Overview: A Graphics System

Raster Scan CRT:

- Electron gun
- Vertical deflection
- Interior metallic coating
- (at high positive voltage)
- Electron bean
- Phospher coating
- Control Grid
- Focusing System
  (Electron lens)
- Heating filament
- Cathode
**Control-grid voltage:** control the picture’s intensity

**Focusing system:** force the electrons to converge

**Deflection systems:** to trace a picture on screen (most crucial part of the CRT)

**Phospher:** when struck by electron beams, most of the "kinetic energy" carried by the electrons is transferred to the electrons of the phospher atoms, so the electrons of the phospher atoms jump to a higher quantum energy levels. These excited electrons return to their previous quantum levels by giving up their extra energy in the form of light at frequency depicted by the quantum theory
**Persistence:** the time from the removal of excitation to the moment when phosphersense decayed to 10% of the initial light output

**Refresh rate:** number of times per second a picture is redrawn (determined by persistence)

**Fusion frequency:** the refresh rate above which a picture stops flickering and fuses into a steady picture

**Note:**
1. Refresh rate for raster scan display is fixed (30 to 60), independent of the picture complexity

2. Highly dynamic applications need low-persistence phospher. CAD applications tend to use long-persistence phospher.

3. The relationship between fusion frequency and persistence is nonlinear.
Raster Scan Display processing Unit:

A simple two-color raster-scan system
Image Storage System (frame buffer, bitmap):

- refresh memory arranged as a 2D array; each entry corresponds to a screen pixel

- each entry is composed of a number of bits; brightness and/or color value of each pixel of the screen is stored in corresponding entry in frame buffer

- implemented with solid state RAM

Image Display System (video/image controller):

- cycle through frame buffer row by row, 30 or 60 times/sec

- memory reference addresses are generated in synchronism with the raster scan; contents of the memory are used to control CRT beam’s intensity

- changes in frame buffer is done during the 1.3 millisecond fly-back (or, vertical retrace) time

- interlaced raster scan (to produce a picture whose effective refresh rate is closer to 60 than to 30 Hz.

Image Creation System:

- scan convert abstract representation of an image into appropriate pixel values in the frame buffer
Shadow Mask Color CRT

- phospher dots (red, green, blue) are arranged in triangular pattern called triad (or, pixel)

- three electron guns are used

- a shadow mask, behind the view surface, is equipped so that each small hole for each triad (holes are aligned so that each electron gun excites its corresponding phospher dot)

- resolution of these CTRs is limited
  (high resolution: triads are on about .21mm centers)
  (home TV: triads are on about .60mm centers)
- each number stored in frame buffer is an index (address) into a lookup table (color table or color map)

- Lookup table provides significant saving on memory while gives the ability to change colors from picture to picture
2D Graphics

- How to paint on screen
  - Device-dependent approach
  - Device-independent approach
- Rendering pipeline
- Graphical system notions
  - Coordinate systems
  - Primitives: points, lines, polygons, shapes, characters,
  - Attributes: colors, fonts, styles, thickness, etc
How to paint on screen?

Issues:

- Representation techniques
- Mapping techniques (transformations)
- Clipping
- Scan conversion (rendering, rasterization)
- Attribute setting
What is 2D Graphics?

• Representation and manipulation of 2D shapes in the computer

Representations

• How should a **polygon** be represented?

• How should a **curve** be represented?

• How should a **scene** be represented?

Manipulation

• How should an object be **transformed**?

• How should a scene be **mapped**?
Device-dependent approach

- working with the device coordinate system (DCS)
Device-independent approach

- working with the world coordinate system (WCS)

- Clipping and mapping are not the responsibility of application programmer
Device coordinate system

- (integer) coordinate system of the screen
- the application programmer has to define the display area (viewport) of the screen

World coordinate system

- (floating-point) working coordinate system of the user
- the application programmer has to define the area of the scene (window) to be displayed
- the application programmer has to define the display area (viewport) of the screen
Java’s graphics capabilities:

• A graphics object must be used to draw within the paint method of the Component class

```java
public void paint ( Graphics g ) {
    //g.drawLine(x1, y1, x2, y2);
    g.drawString( "Hello, World." , 25, 25 );
    .
    .
}
```

• paint is an Abstract method. It must be overridden.

• For paint to be called again (using repaint), an event must occur.
• **repaint** is not an Abstract method. It should not be overridden (it performs system-dependent tasks).

• **graphics** is an Abstract class. The **graphics** object \( g \) in the header is an object of the system’s derived **graphics** class.
Drawing Strings, Characters, ...

g.drawString("Hello", x, y )

• How to erase a string?
Color Control

The **Color** class

Instance variables: (public final static **Color**)

- red (255, 0, 0)
- green (0, 255, 0)
- blue (0, 0, 255)
- black (0, 0, 0)
- darkGray (64, 64, 64)
- gray (128, 128, 128)
- lightGray (192, 192, 192)
- white (255, 255, 255)
- cyan (0, 255, 255)
- magenta (255, 0, 255)
- orange (255, 200, 0)
- pink (255, 175, 175)
- yellow (255, 255, 0)
**Color** methods:

- public Color( int r, int g, int b )
- public Color( float r, float g, float b )
- public Color( int rgb )
- public int getRed()
- public int getGreen()
- public int getBlue()

**Graphics** methods:

- public abstract Color getColor()
- public abstract void setColor( Color c )

**Component (Applet)** methods:

- public void setBackground( Color c )
- public Color getBackground( )
- public void setForeground( Color c )
- public Color getForeground( )
Drawing Lines

g.drawLine( x1, y1, x2, y2 )

- How to draw a point?
- How to erase a line?
- How to draw a curve?
To erase a line:
• Redraw the line in background color

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Line extends Applet {
    private Color c1, c2;

    public void init () {
        addMouseListener( new DetailHandler( this ) );
    }

    public void paint( Graphics g ) {
        g.drawLine( 10, 10, 230, 95 );
    }

    public void eraseLine( MouseEvent e ) {
        c1 = getForeground();
        setForeground( c2 = getBackground() );
        repaint();
        //setForeground( c1 = Color.red );
        //repaint();
    }
}
```
class DetailHandler extends MouseAdapter {
    private Line l;

    public DetailHandler( Line line ) { l = line; }

    public void mouseClicked( MouseEvent e )
    {
        l.eraseLine( e );
    }
}

- 220 -
To draw a curve:

- Compute a set of discrete points of the curve and then draw the line segments that connect these points.

Example: draw a Bezier curve of degree 3
// Demonstrating curve drawing
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class BezierCurve extends Applet {
    private int count;
    private int px[], py[];
    private Color c1, c2;
    private final int no_points=20;
    private int cx, cy, nx, ny;

    public void init ( )
    {
        px = new int[ 4 ];
        py = new int[ 4 ];
        addMouseListener( new DetailHandler( this ) );
    }

    public void paint( Graphics g )
    {
        if ( count >= 4 ){
            for (int i=0; i < count; i++) { // draw control points
                g.drawLine( px[ i ]-3, py[ i ], px[ i ]+3, py[ i ]);
                g.drawLine( px[ i ], py[ i ]-3, px[ i ], py[ i ]+3 );
            }
        }
    }
}
g.setColor( Color.red );
//setForeground( c1 );
for (int i=0; i < count-1; i++) { // draw control polygon
    g.drawLine( px[ i ], py[ i ], px[ i+1 ], py[ i+1 ]);
}
cx = px[ 0 ];  cy = py[ 0 ];
for (int i=1; i < no_points; i++ ) { // draw the curve
    nx = nextPoint( px, i, no_points );
    ny = nextPoint( py, i, no_points );
    g.drawLine( cx, cy, nx, ny );
cx = nx;  cy = ny;
}
}
else { // prompt the user
    g.drawString( "Click mouse to input control point ", 10, 20 );
    if ( count > 0 )
        for (int i=0; i < count; i++) { // show the control points
            g.drawLine( px[ i ]-3, py[ i ], px[ i ]+3, py[ i ]);
            g.drawLine( px[ i ], py[ i ]-3, px[ i ], py[ i ]+3 );
        }
}
}

public int getCount() { return count; }
public void update( MouseEvent e )
{
    count += 1;
    px[ count - 1 ] = e.getX();
    py[ count - 1 ] = e.getY();
}

public void drawControlPoint() { repaint(); }  

public int nextPoint( int a[], int j, int k )
{
    double v, t;
    double t = (double) j/k ;
    v = (double) (1-t)*(1-t)*(1-t)*a[0] + 3*(1-t)*(1-t)*t*a[1]
    +3*(1-t)*t*t*a[2] + t*t*t*a[3];
    return (int) v;
}
class DetailHandler extends MouseAdapter {
    private BezierCurve c;

    public DetailHandler( BezierCurve curve ) { c = curve; }

    public void mouseClicked( MouseEvent e )
    {
        c.update( e );
        c.drawControlPoint();
        if ( c.getCount() == 4 )
            c.repaint();
    }
}
How is a line drawn?

- Need an algorithm that can rapidly generate the pixels comprising the line in a very accurate way

**Midpoint (Line) Algorithm**
Intuitive Approach
• For each vertical line, find the intersection point and then round to the nearest pixel

Midpoint (Line) Algorithm
• An incremental method
• Use only integer arithmetic
Assumption

• Slope of the line to be drawn is 0 ≤ m ≤ 1

• \( x_{\text{start}} < x_{\text{end}}; \quad y_{\text{start}} < y_{\text{end}} \)

Basic idea

• At each step, decide which pixel (E or NE) is closest to the line. Choose that pixel and draw it.

• Strategy: if M lies above the line, choose E. Otherwise, choose NE.
How to make efficient decision?

Implicit line equation: \( F(x, y) \equiv ax + by + c = 0 \)

\[
a = \Delta y \quad (\Delta y \equiv y_{end} - y_{start})
\]

\[
b = -\Delta x \quad (\Delta x \equiv x_{end} - x_{start})
\]

\[
c = B \cdot \Delta x \quad (B: y \text{ intercept})
\]

Note that

\[
F(x, y) = 0 \quad \text{for points on the line}
\]

\[
F(x, y) > 0 \quad \text{for points below the line}
\]

\[
F(x, y) < 0 \quad \text{for points above the line}
\]
Let \((x_{i-1}, y_{i-1})\) be the pixel selected for the line \(x_{i-1}\).

Consider the sign of \(F(x, y)\) at \(M\)

\[
s_i \equiv F(M) = F(x_i, y_{i-1} + \frac{1}{2})
\]

Then the decision for the line \(x_i\) can be made as follows

\[
\begin{align*}
NE_i & \text{ is selected if } s_i \geq 0 \\
E_i & \text{ is selected if } s_i < 0
\end{align*}
\]
To avoid floating-point arithmetic, we consider the sign of

\[ S_i \equiv 2F(M) = 2F(x_i, y_{i-1} + \frac{1}{2}) \]

\[ = 2 \cdot \Delta y \cdot x_i - \Delta x \cdot (2y_{i-1} + 1) + 2B \cdot \Delta x \]

To compute \( S_i \), note that

\[ S_{i+1} = 2F(x_{i+1}, y_i + \frac{1}{2}) \]

\[ = 2 \cdot \Delta y \cdot x_{i+1} - \Delta x \cdot (2y_i + 1) + 2B \cdot \Delta x \]

Therefore

\[ S_{i+1} - S_i = 2\Delta y - 2\Delta x(y_i - y_{i-1}) \]
Since

\[ y_i = \begin{cases} y_{i-1} + 1, & \text{if } S_i \geq 0 \\ y_{i-1}, & \text{if } S_i < 0 \end{cases} \]

we have the following recurrence formula

\[ S_{i+1} = \begin{cases} S_i + 2(\Delta y - \Delta x), & S_i \geq 0 \\ S_i + 2\Delta y, & S_i < 0 \end{cases} \]

with the initial condition

\[ S_1 = 2\Delta y - \Delta x \]
Example:

Which pixel should be plotted for the line $x = 2$, $x = 6$?

1. Plot $(x_0, y_0) = (x_{start}, y_{start}) = (0, 0)$
   
   Compute $S_1 = 2\Delta y - \Delta x = 5$

2. Since $S_1 > 0$, plot $NE_1 = (1, 1)$ and
   
   compute $S_2 = S_1 + 2(\Delta y - \Delta x) = 5 - 6 = -1$

3. Since $S_2 < 0$, plot $E_2 = (2, 1)$ for $x = 2$ and ...
A **polygon** is a piecewise-linear, closed curve.

The **edges** of a polygon could intersect themselves.

The **interior** of a polygon could contain holes.

But how is the **interior** of a polygon defined?
Polygon Representation

- Vertex-based approach
- Edge-based approach

Vertex-based approach

A polygon is represented as an ordered array of vertex coordinates.

\[ v_i = (x_i, y_i) \]

To transform a polygon, simply apply the transformation to all the vertices of the polygon.
**Edge-based approach**

A polygon is represented as an ordered array of edges

\[ v_i = (x_i, y_i) \]
Comparison

• **Space:**
  - Edge-based approach requires more space and bookkeeping
  - However, if a set of polygons with many shared vertices is considered, edge-based approach actually isn’t that bad

• **Speed:**
  - Edge-based approach can answer adjacency queries very quickly.

• **Representation:**
  - Edge-based approach allows information about edges to be stored in edge nodes, which are explicit structures
How to fill a polygon?

- identify and paint all the pixels on or inside a polygon
Basic idea:

1. **Compute** $x$ coordinates of intersection points of current scan line with all edges
2. **Sort** intersection points by increasing $x$ values
3. **Group** intersection points by pairs
4. **Fill in** the pixels between each pair of intersection points on the current scan line
Two potential problems:

1. When a scan line intersects a vertex of the polygon which is not a local minimum or a local minimum, an odd number of intersection points would be reported.

   **Remedy:** shorten the higher adjacent edge by 1 in y direction.

2. What if an edge of the polygon is horizontal?

   **Remedy:** ignore horizontal edges
The Scan-Conversion Process:

Creating **bucket-sorted edge table (ET)** first:

- To determine which edges intersect current scan line
- To efficiently compute intersection points of these edges with the current scan line
- Vertical edges need to be shortened by 1 in y direction

**Example:**

![Diagram showing the scan-converted process](image)
Also need to maintain an **active-edge table (AET)**

**Purpose:** keep track of the edges the current scan line intersects

**How:** when we move to a new scan line (bottom to top), new edges intersecting the new scan line are added into the AET, edges in AET which are no longer active (not intersected by the new scan line) are deleted

**Example:**

In the previous example, when \( y = 2 \),

\[
AET \rightarrow
\]

when \( y = 4 \),

\[
AET \rightarrow
\]

when \( y = 8 \),

\[
AET \rightarrow
\]
Algorithms:

1. Set $y$ to the $y$-coordinate of the first non-empty bucket
2. Set AET to empty
3. **Pepeat until** the AET and ET are both empty
   3.1 **Merge** edges from ET bucket $y$ with edges in AET, maintaining AET sort order on $x$
   3.2 **Fill in** pixels on scan line $y$ bounded by pairs of $x$-coordinates from edges in AET
   3.3 **Remove** from AET those edges for which $y = y_{top}$
   3.4 For each edge remaining in AET, **replace** $x$ with $x + 1/m$
   3.5 **Increment** $y$ by 1, to the coordinate of the next scan line
Drawing and filling polygons in Java:

```java
public abstract void drawPolygon( // Graphics class
    int xPoints[ ],
    int yPoints[ ],
    int points )

public abstract void drawPolygon( // Graphics class
    Polygon p )

public abstract void fillPolygon( // Graphics class
    int xPoints[ ],
    int yPoints[ ],
    int points )

public abstract void fillPolygon( // Graphics class
    Polygon p )

To construct a polygon object:

public Polygon( ) // Graphics class

public Polygon( // Graphics class
    int xValues[ ],
    int yValues[ ],
    int numberOfPoints )
```