Binghamton University

EngiNet™

State University of New York

Thomas J. Watson

School of Engineering
and Applied Science

EngiNet™

WARNING

CS 560

Computer Graphics

Professor Richard Eckert

Lecture # 9

February 19, 2001

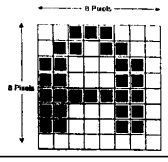
I. Scan Converting Text
II. Attributes
III. Area Fill

Text and Characters

- (See CS-460/560 Notes Web Page:
  - Week 5-A: Text and Fonts)
- Very important output primitive
- Many pictures require text
- Two general techniques used
  - Bitmapped
  - Stroked

Bitmapped Characters

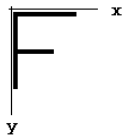
- Each character represented (stored) as a 2-D array
  - Each element corresponds to a pixel in a rectangular character cell
  - Simplest: each element is a bit (1=pixel on, 0=pixel off)



```
00111000
01101100
11000110
11000110
11111110
11000110
11000110
00000000
```

Stroked Characters

- Each character represented (stored) as a series of line segments
  - sometimes as more complex primitives
- Parameters needed to draw each stroke
  - endpoint coordinates for line segments




```
Strokes:
(0,0), (0,10)
(0,0), (10,0)
(0,5), (6,5)
```

Characteristics of Bitmapped Characters

- Each character in set requires same amount of memory to store
- Characters can only be scaled by integer scaling factors
- --> "Blocky" appearance
- Difficult to rotate characters by arbitrary angles
- Fast

Characteristics of Stroked Characters

- Number of stokes (storage space) depends on complexity of character
- Each stroke must be scan converted ==> more time to display
- Easily scaled and rotated arbitrarily
  - just transform each stroke

Example Character-Display Algorithms

- See CS-460/560 Notes Web Pages:
- Links to:
  - An illustration of how to display bitmapped characters
  - An illustration of how to display stroked characters

Algorithm for Bitmapped Characters--an Example

- 1. Define bitmap for the letter--e.g. 'T'

```
int t[7][7] = { {0,0,0,0,0,0,0}, {0,1,1,1,1,1,0},
  {0,0,0,1,0,0,0}, {0,0,0,1,0,0,0}, {0,0,0,1,0,0,0},
  {0,0,0,1,0,0,0}, {0,0,0,0,0,0,0}};  // bitmap for 'T'
```

  - [Could have a file with the bitmap descriptions of each character in the character set to be displayed]
  - Not the most efficient way of doing it
    - Could have used individual bits
    - Algorithm would be more complex

Bitmapped Character Alg., Continued

- 2. Define function to display bitmap letter[][] at pixel coordinates (x,y)

```
disp_letter (int x, int y, int letter[7][7])
{ int i,j;
  for (i=0; i<7; i++)
      for (j=0; j<7; j++)
          if (letter[i][j] == 1)
              Setpixel(x+j,y+i);  // plot from bitmap}
```

- 3. Call function, passing it desired bitmap

```
disp_letter (50,100,t);  // draw a 'T' at (50,100)
```

Algorithm for Stroked Characters

- 1. Define a character (CH) type

```
typedef struct tagCH
{ int n;
  POINT * pts; } CH;
```

- pts is an array of stroke endpoint vertices
- n is the number of vertices

Stroked Character Alg., Continued

- 2. Define generic display-character function
  - Strokes are specified in c (type CH)
  - Display at pixel coordinates (xx,yy):

```
disp_char (int xx, int yy, CH c)
{ int i,j;
  j=c.n/2;    /* n points ==> n/2 strokes */
  for (i=0; i<j; i++)
      line(xx+c.pts[2*i].x, yy+c.pts[2*i].y,
    xx+c.pts[2*i+1].x, yy+c.pts[2*i+1].y);
  }
```

Stroked Character Alg., Continued

- 3. Define the character's CH structure
- The following could be for an 'F':

```
POINT p[6];   CH f;
p[0].x=0;   p[0].y=0;   p[1].x=0;   p[1].y=10;
p[2].x=0;   p[2].y=0;   p[3].x=10;   p[3].y=0;
p[4].x=0;   p[4].y=5;   p[5].x=6;   p[5].y=5;
f.n = 6;   f.pts = p;
```

- [Descriptions of each character in the character set could be stored in a file]

Stroked Character Algorithm, Continued

- 4. Call the character-display function, passing it the desired character (CH)

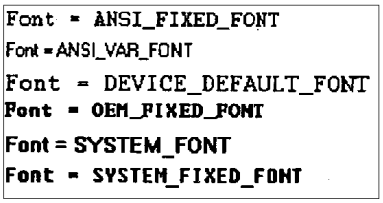
```
disp_char (50,100,f); // draw 'F' at (50,100)
```

Character Fonts in Windows

- FONT--Typeface, style, size of characters in a character set
- Three kinds of Windows Fonts
  - Stock Fonts
  - Logical or GDI Fonts
  - Device Fonts

Windows Stock Fonts

- Built into Windows
- Always available



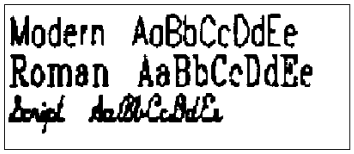
Windows Stock Fonts

Windows Logical or GDI Fonts

- Defined in separate font resource files on disk
  - .fon file
    - (Stroke or Raster)
  - .fot/.ttf file
    - (TrueType)
- Specific instance must be "created"

Windows Stroke Fonts

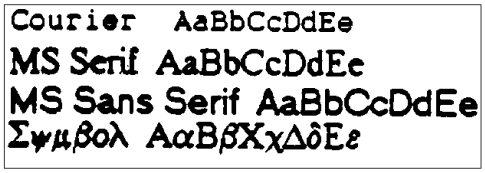
- Consist of line/curve segments
- Continuously scalable
- Slow to draw
- Legibility not too good



Windows Stroke Fonts

Windows Raster Fonts

- Bitmaps so:
  - Scaling by non-integer factors difficult
  - Fast to display
  - Legibility very good



Windows Raster Fonts

Windows TrueType Fonts

- Rasterized stroke fonts so:
  - Stored as strokes with hints to convert to bitmap
  - Conversion called rasterization
  - Continuously scalable
  - Fast to display
  - Legibility very good
  - Combine best of both stroke and raster fonts

Windows TrueType Fonts

Courier New AaBbCcDdEe
Courier New Bold AaBbCcDdEe
Courier New Italic AaBbCcDdEe
Courier New Bold Italic AaBbCcDdEe
Times New Roman AaBbCcDdEe
Times New Roman Bold AaBbCcDdEe
Times New Roman Italic AaBbCcDdEe
Times New Roman Bold Italic AaBbCcDdEe
Arial AaBbCcDdEe
Arial Bold AaBbCcDdEe
Arial Italic AaBbCcDdEe
Arial Bold Italic AaBbCcDdEe
Σψμβολ ΑαΒβΧχΔδΕε

Device Fonts

- Native to output device
- e.g., built-in printer fonts
  - Postscript

Using Windows Stock Fonts

- Like stock pens, brushes
- Accessed with:

  GetStockObject(font_name)
  - Returns a handle to a font
  - Use by selecting into DC with SelectObject():

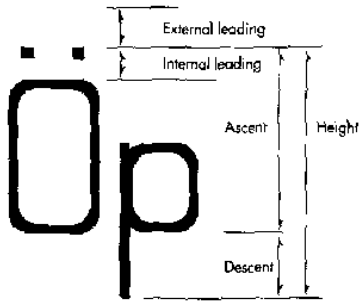
Using Windows Logical Fonts

- Instantiate a CFont object
- Use CFont::CreateFont(14 params!!)
  - Specify characteristics
  - Interpolates data from font file
  - --> new sizes, bold, rotated, etc.
- Select CFont object into the DC
- Called logical since determined by program logic not just file contents
- See online help

Windows Text Metrics

- CreateFont() may not give you exactly what you ask for
- Can use CDC::GetTextMetrics() to find out font details
- --> lots of information in a TEXTMETRIC structure
- Commonly used to determine font size
  - can be used to set line spacing, caret size, sizes of buttons, etc.

Windows Text Metrics



II. Attributes

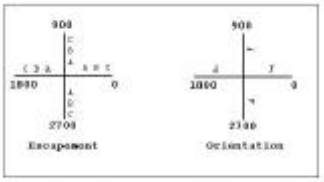
- How primitives are to be displayed
- Most systems use modal attributes
  - Values in effect until changed
  - Like a state machine

Text Attributes

- Font (typeface)
  - Character set with particular design style
- Display style
  - underlined, italic, boldface, outlined, strikeout, spacing, etc.
- Color
- Size (width, height)--specified in points
  - Point = 1/72 inch

Text Attributes, continued

- Orientation--how much character is rotated
- Escapement--orientation of line between first & last character in a string



Character Escapement & Orientation

Line Attributes

- Color
- Width
- Style--solid, dotted, dashed, etc.

  Can be specified by giving a pattern array

  e.g., pat[]={1,1,1,1,1,1,0,0}

  Repeat this pattern on entire line:

  i^{th} pixel along line:

  ```
  if (pat[i%8]==1) SetPixel(x,y)
  ```
- In Windows, use a pen (CPen)

III. Area Fill

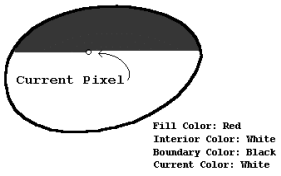
- Attributes:
  - fill color
  - fill pattern
- 2 Classes of area fill algorithms:
  - Boundary/Flood Fill Algorithms
  - Scanline Algorithms

Area Fill Algorithms

- See CS-460/560 Notes Web Page
- Link to:
  - Week 5-BC: Area Fill Algorithms
- URL:
  - http://www.cs.binghamton.edu/~reckert/460/fillalgs.htm

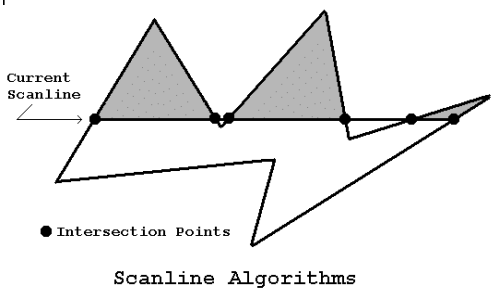
Boundary/Flood Fill Algorithms

- Determine which points are inside from pixel color information
  - e.g., interior color, boundary color, fill color, current pixel color
  - Color the ones that are inside.



Scanline Algorithms

- Examine horizontal scanlines spanning area
- Find intersection points between current scanline and borders
- Color pixels along the scanline between alternate pairs of intersection points
- Especially useful for filling polygons
  - polygon int. pt. calculations are very simple
  - Use vertical and horizontal coherence to get new intersection points from old rapidly



1. Connected Area Boundary Fill Algorithm

- For arbitrary closed areas
- Input:
  - Boundary Color (BC), Fill Color (FC)
  - (x,y) coords of seed point known to be inside
- Define a recursive BndFill(x,y,BC,FC) function:

  If pixel (x,y) not set to BC or FC, then set to FC
  Call BndFill() for neighboring points

- To be able to implement this, need an inquire function
- e.g., Windows GetPixel(x,y)
  - returns color of pixel at (x,y)

The BndFill() Function

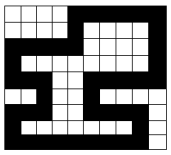
```
BndFill(x,y,BC,FC)
{
   color = GetPixel(x,y)
   if ( (color != BC) && (color != FC) )
   {
      SetPixel(x,y,FC);
      BndFill(x+1,y,BC,FC);   BndFill(x,y+1,BC,FC);
      BndFill(x-1,y,BC,FC);  BndFill(x,y-1,BC,FC);
   }
}
```

This would be called by code like:

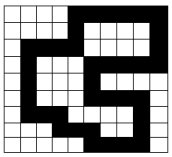
BndryFill(50,100,5,8); //5,8 are colors

- Windows GDI: colors are COLORREFs
- RGB() macro could be used

As given, only works with 4-connected regions



A 4-connected Region



An 8-connected Region

Flood Fill Algorithm

- A variation Boundary Fill
- Fill area identified by the interior color
  - instead of boundary color
- Good for single colored area with multicolor border

Ups/Downs of Bndry/Flood Fill

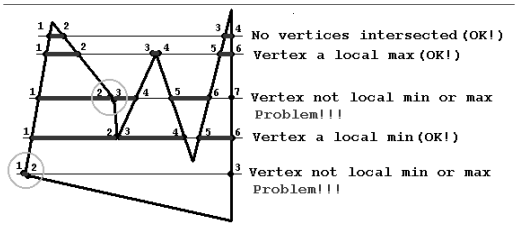
- Big Up: Can be used for arbitrary areas!
- BUT: Deep Recursion so:
  - Uses enormous amounts of stack space
    - (Adjust stack size before building in Windows!)
- Also very slow since:
  - Extensive pushing/popping of stack
  - Pixels may be visited more than once
  - GetPixel() & SetPixel() called for each pixel
    - 2 accesses to frame buffer for each pixel plotted

Scanline Polygon Fill Algorithm

- Look at individual scan lines
- Compute intersection points with polygon edges
- Fill between alternate pairs of intersection points

More specifically:

- For each scanline spanning the polygon:
  - Find int. pts. with all edges scanline cuts
  - Sort intersection points by increasing x
  - Turn on all pixels between alternate pairs of intersection points
- But--
  - Look at intersection points that are polygon vertices



Dealing With Vertex Intersection Points

Vertex intersection points that are not local max or min must be preprocessed!

Preprocessing non-max/min intersection points

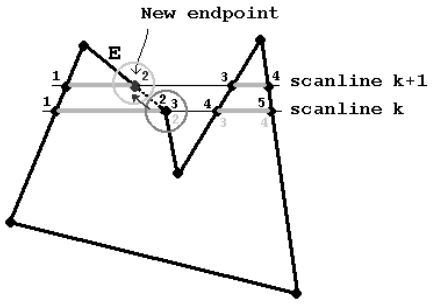
- Move lower endpoint of upper edge up by one pixel
- i.e., y <-- y + 1
- What about x?

  m = $\Delta y/\Delta x$, so $\Delta x = (1/m) * \Delta y$

  But $\Delta y = 1$, so:

  x <-- x + 1/m

Preprocessing



Preprocessing Edge E