/2) + c \]

\[ = f(x_p, y_p) + a + b/2 \]

\[ = a + b/2 \]

เพื่อไม่ให้เกิดเศษส่วนในการหาร \( b/2 \) สามารถเขียนสูตรใหม่ โดยใช้ 2 คู่ \( (x,y) = 2(ax+by+c) \) หลังจากคำนวณใหม่ จะได้ว่า

\[ dx = x_1 - x_0 \]

\[ dy = y_1 - y_0 \]

\[ d_{\text{start}} = 2a+b = 2*dy - dx \]

ถ้าเลือก E \( \Rightarrow x = x+1 \) และ \( d_{\text{new}} = d_{\text{old}} + 2dy \)
ถ้าเลือก NE \( \Rightarrow x = x+1, y = y+1 \) และ \( d_{new} = d_{old} + 2(dy-dx) \)

ในการเขียนโปรแกรมเราจะใช้ตัวแปร \( d \) แทน \( d_{start}, d_{new}, \) และ \( d_{old} \) โดยเริ่มจากจุด \((x_0,y_0)\) คำนวณ \( d \) ซึ่งเริ่มต้นเท่ากับ \( d_{start} \) หรือ \( 2*dy-dx \) ถ้า \( d \) ไปทดสอบว่าจุดต่อไปอยู่ที่ \( E \) หรือ \( NE \) ถ้าอยู่ที่ \( E \) จุดต่อไปคือ \((x+1,y)\) และค่า \( d \) ที่จะนำไปทดสอบตัวแหน่งของจุดต่อไปจะเท่ากับ \( d+2dy \) แต่ถ้าเลือก \( NE \) จุดต่อไปคือ \((x+1,y+1)\) และค่า \( d \) ที่จะนำไปทดสอบตัวแหน่งของจุดต่อไปจะเท่ากับ \( 2(dy-dx) \)

ขั้นตอนวิธี:

```pascal
procedure Line(x0,y0,x1,y1,color : integer)
var dx, dy, incrE, incrNE, d, x, y : integer;
begnin
  dx := x1 – x0;
  dy := y1 – y0;
  d := 2*dy-dx;  \{ initial value of d \}
  incrE := 2 * dy;  \{ increment used for move to E \}
  incrNE := 2*(dy-dx); \{ increment used for move to NE \}
  x := xo;
  y := y0;
  putpixel(x,y,color);
  while x < x1 do
  begin
    if d <= 0 then        \{ choose E \}
      begin
        d := d + incrE;
        x := x + 1;
      end
    else
      begin   \{ choose NE \}
        d := d + incrNE;
        x := x + 1;
        y := y + 1;
      end;
      putpixel(x,y,color);  \{ The selected pixel closest to the line \}
  end { while }
end {Line}
```
void swap(int *i1, int *i2)
{
    int temp;
    temp = *i2;
    *i2 = *i1;
    *i1 = temp;
}

void Line(int x0, int y0, int x1, int y1, int color)
{
    int d, dx, dy, aincr, bincr, xincr, yincr, x, y;

    if (abs(x1-x0) < abs(y1-y0)) { /* y dominate */
        if (y0>y1) {
            swap(&x1,&x2);
            swap(&y1,&y2);
        }
        if (x1>x0) xincr = 1;
        else xincr = (-1);
        dy = y1 - y0;
        dx = abs(x1 - x0);
        d = 2 * dx - dy;
        aincr = 2 * (dx - dy);
        bincr = 2 * dx;
        x = x0;
        y = y0;
        putpix(x,y,color);
        for (y = y0+1; y <= y1; y++) {
            if (d>=0) {
                x = x + xincr;
                d = d + aincr;
            } else
d = d + bincr;
putpix(x,y,color);
}
else { /* x dominate */
if (x0 > x1) {
    swap(&x0,&x1);
    swap(&y0,&y1);
}
if (y1>y0) yincr=1; /* slope positive */
else yincr = (-1); /* slope negative */
dx = x1 - x0;
dy = abs(y1 - y0);
d = 2 * dy - dx;
aincr = 2 * (dy - dx); /* aincr is incrNE */
bincr = 2 * dy; /* bincr is incrE */
x = x0;
y = y0;
putpix(x,y,color);
for (x = x0+1; x <= x1; x++) {
    if (d>=0) {
        y = y + yincr;
        d = d + aincr;
    }
    else
        d = d + bincr;
    putpix(x,y,color);
}
}


procedure line(x0, y0, x1, y1 :integer; color : colors);
label 99;

var d, x, y, ax, ay, sx, sy, dx, dy : integer;
begin (* line *)
  dx := x1 - x0;
  ax := abs(dx)*2;
  if dx > 0 then sx := 1 else sx := -1;
  dy := y1 - y0;
  ay := abs(dy)*2;
  if dy > 0 then sy := 1 else sy := -1;
  x := x0;
  y := y0;
  if ax > ay then begin (* x dominat *)
    d := ay - ax div 2;
    while true do begin
      putpix(x,y,color);
      if x = x1 then goto 99;
      if d >= 0 then begin
        y := y + sy;
        d := d - ax;
      end;
      x := x + sx;
      d := d + ay;
    end (* while *)
  end
  else begin (* y dominate *)
    d := ax - ay div 2;
    while true do begin
      putpix(x,y,color);
      if y = y1 then goto 99;
      if d >= 0 then begin
        x := x + sx;
        d := d - ay;
      end;
      y := y + sy;
      d := d + ax;
    end (* while *)
  end
end (* line *)
Scan Converting Circles

แนวคิดเหมือน Midpoint Line Algorithm

พิจารณาวงกลมที่มีจุดศูนย์กลางอยู่ที่ (0,0) และมีรัศมี R ต่อไปนี้

เราจะพิจารณาวงกลมแบ่งเป็น quadrants คือ quadrants ที่อยู่ระหว่าง (0,R) ถึง (R/ 2 , R / 2 ) (ที่มุม 45°) เพราะ quadrant อื่น ๆ มีลักษณะสมมาตรกับ quadrant ที่กำลังพิจารณา ถ้าเราสอดจุดบน quadrant ที่กำลังพิจารณา เราสามารถหาจุดบน quadrant อื่น ๆ ที่สมมาตรกัน โดยเขียนเป็นโปรแกรมย่อยเพื่อแสดงจุดต่าง ๆ ได้ดังนี้

```pascal
procedure CirclePoints(x,y,color:integer);
begin
    putpixel(x,y,color);
    putpixel(y,x,color);
    putpixel(y,-x,color);
    putpixel(x,-y,color);
    putpixel(-x,-y,color);
    putpixel(-y,-x,color);
    putpixel(-y,x,color);
    putpixel(-x,y,color);
end;
```

พิจารณาวงกลมที่ต้องการ ดังนี้

\[ P = (x_p, y_p) \]
แนวคิด: ถ้าจุด M อยู่ภายในวงกลมเลือกจุด E ถ้าไม่ในวงกลมเลือกจุด SE
ถ้าเลือกจุด E การคำนวณหาจุด M ถัดไปจะต้องบวก x 1 และลบ y 1
dังนั้นเลือกจุด SE การคำนวณหาจุด M ถัดไปจะต้องบวก x 1 และลบ y 1

หาสูตร:
\[ f(x,y) = x^2+y^2-R^2 = 0 \]

ถ้าต้องการตรวจสอบเส้นตรงที่ต้องการอยู่ใกล้ SE หรือ E ทำได้โดยแทนค่าที่จุด M คือ \((x_p+1,\ y_{p-1/2})\) หรือ \((x_{p+1},\ y_{p-1/2})\) ใน \(f(x,y)\) ถ้ามีค่าเป็นลบแสดงว่าจุด M อยู่ข้างในวงกลม หรือใกล้ E นั่นเอง แสดงว่าอยู่ข้างนอกวงกลม หรือใกล้ SE เขียนเป็นสูตรได้ดังนี้
\[ d = f(x_p+1,\ y_{p-1/2}) = (x_p+1)^2 + (y_{p-1/2})^2 - R^2 \]

นั่นคือ ถ้า \(d > 0\) เลือก SE ถ้า \(d < 0\) เลือก E ถ้า \(d = 0\) เลือกอันไหนก็ได้ สมมุติว่าเลือก SE
- กรณีที่เลือก E ค่า x ที่จะนำไปคำนวณ M ต้องบวกค่า 1 ได้ \(d_{\text{new}}\) ดังนี้
\[ d_{\text{new}} = f(x_p+2,\ y_{p-1/2}) = (x_{p+1})^2 + (y_{p-1/2})^2 - R^2 \]

แต่
\[ d_{\text{old}} = f(x_p+1,\ y_{p-1/2}) = (x_p+1)^2 + (y_{p-1/2})^2 - R^2 \]
\[ d_{\text{new}} - d_{\text{old}} = (2x_p + 3) \] หรือ \(d_{\text{new}} = d_{\text{old}} + (2x_p + 3)\)
- กรณีที่เลือก SE ค่า x ที่จะนำไปคำนวณ M ต้องบวกค่า 1 และลบ y 1 ได้ \(d_{\text{new}}\) ดังนี้
\[ d_{\text{new}} = f(x_p+2,\ y_{p-3/2}) = (x_{p+1})^2 + (y_{p-3/2})^2 - R^2 \]

แต่
\[ d_{\text{old}} = d_{\text{old}} = f(x_p+1,\ y_{p-1/2}) = (x_p+1)^2 + (y_{p-1/2})^2 - R^2 \]
\[ d_{\text{new}} - d_{\text{old}} = 2x_p - 2y_p + 5 \] หรือ \(d_{\text{new}} = d_{\text{old}} + (2x_p - 2y_p + 5)\)

เนื่องจากจุดเริ่มต้นคือ \((x_0,y_0) = (0,R)\) เราสามารถคำนวณค่า \(d\) เริ่มต้นได้ ดังนี้
\[ d_{\text{start}} = f(0+1, R-1/2) = 1 + (R-1/2)^2 - R^2 \]
\[ = 5/4 - R \]

ขั้นตอนวิธี 1:

procedure Circle(R, color : integer)
{ assume center of circle is at origin }

var x, y : integer; d : real;

begin
  x := 0;;
  y := R;
  d := 5/4-R;
  CirclePoints(x, y, color);
  while y > x do
    begin
      if d < 0 then  { select E }  
      begin
        d := d + 2*x+3;
        x := x + 1;
      end
      else
        begin  { select SE }
          d := d + 2*(x-y) + 5;
          x := x + 1;
          y := y – 1;
        end;
      CirclePoints(x,y,color);
    end; { Circle }
end;

ขั้นตอนวิธี 1 มีข้อเสียคือ d เป็น real เราต้องการให้ d เป็น integer ทำได้โดยการกำหนดให้
h = d – ¼ หรือ d = h + ¼ แทนค่า d ด้วย h + ¼ จะได้

\[
d_{\text{start}} = \frac{5}{4} - R \Rightarrow h_{\text{start}} + \frac{1}{4} = \frac{5}{4} - R \Rightarrow h_{\text{start}} = 1 - R
\]

ถ้าเลือก E \( \Rightarrow d_{\text{new}} = d_{\text{old}} + (2x + 3) \) หรือ \( h_{\text{new}} + \frac{1}{4} = h_{\text{old}} + \frac{1}{4} + (2x + 3) \) หรือ \( h_{\text{new}} = h_{\text{old}} + (2x + 3) \)

ในท่าจะเลือกหน้า \( \Rightarrow h_{\text{new}} = h_{\text{old}} + (2x - 2y + 5) \)

ในการเลือกภาวะทดสอบ \( d < 0 \) เป็นข้อเสียในภาวะทดสอบ \( h + \frac{1}{4} < 0 \) หรือ \( h < -\frac{1}{4} \) แต่เนื่องจากว่า h เป็นเลขจำนวนเต็มและเวลาจะเปลี่ยนจำนวนเต็มเสมอ ดังนั้นจะเป็นภาวะทดสอบ \( h < 0 \) ได้

ขั้นตอนวิธี 2:

procedure Circle(R,color : integer)
{ assume center of circle is at origin, d is h }
var x, y, d: integer;

begin
  x := 0;;
  y := R;
  d := 1-R;
  CirclePoints(x, y, color);
  while y > x do
    begin
      if d < 0 then { select E }
        begin
          d := d + 2*x+3;
          x := x + 1;
        end
      else
        begin { select SE }
          d := d + 2*(x-y) + 5;
          x := x + 1;
          y := y – 1;
        end;
      CirclePoints(x,y,color);
    end; {Circle}
end;

ขั้นตอนวิธี 2 ยังมีข้อเสียตรงการคำนวณ ค่าด ภายในข้อความสั่ง while มีการคูณ เราสามารถปรับปรุงให้ดีขึ้นได้โดยการใช้ second-order differences ดังนี้

1. ถ้าเราเลือก E จุดจะเลื่อนจาก M สมมุติว่าแทนด้วย (x_q, y_q) ไปที่ M_E = (x_q+1, y_q)

การคำนวณ d_new – d_old = 2x_p + 3 เรียกว่า ทำ first-order difference ก็จะได้ deltaE(x_p,y_p) = 2x_p + 3

deltaE_{old}(x_q,y_q) = 2x_q + 3

นั่นคือ deltaE_{new} – deltaE_{old} = 2 เรียกว่า second-order difference

ให้ตัวอย่างว่าเลือก SE จุดจะเลื่อนจาก M ไปที่ M_{SE} = (x_q+1, y_q-1)

การคำนวณ d_new – d_old = 2x_p -2y_p + 5 เรียกว่า ทำ first-order difference ก็จะได้ deltaSE(x_p,y_p) = 2x_p - 2y_p + 5

 deltaSE_{old}(x_q,y_q) = 2x_q -2y_q + 5

deltaSE_{new}(x_q+1,y_q-1) = 2(x_q+1)-2y_q + 5

นั่นคือ deltaSE_{new} – deltaSE_{old} = 2 เรียกว่า second-order difference
2. ถ้าเราเลือก SE จุดจะเลื่อนจาก $M = (x_q, y_q)$ ไปที่ $M_{SE} = (x_q+1, y_q-1)$

\[
\begin{align*}
\delta E_{old}(x_q, y_q) &= 2x_q + 3 \\
\delta E_{new}(x_q+1, y_q-1) &= 2(x_q+1) + 3
\end{align*}
\]

นั่นคือ $\delta E_{new} - \delta E_{old} = 2$

ไม่ก่อนสมการกัน

\[
\begin{align*}
\delta E_{old}(x_q, y_q) &= 2x_q - 2y_q + 5 \\
\delta E_{new}(x_q+1, y_q-1) &= 2(x_q+1) - 2(y_q - 1) + 5
\end{align*}
\]

นั่นคือ $\delta E_{new} - \delta E_{old} = 4$

3. ที่จุดเริ่มต้น $(x_p=0, \ y_p=R) \Rightarrow \delta E_{start}(x_p, y_p) = 2*x_p + 3 = 2*0 + 3$ และ $\delta E_{start}(x_p, y_p) = 2*x_p - 2*y_p + 5 = 2*0 - 2*R + 5 = -2*R + 5$

ขั้นตอนวิธี 3:

```python
procedure Circle(R, color : integer)
{ assume center of circle is at origin , this algorithm use second-order difference }

var x, y , d, deltaE, deltaSE: integer;
begin
x := 0;;
y := R;
d := 1-R;
deltaE := 3;
deltaSE := -2*R+5;
CirclePoints(x, y, color);
while y > x do
begin
if d < 0 then { select E }
begin
    d := d + deltaE;
deltaE := deltaE+2;
deltaSE := deltaSE+2;
x := x + 1;
end
else
begin { select SE }
    d := d + deltaSE;
deltaE := deltaE+2;
deltaSE := deltaSE + 4;
end
end
end
```
\[
x := x + 1; \\
y := y - 1;
\]
end;
CirclePoints(x,y,color);
end; \{ Circle\}

เนื่องจากเราสมมุติว่าจุดศูนย์กลางของวงกลมอยู่ที่ (0,0) ถ้าต้องการให้จุดศูนย์กลาง (xx,yy) ใด ๆ เขียนเป็นภาษาซีได้ดังนี้

```c
void CirclePoints(xx,yy,x,y,color)
int xx,yy,x,y,color;
{
    putpixel(x+xx,y+yy,color);
    putpixel(y+xx,x+yy,color);
    putpixel(y+xx,-x+yy,color);
    putpixel(x+xx,-y+yy,color);
    putpixel(-x+xx,-y+yy,color);
    putpixel(-y+xx,-x+yy,color);
    putpixel(-y+xx,x+yy,color);
    putpixel(-x+xx,y+yy,color);
}

void Circle(xx,yy,r,color)
int xx,yy,r,color;
{
    int x,y,d,deltaE,deltaSE;

    x=0;
y=r;
d=1-r;
deltaE = 3;
deltaSE = (-2*r)+5;
CirclePoints(xx,yy,x,y,color);
while (y>x) {
    ++x;
```
\[ \text{deltaE} = \text{deltaE} + 2; \]
if \( d < 0 \) {
\[ d += \text{deltaE}; \]
\[ \text{deltaSE} += 2; \]
}
else {
\[ d += \text{deltaSE}; \]
\[ \text{deltaSE} += 4;; \]
\[ --y; \]
}
\text{CirclePoints}(xx,yy,x,y,color);
ภาคผนวก ข
Inverse Matrix

1. determinants
determinant ของ matrix $A = \det(A)$ หรือ $|A|$

$$|A_{2x2}| = a_{11}a_{22} - a_{21}a_{12}$$

หรือ

$$|A_{nxn}| = \sum_{j=1}^{n} a_{ij} |A_{ij}|$$

เมื่อใช้แถวที่ $i$ เป็นหลักในการกระจาย

หรือ

$$|A_{nxn}| = \sum_{i=1}^{n} a_{ij} |A_{ij}|$$

เมื่อใช้สดมภ์ที่ $j$ เป็นหลักในการกระจาย

ซึ่ง $|A_{ij}| = (-1)^{i+j} |M_{ij}|$ เรียกว่า cofactor ของสมาชิก $a_{ij}$ ของ $A$ และ $|M_{ij}|$ เรียกว่า minor ของสมาชิก $a_{ij}$ ของ $A$

คือ determinant ของ matrix ที่ได้จากการตัดแถวที่ $i$ และสดมภ์ที่ $j$ ของ matrix $A$ ออกเช่น

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{bmatrix}$$

เสมาแกว

$$\text{minor ของ a}_{12} = |M_{12}| = \begin{vmatrix}
a_{21} & a_{23} \\
a_{31} & a_{33}
\end{vmatrix} = a_{21}a_{33} - a_{31}a_{23}$$

cofactor ของ $a_{12} = |A_{12}| = (-1)^{1+2} |M_{12}|$

$$\begin{vmatrix}
a_{21} & a_{23} \\
a_{31} & a_{33}
\end{vmatrix} = -(a_{21}a_{33} - a_{31}a_{23})$$

ตัวอย่าง กำหนดให้

$$A = \begin{bmatrix}
2 & 0 & 1 \\
3 & 2 & -3 \\
-1 & -3 & 5
\end{bmatrix}$$

จงหา $|A|$
Using the first row as the pivot:

\[ |A| = \sum_{j=1} a_{1j} |A_{1j}| = a_{11} |A_{11}| + a_{12} |A_{12}| + a_{13} |A_{13}| \]

For the 1st cofactor:

\[ |A_{11}| = (-1)^{1+1} \begin{vmatrix} -3 & 5 \\ 1 & 3 \end{vmatrix} = 10 - 9 = 1 \]

\[ |A_{12}| = (-1)^{1+2} \begin{vmatrix} -1 & 5 \\ 3 & 2 \end{vmatrix} = -(15 - 3) = -12 \]

\[ |A_{13}| = (-1)^{1+3} \begin{vmatrix} -3 & 5 \\ -1 & -3 \end{vmatrix} = (-9 + 2) = -7 \]

Hence:

\[ |A| = 2(1) + 0(-12) + 1(-7) = -5 \]

Note: Will use rows 2, 3 or columns 1, 2, 3 as the pivot to get the same result.

2. **adjoint**

The adjoint of matrix \( A \) is denoted as \( \text{adj} \ A \). It is the transpose of the matrix of cofactors of the matrix elements \( a_{ij} \) of \( A \).

\[
\begin{bmatrix}
|A_{11}| & |A_{12}| & ... & |A_{1n}|
|A_{21}| & |A_{22}| & ... & |A_{2n}|
... & ... & ... & ...
|A_{n1}| & |A_{n2}| & ... & |A_{nn}|
\end{bmatrix}
\]

Hence: \( |A| \) is the cofactor of \( a_{ij} \)

3. **inverse of a square matrix**

\[
A^{-1} = \frac{1}{|A|} \text{adj} \ A
\]
ตัวอย่าง

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
2 & 3 & 4 \\
1 & 5 & 7
\end{bmatrix}
\]

จงหา

\[|A|\]
\[\text{adj } A\]
\[A^{-1}\]

วิธีทำ

\[|A| = \begin{vmatrix}
1 & 2 & 3 \\
2 & 3 & 4 \\
1 & 5 & 7
\end{vmatrix}
= 1(-1)^{1+1}(3 - 4) + 2(-1)^{1+2}(2 - 7) + 3(-1)^{1+3}(5 - 4)
= 1 - 20 + 21 = 2
\]

\[\text{adj } A = \begin{vmatrix}
(-1)^{1+1} (21-20) & (-1)^{2+1}(14-15) & (-1)^{3+1}(8-9) \\
(-1)^{1+2}(14-4) & (-1)^{2+2}(7-3) & (-1)^{3+2}(4-6) \\
(-1)^{1+3}(10-3) & (-1)^{2+3}(5-2) & (-1)^{3+3}(3-4)
\end{vmatrix}
= \begin{bmatrix}
1 & 1 & -1 \\
-10 & 4 & 2 \\
7 & -3 & -1
\end{bmatrix}
\]

\[A^{-1} = \frac{1}{|A|} \text{adj } A = \begin{bmatrix}
\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\
-5 & 2 & 1 \\
7/2 & -3/2 & -1/2
\end{bmatrix}
\]

หมายเหตุ ถ้า \(|A| = 0\) จะหา \(A^{-1}\) ไม่ได้ เรียก \(A\) ว่า singular matrix ถ้าหาได้ \(|A| \neq 0\) เรียก \(A\) ว่า non-singular matrix
ภาคผนวก ก
Vector Calculus

1. ระบบพิกัด (Coordinate Systems)

พิกัดฉาก (Rectangular Coordinates) แทนจุดด้วยระยะห่างจากแกน $x$ และ $y$

![Diagram of rectangular coordinates]

พิกัดเชิงขั้ว (Polar Coordinates) แทนจุดด้วยระยะห่าง $(r)$ และมุม $(\theta)$

![Diagram of polar coordinates]

ความสัมพันธ์ระหว่างพิกัดฉากกับพิกัดเชิงขั้ว

|x| = r \cos \theta
|y| = r \sin \theta

2. เวกเตอร์ (Vector)

เวลาเวกเตอร์ $\vec{A} = (a_1, a_2, a_3)$ สามารถเขียนแทนในระบบพิกัดฉากสามมิติได้ด้วยส่วนของเส้นตรง $PQ$ ที่มีทิศทางด้วยจุดเริ่มต้น $P(x, y, z)$ และจุดสิ้นสุด $Q(x+a_1, y+a_2, z+a_3)$ หรือส่วนของเส้นตรงที่มีจุดเริ่มต้นที่ $P$ และมีจุดสิ้นสุดที่ระยะห่างจากจุดเริ่มต้น ไปทางแกน $X$ ขนาด $a_1$, ไปทางแกน $Y$ ขนาด $a_2$ และไปทางแกน $Z$ ขนาด $a_3$ ตามลำดับ
หมายเหตุ ระบบพิกัด 3 มิติ แบบ Right Hand ในทางคณิตศาสตร์ มักจะเขียนแบบนี้

\[ \mathbf{PQ} = (x_2 - x_1, y_2 - y_1, z_2 - z_1) \]

หรือ  \[ \mathbf{P2P1} = (x_1 - x_2, y_1 - y_2, z_1 - z_2) = -(x_2 - x_1), -(y_2 - y_1), -(z_2 - z_1) \] นั่นคือ  \[ \mathbf{P1P2} = - \mathbf{P2P1} \]

หรือมีระยะห่างเท่ากันแต่ทิศทางตรงกันข้าม

ขนาดของเวกเตอร์ \( \mathbf{A} = (a_1, a_2, a_3) \) หมายถึง ความยาวของเวกเตอร์ เช่นที่ดังนี้

\[ |\mathbf{A}| = \sqrt{a_1^2 + a_2^2 + a_3^3} \]

เวกเตอร์ \( \mathbf{A} \) จะเป็นเวกเตอร์หนึ่งหน่วย (unit vector) ถ้า \( |\mathbf{A}| = 1 \) การทำเวกเตอร์ให้เป็นเวกเตอร์หนึ่งหน่วยโดยการหารเวกเตอร์ด้วยขนาดของเวกเตอร์ \( \mathbf{A} \) เพื่อว่าการทำ normalized

เวกเตอร์หนึ่งหน่วยชนิดพิเศษ ได้แก่ \( \mathbf{i} = (1, 0, 0), \mathbf{j} = (0, 1, 0) \) และ \( \mathbf{k} = (0, 0, 1) \) เป็นเวกเตอร์ที่สำคัญ เพราะเวกเตอร์ \( \mathbf{A} = (a_1, a_2, a_3) \) ใด ๆ สามารถเขียนในรูปของเวกเตอร์ข้างต้น (linear combination) จาก \( \mathbf{i}, \mathbf{j} \) และ \( \mathbf{k} \) ได้ ดังนี้ \( \mathbf{A} = (a_1, a_2, a_3) = a_1 \mathbf{i} + a_2 \mathbf{j} + a_3 \mathbf{k} \)
คุณสมบัติที่สำคัญ

\[ v_1 = (a_1, a_2, a_3) \quad v_2 = (b_1, b_2, b_3) \quad v_1 + v_2 = (a_1+b_1, a_2+b_2, a_3+b_3) \]

ผลคูณเชิงสเกลาร์ (scalar product or dot product or inner product) ของ \( A = (a_1, a_2, a_3) \) และ \( B = (b_1, b_2, b_3) \) ก้านต่อโดย

\[ A \cdot B = a_1b_1 + a_2b_2 + a_3b_3 \]

หรือ

\[ A \cdot B = |A||B| \cos \theta \]

\( \theta \) เป็นมุมระหว่างเวกเตอร์ \( A \) และ \( B \) ที่ไม่ใช่เวกเตอร์ 0

ตัวอย่างการนำไปใช้ประโยชน์ในทาง Computer Graphics คือ ใช้ตรวจสอบว่า polygon ที่มีหน้าหรือหน้าหลังให้ผู้มองดูที่ 6

ผลคูณเชิงเวกเตอร์ (vector product or cross product) ของเวกเตอร์ \( A = (a_1, a_2, a_3) \) และ \( B = (b_1, b_2, b_3) \)

\[ A \times B = \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \\ a_1 & a_2 \end{vmatrix} i - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \\ a_1 & a_2 \end{vmatrix} j + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \\ a_1 & a_2 \end{vmatrix} k \]

\[ |\begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \\ a_1 & a_2 \end{vmatrix}| \]

หมายถึงตัวกำหนด (determinant)

เพื่อช่วยให้ง่ายจะเขียนแทนดังนี้

\[ \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} \]

หมายถึงเวกเตอร์ \( A \times B \) จะเป็นเวกเตอร์ที่ตั้งฉากกับเวกเตอร์ \( A \) และเวกเตอร์ \( B \)

ตัวอย่างการนำไปใช้ประโยชน์ในทาง Computer Graphics คือ ใช้หา normal vector ของ ระนาบ (plane)

เช่น ถ้าจุด 3 จุด \( P_1, P_2 \) และ \( P_3 \) เป็นจุดบนระนาบเดียวกันและไม่เรียงเป็นเส้นตรงเดียวกัน (non colinear)

สามารถหา normal vector ที่ตั้งฉากกับระนาบดังกล่าวได้ดังนี้

\[ P_1P_2 \times P_1P_3 \]

\[ P_1P_3 \times P_1P_2 \]
หมายเหตุ การคำนวณหา $\mathbf{N}$ ที่จุดสามจุดบนระนาบเดียวกัน อาจจะได้ขนาดไม่เท่ากัน ถ้าต้องการให้มีขนาดเท่ากันต้องหารด้วยขนาดเพื่อให้เป็น unit vector.

ถ้า $\mathbf{PQ}$ และ $\mathbf{PR}$ มีจุดเริ่มต้นจุดเดียวกัน และ $\mathbf{S}$ เป็นภาพฉาย (projection) ของ $\mathbf{Q}$ บนส่วนเส้นตรงที่ผ่าน $\mathbf{P}$ และ $\mathbf{R}$ จะได้ว่า ขนาด $\|\mathbf{PS}\|$เท่ากับ $\|\mathbf{PQ}\| \cos \theta$.

Perspective of Planes

A plane is specified by a single equation of the form,

$A \cdot x + B \cdot y + C \cdot z + D = 0$

How to get coefficients of plane equation? Consider three points, $P_1$, $P_2$, and $P_3$, are distinct and noncollinear on the plane.

Find vectors $\mathbf{v}_1$ and $\mathbf{v}_2$:

$\mathbf{v}_1 = \mathbf{P}_2 - \mathbf{P}_1$ (from $x_1$, $y_1$, $z_1$ and $x_2$, $y_2$, $z_2$)

$\mathbf{v}_2 = \mathbf{P}_3 - \mathbf{P}_1$ (from $x_1$, $y_1$, $z_1$ and $x_3$, $y_3$, $z_3$)

Find cross product of $\mathbf{v}_2$ and $\mathbf{v}_1$

$\mathbf{N} = \mathbf{v}_2 \times \mathbf{v}_1$
\[
N = \mathbf{v}_2 \times \mathbf{v}_1 \text{ is a normal vector to the plane that contains } \mathbf{v}_1 \text{ and } \mathbf{v}_2.
\]

\[
\begin{vmatrix}
\mathbf{i} & \mathbf{j} & \mathbf{k} \\
v_2x & v_2y & v_2z \\
v_1x & v_1y & v_1z
\end{vmatrix}
= \mathbf{i} (v_2y \cdot v_1z - v_1y \cdot v_2z) + \mathbf{j} (v_2z \cdot v_1x - v_2x \cdot v_1z) + \mathbf{k} (v_1y \cdot v_2x - v_2y \cdot v_1x)
\]

Cross product is now \( \mathbf{N} = (v_2y \cdot v_1z - v_1y \cdot v_2z, v_2z \cdot v_1x - v_2x \cdot v_1z, v_1y \cdot v_2x - v_2y \cdot v_1x) \)

Let normal vector be \( \mathbf{N} = (N_x, N_y, N_z) \). So far, from three points, we have derived a vector perpendicular to a plane (plane normal vector). In plane equation, \( A \cdot x + B \cdot y + C \cdot z + D = 0 \),

\[
\begin{align*}
A &= N_x \\
B &= N_y \\
C &= N_z
\end{align*}
\]

Now, use any point on plane to solve for \( D \) coefficient, e.g. use \( P_1 = (x_1, y_1, z_1) \)

\[
D = -A \cdot x_1 - B \cdot y_1 - C \cdot z_1
\]

หมายเหตุ \( \mathbf{N} = \mathbf{v}_2 \times \mathbf{v}_1 \) จะตั้งฉากในแนวหัวแม่เมาถ้าเรากำมือขวาจากเวกเตอร์ \( \mathbf{v}_2 \)ไป \( \mathbf{v}_1 \)แต่ถ้าหา cross product of \( \mathbf{v}_1 \times \mathbf{v}_2 \)จะได้ \( \mathbf{N} \) ในทิศทางตรงข้าม (หรือ – \( \mathbf{N} \)) ดังภาพ

เอกสารอ้างอิง จันทนา ไอยรักกุล, "คณิตศาสตร์พื้นฐาน 2", ภ.คณิตศาสตร ม.สงขลานครินทร์