Computer Graphics in Java

Shannon Hale

COMPUTER GRAPHICS IN JAVA

COMPUTER GRAPHICS IN JAVA

Shannon Hale

BIBLIOTEX
Digital Library

Computer Graphics in Java
by Shannon Hale

www.bibliotex.com

Ebook ISBN: 9781984664068

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Introduction to Computer Graphics in Java

Although computer graphics is a vast field that encompasses almost any graphical aspect, we are mainly interested in the generation of images of 3-dimensional scenes. Computer imagery has applications for film special effects, simulation and training, games, medical imagery, flying logos, etc. Computer graphics relies on an internal model of the scene, that is, a mathematical representation suitable for graphical computations. The model describes the 3D shapes, layout and materials of the scene.

This 3D representation then has to be projected to compute a 2D image from a given viewpoint, this is the rendering step. Rendering involves projecting the objects (perspective), handling visibility (which parts of objects are hidden) and computing their appearance and lighting interactions. Finally, for animated sequence, the motion of objects has to be specified. We will not discuss animation in this document.

The Evolution of Computer Graphics

CGI was first used in movies in 1973, in the science fiction film, *Westworld*. The film was the story of a society in which humans and robots were integrated, working and living together. Its sequel, *Futureworld* (1976) featured the first use of 3D wireframe imagery. The third film ever to use this technology was *Star Wars* (1977), designing the Death Star and the targeting computers in the X-wings and the Millenium Falcon, Han Solo's ship. Later on, *The Black Hole* (1979) used raster wire-frame model rendering to create a black hole onscreen. That same year, James Cameron's *Alien* used the raster wireframe model to render the image of navigation monitors in the scene where the spaceship follows a beacon for landing guidance.

Long before this, computer engineers at MIT and Cornell were in the midst of creating the very basics that eventually enabled these filmmakers to utilize computer animation technology. It all began in 1963.

1960s

- 1963: Ivan Sutherland presented his Ph. D. dissertation, an interactive design on a vector graphics display monitor with a light pen input device called *Sketchpad*. This instance is often credited as the event that marks the beginning of computer graphics.
- Jack Bresenham develops a system of drawing lines and circles on a raster device, and Steve Coons introduces parametric surfaces and computer-aided geometric design concepts.
- Arthur Appel at IBM introduces hidden surface and shadow algorithms.

- The fast Fourier transform was discovered by J. W. Cooley and John Tukey, allowing computer engineers to better understand signals to develop antialiasing techniques.
- Doug Englebart develops the mouse at Xerox PARC.
- Evans & Sutherland Corps. and GE start building flight simulators with raster graphics.

1970s

- Rendering and a reflection model were discovered and developed by H. Gouraud and Bui Tuong Phong at the University of Utah.
- Xerox PARC develops a "paint programme. "
- Edward Catmull introduces parametric patch rendering, the z-buffer algorithm and texture mapping.
- Turner Whitted develops recursive ray tracing that would become the standard for photorealism.
- Apple I and Apple II computers were the first commercially successful options for personal computing.
- Arcade games Pong and Pac Man become popular.

1980s

- Microprocessors begin to take off but remain in early stages of development.
- Loren Carpenter begins exploring fractals in computer graphics.
- Adobe formed by John Warnock, who discovers Postscript. Adobe markets Photoshop.
- Steve Cook introduces stochastic sampling.
- Character animation becomes a goal for animators.

- Video arcade games take off.
- C++, C, and MS-DOS programming gain popularity.

1990s

- Shaded raster graphics appear in films.
- Computers have 24-bit raster display and hardware support for Gouraud shading.
- Laser printers and single-frame video recorders become standard.
- Mosaic, the first graphical internet browser is created.
- Dynamical systems that allowed programmers to animate collisions, friction and cause and effects are introduced.
- Handheld computers are invented at Hewlett-Packard and zip drives invented at Iomega.
- Nintendo 64 game console arrives on the market.
- Linux and open source software emerges.
- Pixar is first studio to fully embrace an entirely computer-generated film with *Toy Story.*

2000s

- Graphic software reaches a peak in quality and user accessibility.
- PC displays support real-time texture mapping.
- Flatbed scanners, laser printers, digital video cameras, etc. , become commonplace.
- Programme language moves towards Java and C++.
- 3D modeling captures facial expressions, human face, hair, water, and other elements formerly difficult to render.

Displaying graphics on a component

Now that you have a Canvas (an area to display graphics on) how do you actually display those graphics? With the paint() method of the Frame class. The paint() method takes one attribute-an instance of the Graphics class. The Graphics class contain methods which are used for displaying graphics. The Graphics class lets a component draw on itself.

Syntax:

public void paint(Graphics g){//methods for drawing graphics here; }

Drawing Lines

To draw lines, the drawLine() method of the Graphics class is used. This method takes four numeric attributes-the first two indicating the x/y starting point of the line, the last two indicating the x/y ending point of the line.

Example:

public void paint(Graphics g){//draw a line starting at point 10, 10 and ending at point 50, 50. g. drawLine(10, 10, 50, 50); }

Drawing Rectangles

To draw rectangles, the drawRect() method is used. This method takes four numeric attributes-the first two indicating the x/y starting point of the rectangle, the last two indicating the width and height of the rectangle.

Example:

Public void paint(Graphics g){//draw a rectangle starting at 100, 100 width a width and height of 80 g. drawRect(100, 100, 80, 80); }.

Filling a Rectangle

By default a rectangle will have no colour on the inside (it will just look like a box). You can use the fillRect() method to fill a rectangle. The fillRect() method has four numeric attributes indicating the x/y starting position to begin filling and the height and width. Set these values the same as you did for the drawRect() method to properly fill the rectangle.

Example:

Public void paint(Graphics g){//draw a rectangle starting at 100, 100 width a width and height of 80 g. drawRect(100, 100, 80, 80); g. fillRect(100, 100, 80, 80); }.

Something's Missing. . .

The rectangle is filled, but we didn't set a colour for it! To do this, we will use the setColor() method.

g. setColor(Colour. orange);

Drawing Ovals

To draw ovals, the drawOval() method is used. This method takes four numeric attributes-the first two indicating the x/y starting point of the oval, the last two indicating the width and height of the oval. Fill an oval with the fillOval() method which also takes four numeric attributes indicating the starting position to begin filling and the height and width. Set these values the same as you did for the drawOval() method to properly fill the oval.

Example:

public void paint(Graphics g){ g. setColor(Colour. gray);// draw an oval starting at 20, 20 with a width and height of 100 and fill it g. drawOval(20, 20, 100, 100); g. fillOval(20, 20, 100, 100); }

Displaying Images

To display images, the Image class is used together with the Toolkit class. Use these classes to get the image to display. Use the drawImage() method to display the image.

Example:

public void paint(Graphics g){ Image img1 = Toolkit. get Default Toolkit(). getImage("sky. jpg");//four attributes: the image, x/y position, an image observer g. drawImage(img1, 10, 10, this); }

An entire Java graphics programme:

import java. awt. *; class GraphicsProgram extends Canvas{ public GraphicsProgram(){ setSize(200, 200); setBackground(Colour. white); } public static void main(String[] argS){//GraphicsProgram class is now a type of canvas//since, it extends the Canvas class//lets instantiate it GraphicsProgram GP = new GraphicsProgram();//create a new frame to which we will add a canvas Frame aFrame = new Frame(); aFrame. setSize(300, 300);//add the canvas aFrame. add(GP); aFrame. setVisible(true); } public void paint(Graphics g){ g. setColor(Colour. blue); g. drawLine(30, 30, 80, 80); g. drawRect(20, 150, 100, 100); g. fillRect(20, 150, 100, 100); g. fillOval(150, 20, 100, 100); Image img1 = Toolkit. getDefaultToolkit(). getImage("sky. jpg"); g. drawImage(img1, 140, 140, this); } }

What it will look like:

Overview of the Java 2D API Concepts

The Java 2D™ API provides two-dimensional graphics, text, and imaging capabilities for Java™ programmes through extensions to the Abstract Windowing Toolkit (AWT).

This comprehensive rendering package supports line art, text, and images in a flexible, full-featured framework for developing richer user interfaces, sophisticated drawing programmes, and image editors. Java 2D objects exist on a plane called user coordinate space, or just *user space*. When objects are rendered on a screen or a printer, user space coordinates are transformed to *device space coordinates*.

The following links are useful to start learning about the Java 2D API:

- Graphics class.
- Graphics2D class.

The Java 2D API provides following capabilities:

- A uniform rendering model for display devices and printers.
- A wide range of geometric primitives, such as curves, rectangles, and ellipses, as well as a mechanism for rendering virtually any geometric shape.
- Mechanisms for performing hit detection on shapes, text, and images.
- A compositing model that provides control over how overlapping objects are rendered.
- Enhanced colour support that facilitates color management.
- Support for printing complex documents.
- Control of the quality of the rendering through the use of rendering hints.

Introduction to Java2D

In Java 1. 2, the paintComponent method is supplied with a Graphics2D object (a subclass of Graphics), which contains a much richer set of drawing operations. It includes pen widths, dashed lines, image and gradient colour fill patterns, the use of arbitrary local fonts, a floating point coordinate system, and a number of coordinate transformation operations. However, to maintain compatibility with Swing as used in Java 1. 1, the declared type of the paintComponent argument is Graphics, so you have to cast it to Graphics2D before using it.

Java 1. 1

```
public void paint(Graphics g) {
// Set pen parameters
g. setColor(someColor);
g. setFont(someLimitedFont);
// Draw a shape
g. drawString(. . . );
g. drawLine(. . . )
g. drawRect(. . . ); // outline
g. fillRect(. . . ); // solid
g. drawPolygon(. . . ); // outline
g. fillPolygon(. . . ); // solid
g. drawOval(. . . ); // outline
g. fillOval(. . . ); // solid
. . .
}
```

Java 1. 2

```
public void paintComponent(Graphics g) {
// Clear off-screen bitmap
super. paintComponent(g);
// Cast Graphics to Graphics2D
Graphics2D g2d = (Graphics2D)g;
// Set pen parameters
g2d. setPaint(fillColorOrPattern);
g2d. setStroke(penThicknessOrPattern);
g2d. setComposite(someAlphaComposite);
g2d. setFont(anyFont);
g2d. translate(. . . );
```

```
g2d. rotate(. . . );
g2d. scale(. . . );
g2d. shear(. . . );
g2d. setTransform(someAffineTransform);
// Allocate a shape
SomeShape s = new SomeShape(. . . );
// Draw shape
g2d. draw(s); // outline
g2d. fill(s); // solid
}
```

Main New Features

- Colours and patterns: gradient fills, fill patterns from tiled images, transparency.
- Local fonts.
- Pen thicknesses, dashing patterns, and segment connection styles.
- Coordinate transformations.

General Approach

- Cast the Graphics object to a Graphics2D object.

```
public void paintComponent(Graphics g) {.
  super. paintComponent(g); // Typical Swing approach.
  Graphics2D g2d = (Graphics2D)g;.
  g2d. doSomeStuff(. . . );.
  . . .
}
```

- Create a Shape object

```
Rectangle2D. Double rect =. . . ;
Ellipse2D. Double ellipse =. . . ;
Polygon poly =. . . ;
GeneralPath path =. . . ;
SomeShapeYouDefined shape =. . . ;// Satisfies Shape
interface.
. . .
```

- Optional: modify drawing parameters.

```
g2d. setPaint(fillColorOrPattern);
g2d. setStroke(penThicknessOrPattern);
```

```
g2d. setComposite(someAlphaComposite);
g2d. setFont(someFont);
g2d. translate(. . . );
g2d. rotate(. . . );
g2d. scale(. . . );
g2d. shear(. . . );
g2d. setTransform(someAffineTransform);
```

- Draw an outlined or solid version of the Shape

```
g2d. draw(someShape);
g2d. fill(someShape);
```

Drawing Shapes in Java2D

Drawing Shapes: Overview

With the AWT, you generally drew a shape by calling the drawXxx or fillXxx method of the Graphics object. In Java2D, you generally create a Shape object, then call either the draw or fill method of the Graphics2D object, supplying the Shape object as an argument.

For example:

```
public void paintComponent(Graphics g) {
super. paintComponent(g);
Graphics2D g2d = (Graphics2D)g;
// Assume x, y, and diameter are instance variables
Ellipse2D. Double circle =
new Ellipse2D. double(x, y, diameter, diameter);
g2d. fill(circle);
. . .
}
```

You can still call the drawXxx methods if you like, however. This is necessary for drawString and drawImage, and possibly convenient for draw3DRect. Several classes have similar versions that store coordinates as either double precision numbers (Xxx. Double) or single precision numbers (Xxx. Float). The idea is that single precision coordinates might be slightly faster to manipulate on some platforms.

Shape Classes

Arguments to the Graphics2D draw and fill methods must implement the Shape interface. You can create your own shapes, of course, but following are the major built-in ones.

Except for Rectangle and Polygon, which are Java 1. 1 holdovers, these appear in the java. awt. geom package.

- Arc2D. Double, Arc2D. Float.
- Area (a shape built by adding/subtracting other shapes).
- CubicCurve2D. Double, CubicCurve2D. Float.
- Ellipse2D. Double, Ellipse2D. Float.
- GeneralPath (a series of connected shapes).
- Line2D. Double, Line2D. Float.
- Polygon.
- QuadCurve2D. Double, QuadCurve2D. Float.
- Rectangle2D. Double, Rectangle2D. Float, Rectangle.
- RoundRectangle2D. Double, RoundRectangle2D. Float.

Drawing Shapes: Example Code

ShapeExample. java

```
import javax. swing. *;// For JPanel, etc.
import java. awt. *; // For Graphics, etc.
import java. awt. geom. *; // For Ellipse2D, etc.
/** An example of drawing/filling shapes with Java2D in
Java 1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class ShapeExample extends JPanel {
private Ellipse2D. Double circle =
new Ellipse2D. Double(10, 10, 350, 350);
```

```
private Rectangle2D. Double square =
new Rectangle2D. Double(10, 10, 350, 350);
public void paintComponent(Graphics g) {
clear(g);
Graphics2D g2d = (Graphics2D)g;
g2d. fill(circle);
g2d. draw(square);
}
// super. paintComponent clears offscreen pixmap,
// since, we're using double buffering by default.
protected void clear(Graphics g) {
super. paintComponent(g);
}
protected Ellipse2D. Double getCircle() {
return(circle);
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new ShapeExample(), 380,
400);
}
}
```

WindowUtilities. java

```
import javax. swing. *;
import java. awt. *;
/** A few utilities that simplify testing of windows in
Swing.
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class WindowUtilities {
/** Tell system to use native look and feel, as in previous
* releases. Metal (Java) LAF is the default otherwise.
*/
public static void setNativeLookAndFeel() {
try {
UIManager.            setLookAndFeel(UIManager.
getSystemLookAndFeelClassName());
} catch(Exception e) {
System. out. println("Error setting native LAF: " + e);
}
}
/** A simplified way to see a JPanel or other Container.
* Pops up a JFrame with specified Container as the content
pane.
*/
```

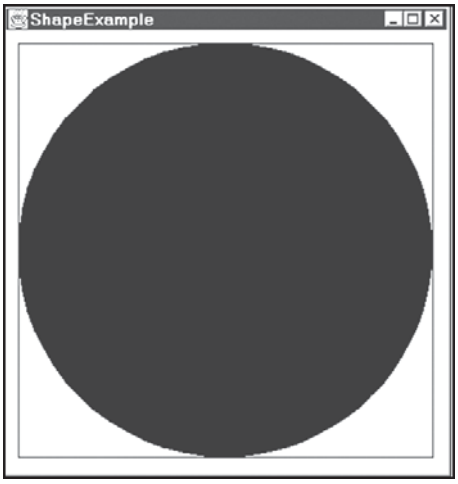
```
public static JFrame openInJFrame(Container content,
int width,
int height,
String title,
Colour bgColor) {
JFrame frame = new JFrame(title);
frame. setBackground(bgColor);
content. setBackground(bgColor);
frame. setSize(width, height);
frame. setContentPane(content);
frame. addWindowListener(new ExitListener());
frame. setVisible(true);
return(frame);
}
/** Uses Colour. white as the background colour. */
public static JFrame openInJFrame(Container content,
int width,
int height,
String title) {
return(openInJFrame(content, width, height, title, Colour.
white));
}
/** Uses Colour. white as the background colour, and the
* name of the Container's class as the JFrame title.
*/
public static JFrame openInJFrame(Container content,
int width,
int height) {
return(openInJFrame(content, width, height,
content. getClass(). getName(),
Colour. white));
}
}
```

ExitListener. java

```
import java. awt. *;
import java. awt. event. *;
/** A listener that you attach to the top-level Frame or
JFrame of
* your application, so quitting the frame exits the
application.
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class ExitListener extends WindowAdapter {
public void windowClosing(WindowEvent event) {
```

```
System. exit(0);
}
}
```

Drawing Shapes: Example Output



Paint Styles in Java2D

Paint Styles: Overview

When you fill a Shape, the current Paint attribute of the Graphics2D object is used. This can be a Colour (solid colour), a GradientPaint (gradient fill gradually combining two colours), a TexturePaint (tiled image), or a new version of Paint that you write yourself. Use setPaint and getPaint to change and retrieve the Paint settings. Note that setPaint and getPaint supersede the setColor and getColor methods that were used in Graphics.

Paint Classes

Arguments to the Graphics2D setPaint method (and return values of getPaint) must implement the Paint interface.

Here are the major built-in Paint classes:

- *Colour:* Has the same constants (Colour. red, Colour. yellow, etc.) as the AWT version, plus some extra constructors ·

- *GradientPaint:* Constructorstakes two points, two colours, and optionally a boolean flag that indicates that the colour pattern should cycle. The first colour is used at the first point, the second colour at the second point, and points in between are colored based on how close they are to each of the points.
- *TexturePaint:* Constructor takes a BufferedImage and a Rectangle2D, maps the image to the rectangle, then tiles the rectangle. Creating a BufferedImage from a GIF or JPEG file is a pain. First load an Image normally, get its size, create a BufferedImage that size with BufferedImage. TYPE_INT_ARGB as the image type, get the BufferedImage's Graphics object via createGraphics, then draw the Image into the BufferedImage using drawImage. An example of this process is shown later.

Transparency

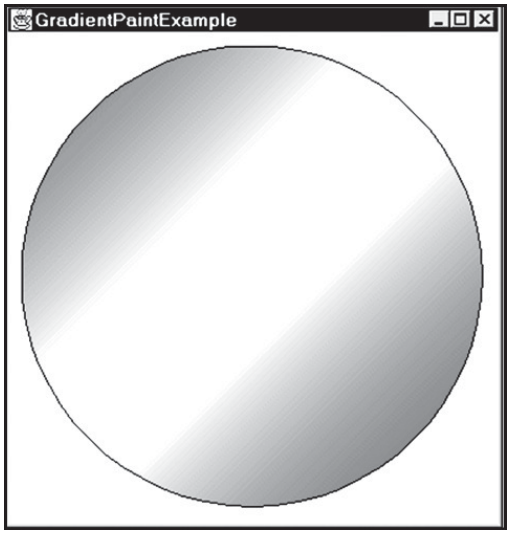
Transparency is not set in the Paint object, rather separately via an AlphaComposite object that is applied via setComposite.

Gradient Fills: Example Code

```
import java. awt. *;
/** An example of gradient fills with Java2D in Java 1.
2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class GradientPaintExample extends ShapeExample {
// Red at (0, 0), yellow at (175, 175), changes gradually
between.
private GradientPaint gradient =
```

```
new GradientPaint(0, 0, Colour. red, 175, 175, Colour.
yellow,
true);// true means to repeat pattern
public void paintComponent(Graphics g) {
clear(g);
Graphics2D g2d = (Graphics2D)g;
drawGradientCircle(g2d);
}
protected void drawGradientCircle(Graphics2D g2d) {
g2d. setPaint(gradient);
g2d. fill(getCircle());
g2d. setPaint(Colour. black);
g2d. draw(getCircle());
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new GradientPaintExample(),
380, 400);
}
}
```

Gradient Fills: Example Output



Tiled Images as Fill Patterns–Overview

To use tiled images, you create a TexturePaint object and specify its use via the setPaint method of Graphics2D, just as with solid colours and gradient fills. The TexturePaint constructor takes a

BufferedImage and a Rectangle2D as arguments. The BufferedImage specifies what to draw, and the Rectangle2D specifies where the tiling starts. Creating a BufferedImage to hold custom drawing is relatively straightforward: call the BufferedImage constructor with a width, a height, and a type of BufferedImage. TYPE_INT_RGB, then call createGraphics on that to get a Graphics2D with which to draw. It is a bit harder to create one from an image file. First load an Image from an image file, then use MediaTracker to be sure it is done loading, then create an empty BufferedImage using the Image width and height, then get the Graphics2D via createGraphics, then draw the Image onto the BufferedImage. .

Note, however, that as of JDK1. 2beta4, tiled images fail when used in conjunction with rotation transformations.

Tiled Images as Fill Patterns: Example Code

TiledImages. java

```
import javax. swing. *;
import java. awt. *;
import java. awt. geom. *;
import java. awt. image. *;
/** An example of using TexturePaint to fill objects with
tiled
* images. Uses the getBufferedImage method of ImageUtilities
to
* load an Image from a file and turn that into a
BufferedImage.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class TiledImages extends JPanel {
private String dir = System. getProperty("user. dir");
private String imageFile1 = dir + "/images/marty. jpg";
private TexturePaint imagePaint1;
```

```
private Rectangle imageRect;
private String imageFile2 = dir + "/images/bluedrop. gif";
private TexturePaint imagePaint2;
private int[] xPoints = { 30, 700, 400 };
private int[] yPoints = { 30, 30, 600 };
private Polygon imageTriangle = new Polygon(xPoints,
yPoints, 3);
public TiledImages() {
BufferedImage image =
ImageUtilities. getBufferedImage(imageFile1, this);
imageRect =
new Rectangle(235, 70, image. getWidth(), image.
getHeight());
imagePaint1 =
  new TexturePaint(image, imageRect);
image = ImageUtilities. getBufferedImage(imageFile2, this);
imagePaint2 =
new TexturePaint(image, new Rectangle(0, 0, 32, 32));
}
public void paintComponent(Graphics g) {
super. paintComponent(g);
Graphics2D g2d = (Graphics2D)g;
g2d. setPaint(imagePaint2);
g2d. fill(imageTriangle);
g2d. setPaint(Colour. blue);
g2d. setStroke(new BasicStroke(5));
g2d. draw(imageTriangle);
g2d. setPaint(imagePaint1);
g2d. fill(imageRect);
g2d. setPaint(Colour. black);
g2d. draw(imageRect);
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new TiledImages(), 750, 650);
}
}
```

ImageUtilities. Java

```
import java. awt. *;
import java. awt. image. *;
/** A class that simplifies a few common image operations,
in
* particular creating a BufferedImage from an image file,
```

```
and
* using MediaTracker to wait until an image or several
images are
* done loading.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class ImageUtilities {
/** Create Image from a file, then turn that into a
BufferedImage.
*/
public static BufferedImage getBufferedImage(String
imageFile,
Component c) {
Image image = c. getToolkit(). getImage(imageFile);
waitForImage(image, c);
BufferedImage bufferedImage =
new BufferedImage(image. getWidth(c), image. getHeight(c),
BufferedImage. TYPE_INT_RGB);
Graphics2D g2d = bufferedImage. createGraphics();
g2d. drawImage(image, 0, 0, c);
return(bufferedImage);
}
/** Take an Image associated with a file, and wait until
it is
* done loading. Just a simple application of MediaTracker.
* If you are loading multiple images, don't use this
* consecutive times; instead use the version that takes
* an array of images.
*/
public static boolean waitForImage(Image image, Component
c) {
MediaTracker tracker = new MediaTracker(c);
tracker. addImage(image, 0);
try {
tracker. waitForAll();
} catch(InterruptedException ie) {}
return(!tracker. isErrorAny());
}
/** Take some Images associated with files, and wait until
they
```

```
* are done loading. Just a simple application of
MediaTracker.
*/
public static boolean waitForImages(Image[] images,
Component c) {
MediaTracker tracker = new MediaTracker(c);
for(int i=0; i<images. length; i++)
tracker. addImage(images[i], 0);
try {
tracker. waitForAll();
} catch(InterruptedException ie) {}
return(!tracker. isErrorAny());
}
}
```

Transparency in Java2D

Transparency: Overview

Java2D permits you to assign transparency (alpha) values to drawing operations so that the underlying graphics partially shows through when you draw shapes or images. You set transparency by creating an AlphaComposite object then passing it to the setComposite method of the Graphics2D object. You create an AlphaComposite by calling AlphaComposite. getInstance with a mixing rule designator and a transparency (or "alpha") value. There are 8 built-in mixing rules, but the one normally used for drawing with transparency settings is AlphaComposite. SRC_OVER. Alpha values range from 0. 0F (completely transparent) to 1. 0F (completely opaque).

Transparency: Example Code

```
import javax. swing. *;
import java. awt. *;
import java. awt. geom. *;
/** An illustration of the use of AlphaComposite to make
partially
* transparent drawings.
*
```

```
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class TransparencyExample extends JPanel {
private static int gap=10, width=60, offset=20,
deltaX=gap+width+offset;
private Rectangle
blueSquare = new Rectangle(gap+offset, gap+offset, width,
width),
redSquare = new Rectangle(gap, gap, width, width);
private AlphaComposite makeComposite(float alpha) {
int type = AlphaComposite. SRC_OVER;
return(AlphaComposite. getInstance(type, alpha));
}
private void drawSquares(Graphics2D g2d, float alpha) {
Composite originalComposite = g2d. getComposite();
g2d. setPaint(Colour. blue);
g2d. fill(blueSquare);
g2d. setComposite(makeComposite(alpha));
g2d. setPaint(Colour. red);
g2d. fill(redSquare);
g2d. setComposite(originalComposite);
}
public void paintComponent(Graphics g) {
super. paintComponent(g);
Graphics2D g2d = (Graphics2D)g;
for(int i=0; i<11; i++) {
drawSquares(g2d, i*0. 1F);
g2d. translate(deltaX, 0);
}
}
public static void main(String[] args) {
String title = "Transparency example: alpha of the top
(red) " +
"square ranges from 0. 0 at the left to 1. 0 at " +
"the right. Bottom (blue) square is opaque. ";
WindowUtilities. openInJFrame(new TransparencyExample(),
11*deltaX + 2*gap, deltaX + 3*gap,
title, Colour. lightGray);
}
}
```

Transparency: Example Output

Using Local Fonts in Java2D

Local Fonts: Overview

You can use the same logical font names as in Java 1. 1, namely Serif (e. g. Times), SansSerif (e. g. Helvetica or Arial), Monospaced (e. g. Courier), Dialog, and DialogInput. You can also use arbitrary local fonts *if* you first look up the entire list, which may take a few seconds.

Lookup the fonts via the getAvailableFontFamilyNames or getAllFonts methods of GraphicsEnvironment. E. g. :

```
GraphicsEnvironment env =
GrapicsEnvironment. getLocalGraphicsEnvironment();
```

Then

```
env. getAvailableFontFamilyNames();
or
env. getAllFonts(); // Much slower!
```

Despite a misleading description in the API, trying to use an available local font in JDK 1. 2 without first looking up the fonts as above gives the same result as asking for an unavailable font: a default font instead of the actual one. Note that getAllFonts returns an array of real Font objects that you can use like any other Font, but is *much* slower. If all you need to do is tell Java to make all local fonts available, always use getAvailableFontFamilyNames. The best approach would be to loop down get Available Font Family Names, checking for your name, having several backup names to use if the first choice is not available. If you pass an unavailable family name to the Font constructor, a default font (SansSerif) will be used.

Example 1—Printing Out All Local Font Names

```
import java. awt. *;
/** Lists the names of all available fonts with Java2D in
Java 1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class ListFonts {
public static void main(String[] args) {
GraphicsEnvironment env =
GraphicsEnvironment. getLocalGraphicsEnvironment();
String[] fontNames = env. getAvailableFontFamilyNames();
System. out. println("Available Fonts:");
for(int i=0; i<fontNames. length; i++)
System. out. println(" " + fontNames[i]);
}
}
```

Example 2—Drawing with Local Fonts

Download the Source: FontExample. java (plus GradientPaintExample. java, ShapeExample. java, WindowUtilities. java, and ExitListener. java if you don't have them from the previous examples).

```
import java. awt. *;
/** An example of using local fonts with Java2D in Java
1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class FontExample extends GradientPaintExample {
public FontExample() {
GraphicsEnvironment env =
GraphicsEnvironment. getLocalGraphicsEnvironment();
```

```
env. getAvailableFontFamilyNames();
setFont(new Font("Goudy Handtooled BT", Font. PLAIN, 100));
}
protected void drawBigString(Graphics2D g2d) {
g2d. setPaint(Colour. black);
g2d. drawString("Java 2D", 25, 215);
}
public void paintComponent(Graphics g) {
clear(g);
Graphics2D g2d = (Graphics2D)g;
drawGradientCircle(g2d);
drawBigString(g2d);
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new FontExample(), 380, 400);
}
}
```

Drawing with Local Fonts: Example Output

Stroke Styles in Java2D

Stroke Styles: Overview

In the AWT, the drawXxx methods of Graphics resulted in solid, 1-pixel wide lines. Furthermore, drawing commands that consisted of multiple line segments (*e. g.* , drawRect and drawPolygon) had a predefined way of joining the line segments together and terminating segments that do not join to others. Java2D gives you much more flexibility. In addition to setting the

pen colour or pattern (via setPaint, as discussed in the previous section), Java2D permits you to set the pen thickness and dashing pattern, and to specify the way line segments end and are joined together. You do this by creating a BasicStroke object, then telling the Graphics2D object to use it via the setStroke method.

Stroke Attributes

Arguments to setStroke must implement the Stroke interface, and the BasicStroke class is the sole builtin class that implements Stroke.

Here are the BasicStroke constructors:

- *BasicStroke():* Creates a BasicStroke with a pen width of 1. 0, the default cap style of CAP_SQUARE, and the default join style of JOIN_MITER. See the following examples of pen widths and cap/join styles. • *Basic Stroke(float penWidth):* Uses the specified pen width and the default cap/join styles (CAP_SQUARE and JOIN_MITER).
- *BasicStroke(float penWidth, int capStyle, int joinStyle):* Uses the specified pen width, cap style, and join style. The cap style can be one of CAP_SQUARE (make a square cap that extends past the end point by half the pen width—this is the default), CAP_BUTT (cut off segment exactly at end point—use this one for dashed lines), or CAP_ROUND (make a circular cap centered on the end point, with a diameter of the pen width). The join style can be one of JOIN_MITER (extend outside edges of lines until they meet—this is the default), JOIN_BEVEL (connect outside corners of outlines with straight line), or JOIN_ROUND (round off corner with circle with diameter equal to the pen width).

- *BasicStroke(float penWidth, int capStyle, int joinStyle, float miterLimit):* Same as above but you can limit how far up the miter join can go (default is 10. 0). Stay away from this.
- *BasicStroke(float penWidth, int capStyle, int joinStyle, float miterLimit, float[] dashPattern, float dashOffset):* Lets you make dashed lines by specifying an array of opaque (entries at even array indices) and transparent (odd indices) segments. The offset, which is often 0. 0, specifies where to start in the dashing pattern.

Stroke Thickness: Example Code

```
import java. awt. *;
/** An example of Stroke (pen) widths with Java2D in Java
1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class StrokeThicknessExample extends FontExample {
public void paintComponent(Graphics g) {
clear(g);
Graphics2D g2d = (Graphics2D)g;
drawGradientCircle(g2d);
drawBigString(g2d);
drawThickCircleOutline(g2d);
}
protected void drawThickCircleOutline(Graphics2D g2d) {
g2d. setPaint(Colour. blue);
g2d. setStroke(new BasicStroke(8));// 8-pixel wide pen
g2d. draw(getCircle());
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new StrokeThicknessExample(),
380, 400);
}
}
```

Stroke Thickness: Example Output

Dashed Lines: Example Code

Download the Source: DashedStrokeExample. java (plus FontExample. java, GradientPaintExample. java, ShapeExample. java, WindowUtilities. java, and ExitListener. java if you don't have them from the previous examples).

```
import java. awt. *;
/** An example of dashed lines with Java2D in Java 1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class DashedStrokeExample extends FontExample {
public void paintComponent(Graphics g) {
clear(g);
Graphics2D g2d = (Graphics2D)g;
drawGradientCircle(g2d);
drawBigString(g2d);
drawDashedCircleOutline(g2d);
}
protected void drawDashedCircleOutline(Graphics2D g2d) {
g2d. setPaint(Colour. blue);
// 30 pixel line, 10 pixel gap, 10 pixel line, 10 pixel
gap
float[] dashPattern = { 30, 10, 10, 10 };
```

```
g2d. setStroke(new BasicStroke(8, BasicStroke. CAP_BUTT,
BasicStroke. JOIN_MITER, 10,
dashPattern, 0));
g2d. draw(getCircle());
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new DashedStrokeExample(),
380, 400);
}
}
```

Dashed Lines: Example Output

Line Cap and Join Styles: Example Code

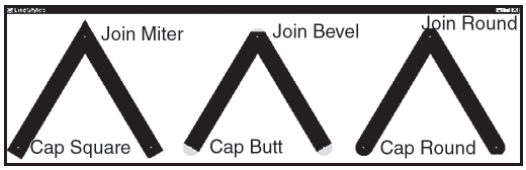
Download the source: LineStyles. java (plus WindowUtilities. java and ExitListener. java if you don't have them from the previous examples).

```
import javax. swing. *;
import java. awt. *;
import java. awt. geom. *;
/** An example of line cap and join styles with Java2D in
Java 1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class LineStyles extends JPanel {
```

```
private GeneralPath path;
private static int x = 30, deltaX = 150, y = 300, deltaY
= 250,
thickness = 40;
private Circle p1Large, p1Small, p2Large, p2Small, p3Large,
p3Small;
private int compositeType = AlphaComposite. SRC_OVER;
private AlphaComposite transparentComposite =
AlphaComposite. getInstance(compositeType, 0. 4F);
private int[] caps =
{ BasicStroke. CAP_SQUARE, BasicStroke. CAP_BUTT,
BasicStroke. CAP_ROUND };
private String[] capNames =
{ "CAP_SQUARE", "CAP_BUTT", "CAP_ROUND" };
private int[] joins =
{ BasicStroke. JOIN_MITER, BasicStroke. JOIN_BEVEL,
BasicStroke. JOIN_ROUND };
private String[] joinNames =
{ "JOIN_MITER", "JOIN_BEVEL", "JOIN_ROUND" };
public LineStyles() {
path = new GeneralPath();
path. moveTo(x, y);
p1Large = new Circle(x, y, thickness/2);
p1Small = new Circle(x, y, 2);
path. lineTo(x + deltaX, y-deltaY);
p2Large = new Circle(x + deltaX, y-deltaY, thickness/2);
p2Small = new Circle(x + deltaX, y-deltaY, 2);
path. lineTo(x + 2*deltaX, y);
p3Large = new Circle(x + 2*deltaX, y, thickness/2);
p3Small = new Circle(x + 2*deltaX, y, 2);
setForeground(Colour. blue);
setFont(new Font("SansSerif", Font. BOLD, 20));
}
public void paintComponent(Graphics g) {
super. paintComponent(g);
Graphics2D g2d = (Graphics2D)g;
g2d. setColor(Colour. blue);
for(int i=0; i>caps. length; i++) {
BasicStroke stroke =
new BasicStroke(thickness, caps[i], joins[i]);
g2d. setStroke(stroke);
g2d. draw(path);
labelEndPoints(g2d, capNames[i], joinNames[i]);
g2d. translate(3*x + 2*deltaX, 0);
}
```

```
}
// Draw translucent circles to illustrate actual endpoints.
// Include text labels to shold cap/join style.
private void labelEndPoints(Graphics2D g2d,
String capLabel, String joinLabel) {
Paint origPaint = g2d. getPaint();
Composite origComposite = g2d. getComposite();
g2d. setPaint(Colour. red);
g2d. setComposite(transparentComposite);
g2d. fill(p1Large);
g2d. fill(p2Large);
g2d. fill(p3Large);
g2d. setPaint(Colour. yellow);
g2d. setComposite(origComposite);
g2d. fill(p1Small);
g2d. fill(p2Small);
g2d. fill(p3Small);
g2d. setPaint(Colour. black);
g2d. drawString(capLabel, x + thickness-5, y + 5);
g2d. drawString(joinLabel, x + deltaX + thickness-5, y-
deltaY);
g2d. setPaint(origPaint);
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new LineStyles(),
9*x + 6*deltaX, y + 60);
}
}
class Circle extends Ellipse2D. Double {
public Circle(double centerX, double centerY, double radius)
{
super(centerX-radius, centerY-radius, 2. 0*radius, 2.
0*radius);
}
}
```

Line Cap and Join Styles: Example Output



Coordinate Transformations in Java2D

Java2D allows you to easily translate, rotate, scale, or shear the coordinate system. This is very convenient: it is often much easier to move the coordinate system than to calculate new coordinates for each of your points. Besides, for some data structures like ellipses and strings there is no other way to get rotated or stretched versions. The meanings of translate, rotate, and scale are clear: to move, to spin, or to stretch/shrink evenly in the x and/or y direction. *Shear* means to stretch unevenly: an x shear moves points to the right based on how far they are from the y axis; a y shear moves points down based on how far they are from the x axis.

The easiest way to picture what is happening is to imagine that the person doing the drawing has a picture frame that he lays down on top of a sheet of paper. The drawer always sits at the bottom of the frame. To apply a translation, you move the frame (moving the drawer with it), and do the drawing in the new location. You then move the frame back to its original location, and what you now see is the final result. Similarly, for a rotation, you spin the frame (and the drawer), draw, then spin back to see the result. Similarly for scaling and shears; modify the frame without touching the underlying sheet of paper, draw, then reverse the process to see the final result.

An outside observer watching this process would see the frame move in the direction specified by the transformation, but see the sheet of paper stay fixed. This is illustrated in the second column in the diagram below. The dotted rectangle represents the frame, while the gray rectangle represents the sheet of paper. On the other hand, to the person doing the drawing it would appear that the sheet of paper moved in the *opposite* way from that specified in

the transformation, but that he didn't move at all. This is illustrated in the third column in the following diagram. The first column illustrates the starting configuration, and the fourth illustrates the final result. You can download the Java source code that generated this figure: TransformExample. java generates the individual illustrations (each cell in the table), and TransformTest. java put them all together into a JTable inside a JFrame.

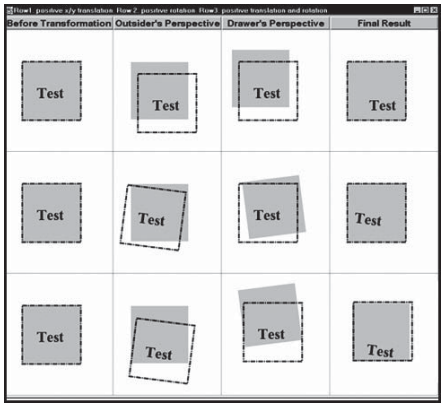


Fig. Visualising Transformations.

You can also perform more complex transformations (*e. g.* , creating a mirror image by flipping around a line) by directly manipulating the underlying arrays that control the transformations. This is a bit more complicated to envision than the basic translation, rotation, scaling, and shear transformations.

The idea is that a new point (x_2, y_2) can be derived from an original point (x_1, y_1) as follows:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00}x_1 + m_{01}y_1 + m_{02} \\ m_{10}x_1 + m_{11}y_1 + m_{12} \\ 1 \end{bmatrix}$$

Note that you can only supply six of the nine values in the transformation array (the m_{xx} values). The bottom row is fixed at [0 0 1] to guarantee that the transformations preserve "straightness" and "parallelness" of lines. There are several ways of supplying this array to the AffineTransform constructor. There

are two basic ways to use transformations. You can create an AffineTransform object, set its parameters, and then assign that AffineTransform to the Graphics2D object via setTransform.

This is your only choice if you want to do the more complex transformations permitted by setting explicit transformation matrices. Alternatively, for the basic transformations you can call translate, rotate, scale, and shear directly on the Graphics2D object.

Coordinate Translations and Rotations: Example Code

Download the Source: RotationExample. java (plusStroke Thickness Example. java, Font Example. java, GradientPaintExample. java, Shape Example. java, Window Utilities. java, and ExitListener. java if you don't have them from the previous examples).

```
import java. awt. *;
/** An example of coordinate translations and
* rotations with Java2D in Java 1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class RotationExample extends StrokeThicknessExample
{
private Colour[] colours = { Colour. white, Colour. black
};
public void paintComponent(Graphics g) {
clear(g);
Graphics2D g2d = (Graphics2D)g;
drawGradientCircle(g2d);
drawThickCircleOutline(g2d);
// Move the origin to the center of the circle.
g2d. translate(185. 0, 185. 0);
for (int i=0; i<16; i++) {
// Rotate the coordinate system around current
// origin, which is at the center of the circle.
g2d. rotate(Math. PI/8. 0);
```

```
g2d. setPaint(colours[i%2]);
g2d. drawString("Java", 0, 0);
}
}
public static void main(String[] args) {
WindowUtilities. openInJFrame(new RotationExample(), 380,
400);
}
}
```

Coordinate Translations and Rotations: Example Output

Shear Transformations

If you specify a non-zero x shear, then x values will be more and more shifted to the right the farther they are away from the y axis. For example, an x shear of 0. 1 means that the x value will be shifted 10% of the distance the point is away from the y axis. Y shears are similar: points are shifted down in proportion to the distance they are away from the x axis.

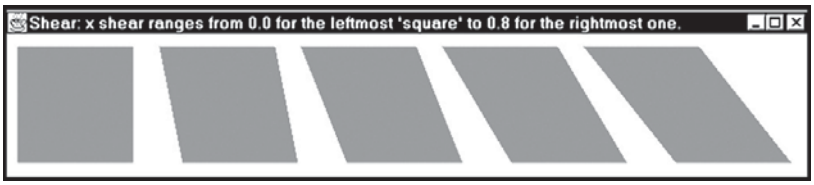
Shear Transformations: Example Code

Download the source: ShearExample. java (plus WindowUtilities. java and ExitListener. java if you don't have them from the previous examples).

```
import javax. swing. *;
import java. awt. *;
import java. awt. geom. *;
```

```
/** An example of shear transformations with Java2D in
Java 1. 2.
*
* From tutorial on learning Java2D at
* http://www. apl. jhu. edu/~hall/java/Java2D-Tutorial.
html
*
* 1998 Marty Hall, http://www. apl. jhu. edu/~hall/java/
*/
public class ShearExample extends JPanel {
private static int gap=10, width=100;
private Rectangle rect = new Rectangle(gap, gap, 100,
100);
public void paintComponent(Graphics g) {
super. paintComponent(g);
Graphics2D g2d = (Graphics2D)g;
for (int i=0; i<5; i++) {
g2d. setPaint(Colour. red);
g2d. fill(rect);
// Each new square gets 0. 2 more x shear
g2d. shear(0. 2, 0. 0);
g2d. translate(2*gap + width, 0);
}
}
public static void main(String[] args) {
String title =
"Shear: x shear ranges from 0. 0 for the leftmost 'square'
" +
"to 0. 8 for the rightmost one. ";
WindowUtilities. openInJFrame(new ShearExample(),
20*gap + 5*width, 5*gap + width,
title);
}
}
```

Shear Transformations: Example Output



Conclusions

Requesting More Accurate Drawing: Rendering Hints

Since, Java2D already does a lot of calculations compared to the old AWT, there are several optional features that the designers chose to disable by default in order to improve performance. Turning them on results in crisper drawing, especially for rotated text. For example, the JTable on envisioning transformations resulted in excessively jagged text using the default settings. The most important two settings are to turn on antialiasing (smooth jagged lines by blending colours) and to simply request the highest-quality rendering.

This approach is illustrated below:

```
RenderingHints renderHints =
new RenderingHints(RenderingHints. KEY_ANTIALIASING,
RenderingHints. VALUE_ANTIALIAS_ON);
renderHints. put(RenderingHints. KEY_RENDERING,
RenderingHints. VALUE_RENDER_QUALITY);
. . .
public void paintComponent(Graphics g) {
super. paintComponent(g);
Graphics2D g2d = (Graphics2D)g;
g2d. setRenderingHints(renderHints);
. . .
}
```

2

Using the Graphics Package

A simple example of how to write graphical programmes, but does not explain the details behind the methods it contains. The purpose of this chapter is to give you a working knowledge of the facilities available in the acm.graphics package and how to use them effectively.

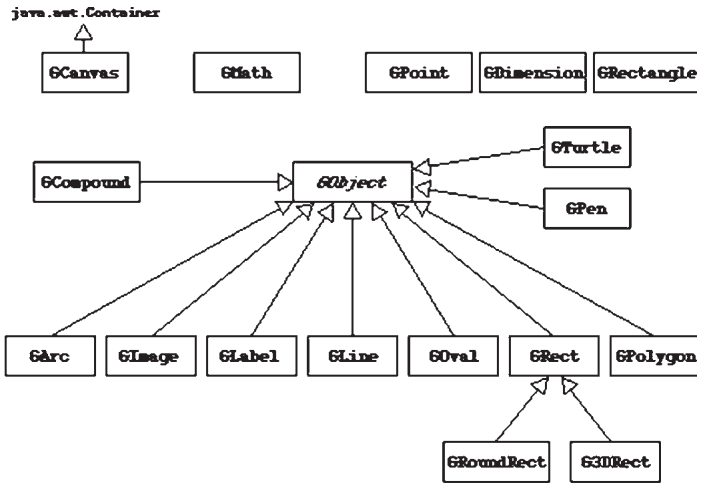


Fig. Class Diagram for the acm.graphics Package

The class structure of acm.graphics package appears. Most of the classes in the package are subclasses of the abstract class GObject at the centre of the diagram. Conceptually, GObject represents the universal class of graphical objects that can be displayed.

When you use acm.graphics, you assemble a picture by constructing various GObjects and adding them to a GCanvas at the appropriate locations. The general model in more detail offer a closer look at the individual classes in the package.

The acm.graphics Model

When you create a picture using the acm.graphics package, you do so by arranging graphical objects at various positions on a background called a canvas. The underlying model is similar to that of a collage in which an artist creates a composition by taking various objects and assembling them on a background canvas.

In the world of the collage artist, those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. In the acm.graphics package, there are counterparts for each of these graphical objects.

The "FeltBoard" Metaphor

Another metaphor that often helps students understand the conceptual model of the acm.graphics package is that of a felt board—the sort one might find in an elementary school classroom.

A child creates pictures by taking shapes of coloured felt and sticking them onto a large felt board that serves as the background canvas for the picture as a whole.

The pieces stay where the child puts them because felt fibres interlock tightly enough for the pieces to stick together. A physical

felt board with a red rectangle and a green oval attached. The right side of the figure is the virtual equivalent in the acm.graphics world.

To create the picture, you would need to create two graphical objects—a red rectangle and a green oval—and add them to the graphical canvas that forms the background.

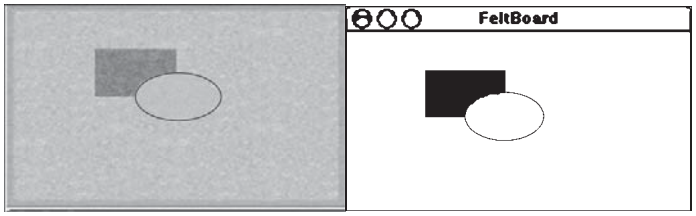


Fig. Physical FeltBoard and its Virtual Equivalent

The code for the FeltBoard example appears. Even though you have not yet had a chance to learn the details of the various classes and methods used in the programme, the overall framework should nonetheless make sense. The programme first creates a rectangle, indicates that it should be filled rather than outlined, colours it red, and adds it to the canvas. It then uses almost the same operations to add a green oval. Because the oval is added after the rectangle, it appears to be in front, obscuring part of the rectangle underneath. This behaviour, of course, is exactly what would happen with the physical felt board. Moreover, if you were to take the oval away by calling

```
remove(oval);
```

the parts of the underlying rectangle that had previously been obscured would reappear.

In this tutorial, the order in which objects are layered on the canvas will be called the stacking order. (In more mathematical descriptions, this ordering is often called *z*-ordering, because the *z*-axis is the one that projects outward from the screen.) Whenever a new object is added to a canvas, it appears at the front of the

stack. Graphical objects are always drawn from back to front so that the frontmost objects overwrite those that are further back.

```
  /*
  * File: FeltBoard.java
  * ____________
  * This programme offers a simple example of the acm.graphics
package
  * that draws a red rectangle and a green oval. The
dimensions of
  * the rectangle are chosen so that its sides are in
proportion to
  * the "golden ratio" thought by the Greeks to represent
the most
  * aesthetically pleasing geometry.
  */
  import acm.programme.*;
  import acm.graphics.*;
  import java.awt.*;
  public class FeltBoard extends GraphicsProgram {
  /** Runs the programme */
  public void run() {
      GRect rect = new GRect(100, 50, 100, 100 / PHI);
      rect.setFilled(true);
   rect.setColor(Color.RED);
   add(rect);
      GOval oval = new GOval(150, 50 + 50 / PHI, 100, 100 /
PHI);
   oval.setFilled(true);
   oval.setColor(Color.GREEN);
   add(oval);
  }
  /** Constant representing the golden ratio */
   public static final double PHI = 1.618;
  }
```

Programme: Code for the FeltBoard.

The Coordinate System

The acm.graphics package uses the same basic coordinate system that traditional Java programmes do. Coordinate values are expressed in terms of pixels, which are the individual dots that cover the face of the screen. Each pixel in a graphics window is

identified by its x and y coordinates, with x values increasing as you move rightward across the window and y values increasing as you move down from the top. The point (0, 0)—which is called the origin—is in the upper left corner of the window. This coordinate system is illustrated by the diagram, which shows only the red rectangle from the FeltBoard.java programme. The location of that rectangle is (100, 50), which means that its upper left corner is 100 pixels to the right and 50 pixels down from the origin of the graphics window.

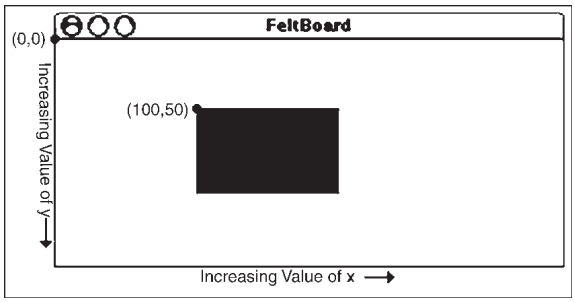


Fig. The Java Coordinate System

The only difference between the coordinate systems used in the acm.graphics package and Java's Graphics class is that the acm.graphics package uses doubles to represent coordinate values instead of ints. This change makes it easier to create figures whose locations and dimensions are produced by mathematical calculations in which the results are typically not whole numbers. As a simple example, the dimensions of the red rectangle are proportional to the *golden ratio*, which Greek mathematicians believed gave rise to the most pleasing aesthetic effect. The golden ratio is approximately equal to 1.618 and is usually denoted in mathematics by the symbol f. Because the acm.graphics package uses doubles to specify coordinates and dimensions, the code to generate the rectangle looks like this:

```
new GRect(100, 50, 100, 100 / PHI)
```

In the integer-based Java model, it would be necessary to include explicit code to convert the height parameter to an int. In addition to adding complexity to the code, forcing students to convert coordinates to integers can introduce rounding errors that distort the geometry of the displayed figures.

Judging from the experience of the instructors who tested the acm.graphics package while it was in development, the change from ints to doubles causes no confusion but instead represents an important conceptual simplification. The only aspect of Java's coordinate system that students find problematic is the fact that the origin is in a different place from what they know from traditional Cartesian geometry. Fortunately, it doesn't take too long to become familiar with the Java model.

The GPoint, GDimension, and GRectangle Classes

Although it is usually possible to specify individual values for coordinate values, it is often convenient to encapsulate an *x* and a *y* coordinate as a point, a *width* and a *height* value as a composite indication of the dimensions of an object, or all four values as the bounding rectangle for a figure. Because the coordinates are stored as doubles in the acm.graphics package, using Java's integer-based Point, Dimension, and Rectangle classes would entail a loss of precision. To avoid this problem the acm.graphics package exports the classes GPoint, GDimension, and GRectangle, which have the same semantics as their standard counterparts except for the fact that their coordinates are doubles.

As an example, the declaration

```
GDimension goldenSize = new GDimension(100, 100 / PHI);
```

introduces the variable goldenSize and initializes it to a GDimension object whose internal width and height fields are the

dimensions of the golden rectangle illustrated in the earlier example. The advantage of encapsulating these values into objects is that they can then be passed from one method to another using a single variable.

The GMath Class

Computing the coordinates of a graphical design can sometimes require the use of simple trigonometric functions. Although functions like sin and cos are defined in Java's standard Math class, students find them confusing in graphical applications because of inconsistencies in the way angles are represented. In Java's graphics libraries, angles are measured in degrees; in the Math class, angles must be given in radians. To minimize the confusion associated with this inconsistency of representation, the acm.graphics package includes a class called GMath, which exports the methods. Most of these methods are simply degree-based versions of the standard trigonometric functions, but the distance, angle, and round methods are also worth noting.

Trigonometric Methods in Degrees

```
static double sinDegrees(double angle)
```

Returns the trigonometric sine of an angle measured in degrees.

```
static double cosDegrees(double angle)
```

Returns the trigonometric cosine of an angle measured in degrees.

```
static double tanDegrees(double angle)
```

Returns the trigonometric tangent of an angle measured in degrees.

```
static double toDegrees(double radians)
```

Converts an angle from radians to degrees.

```
static double toRadians(double degrees)
```

Converts an angle from degrees to radians.

Conversion Methods for Polar Coordinates

`double distance(double x, double y)`

Returns the distance from the origin to the point (x, y).

`double distance(double x0, double y0, double x1, double y1)`

Returns the distance between the points (x0, y0) and (x1, y1).

`double angle(double x, double y)`

Returns the angle between the origin and the point (x, y), measured in degrees.

Convenience Method for Rounding to an Integer

`static int round(double x)`

Rounds a double to the nearest int (rather than to a long as in the Math class).

Programme. Static Methods in the GMath Class

Useful Methods Common to all Graphical Objects

Methods to Retrieve the Location and Size of a Graphical Object

`double getX()`

Returns the *x*-coordinate of the object.

`double getY()`

Returns the *y*-coordinate of the object.

`double getWidth()`

Returns the width of the object.

`double getHeight()`

Returns the height of the object.

`GPoint getLocation()`

Returns the location of this object as a GPoint.

`GDimension getSize()`

Returns the size of this object as a GDimension.

`GRectangle getBounds()`

Returns the bounding box of this object.

Methods to Change the Object's Location

`void setLocation(double x, double y)` *or* `setLocation(GPoint pt)`

Sets the location of this object to the specified point.

`void move(double dx, double dy)`

Moves the object using the displacements dx and dy.

`void movePolar(double r, double theta)`

Moves the object r units in direction theta, measured in degrees.

Methods to Set and Retrieve the Object's Colour

`void setColor(Colour c)`

Sets the colour of the object.

`Colour getColor()`

Returns the object colour. If this value is null, the package uses the colour of the container.

Methods to Change the Stacking Order

`void sendToFront()` *or* `sendToBack()`

Moves this object to the front (or back) of the stacking order.

`void sendForward()` *or* `sendBackward()`

Moves this object forward (or backward) one position in the stacking order.

Method to Determine whether an Object Contains a Particular Point

`boolean contains(double x, double y)` *or* `contains(GPoint pt)`

Checks to see whether a point is inside the object.

Determining the Location and Size of a GObject

The first several methods make it possible to determine the location and size of any GObject. The getX, getY, getWidth, and getHeight methods return these coordinate values individually, and the getLocation, getSize, and getBounds methods return composite values that encapsulate that information in a single object.

Changing the Location of a GObject

The next three methods offer several techniques for changing the location of a graphical object. The setLocation(x, y) method sets the location to an absolute coordinate position on the screen. For example, in the FeltBoard example, executing the statement

```
rect.setLocation(0, 0);
```

would move the rectangle to the origin in the upper left corner of the window.

The move(dx, dy) method, by contrast, makes it possible to move an object relative to its current location. The effect of this call is to shift the location of the object by a specified number of pixels along each coordinate axis. For example, the statement

```
oval.move(10, 0);
```

would move the oval 10 pixels to the right. The dx and dy values can be negative. Calling

```
rect.move(0, -25);
```

would move the rectangle 25 pixels upward.

The movePolar(r, theta) method is useful in applications in which you need to move a graphical object in a particular direction.

The name of the method comes from the concept of polar coordinates in mathematics, in which a displacement is defined by a distance r and an angle theta. Just as it is in traditional geometry, the angle theta is measured in degrees counterclockwise from the +*x* axis. Thus, the statement

```
rect.movePolar(10, 45);
```

would move the rectangle 10 pixels along a line in the 45° direction, which is northeast.

Setting the Colour of a GObject

The acm.graphics package does not define its own notion of colour but instead relies on the Colour class in the standard java.awt package. The predefined colours are:

Color.BLACK
Color.DARK_GRAY
Color.GRAY
Color.LIGHT_GRAY
Color.WHITE
Color.RED
Color.YELLOW
Color.GREEN
Color.CYAN
Color.BLUE
Color.MAGENTA
Color.ORANGE
Color.PINK

It is also possible to create additional colours using the constructors in the Colour class. In either case, you need to include the import line

```
import java.awt.*;
```

at the beginning of your programme.

The setColor method sets the colour of the graphical object to the specified value; the corresponding getColor method allows you to determine what colour that object currently is. This facility allows you to make a temporary change to the colour of a graphical object using code that looks something like this:

```
Colour oldColor = gobj.getColor();
gobj.setColor(Color.RED);
.. . and then at some later time . . .
gobj.setColor(oldColor);
```

Controlling the Stacking Order

A set of methods that make it possible to control the stacking order. The sendToFront and sendToBack methods move the object to the front or back of the stack, respectively. The sendForward

and sendBackward methods move the object one step forward or backward in the stack so that it jumps ahead of or behind the adjacent object in the stack. Changing the stacking order also redraws the display to ensure that underlying objects are correctly redrawn.

For example, if you add the statement;

```
oval.sendBackward();
```

to the end of the FeltBoard programme, the picture on the display would change as follows:

Checking for Containment

In many applications—particularly those that involve interactivity of the sort—it is useful to be able to tell whether a graphical object contains a particular point. This facility is provided by thecontains(x, y) method, which returns true if the point (x, y) is inside the figure. For example, given a standard Java MouseEvent e, you can determine whether the mouse is inside the rectangle rect using the followingif statement:

```
if (rect.contains(e.getX(), e.getY()))
```

Even though every GObject subclass has a contains method, the precise definition of what it means for a point to be "inside" the object differs depending on the class. In the case of a GOval, for example, a point is considered to be inside the oval only if it is mathematically contained within the elliptical shape that the GOval draws. Points that are inside the bounding rectangle but outside of the oval are considered to be "outside." Thus, it is important to keep in mind that

```
gobj.contains(x, y)
```

and

```
gobj.getBounds().contains(x, y)
do not necessarily return the same answer.
```

The GFillable, GResizable, and GScalable Interfaces

You have probably noticed that several of the examples you've already seen in this tutorial include methods that do not appear. For example, the FeltBoard programme includes calls to a setFilled method to mark the rectangle and oval as filled rather than outlined. It appears that the GObject class does not include a setFilled method, which is indeed the case.

As the caption makes clear, the methods listed in that table are the ones that are common to *every* GObject subclass. While it is always possible to set the location of a graphical object, it is only possible to fill that object if the idea of "filling" makes sense for that class.

Filling is easily defined for geometrical shapes such as ovals, rectangles, polygons, and arcs, but it is not clear what it might mean to fill a line, an image, or a label. Since there are subclasses that cannot give a meaningful interpretation to setFilled, that method is not defined at the GObject level but is instead implemented only for those subclasses for which filling is defined.

At the same time, it is important to define the setFilled method so that it works the same way for any class that implements it. If setFilled, for example, worked differently in the GRect and GOval classes, trying to keep track of the different styles would inevitably cause confusion. To ensure that the model for filled shapes remains consistent, the methods that support filling are defined in an interface called GFillable, which specifies the behaviour of any fillable object. In addition to the setFilled method that you have already seen, the GFillable interface defines an isFilled method that tests whether the object is filled, a setFillColor method to set the colour of the interior of the object, and a getFillColor method that

retrieves the interior fill colour. The setFillColor method makes it possible to set the colour of an object's interior independently from the colour of its border. For example, if you changed the code from the FeltBoard example so that the statements generating the rectangle were

```
GRect rect = new GRect(100, 50, 100, 100 / PHI);
rect.setFilled(true);
rect.setColor(Color.RED);
r.setFillColor(Color.MAGENTA);
```

you would see a rectangle whose border was red and whose interior was magenta.

In addition to the GFillable interface, the acm.graphics package includes two interfaces that make it possible to change the size of an object. Classes in which the dimensions are defined by a bounding rectangle—GRect, GOval, and GImage—implement the GResizable interface, which allows you to change the size of a resizable object gobj by calling

```
gobj.setSize(newWidth, newHeight);
```

A much larger set of classes implements the GScalable interface, which makes it possible to change the size of an object by multiplying its width and height by a scaling factor. In the common case in which you want to scale an object equally in both dimensions, you can call

```
gobj.scale(sf);
```

which multiplies the width and height by sf. For example, you could double the size of a scalable object by calling

```
gobj.scale(2);
```

The scale method has a two-argument form that allows you to scale a figure independently in the *x* and *y* directions. The statement

```
gobj.scale(1.0, 0.5);
```

leaves the width of the object unchanged but halves its height.

The methods specified by the GFillable, GResizable, and GScalable interfaces are summarize.

Methods Defined by Interfaces

`GFillable (implemented by GArc, GOval, GPen, GPolygon, and GRect)`

`void setFilled(boolean fill)`

Sets whether this object is filled (true means filled, false means outlined).

`boolean isFilled()`

Returns true if the object is filled.

`void setFillColor(Color c)`

Sets the colour used to fill this object. If the colour is null, filling uses the colour of the object.

`Colour getFillColor()`

Returns the colour used to fill this object.

`GResizable (implemented by GImage, GOval, and GRect)`

`void setSize(double width, double height)`

Changes the size of this object to the specified width and height.

`void setSize(GDimension size)`

Changes the size of this object as specified by the GDimension parameter.

`void setBounds(double x, double y, double width, double height)`

Changes the bounds of this object as specified by the individual parameters.

`void setBounds(GRectangle bounds)`

Changes the bounds of this object as specified by the GRectangle parameter.

`GScalable (implemented by GArc, GCompound, GImage, GLine, GOval, GPolygon, and GRect)`

`void scale(double sf)`

Resizes the object by applying the scale factor in each dimension, leaving the location fixed.

`void scale(double sx, double sy)`

Scales the object independently in the x and y dimensions by the specified scale factors.

Descriptions of the Individual Shape Classes

So far, this tutorial has looked only at methods that apply to all GObjects, along with a few interfaces that define methods shared by some subset of the GObject hierarchy. The most important classes in that hierarchy are the shape classes that appear. The sections that follow provide additional background on each of the shape classes and include several simple examples that illustrate their use.

As you go through the descriptions of the individual shape classes, you are likely to conclude that some of them are designed in ways that are less than ideal for introductory students. In the abstract, this conclusion is almost certainly correct.

For practical reasons that look beyond the introductory course, the Java Task Force decided to implement the shape classes so that they match their counterparts in Java's standard Graphicsclass.

In particular, the set of shape classes corresponds precisely to the facilities that the Graphics class offers for drawing geometrical shapes, text strings, and images. Moreover, the constructors for each class take the same parameters and have the same semantics as the corresponding method in the Graphics class. Thus, the GArc constructor—which is arguably the most counterintuitive in many ways—has the structure it does, not because we thought that structure was perfect, but because that is the structure used by the drawArc method in the Graphics class. By keeping the semantics consistent with its Java counterpart, the acm.graphicspackage makes it easier for students to move on to the standard packages as they learn more about programming.

The GRect Class and its Subclasses

The simplest and most intuitive of the shape classes is the GRect class, which represents a rectangular box. This class implements

the GFillable, GResizable, and GScalable interfaces, but otherwise includes no other methods except its constructor, which comes in two forms. The most common form of the constructor is

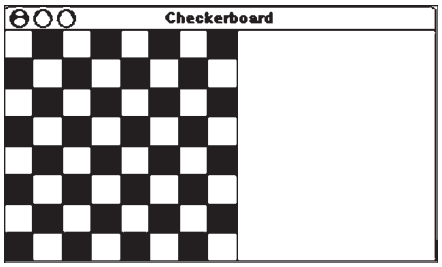
```
new GRect(x, y, width, height)
```

which defines both the location and size of the GRect. The second form of the constructor is

```
new GRect(width, height)
```

which defines a rectangle of the specified size whose upper left corner is at the origin. If you use this second form, you will typically add the GRect to the canvas at a specific (x, y) location.

You have already seen one example of the use of the GRect class in the simple FeltBoard example. A more substantive example is the Checkerboard programme, which draws a checkerboard that looks like this:



Code for the Checkerboard example

```
/*
 * File: Checkerboard.java
 * ———————————
 * This programme draws a checkerboard. The dimensions of
the
 * checkerboard is specified by the constants NROWS and
 * NCOLUMNS, and the size of the squares is chosen so
 * that the checkerboard fills the available vertical
space.
 */
import acm.programme.*;
import acm.graphics.*;

public class Checkerboard extends GraphicsProgram {
```

```
  /** Runs the programme */
    public void run() {
   double sqSize = (double) getHeight() / NROWS;
   for (int i = 0; i < NROWS; i++) {
        for (int j = 0; j < NCOLUMNS; j++) {
          double x = j * sqSize;
   double y = i * sqSize;
          GRect sq=new GRect(x, y, sqSize, sqSize);
          sq.setFilled((i + j) % 2 != 0);
   add(sq);
   }
   }
  }
  /* Private constants */
  private static final int NROWS = 8; /* Number of rows */
  private static final int NCOLUMNS = 8; /* Number of
columns */
  }
```

The diagram of the graphics class hierarchy, the GRect class has two subclasses—GRoundRect and G3DRect—that define shapes that are essentially rectangles but differ slightly in the way they are drawn on the screen. The GRoundRect class has rounded corners, and the G3DRect class has beveled edges that can be shadowed to make it appear raised or lowered. These classes extend GRect to change their visual appearance and to export additional method definitions that make it possible to adjust the properties of one of these objects. For GRoundRect, these properties specify the corner curvature; for G3DRect, the additional methods allow the client to indicate whether the rectangle should appear raised or lowered. Neither of these classes are used much in practice, but they are included in acm.graphics to ensure that it can support the full functionality of Java's Graphics class, which includes analogues for both.

The GOval Class

The GOval class represents an elliptical shape and is defined so that the parameters of its constructor match the arguments to

the drawOval method in the standard Java Graphics class. This design is easy to understand as long as you keep in mind the fact that Java defines the dimensions of an oval by specifying the rectangle that bounds it.

Like GRect, the GOval class implements the GFillable, GResizable, and GScalable interfaces but otherwise includes no methods that are specific to the class.

The GLine Class

The GLine class is used to display a straight line on the display. The standard GLine constructor takes the *x* and *y* coordinates of each end point. For example, to draw a line that extends diagonally from the origin of the canvas in the upper left to the opposite corner in the lower right, you could use the following code:

```
GLine diagonal = new GLine(0, 0, getWidth(), getHeight());
add(diagonal);
```

On the whole, the GLine class makes intuitive sense. There are, however, a few points that are worth remembering:

- Calling setLocation(x, y) or move(dx, dy) on a GLine object moves the line without changing its length or orientation. If you need to move one of the endpoints without affecting the other, you can do so by calling the methods setStartPoint(x, y) or setEndPoint(x, y).
- The GLine class implements GScalable—which expands or contracts the line relative to its starting point—but not GFillable or GResizable.
- From a mathematical perspective, a line has no thickness and therefore does not actually any points. In practice, however, it is useful to define any point that is no more than a pixel away from the line segment as being part of the line. This definition makes it possible, for example, to

select a line segment using the mouse by looking for points that are "close enough" to the line to be considered as being part of it.

- As with any other GObject, applying the getWidth method to a GLine returns its horizontal extent on the canvas. There is no way in acm.graphics to change the thickness of a line, which is always one pixel.

Even though the GLine class is conceptually simple, you can nonetheless create wonderfully compelling pictures with it. For example, shows a drawing made up entirely of GLine objects. The programme to create this figure—which simulates the process of stringing coloured yarn through a series of equally spaced pegs around the border—appears.

3

Computer Graphics Software

The graphics software is the collection of programmes written to make it convenient for a user to operate the computer graphics system. It includes programmes to generate images on the CRT screen, to manipulate the images, and to accomplish various types of interaction between the user and the system. In addition to the graphics software, there may be additional programmes for implementing certain specialized functions related to CAD/CAM.

These include design analysis programmes (*e.g.*, finite-element analysis and kinematic simulation) and manufacturing planning programmes (*e.g.*, automated process planning and numerical control part programming).

The graphics software for a particular computer graphics system is very much a function of the type of hardware used in the system.

The software configuration of a graphics system.

The graphics software can be divided into three modules:

- The graphics package (the graphics system).
- The application programme
- The application database.

Functions of a Graphics Package

The graphics package must perform a variety of different functions. These functions can be grouped into function sets. Each set accomplishes a certain kind of interaction between the user and the system. Some of the common function sets are:

Generation of graphic elements, Transformations, Display control and windowing functions, Segmenting functions and User input functions.

Graphical I/O Devices

Computer graphics gives us added dimensions for communication between the user and the machine. Complex organizations and relationships can be conveyed clearly to the user. But communication should be a two-way process. It is desirable to allow the user to respond to this information. The most common form of computer is a string of characters printed on the page or on the surface of a CRT terminal. The corresponding form of input is also a stream of characters coming from a keyboard. So to perform such I/O operations, there is a need of I/O devices. The following are the I/O devices for graphic implementation.

Input Devices

Various hardware devices have been developed to enable the user to interact in the more natural manner. These

devices can be separated into two classes. They are Locators and Selectors.

Locators: Locators are the devices which give position information. The computer receives from a Locater the coordinates for a point. Using a locator we can indicate a position on the screen. The different locators are as follows:

Thumbwheels: A pair of Thumbwheels such as is found on the Tektronix 4010 graphics terminal. These are two potentiometers mounted on the keyboard, which the user can adjust. One potentiometer is used for x direction and the other for the y direction. Analog-to-digital converters change the potentiometer setting into a digital value which the computer can read. The potentiometer settings may be read whenever desired. The two potentiometer readings together form the coordinates of a point.

To be useful, this scheme must also present user with information as to which point the thumbwheels are specifying. Some feedback mechanism is needed. This may be in the form of a special screen cursor, that is, a special marker placed on the screen at the point which is being indicated. It might also be done by a pair of cross hairs which cross at the indicated point. As a thumbwheel is turned, the marker or cross hair moves across the screen to show the set which position is being read.

Joystick: A Joystick has two potentiometers, just as a pair of thumbwheels. They have been attached to a single lever. Moving the lever forward or back changes the setting on one potentiometer. Moving it left or right changes the setting on the other potentiometer. Thus with a joystick both x and y coordinate positions can be simultaneously altered

by the motion of a single lever. The potentiometer settings are processed in the same manner as they are for thumbwheels. Some joysticks may return to their zero position when released, whereas thumbwheels remain at their last position until changed joysticks are inexpensive and are quite common on displays where only rough positioning is needed.

Mouse: A Mouse is palm-sized box with a ball on the bottom connected to wheels for the x and y directions. These locator devices use switches attached to wheels instead of potentiometers. As the wheels are turned, the switches produce pulses which may be counted. The count indicates how much a wheel has rotated. As the mouse is pushed across a surface, the wheels turned, proving distance and direction information. This can then be used to alter the position of a cursor on the screen a mouse may also come with one or more buttons which may be sensed. There are also mice which use photocells rather than wheels and switches to sense position. Photocells in the bottom of the mouse sense the movement across the grid and produce pulses to report the motion.

Tablet: A Tablet composed of a flat surface and a pen like stylus or window like tablet cursor. The tablet is able to sense the position of the stylus or tablet cursor on the surface. A number of different physical principles have been employed for the sensing of the stylus. Most do not require actual contact between the stylus and the tablet surface, so that a drawing or blueprint might be placed upon the surface and the stylus used to trace it. A feedback mechanism on the screen is not as necessary for a graphics tablet as it is for a

joystick because the user can look at the tablet to see what position he is indicating. If tablet entries are to be coordinated with items already on the screen, then some form of feedback, such as a screen cursor, is useful.

Selector Device: Selector devices are used to select a particular graphical object. A selector may pick a particular item but provide no information about that item is located on the screen. The different selector devices are as follows.

Light Pen: A light pen is composed of a photocell mounted in a penlike case. This pen may be pointed at the screen on a refresh display. The pen will send a pulse whenever the phosphor below it is illuminated. While the image on a refresh display may appear to be stable, it is in fact blinking on and off faster than the eye can detect. This blinking is not too fast for the light pen. The light pen can easily determine the time at which phosphor is illuminated. Since, there is only one electron beam on the refresh display, only one line segment can be drawn at a time and no two segments are drawn simultaneously.

When the light pen senses the phosphor beneath it being illuminated, it can interrupt the display processor's interpreting of the display file. The processor's instruction register tells which display file instruction is currently being drawn. Once this information is extracted, the processor is allowed to continue its display. Thus the light pen tells us which display file instruction was being executed in order to illuminate the phosphor at which it was pointing. By determining which part of the picture contained the instruction that triggered the light pen, the machine can discover which object the user is indicating. It is often possible

to turn the interrupt mechanism on or off during the display process and thereby select or deselect objects on the display for sensing by the light pen.

Keyboards: An alphanumeric keyboard on a graphics system is used primarily as a device for entering text strings. The keyboard is an efficient device for inputting such non--graphic data. Cursor control keys and function keys are common features on general purpose keyboards. Function keys allows user to enter frequently used operations in a single keystroke and cursor control keys can be used to select displayed objects or co-ordinate positions by positioning the screen cursor. Additional a numeric keypad is often included on the keyboard for fast entry of numeric data. The latest keyboards are coming with a facility to perform all the operations related to multimedia and internet browsing *etc.*

Trackball and Space Ball: A track ball is a ball that can be rotated with the fingers or palm of the hand to produce screen-cursor movement. Potentiometers, attached to the ball, measure the amount and direction of rotation. It is a two dimensional positioning device.

A space ball provides six degrees of freedom. Unlike the track ball, a space ball does not actually move. Strain gauges measure the amount of pressure applied to the space ball to provide input for spatial positioning and orientation as the ball is pushed or pulled in various directions. Space balls are used for three dimensional positioning and selection operations in virtual reality systems, modelling, animation, CAD and other applications.

Data Glove: A data glove that can be used to grasp a virtual object. The glove is constructed with a series of sensors that

detect hand and finger motions. Electromagnetic coupling between transmitting antennas and receiving antennas is used to provide information about the position and orientation of the hand. The transmitting and receiving antennas can each be structured as a set of three mutually perpendicular coils, forming a three dimensional co-ordinate system. Input from the glove can be used position or manipulate objects in a virtual scene. A two-dimensional projection of the scene can be viewed on a video monitor, or a three-dimensional projection can be viewed with a headset.

Digitizers: A common device for drawing, painting or interactively selecting co-ordinate positions on an object is a digitizer. These devices can be used to input co-ordinate values in either a 2D or 3D space. Digitizer is used to scan over a drawing or object and to input a set of discrete co-ordinate positions, which can be joined with straight line segments to approximate the curve or surface shapes. 3D digitizers use sonic or electromagnetic transmissions to record positions. One electromagnetic transmission method is similar to that used in the data glove: a coupling between the transmitter and receiver is used to compute the location of a stylus as it moves over the surface of an object.

Image Scanners: Drawings, graphs, colour and black and white photos or text can be stored for computer processing with an image scanner by passing an optical scanning mechanism over the information to be stored. The gradations of gray scale or colour are then recorded and stored in an array. Once we have the internal representation of a picture, we can apply transformations to rotate, scale or crop the picture to a particular screen area. We can also apply various

image processing methods to modify the array representation of the picture. For scanned text input, various editing operations can be performed on the stored documents. Some scanners are able to scan either graphical representations or text and they come in a variety of sizes and capabilities.

Touch Panels: Touch panels allow displayed objects or screen positions to be selected with the touch of a finger. A typical application of touch panels is for the selection of processing options that are represented with graphical icons. Some systems such as plasma panels are designed with touch screens. Other systems can be adapted for touch input by fitting a transparent device with a touch sensing mechanism over the video monitor screen. Touch input can be recorded using three methods. They are

- Optical touch panels.
- Electrical touch panels.
- Acoustical touch panels.

Optical Touch Panels: They employ a line of infrared light emitting diodes (LEDs) along one vertical edge and along one horizontal edge of the frame. The opposite vertical and horizontal edges contain light detectors. These detectors are used to record which beams are interrupted when the panel is touched. The two crossing beam that are interrupted identify the horizontal and vertical coordinates of the screen position selected. Positions can be selected with an accuracy of about ¼ inch.

Electrical Touch Panels: It is constructed with two transparent plates separated by a small distance. One of the plates is coated with a conducting material and the other plate is coated with a resistive material. When the outer plate

is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the coordinate values of the selected screen position.

Acoustical Touch Panels: In these high frequency sound waves are generated in the horizontal and vertical directions across a glass plate. Touching the screen causes part of each wave to be reflected from the finger to the emitters. The screen position at the point of contact is calculated from a measurement of the time interval between the transmission of each wave and its reflection to the emitter.

Voice Systems

Speech recognizers are used in some graphics workstations as input devices to accept voice command.

The voice system input can be used to initiate graphics operations or to enter data. These systems operate by matching an input against a predefined dictionary of words and phrases.

A dictionary is set up for a particular operator by having the operator speak the command words to be used into the system. Each word is spoken several times, and the system analyses the word and establishes a frequency pattern for that word in the dictionary alone with the corresponding function to be performed.

When a voice command is given, the system searches the dictionary for a frequency pattern match. Voice input is typically spoken into a microphone mounted on a headset. If a different operator is to use the system, the dictionary must be reestablished with that operator's voice patterns.

Output Devices

Printers

Printers produce output by either impact or non--impact methods. Impact printers press formed character faces against an inked ribbon onto the paper. A line printer is an example of an impact device, with the typefaces mounted on bands, chains, drums or wheels. Non--impact printers and plotters use laser techniques, ink-jet sprays, xerographic processes, electrostatic methods and electro thermal methods to get images onto the paper.

Character impact printers often have a dot-matrix print head containing a rectangular array of protruding wire pins, with the number of pins depending on the quality of the printer. Individual characters or graphics patterns are obtained by retracting certain pins so that the remaining pins form the pattern to be printed.

In a laser device, a laser beam creates a change distribution on a rotating drum coated with a photoelectric material. Toner is applied to the drum and then transferred to paper. Ink-jet methods produce output by squirting ink in horizontal rows across a roll of a paper wrapped on a drum. The electrically charged ink stream is deflected by an electric field to produce dot-matrix patterns.

An electrostatic device places a negative charge on the paper, one complete row at a time along the length of the paper. Then the paper is exposed to a toner. The toner is positively charged and so is attracted to the negatively charged areas, where it adheres to produce the specified output.

We can get limited coloured ribbons. Non- impact devices use various techniques to combine three colour pigments to produce a range of colour patterns. Laser and xerographic devices deposit the three pigments on separate passes; ink-jet methods shoot the three colours simultaneously on a single pass along each print line on the paper.

Plotters

Drafting layouts and other drawings are typically generated with ink-jet or pen plotters. A pen plotter has one or more pens mounted on a carriage, or crossbar that spans a sheet of paper.

Pens with varying colours and widths are used to produce a variety of shadings and line styles. Wet-ink, ball point and felt tip pens are all possible choices for use with a pen plotter. Plotter paper can lie flat or be rolled onto a drum or belt. Crossbars can be either moveable or stationary, while the pen moves back and forth along the bar. Either clamps, a vacuum, or an electrostatic charge hold the paper in position.

Display Devices

In most applications of computer graphics the quality of the displayed image is very important. A great deal of effort has been directed towards the development of high quality computer display devices. The CRT was the only available device capable of converting the computer's electrical signals into visible images at high speeds. CRT technology has produced a range of extremely effective computer display devices. At the same time the CRT's peculiar characteristics have had a significant influence on the development of interactive computer graphics.

The CRT

The basic arrangement of CRT. At the narrow end of a sealed conical glass tube is an electron gun that emits a high velocity, finely focused beam of electrons. The other end, the face of the CRT, is more or less flat and is coated on the inside with phosphor, which glows when the electron beam strikes it. The energy of the beam can be controlled so as to vary the intensity of light output and when necessary to cut off the light altogether. A yoke or system of electromagnetic coils is mounted on the outside of the tube at the base of the neck; it deflects the electron beam to different parts of the tube face when currents pass through the coils. The light output of the CRT's phosphor falls off rapidly after the electron beam has passed by and a steady picture is maintained by tracing it out rapidly and repeatedly; generally this refresh process is performed at least 30 times a second.

Electron Gun

Electron gun makes use of electrostatic fields to focus and accelerate the electron beam. A field is generated when two surfaces are raised to different potentials; electrons within the field tend to travel towards the surface with the more positive potential. The force attracting the electron is directly proportional to the field potential.

The purpose of the electron gun in the CRT is to produce an electron beam with the following properties:

- It must be accurately focused so that it produces a sharp spot of light where it strikes the phosphor.
- It must have high velocity, since, the brightness of the image depends on the velocity of the electron beam.

- Means must be provided to control the flow of electrons so that the intensity of the trace of the beam can be controlled.

Electrons are generated by a cathode heated by an electric filament. Surrounding the cathode is a cylindrical metal control grid, with a hole at one end that allows electrons to escape. The control grid is kept at a lower potential than the cathode, creating an electrostatic field that directs the electrons through a point source; this simplifies the subsequent focusing process. By altering the control grid potential, we can modify the rate of flow of electrons, or beam current and can thus control the brightness of the image; we can even cut off the flow of electrons altogether. Focusing is achieved by a focusing structure, used to focus finely and highly concentrated at the precise moment at which it strikes the phosphor. An accelerating structure is generally combined with the focusing structure. It consists of two metal plates mounted perpendicular to the beam axis with holes at their centres through which the beam can pass. The two plates are maintained at a sufficiently high relative potential to accelerate the beam to the necessary velocity; accelerating potentials of several thousand volts are not uncommon. The resulting electron gun structure has the advantage that it can be built as a single physical unit and mounted inside the CRT envelope. Other types of gun exist, whose focusing is performed by a coil mounted outside the tube; this is called electromagnetic focusing.

The Deflection System

A set of coils or yoke, mounted at the neck of the tube, forms part of the deflection system responsible for addressing

in the CRT. Two pairs of coils are used, one to control horizontal deflection and the other for vertical. A primary requirement of the deflection system is that it deflects rapidly, since, speed of deflection determines how much information can be displayed without flicker. To achieve fast deflection, we must use large amplitude currents in the yoke. An important part of the deflection system is therefore the set of amplifiers that convert the small voltages received from the display controller into currents of the appropriate magnitude. The voltages used for deflection are generated by the display controller from digital values provided by the computer. These values normally represent coordinates that are converted into voltages by digital to analog conversion. To draw a vector a pair of gradually changing voltages must be generated for the horizontal and vertical deflection coils.

Phosphors

The phosphors used in a graphic display are normally chosen for their colour characteristics and persistence. Ideally the persistence, measured as the time for the brightness to drop to one tenth of its initial value, should last about 100 milliseconds or less allowing refresh at 30Hz rates without noticeable smearing as the image moves. Colour should preferably be white, particularly for applications where dark information appears on a light background. The phosphor should also possess a number of other attributes: small grain size for added resolution, high efficiency in terms of electric energy converted to light and resistance to burning under prolonged excitation.

In attempts to improve performance in one or another of these respects, many different phosphors have been

produced, using various compounds of calcium, cadmium and zinc together with traces of rare earth elements. These phosphors are identified by a numbering system like P1, P4, P7 *etc.*

Raster-scan Displays

The most common type of graphics monitor employing a CRT is the raster scan display. In a raster-scan system, the electron beam is swept across the screen, one row at a time from top to bottom. As the electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots. Picture definition is stored in a memory area called the refresh buffer or frame buffer. This memory area holds the set of intensity values for all the screen points. Stored intensity values are then retrieved from the refresh buffer and painted on the screen one row at a time.

Each screen point is referred to as a pixel or pel (picture element). The capability of a raster-scan system to store intensity information for each screen point makes it well suited for the realistic display of scenes. Home televisions and printers are examples of other systems using raster-scan methods.

Intensity range for pixel positions depends on the capability of the raster system. In a simple black and white system, each screen point is either on or off, so only one bit per pixel is needed to control the intensity of screen positions. Here 1 indicates that the electron beam is to be turned on at that position, and value 0 indicates that the electron beam intensity is to be off. Additional bits are needed when colour and intensity variations can be displayed. Up to 24 bits per pixel are included in high quality systems, which can require

several megabytes of storage for the frame buffer, depending on the resolution of 1024 by 1024 requires 3 megabytes of storage for the frame buffer. On a black and white system with one bit per pixel, the frame buffer is commonly called a bitmap. For systems with multiple bits per pixel, the frame buffer is often referred to as a pixmap.

Refreshing on raster-scan displays is carried out at the rate of 60 to 80 frames per second. At the end of each scan line, the electron beam returns to the left side of the screen to begin displaying the next scan line. The return to the left of the screen, after refreshing each scan line, is called the horizontal retrace of the electron beam and at the end of each frame the electron beam returns to the left corner of the screen to begin the next frame. On some raster-scan systems, each frame is displayed in two passes using an interlaced refresh procedure. In the first pass, the beam sweeps across every other scan line from top to bottom. Then after the vertical retrace, the beam sweeps out the remaining scan lines.

Random-scan Displays

When operated as a random-scan display unit, a CRT has the electron beam directed only to the parts of the screen where a picture is to be drawn. Random-scan monitors draw a picture one line at a time and for this reason are also referred to as vector displays. A pen plotter operates in a similar way and is an example of a random-scan, hard copy device. Refresh rate on a random-scan system depends on the number of lines to be displayed. Picture definition is now stored as a set of line drawing commands in an area of memory referred to as the refresh display file. Sometime the

refresh display file is called the display list or display Programme or refresh buffer. To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn. After all line drawing commands have been processed, the system cycles back to the first line command in the list.

Colour CRT Monitors

A CRT monitor displays colour pictures by using a combination of phosphors that emit different-coloured light. By combining the emitted light from the different phosphors, a range of colours can be generated. The two basic techniques for producing colour displays with a CRT are the *beam-penetration method* and the *shadow-mask method*.

The beam-penetration method for displaying colour pictures has been used with random-scan monitors. Two layers of phosphor, usually red and green are coated onto the inside of the CRT screen, and the displayed colour depends on how far the electron beam penetrates into the phosphor layers. A beam of slow electrons excites only the outer red layer. A beam of very fast electrons penetrates through the red layer and excites the inner green layer. At intermediate beam speeds, combinations of red and green light are emitted to show two additional colours, orange and yellow. The speed of the electrons and hence the screen colour at any point is controlled by the beam-acceleration voltage. Four colours are possible, and the quality of pictures is not as good as with other methods.

Shadow-mask methods are commonly used in raster-scan systems because they produce a much wider range of colours than the beam-penetration method. A shadow-mask CRT

has three phosphor colour dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. This type of CRT has three electron guns, one for each colour dot and a shadow-mask grid just behind the phosphor-coated screen. We obtain colour variations in a shadow-mask CRT by varying the intensity levels of the three electron beams. By turning off the red and green guns, we get only the colour coming from the blue phosphor. A white area is the result of activating all three dots with equal intensity.

Direct-View StorageTubes

An alternative method for maintaining a screen image is to store the picture information inside the CRT instead of refreshing the screen. A direct-view storage tube (DVST) stores the picture information as a charge distribution just behind the phosphor-coated screen. Two electron guns are used in a DVST. One, the primary gun, is used to store the picture pattern; the second, the flood gun, maintains the picture display. A DVST monitor has both disadvantages and advantages compared to the refresh CRT. Because no refreshing is needed, very complex pictures can be displayed at very high resolutions without flicker.

The disadvantages of DVST systems are that they ordinarily do not display colour and that selected parts of a picture cannot be erased. The entire screen must be erased and the modified picture redrawn. The erasing and redrawing process can take several seconds for a complex picture.

Flat-Panel Displays

The term flat-panel display refers to a class of video devices

that have reduced volume, weight and power requirements compared to a CRT. Flat panel displays into two categories: emissive displays and noon-emissive displays. The emissive displays are devices that convert electrical energy into light. Plasma panels, thin-film electroluminescent displays, and light emitting diodes are examples of emissive displays. Non--emissive displays use optical effects to convert sunlight or light from some other source into graphics patterns. The most important example of a non--emissive flat-panel display is a liquid crystal device.

Plasma panels also called gas-discharge displays are constructed by filling the region between two glass plates with a mixture of gases that usually includes neon. A series of vertical conducting ribbons is placed on one glass panel, and a set of horizontal ribbons is built into the other glass panel. Firing voltages applied to a pair of horizontal and vertical conductors cause the gas at the intersection of the two conductors to break down into glowing plasma of electrons and ions. Picture definition is stored in a refresh buffer, and the firing voltages are applied to refresh the pixel positions 60 times per second. One disadvantage of plasma panels has been that they were strictly monochromatic devices, but systems have been developed that are now capable of displaying colour and grayscale.

LCD Technology

Borrowing technology from laptop manufacturers, some companies provide LCD (Liquid Crystal Display) displays. LCDs have low glare flat screens and low power requirements. The colour quality of an active matrix LCD panel actually

exceeds that of most CRT displays. At this point, however, LCD screens usually are more limited in resolution than typical CRTs and are much more expensive. There are three basic LCD choices.

They are…….

- Passive matrix monochrome.
- Passive matrix colour.
- Active matrix colour.

In a LCD, a polarizing filter creates two separate light waves. In a colour LCD, there is an additional filter that has three cells per each pixels – one each for displaying red, green and blue.

The light wave passes through a liquid crystal cell, with each colour segment having its own cell. The liquid crystals are rod-shaped molecules that flow like a liquid. They enable light to pass straight through them. Although monochrome LCDs do not have colour filters, they can have multiple cells per pixel for controlling shades of grey.

In passive matrix LCD, each cell is controlled by electrical charges transmitted by transistors according to row and column positions on the screen's edge. As the cell reacts to the pulsing charge, it twists the light wave, with stronger charges twisting the light wave more. In an active matrix LCD, each cell has its own transistor to charge it and twist the light wave. This provides brighter image than passive matrix displays because, the cell can maintain a constant, rather than momentary charge. However, active matrix technology uses more energy than passive matrix. With a dedicated transistor for every cell, active matrix displays are more difficult and expensive to produce.

In both active and passive matrix LCDs, the second polarizing filter controls how much light passes through each cell. Cells twist the wavelength of light that passes through the filter at each cell, the brighter the pixel. The best colour displays are active matrix or thin film transistor panels, in which each pixel is controlled by three transistors for red, green and blue.

Raster-scan Systems

Interactive raster graphics systems typically employ several processing units. In addition to the central processing unit, a special-purpose processor, called the video controller or display controller is used to control the operation of the display device. The video controller accesses the frame buffer to refresh the screen. In addition to the video controller, more sophisticated raster systems employ other processors as co-processors and accelerators to implement various graphics operations.

Video Controller

Frame buffer locations, and the corresponding screen positions, are referenced in Cartesian co-ordinates. For many graphics monitors, the co-ordinate origin is defined at the lower left screen corner. The screen surface is then represented as the first quadrant of a two-dimensional system, with positive x values increasing to the right and positive y values increasing from bottom to top. Scan lines are then labelled from y_{max} at the top of the screen to 0 at the bottom. Along each scan line, screen pixel positions are labelled from 0 to x_{max}. Two registers are used to store the co-ordinates of the screen pixels. Initially, the x register is

set to 0 and the y register is set to ymax. The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam. Then the x register is incremented by 1, and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the scan line. After the last pixel on the top scan line has been processed, the x register is reset to 0 and the y register is decremented by 1. Pixels along this scan line are then processed in turn, and the procedure is repeated for each successive scan line. After cycling through all pixels along the bottom scan line (y = 0), the video controller resets the registers to the first pixel position on the top scan line and the refresh process starts over.

Raster-scan Display Processor

The organization of raster system containing a separate display processor, sometimes referred to as a graphics controller or display co-processor. The purpose of the display processor is to free the CPU from the graphics chores. In addition to the system memory, a separate display processor memory area can also be provided. A major task of the display processor is digitizing a picture definition given in an application Programme into a set of pixel-intensity values for storage in the frame buffer. This digitization process is called scan conversion.

Characters can be defined with rectangular grids. The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher quality displays. Display processors are typically designed to interface with interactive input devices such as mouse.

In an effort to reduce memory requirements in raster systems, methods have been devised for organizing the frame buffer as a linked list and encoding the intensity information. One way to do this is to store each scan line as a set of integer pairs.

One number of each pair indicates an intensity value, and the second number specifies the number of adjacent pixels on the scan line that are to have that intensity. This technique called run-length encoding. A similar approach can be taken when pixel intensities change linearly. Another approach is to encode the raster as a set of rectangular areas (cell encoding).

Random-scan Systems

The organization of a simple random-scan system. An application Programme is input and stored in the system memory along with a graphics package. Graphics commands in the application Programme are translated by the graphics package into a display file stored in the system memory. This display file is then accessed by the display processor to refresh the screen. The display processor cycles through each command in the display file Programme once during every refresh cycle. Sometimes the display processor in a random-scan system is referred to as a display processing or a graphics controller.

Lines are defined by the values for their co-ordinate endpoints, and these input co-ordinate values are converted to x and y deflection voltages. A scene is then drawn one line at a time by positioning the beam to fill in the line between specified endpoints.

Using the acm.graphics Package

A simple example of how to write graphical programmes, but does not explain the details behind the methods it contains. The purpose of this chapter is to give you a working knowledge of the facilities available in the acm.graphics package and how to use them effectively.

The class structure of acm.graphics package appears. Most of the classes in the package are subclasses of the abstract class GObject at the centre of the diagram. Conceptually, GObject represents the universal class of graphical objects that can be displayed. When you use acm.graphics, you assemble a picture by constructing various GObjects and adding them to a GCanvas at the appropriate locations. The general model in more detail offer a closer look at the individual classes in the package.

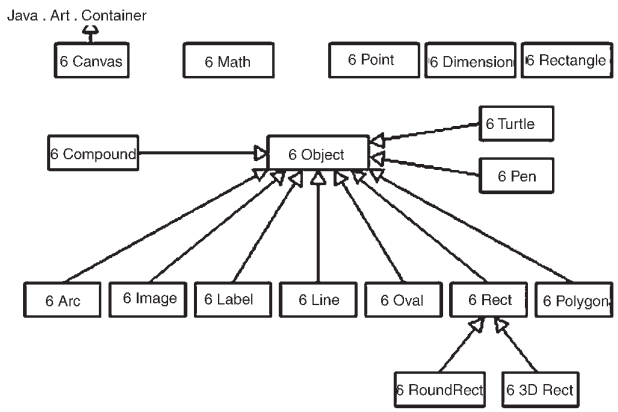


Fig. Class Diagram for the acm.graphics Package

The acm.graphics Model

When you create a picture using the acm.graphics package, you do so by arranging graphical objects at various positions on a background called a canvas. The underlying model is similar to that of a collage in which an artist creates a

composition by taking various objects and assembling them on a background canvas. In the world of the collage artist, those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. In the acm.graphics package, there are counterparts for each of these graphical objects.

The "Felt Board" Metaphor

Another metaphor that often helps students understand the conceptual model of the acm.graphics package is that of a felt board—the sort one might find in an elementary school classroom. A child creates pictures by taking shapes of coloured felt and sticking them onto a large felt board that serves as the background canvas for the picture as a whole. The pieces stay where the child puts them because felt fibres interlock tightly enough for the pieces to stick together. A physical felt board with a red rectangle and a green oval attached. The right side of the figure is the virtual equivalent in the acm.graphics world. To create the picture, you would need to create two graphical objects—a red rectangle and a green oval—and add them to the graphical canvas that forms the background.

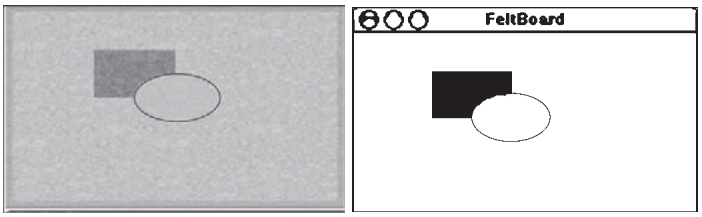


Fig. Physical Felt Board and its Virtual Equivalent

The code for the FeltBoard example appears. Even though you have not yet had a chance to learn the details of the various classes and methods used in the programme, the overall

framework should nonetheless make sense. The programme first creates a rectangle, indicates that it should be filled rather than outlined, colours it red, and adds it to the canvas. It then uses almost the same operations to add a green oval. Because the oval is added after the rectangle, it appears to be in front, obscuring part of the rectangle underneath. This behaviour, of course, is exactly what would happen with the physical felt board. Moreover, if you were to take the oval away by calling

```
remove(oval);
```

the parts of the underlying rectangle that had previously been obscured would reappear.

In this tutorial, the order in which objects are layered on the canvas will be called the stacking order. (In more mathematical descriptions, this ordering is often called *z*-ordering, because the *z*-axis is the one that projects outward from the screen.) Whenever a new object is added to a canvas, it appears at the front of the stack. Graphical objects are always drawn from back to front so that the frontmost objects overwrite those that are further back.

```
/*
 * File: FeltBoard.java
 * ____________
  * This programme offers a simple example of the
acm.graphics package
  * that draws a red rectangle and a green oval. The
dimensions of
  * the rectangle are chosen so that its sides are in
proportion to
  * the "golden ratio" thought by the Greeks to represent
the most
  * aesthetically pleasing geometry.
  */

import acm.programme.*;
import acm.graphics.*;
import java.awt.*;
```

```
  public class FeltBoard extends GraphicsProgram {

  /** Runs the programme */
     public void run() {
       GRect rect = new GRect(100, 50, 100, 100 / PHI);
       rect.setFilled(true);
        rect.setColor(Color.RED);
        add(rect);
       GOval oval = new GOval(150, 50 + 50 / PHI, 100, 100 /
PHI);
        oval.setFilled(true);
        oval.setColor(Color.GREEN);
        add(oval);
     }

  /** Constant representing the golden ratio */
      public static final double PHI = 1.618;

  }
```

Programme : Code for the Felt Board.

The GCanvas Class

In the acm.graphics model, pictures are created by adding graphical objects—each of which is an instance of the GObject class to a background canvas. That background—the analogue of the felt board in the physical world—is provided by the GCanvas class. The GCanvas class is a lightweight component and can be added to any Java container in either the java.awt or javax.swingpackages, which makes it possible to use the graphics facilities in any Java application. For the most part, however, students in introductory courses won't use the GCanvas class directly but will instead use theGraphicsProgram class, which automatically creates a GCanvas and installs it in the programme window, as illustrated in several preceding examples. The GraphicsProgram class forwards operations such as add

andremove to the embedded GCanvas so that students don't need to be aware of the underlying implementation details.

The most important methods supported by the GCanvas class. Many of these methods are concerned with adding and removing graphical objects. These methods are easy to understand, particularly if you keep in mind that a GCanvas is conceptually a container for GObject values. The container metaphor explains the functionality provided by the add, remove, and removeAll, which are analogous to the identically named methods in JComponent and Container.

Constructor

`new GCanvas()`

Creates a new GCanvas containing no graphical objects.

Methods to Add and Remove Graphical Objects from a Canvas

`void add(GObject gobj)`

Adds a graphical object to the canvas at its internally stored location.

`void add(GObject gobj, double x, double y)` *or* `add(GObject gobj, GPoint pt)`

Adds a graphical object to the canvas at the specified location.

`void remove(GObject gobj)`

Removes the specified graphical object from the canvas.

`void removeAll()`

Removes all graphical objects and components from the canvas.

Method to Find the Graphical Object at a Particular Location

`GObject getElementAt(double x, double y)` *or* `getElementAt(GPoint pt)`

Returns the topmost object containing the specified point, or null if no such object exists.

Useful Methods Inherited from Superclasses

```
int getWidth()
```

Return the width of the canvas, in pixels.

```
int getHeight()
```

Return the height of the canvas, in pixels.

```
void setBackground(Color bg)
```

Changes the background colour of the canvas.

The add method comes in two forms, one that preserves the internal location of the graphical object and one that takes an explicit *x* and *y* coordinate. Each method has its uses, and it is convenient to have both available. The first is useful particularly when the constructor for the GObject specifies the location, as it does, for example, in the case of the GRect class. If you wanted to create a 100 x 60 rectangle at the point (75, 50), you could do so by writing the following statement:

```
add(new GRect(75, 50, 100, 60));
```

The second form is particularly useful when you want to choose the coordinates of the object in a way that depends on other properties of the object. For example, the following code taken from the HelloGraphicsexample centres a GLabel object in the window:

```
GLabel label = new GLabel("hello, world");
double x = (getWidth() - label.getWidth()) / 2;
double y = (getHeight() + label.getAscent()) / 2;
add(label, x, y);
```

Because the placement of the label depends on its dimensions, it is necessary to create the label first and then add it to a particular location on the canvas.

The GCanvas method getElement(x, y) returns the graphical object on the canvas that includes the point (x, y).

If there is more than one such object, getElement returns the one that is in front of the others in the stacking order; if there is no object at that position, getElement returns null. This method is useful, for example, if you need to select an object using the mouse. Several of the most useful methods in the GCanvas class are those that are inherited from its superclasses in Java's component hierarchy. For example, if you need to determine how big the graphical canvas is, you can call the methods getWidth and getHeight.

Thus, if you wanted to define a GPoint variable to mark the centre of the canvas, you could do so with the following declaration:

```
GPoint centre = new GPoint(getWidth() / 2.0, getHeight()
/ 2.0);
```

You can also change the background colour by calling setBackground(bg), where bg is the new background colour for the canvas.

The GObject Class

The GObject class represents the universe of graphical objects that can be displayed on a GCanvas. The GObject class itself is abstract, which means that programmes never create instances of the GObject class directly. Instead, programmes create instances of one of the GObject subclasses that represent specific graphical objects such as rectangles, ovals, and lines.

The most important such classes are the ones that appear at the bottom of the class diagram, which are collectively called the shape classes. Before going into those details, however, it makes sense to begin by describing the characteristics that are common to the GObject class as a whole.

Methods common to all GObject Subclasses

All GObjects—no matter what type of graphical object they represent—share a set of common properties. For example, all graphical objects have a *location*, which is the *x* and *y* coordinates at which that object is drawn. Similarly, all graphical objects have a *size*, which is the width and height of the rectangle that includes the entire object. Other properties common to all GObjects include their colour and how the objects are arranged in terms of their stacking order. Each of these properties is controlled by methods defined at the GObject level.

4

Java Package

In simple it is a way of categorising the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorising these classes is a must as well as makes life much easier.

Import Statements

In java if a fully qualified name, which includes the package and the class name, is given then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example following line would ask compiler to load all the classes available in directory java_installation/java/io:

```
import java. io. *;
```

A Simple Case Study

For our case study we will be creating two classes. They are Employee and EmployeeTest. First open notepad and add the

following code. Remember this is the Employee class and the class is a public class. Now save this source file with the name Employee. java. The Employee class has four class variables name, age, designation and salary. The class has one explicitly defined constructor which takes a parameter.

```
import java. io. *;
public class Employee{
String name;
int age;
String designation;
double salary;

// This is the constructor of the class Employee
public Employee(String name){
this. name = name;
}
// Assign the age of the Employee to the variable age.
public void empAge(int empAge){
age = empAge;
}
/* Assign the designation to the variable designation. */
public void empDesignation(String empDesig){
designation = empDesig;
}
/* Assign the salary to the variable salary. */
public void empSalary(double empSalary){
salary = empSalary;
}
/* Print the Employee details */
public void printEmployee(){
System. out. println("Name:"+ name );
System. out. println("Age:" + age );
System. out. println("Designation:" + designation );
System. out. println("Salary:" + salary);
}
}
```

As mentioned previously in this tutorial processing starts from the main method. Therefore in-order for us to run this Employee class there should be main method and objects should be created. We will be creating a separate class for these tasks. Given below is

the *EmployeeTest* class which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest. java file

```
import java. io. *;
public class EmployeeTest{
public static void main(String args[]){
/* Create two objects using constructor */
Employee empOne = new Employee("James Smith");
Employee empTwo = new Employee("Mary Anne");
// Invoking methods for each object created
empOne. empAge(26);
empOne. empDesignation("Senior Software Engineer");
empOne. empSalary(1000);
empOne. printEmployee();
empTwo. empAge(21);
empTwo. empDesignation("Software Engineer");
empTwo. empSalary(500);
empTwo. printEmployee();
}
}
```

Now compile both the classes and then run EmployeeTest to see the result as follows:

```
C:> javac Employee. java
C:> vi EmployeeTest. java
C:> javac EmployeeTest. java
C:> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000. 0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500. 0
```

Fields, Methods, and Access Levels

- Java classes contain *fields* and *methods*. A field is like a C++ data member, and a method is like a C++ member function.
- *Each field and method has an access level:*

  - *Private:* Accessible only in this class.
  - *(package):* Accessible only in this package.
  - *Protected:* accessible only in this package and in all subclasses of this class.
  - *Public:* accessible everywhere this class is available
- *Similarly, each class has one of two possible access levels:*
  - *(Package):* Class objects can only be declared and manipulated by code in this package
  - *Public:* Class objects can be declared and manipulated by code in any package

Note: For both fields and classes, package access is the default, and is used when *no* access is specified.

Simple Example Class

Here's a (partial) example class; a List is an ordered collection of items of any type:

```
class List {
// fields
private Object [] items; // store the items in an array
private int numItems;// the current # of items in the
list
// methods
// constructor function
public List()
{
items = new Object[10];
  numItems = 0;
}

 // AddToEnd: add a given item to the end of the list
 public void AddToEnd(Object ob)
 {. . . }
}
```

Notes:

- *Object:* Object-oriented programming involves *inheritance*. In Java, all classes (built-in or user-defined) are (implicitly) subclasses of Object. Using an array of Object in the List class allows any kind of Object (an instance of any class)

to be stored in the list. However, primitive types (int, char, etc) cannot be stored in the list.

- *Constructor Function: As in C++, constructor functions in Java:*
  - Are used to initialise each instance of a class.
  - Have no return type (not even void).
  - Can be overloaded; you can have multiple constructor functions, each with different numbers and/or types of arguments.

If you don't write any constructor functions, a default (no-argument) constructor (that doesn't do anything) will be supplied. If you write a constructor that takes one or more arguments, no default constructor will be supplied (so an attempt to create a new object without passing any arguments will cause a compile-time error).

It is often useful to have one constructor call another (for example, a constructor with no arguments might call a constructor with one argument, passing a default value). The call must be the *first* statement in the constructor. It is performed using *this* as if it were the name of the method.

For example:

```
this( 10 );
```

is a call to a constructor that expects one integer argument.

- *Initialisation of Fields:* If you don't initialise a field (*i. e.* , either you don't write any constructor function, or your constructor function just doesn't assign a value to that field), the field will be given a default value, depending on its type. The values are the same as those used to initialise newly created arrays (see the "Java vs C++" notes).
- *Access Control:* Note that the access control must be specified for every field and every method; there is no

grouping as in C++. For example, given these declarations:

```
public
   int x;
   int y;
only x is public; y gets the default, package access.
```

Static Fields and Methods

- Fields and methods can be declared *static* (this is true in C++, too). If a field is static, there is only one copy for the entire class, rather than one copy for each instance of the class. (In fact, there is a copy of the field even if there are *no* instances of the class.)

For example, we could add the following field to the List class:

```
static int numLists = 0;
```

And the following statement to the List constructor:

```
numLists++;
```

Now every time a new List object is created, the numLists variable is incremented; so it maintains a count of the total number of Lists created during programme execution. Every instance of a List could access this variable (could both read it and write into it), and they would all be accessing the *same* variable, not their own individual copies. A method should be made static when it does not access any of the non-static fields of the class, and does not call any non-static methods. (In fact, a static method *cannot* access non-static fields or call non-static methods.) Methods that would be "free" functions in C++ (*i. e.* , not members of any class) should be static methods in Java. Also, methods that are logically associated with a particular class, but that only manipulate the class's static fields should be static.

For example, if we wanted a function to print the current value of the numLists field defined above, that function should be defined as a static method of the List class.

A public static field or method can be accessed from outside the class using either the usual notation:

```
class-object. field-or-method-name
```

or using the class name instead of the name of the class object:

```
class-name. field-or-method-name
```

For example, if the numLists field is public, and there is a variable L of type List, the numLists field can be accessed using either L. numLists or List. numLists. Similarly, if the List class includes a public static method PrintNumLists, then the method can be called using either L. PrintNumLists() or List. PrintNumLists(). The *preferred* way to access a static field or a static method is using the class name (not using a class object). This is because it makes it clear that the field or method being accessed is static.

Final Fields and Methods

- Fields and methods can also be declared *final*. A final method cannot be overridden in a subclass. A final field is like a constant: once it has been given a value, it cannot be assigned to again. For example, the constructor function for the List class initialises the "items" field to (point to) an array of size 10. It would probably be better to use a constant for the initial size of the array.

Only a single copy of the constant is needed for the whole class, not one for every class instance, so it would be appropriate to make the field static as well as final:

```
private static final int INITIAL_SIZE = 10;
```

The assignment statement in the constructor function would change to:

```
items = new Object[INITIAL_SIZE];
```

Objects

When you look at the world around you, what do you see? You see objects. In programming, objects are nothing more than representations of things. It's actually pretty simple. The world around you is a representation of stuff. You look at something, and you can probably tell me its characteristics. It's an object. It may be more specific than that, it might be a car object or a keyboard object or a person object, but it is still an object. Java has a slightly different definition but it is almost identical to a real-life object.

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behaviour*. Dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behaviour (changing gear, changing pedal cadence, applying brakes). Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behaviour can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviourss (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and

behaviour (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

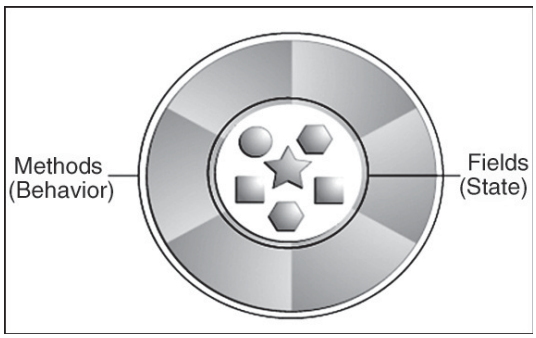


Fig. A Software Object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in *fields* (variables in some programming languages) and exposes its behaviour through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation*—a fundamental principle of object-oriented programming.

Consider a bicycle, for example:

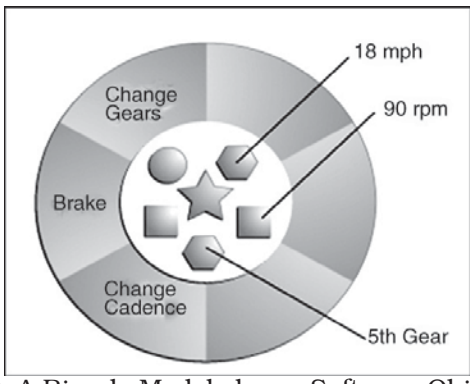


Fig. A Bicycle Modeled as a Software Object.

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

- *Modularity:* The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- *Information-hiding:* By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- *Code re-use:* If an object already exists (perhaps written by another software developer), you can use that object in your programme. This allows specialists to implement/ test/debug complex, task-specific objects, which you can then trust to run in your own code.
- *Pluggability and Debugging Ease:* If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System. out. println method, for example, the system actually executes several

statements in order to display a message on the console. Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the programme design.

Creating a Method

In general, a method has the following syntax:

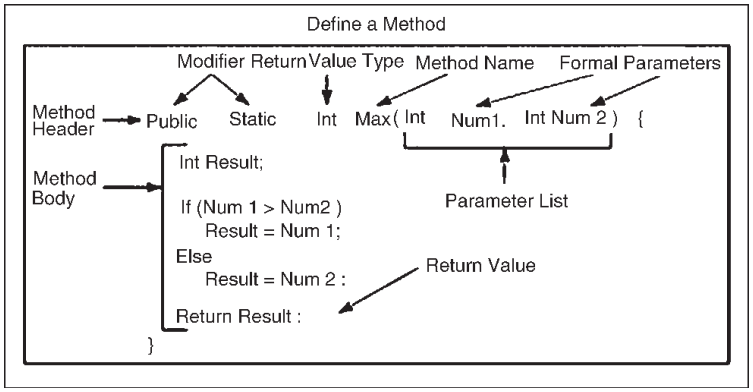
```
modifier returnValueType methodName(list of parameters) {
// Method body;
}
```

A method definition consists of a method header and a method body.

Here are all the parts of a method:

- *Modifiers:* The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.
- *Return Type:* A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword void.
- *Method Name:* This is the actual name of the method. The method name and the parameter list together constitute the method signature.
- *Parameters:* A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

- *Method Body:* The method body contains a collection of statements that define what the method does.



Note: In certain other languages, methods are referred to as procedures and functions. A method with a nonvoid return value type is called a function; a method with a void return value type is called a procedure.

Example

Here is the source code of the above defined method called max().

This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** Return the max between two numbers */
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Calling a Method

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the

method returns a value or not. When a programme calls a method, programme control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(30, 40);
```

If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

```
System. out. println("Welcome to Java!");
```

Example

Following is the example to demonstrate how to define a method and how to call it:

```
public class TestMax {
/** Main method */
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System. out. println("The maximum between " + i +
" and " + j + " is " + k);
}
/** Return the max between two numbers */
public static int max(int num1, +int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

This would produce following result:

```
The maximum between 5 and 2 is 5
```

This programme contains the main method and the max method. The main method is just like any other method except

that it is invoked by the JVM. The main method's header is always the same, like the one in this example, with the modifiers public and static, return value type void, method name main, and a parameter of the String[] type. String[] indicates that the parameter is an array of String.

The Void Keyword

This section shows how to declare and invoke a void method. Following example gives a programme that declares a method named printGrade and invokes it to print the grade for a given score.

Example

```
public class TestVoidMethod {
public static void main(String[] args) {
printGrade(78. 5);
}
public static void printGrade(double score) {
if (score >= 90. 0) {
System. out. println('A');
}
else if (score >= 80. 0) {
System. out. println('B');
}
else if (score >= 70. 0) {
System. out. println('C');
}
else if (score >= 60. 0) {
System. out. println('D');
}
else {
System. out. println('F');
}
}
}
```

This would produce following result:

C

Here the printGrade method is a void method. It does not return any value. A call to a void method must be a statement. So, it is invoked as a statement in line 3 in the main method. This statement is like any Java statement terminated with a semicolon.

Passing Parameters by Values

When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association.

For example, the following method prints a message n times:

```
public static void nPrintln(String message, int n) {
for (int i = 0; i < n; i++)
System. out. println(message);
}
```

Here, you can use nPrintln("Hello", 3) to print "Hello" three times. The nPrintln("Hello", 3) statement passes the actual string parameter, "Hello", to the parameter, message; passes 3 to n; and prints "Hello" three times. However, the statement nPrintln(3, "Hello") would be wrong. When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means passing the value of x to y.

Example

Following is a programme that demonstrates the effect of passing by value. The programme creates a method for swapping

two variables. The swap method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

```
public class TestPassByValue {
public static void main(String[] args) {
int num1 = 1;
int num2 = 2;
System. out. println("Before swap method, num1 is " +
num1 + " and num2 is " + num2);
// Invoke the swap method
swap(num1, num2);
System. out. println("After swap method, num1 is " +
num1 + " and num2 is " + num2);
}
/** Method to swap two variables */
public static void swap(int n1, int n2) {
System. out. println("\tInside the swap method");
System. out. println("\t\tBefore swapping n1 is " + n1
+ " n2 is " + n2);
// Swap n1 with n2
int temp = n1;
n1 = n2;
n2 = temp;
System. out. println("\t\tAfter swapping n1 is " + n1
+ " n2 is " + n2);
}
}
```

This would produce following result:

```
Before swap method, num1 is 1 and num2 is 2
Inside the swap method
Before swapping n1 is 1 n2 is 2
After swapping n1 is 2 n2 is 1
After swap method, num1 is 1 and num2 is 2
```

Overloading Methods

The max method that was used earlier works only with the int data type. But what if you need to find which of two floating-point numbers has the maximum value?

The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2) {
if (num1 > num2)
return num1;
else
return num2;
}
```

If you call max with int parameters, the max method that expects int parameters will be invoked; if you call max with double parameters, the max method that expects double parameters will be invoked. This is referred to as method overloading; that is, two methods have the same name but different parameter lists within one class. The Java compiler determines which method is used based on the method signature. Overloading methods can make programmes clearer and more readable. Methods that perform closely related tasks should be given the same name.

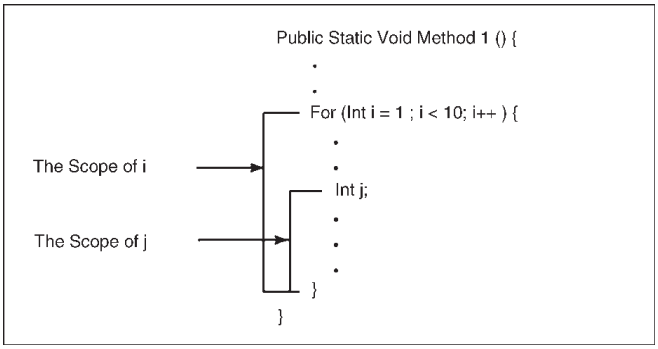
Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types. Sometimes there are two or more possible matches for an invocation of a method due to similar method signature, so the compiler cannot determine the most specific match. This is referred to as ambiguous invocation.

The Scope of Variables

The scope of a variable is the part of the programme where the variable can be referenced. A variable defined inside a method is referred to as a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method. A variable declared in the initial action part of a for loop header has its scope in the entire loop. But

a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable as shown below:



You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

Using Command-Line Arguments

Sometimes you will want to pass information into a programme when you run it. This is accomplished by passing command-line arguments to main( ). A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java programme is quite easy. They are stored as strings in the String array passed to main( ).

Example

The following programme displays all of the command-line arguments that it is called with:

```
class CommandLine {
public static void main(String args[]){
for(int i=0; i<args. length; i++){
System. out. println("args[" + i + "]: " +
args[i]);
}
}
```

```
}
```

Try executing this programme, as shown here:

```
java CommandLine this is a command line 200 -100
```

This would produce following result:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

The Constructors

A constructor initialises an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initialises all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass {
int x;
// Following is the constructor
MyClass() {
x = 10;
}
}
```

You would call constructor to initialise objects as follows:

```
class ConsDemo {
public static void main(String args[]) {
```

```
MyClass t1 = new MyClass();
MyClass t2 = new MyClass();
System. out. println(t1. x + " " + t2. x);
}
}
```

Most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method:just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass {
int x;
// Following is the constructor
MyClass(int i ) {
x = i;
}
}
```

You would call constructor to initialise objects as follows:

```
class ConsDemo {
public static void main(String args[]) {
MyClass t1 = new MyClass( 10 );
MyClass t2 = new MyClass( 20 );
System. out. println(t1. x + " " + t2. x);
}
}
```

This would produce following result:

```
10 20
```

Variable Arguments(var-args)

JDK 1. 5 enables you to pass a variable number of arguments of the same type to a method.

The parameter in the method is declared as follows:

```
typeName. . . parameterName
```

In the method declaration, you specify the type followed by an ellipsis (. . .) Only one variable-length parameter may be

specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example

```
public class VarargsDemo {
public static void main(String args[]) {
// Call method with variable args
  printMax(34, 3, 3, 2, 56. 5);
printMax(new double[]{1, 2, 3});
}
public static void printMax( double. . . numbers) {
if (numbers. length == 0) {
System. out. println("No argument passed");
return;
}
double result = numbers[0];
for (int i = 1; i < numbers. length; i++)
if (numbers[i] > result)
result = numbers[i];
System. out. println("The max value is " + result);
}
}
```

This would produce following result:

```
The max value is 56. 5
The max value is 3. 0
```

The finalise( ) Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called finalise( ), and it can be used to ensure that an object terminates cleanly.

For example, you might use finalise( ) to make sure that an open file owned by that object is closed. To add a finaliser to a class, you simply define the finalise( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class. Inside the finalise( ) method you will specify those actions that must be performed before an object is destroyed.

The finalise( ) method has this general form:

```
protected void finalise( )
{
// finalisation code here
}
```

Here, the keyword protected is a specifier that prevents access to finalise( ) by code defined outside its class.

This means that you cannot know when. or even if. finalise( ) will be executed. For example, if your programme ends before garbage collection occurs, finalise( ) will not execute.

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalised.

Here are some examples:

```
run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty
```

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the

same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type.

It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on.

In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
. . .
public void draw(String s) {
. . .
}
public void draw(int i) {
. . .
}
public void draw(double f) {
. . .
}
public void draw(int i, double f) {
. . .
}
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types. You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Strings

In Java strings are objects designed to represent a sequence of characters. Because character strings are commonly used in programmes, Java supports the ability to declare String constants and perform concatenation of Strings directly without requiring access to methods of the String class. This additional support provided for Java Strings allows programmers to use Strings in a similar manner as other common programming languages.

- A Java String is read-only and once created the contents cannot be modified. Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a String object with its value—in this case, Hello world!. As with any other object, you can create String objects by using the new keyword and a constructor.

The String class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '. ' };
String helloString = new String(helloArray);
System. out. println(helloString);
```

The last line of this code snippet displays hello.

Note: The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since, strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

String Length

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome. length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient programme to reverse a palindrome string. It invokes the String method charAt(i), which returns the i^{th} character in the string, counting from 0.

```
public class StringDemo {
public static void main(String[] args) {
String palindrome = "Dot saw I was Tod";
int len = palindrome. length();
char[] tempCharArray = new char[len];
char[] charArray = new char[len];
// put original string in an
// array of chars
for (int i = 0; i < len; i++) {
tempCharArray[i] =
```

```
palindrome. charAt(i);
}
// reverse array of chars
for (int j = 0; j < len; j++) {
charArray[j] =
tempCharArray[len-1-j];
}
String reversePalindrome =
new String(charArray);
System. out. println(reversePalindrome);
}
}
```

Running the programme produces this output:

```
doT saw I was toD
```

To accomplish the string reversal, the programme had to convert the string to an array of characters (first for loop), reverse the array into a second array (second for loop), and then convert back to a string. The String class includes a method, getChars(), to convert a string, or a portion of a string, into an array of characters so we could replace the first for loop in the programme above with

```
palindrome. getChars(0, len, tempCharArray, 0);
```

Concatenating Strings

The String class includes a method for concatenating two strings:

```
string1. concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

You can also use the concat() method with string literals, as in:

```
"My name is ". concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the + operator, as in

```
"Hello, " + " world" + "!"
```

which results in

```
"Hello, world!"
```

The + operator is widely used in print statements.

For example:

```
String string1 = "saw I was ";
System. out. println("Dot " + string1 + "Tod");
```

which prints

```
Dot saw I was Tod
```

Such a concatenation can be a mixture of any objects. For each object that is not a String, its toString() method is called to convert it to a String.

Note: The Java programming language does not permit literal strings to span lines in source files, so you must use the + concatenation operator at the end of each line in a multi-line string.

For example:

String quote =

```
"Now is the time for all good " +
"men to come to the aid of their country. ";
```

Breaking strings between lines using the + concatenation operator is, once again, very common in print statements.

Creating Format Strings

You have seen the use of the printf() and format() methods to print output with formatted numbers.

The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object. Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of

```
System. out. printf("The value of the float " +
"variable is %f, while " +
"the value of the " +
"integer variable is %d, " +
"and the string is %s",
floatVar, intVar, stringVar);
```

you can write

```
String fs;
```

```
fs = String. format("The value of the float " +
"variable is %f, while " +
"the value of the " +
"integer variable is %d, " +
" and the string is %s",
floatVar, intVar, stringVar);
System. out. println(fs);
```

String Class

The String class is commonly used for holding and manipulating strings of text in Java programmes. It is found in the standard java. lang package which is automatically imported, so you don't need to do anything special to use it. In its simplest form, you use the String class by typing some text surrounded by double quotes.

This is called a String literal.

```
"This is a Java String"
```

Anywhere you need a String object, you can type a String literal as in:

```
Console. println("Hello there!");
```

Concatenation

Strings are special types of objects in that you can add them together using the "+" operator and get a longer string that is a concatenation of the two.

Thus, the code:

```
String name = "Bill";
String introduction = "Hello there, my name is " + name;
Console. println( introduction );
will print Hello there, my name is Bill
```

on the console.

String representation of Objects

All Java objects implement the method toString() which returns a String object that best "describes" that object. For example, if you can print out a red colour object using

```
Colour colour = new Colour( 255, 0, 0 );
 String colorStr = colour. toString();
Console. println( colorStr );
```

This will printout:

```
java. awt. Colour[r=255, g=0, b=0]
```

In this case you do not even need to explicitly call toString(). The println() method has a version that takes an Object as its argument. This version will implicitly call toString() for you and print the result.

This implicit conversion to a String also happens when concatenating a String object with another object type.

For example, you could type:

```
Colour colour = new Colour( 255, 0, 0 );
String str = "My colour looks like: " + colour;
Console. println( str );
```

And you would get the following on the console:

```
My colour looks like: java. awt. Colour[r=255, g=0, b=0]
```

String API

It would probably be useful for you to at least scan the API to see what other interesting things you can do with Strings.

There are methods for:

- Getting the character at a given position within the string—charAt().
- Seeing if a character or string exists within a string—indexOf().
- Getting the number of characters in a string—length().
- Extracting a substring from a string—substring().

5

Distinction from Photorealistic 2D Graphics Design

Not all computer graphics that appear 3D are based on a wireframe model. 2D computer graphics with 3D photorealistic effects are often achieved without wireframe modeling and are sometimes indistinguishable in the final form. Some graphic art software includes filters that can be applied to 2D vector graphics or 2D raster graphics on transparent layers. Visual artists may also copy or visualize 3D effects and manually render photorealistic effects without the use of filters.

Modelling

3D Modelling

Modelling is the process of taking a shape and moulding it into a completed 3D mesh. The most typical means of creating

a 3D model is to take a simple object, called a primitive, and extend or "grow" it into a shape that can be refined and detailed. Primitives can be anything from a single point (called a vertex), a two-dimensional line (an edge), a curve (a spline), to three dimensional objects (faces or polygons). Using the specific features of your chosen 3D software, each one of these primitives can be manipulated to produce an object. When you create a model in 3D, you'll usually learn one method to create your model, and go back to it time and again when you need to create new models. There are three basic methods you can use to create a 3D model, and 3D artists should understand how to create a model using each technique.

- *Spline or Patch Modelling:* A spline is a curve in 3D space defined by at least two control points. The most common splines used in 3D art are bezier curves and NURBS (the software Maya has a strong NURBS modelling foundation.) Using splines to create a model is perhaps the oldest, most traditional form of 3D modelling available. A cage of splines is created to form a "skeleton" of the object you want to create. The software can then create a patch of polygons to extend between two splines, forming a 3D skin around the shape. Spline modelling is not used very often these days for character creation, due to how long it takes to create good models. The models that are produced usually aren't useful for animation without a lot of modification. Spline modelling is used primarily for the creation of hard objects, like cars, buildings, and furniture.

Splines are extremely useful when creating these objects, which may be a combination of angular and curved shapes. When creating a 3D scene that requires curved shapes, spline modelling should be your first choice.

- *Box Modelling:* Box modelling is possibly the most popular technique, and bears a lot of resemblance to traditional sculpting. In box modelling, one starts with a primitive (usually a cube) and begins adding detail by "slicing" the cube into pieces and extending faces of the cube to gradually create the form you're after. People use box modelling to create the basic shape of the model. Once practiced, the technique is very quick to get acceptable results. The downside is that the technique requires a lot of tweaking of the model along the way. Also, it is difficult to create a model that has a surface topology that lends well to animation. Box modelling is useful as a way to create organic models, like characters. Box modellers can also create hard objects like buildings, however precise curved shapes may be more difficult to create using this technique.
- *Poly Modelling/ edge Extrusion:* While it's not the easiest to get started with, poly modelling is perhaps the most effective and precise technique. In poly modelling, one creates a 3D mesh point-by-point, face-by-face. Often one will start out with a single quad (a 3D object consisting of 4 points) and extrude an edge of the quad, creating a second quad attached to the first. The 3D model is created gradually in

this way. While poly modelling is not as fast as box modelling, it requires less tweaking of the mesh to get it "just right," and you can plan out the topology for animation ahead of time. Poly modellers use the technique to create either organic or hard objects, though poly modelling is best suited for organic models.

A Workflow that Works The workflow you choose to create a model will largely depend on how comfortable you are with a given technique, what object you're creating, and what your goals are for the final product. Someone who is creating an architectural scene, for example, may create basic models with cubes and other simple shapes to create an outline of the finished project. Meshes can then be refined or replaced with more detailed objects as you progress through the project. This is an organized, well-planned way to create a scene; it is a strategy used by professionals that makes scene creation straightforward. Beginners, on the other hand, tend to dive in headfirst and work on the most detailed objects first. This is a daunting way to work, and can quickly lead to frustration and overwhelm.

Remember, sketch first, then refine. Likewise, when creating an organic model, beginners tend to start with the most detailed areas first, and flesh out the remaining parts later, a haphazard way to create a character. This may be one reason why box modelling has grown to be so widely popular. A modeller can easily create the complete figure before refining the details, like eyes, lips, and ears. Perhaps the best strategy is to use a hybrid workflow when creating organic models. A well planned organic model is created using

a combination of box modelling and poly modelling. The arms, legs, and torso can be sketched out with box modelling, while the fine details of the head, hands, and feet are poly modelled. This is a compromise professional modellers seek which prevents them from getting bogged down in details.

It can make the difference between a completed character, and one that is never fleshed out beyond the head. Beginners would be wise to follow this advice. Mesh Topology Another aspect of proper workflow is creating a model with an ideal 3D mesh topology. Topology optimization is usually associated with creating models used in animation. Models created without topology that flows in a smooth, circular pattern, may not animate correctly, which is why it is important to plan ahead when creating any 3D object that will be used for animation. The most frequently discussed topology is the proper creation or placement of edgeloops. An edgeloop is a ring of polygons placed in an area where the model may deform, as in the case of animation.

These rings of polygons are usually placed around areas where muscles might be, such as in the shoulder or elbow. Edegeloop placement is critical when creating faces. When edgeloops are ignored, models will exhibit “tearing” when animated, and the model will need to be reworked or scrapped altogether in favour of a properly-planned model. Next Steps The next step to creating great models is simply to practice and examine the work of artists you admire. Some of the best 3D modellers are also fantastic pencil-and-paper artists. It will be well worth your time to practice drawing, whether you’re a character creator or a wanna-be architect.

Good modelling requires a lot of dedication. You'll need to thoroughly understand the software you're using, and the principles of good 3D model creation laid out aboe. Character artists will need to learn proportion and anatomy. The model describes the process of forming the shape of an object. The two most common sources of 3D models are those that an artist or engineer originates on the computer with some kind of 3D modelling tool, and models scanned into a computer from real-world objects. Models can also be produced procedurally or via physical simulation. Basically, a 3D model is formed from points called *vertices* (or *vertexes*) that define the shape and form *polygons*. A polygon is an area formed from at least three vertexes (a triangle). A four-point polygon is a *quad*, and a polygon of more than four points is an *n-gon*. The overall integrity of the model and its suitability to use in animation depend on the structure of the polygons.

Layout and Animation

Before rendering into an image, objects must be placed (laid out) in a scene. This defines spatial relationships between objects, including location and size. Animation refers to the *temporal* description of an object, *i.e.* how it moves and deforms over time. Popular methods include keyframing, inverse kinematics, and motion capture. These techniques are often used in combination. As with modelling, physical simulation also specifies motion.

3D Rendering

For those of us used to working in Photoshop and Illustrator it is important to realise that all that work is 2D, or two-dimensional. Photographs of real objects or painting

them from scratch in Painter, they are still 2D. This is because we are either working with a pixel representation or flat objects, like lines, text, paths, *etc.*

This is true even if we are attempting to simulate a 3D look. In 3D work, or three dimensions, we are producing a description of real objects with depth, scenes comprising many objects and the spatial relationships between them, along with the required lighting arrangements and viewing characteristics. The end result of 3D work is still usually 2D. This is either a still image or an animation, but it's still made up of pixels. In an ideal world our output would be three-dimensional too, as in a holographic projection or even a sculpture. This is a limitation of the output technologies that we have to work with at present, rather than an inherent characteristic of 3D work. Since, 3D printers exist (they are actually more like a numerically controlled milling machine in some ways), as do using LCD shutter glasses for direct 3D display, working completely in 3D is possible, just not the normal use.

Deep down, usually buried deep inside the software, our 3D work consists of rather mathematical descriptions of our scenes, such as place a sphere of radius *k*, with it's centre at *x, y, z* point in space with a surface texture like stone. Thankfully, we rarely have to deal with the numerical level unless we choose to. There are good reasons to dive down to the numerical level at times, such as exact placement. 3D software is largely click and drag operation these days for most common operations. It is important to remember that we are trying to represent things in the three-dimensional world that we are used to living in. Just as navigating around

the real world can get you lost, so is it easy to become disoriented in 3D software.

Keeping Oriented in 3D

In 3D software the convention is to use a set of three coordinates, x, y and z. Co-ordinates can be absolute or relative. Absolute coordinates apply to the entire world that we are creating in the computer. Everything is specified relative to a universal origin, the centre of your digital universe, with coordinates of 0,0,0. Positive x values may lie to the right, negative ones to the left. Positive y values may be up and negative ones down from the origin. Positive z may be in front of and negative ones behind the origin. Absolute coordinates are used to position objects in our scene, to place cameras and lights, *etc.* Relative coordinates have their origin somewhere other than the world origin. For instance, in creating an object made up of many parts it may be more convenient to think in terms of positions relative to what you wish to consider the centre of the object.

How the software works can have an impact on how easy it is to keep oriented. Some Programmes, like Bryce, display only one window, so you only have one view of your objects/ scene at a time. Other Programmes, like Vue d'Esprit or Lightwave, by default give you four views: a front, left and top view plus the view through the main camera. This last solution is generally preferred but does tend to work best when you are using a large, high-resolution screen. This is why most of the consumer level Programmes use the one view approach, assuming home users have small screens, whilst professional software takes the four-view approach.

The Stages of 3D Work

The following are the main stages of creating a 3D work:

- Create objects;
- Place objects in relation to each other in scene;
- Place light sources;
- Place the camera or observer;
- Add textures to objects;
- Add atmospheric effects;
- Render to produce a final image or animation movie.

The exact order of this sequence is partly up to you and partly a function of the software that you are using. For instance, some software separates the creation of objects and their placing in the scene (as in Lightwave), others combine this into one step (as in Bryce). Likewise, sometimes the textures are placed on objects when you create them.

But they can also be added at the scene creation stage. Each person gradually finds their own order of working that suits their needs and the needs of the specific project. For projects involving many people there may be different order, or indeed some stages my be performed in parallel, than for projects where you are doing the whole thing. The order of steps can affect the performance of your software. The sequence given tends to produce the least delays with most software, for reasons that will become clear as we progress through this series.

Creating objects and placing them in the scene is often called 'modelling'. This is because in creating an object and then a scene we are building a 'model' of it in the computer. Some software even separates the modelling function from the rest of the software by splitting the process into two

Programmes. It is quite possible to do the modelling in on manufacturer's Programme and the rest of the process in another. I quite frequently use three different Programmes for this process, making use of the strengths of each, these being Poser and Byrce and Lightwave.

Light sources and a camera are necessary if you are to see anything of the wonderful model you have created. Light sources and cameras can be treated in much the same way as any other object. Light sources will have their own, special characteristics though, like the type of light source, whether it casts shadows, its colour, *etc.* The camera also has special characteristics, like its field of view, resolution of the resulting image(s), *etc.*

Rendering is the process of determining what the scene looks like from the camera position taking into account all the characteristics of the objects, light sources and their interaction. Rendering is usually a time consuming process for any scene of reasonable complexity. This can vary from a 'go get a cup of coffee' to 'lunch' up to a whole week, or more. This is one reason why high complexity rendering of still images or animations tends to require fast computers and lots of memory. One reason that the order with which you create your image(s) is important is that you will usually do lots of little test renders along the way. Thus you want to leave the details which really slow the rendering down to as late in the sequence as possible.

Why Would We Want to Use 3D

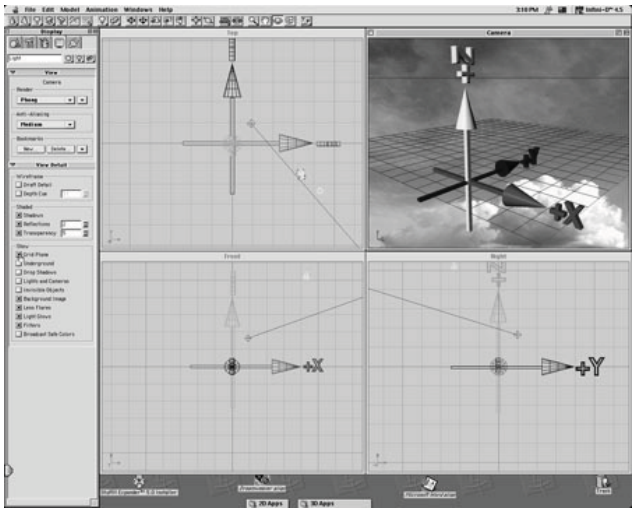
We need to represent solid objects, whether in a still image for an ad or an animation to go in a movie. Since, real world objects are 3D, there will be times when a 3D representation

is needed. Sure, we can paint or airbrush a 3D approximation but it will have a particular look, assuming that we have the skill level to create it. Working with 3D software creates a different look.

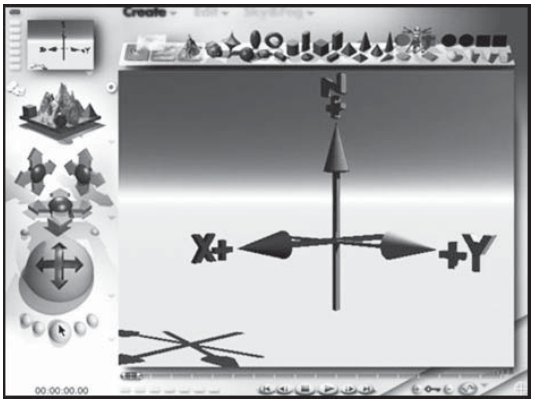
This can vary from one with a very computer feel to a photorealistic one, depending on the software and what we do with it.

The major advantage of working with 3D software is that it is easy to produce changes. To change the viewpoint only requires that we move the camera and render. To change the lighting or reposition objects is equally easy. So having created a scene once, we can produce many different images from it. This is like photographing a real scene in everything from wide-angle to close-up, and from different positions.

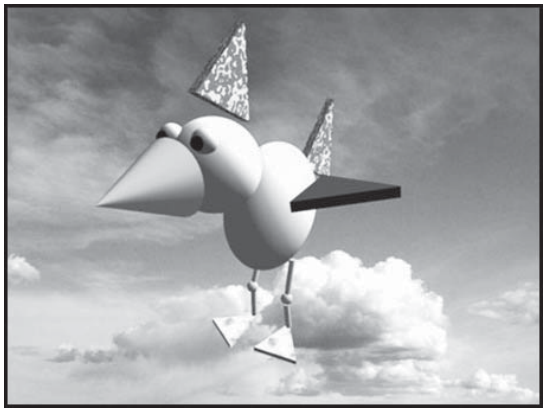
3D software gives you flexibility. This very flexibility allows you to re-purpose images. You may do an illustration for a magazine ad and then the client comes back and wants an animation for a TV ad, or the web. Once you have built the models, you can re-use them repeatedly



This screen grab of the old Metacreation's Infiniti-D 4.5 shows a four window, working environment. Three windows give front, top and side views whilst the fourth shows the camera view. This type of display, common to most of the higher-end 3D packages, works best on a high resolution, large screen.



The single view at atimedisplay, like this one from Bryce, works well on smaller displays. Usually keyboard shortcuts or button sallow you to switch between views. Whilstnot as convenient as the four-window display it is quite workable. It seems natural once you get used to it



This simple cartoon bird was created out of basic object types and rendered in Infiniti-D 4.5.A background image was used.

What Sorts of Objects

In most real scenes, the objects that we might want to incorporate will be complex. Unfortunately most 3D modellers and renderers don't support basic object types like 'tree', 'car', 'person' or 'house'. Such complex objects have to be created out of the actual object types that the renderer supports. The usual basic objects types are flat objects, like planes and polygons, and 3D objects like spheres, cylinders, cones, *etc.* Of course, you can also obtain libraries of already created objects. Some 3D Programmes come with lots of these, others few. There are web sites where people place free 'models' that you can download. There are also companies that specialise in creating 'models' that you can buy.

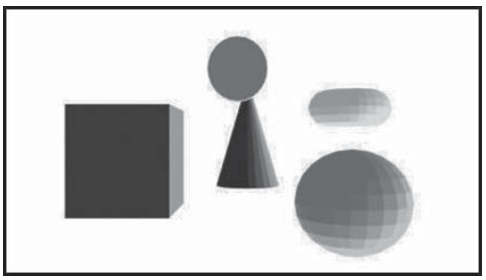
Polygons, for reasons that will become clearer later in this series, are the mainstay of most 3D modellers and renderers. A polygon is simply a shape made up of a number of straight lines, joined together to define a closed shape. The points that define the end of each line are called a vertex. Different Programmes allow variations on the basic polygon. Some Programmes require that polygons be totally flat, that all the vertices lie in a flat plane. Others allow curved polygons. Some require all polygons to have either three or four sides. Others allow you to construct polygons with greater numbers of sides. Many of these latter ones will actually subdivide the polygon into three or four sided ones before rendering, though this is usually hidden from the user. One major advantage of three sided polygons, triangles, is that they have to be flat. Only four sided or higher polygons can have some vertices not in the same plane as the others. A variation on the polygon that you find in most 3D software is the infinite

plane. As its name implies this plane is a flat surface that stretches off into infinity. Infinite planes are useful for things like water levels, cloud layers, *etc.*

Polygons are defined by the *x, y, z* coordinates of their vertices. It is not unusual to be required to define the vertices of a polygon in a particular order, such as clockwise or anticlockwise when looking at the front face of the polygon. Some software requires this to be able to calculate the surface normal. Surface normals are incredibly important in 3D work as they are used to work out how much light is hitting a surface, and thus it's colour. The surface normal points up from the surface of the polygon. Some software treats polygons as single sided, other software as double sided. 3D software that has single sided polygons will not display them if you are looking at their back surface. With such software if you want a bowl, for example, you have to define polygons forming both the inside and outside surfaces. Software that uses double sided polygons does not have this requirement, one layer of polygons can represent both the inside, and outside surfaces, though this is not natural, since, the bowl walls would have no thickness.

Basic 3D objects, like spheres, cylinders, boxes and cones are also incredibly useful. We can construct planets from spheres and tree trunks from cylinders, for instance. Since, these are the basic forms used in the construction of most man-made, and many natural, objects, they are indispensable. Many Programmes, when you use one of these, create the basic object at a standard size. You can then usually modify the object by stretching it into the form you want. Other Programmes allow you to stretch out the shape

when you insert it into the scene. This stretching process allows you to create oval footballs from a sphere, a rectangular building from a square cube and a long spear from a squat cylinder. Most software gives you the choice of doing this either by typing in numbers or by clicking and dragging. This stage of modifying the shape of your objects is usually much easy if you can easily switch between different views of the object, like front, side and top, either through having multiple views open at once or by switching views in the one window.



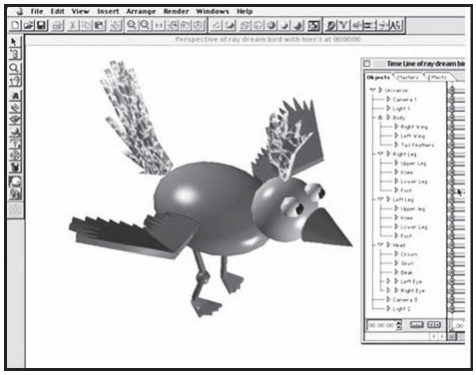
Boxes, spheres, cylinders, cones, polygons and text objects are the basic construction components available in most 3D software, as shown in this render done with Newtek's Inspire 3D. In some Programmes all these objects are constructed out of polygons, in others they are primitive objects that are rendered directly. If you examine the edges of the sphere and cone you can see that they are constructed out of polygons.

Creating Composite Objects

If all objects are treated as individual ones, you end up with a heap of them to try to manage. Since, most basic objects will actually be used to construct more complex objects it is useful to be able to group objects together that form parts of a whole. Thus we might create an object 'person'

with parts 'head', 'body', 'arm12, 'arm22, 'leg12 and 'leg22. Then 'leg12 consists of 'upper', 'lower' and 'foot'. And so on. Building up complex objects out of hierarchies of other parts makes life a lot easier. If you want to move a whole object you can simply select the top level and move it, knowing that all the component parts will move too. Otherwise you would have to separately select every component and move them, and hope you didn't forget some small parts.

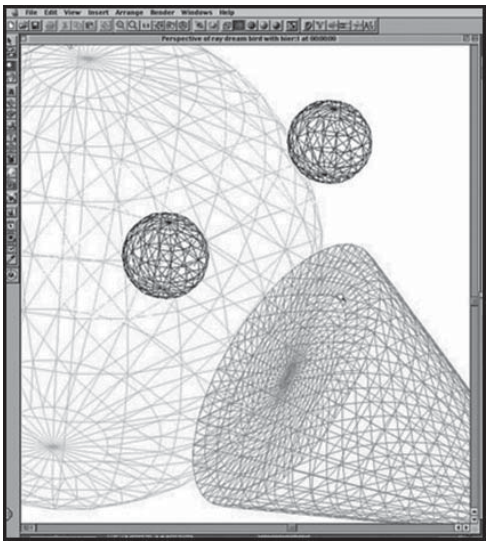
Object hierarchies are most flexible when you can give names to each component part. Such hierarchies are also essential to making character animation easier. Some Programmes allow you to readily display object hierarchies in a diagram form that shows the relationships between parts, similar to the folder hierarchy views that most operating systems allow. Software that doesn't do this is certainly harder to use for some things.



This screen grab, from Ray Dream Studio, shows a cartoon bird and it's hierarchical construction. Unfortunately too few Programmes provide this sort of display.

Another type of object related to the above is a polygon mesh. A mesh is a set of polygons which are joined together to represent a surface of some complexity. A good example of this is the polygon mesh that Bryce 3D uses to represent

the shape of the landscape. The process of creating a polygon mesh usually does not require that the user manually position each vertex of each polygon in the mesh. Various other convenient methods are available. We'll examine these in later chapter.



This close-up of part of a bird model in Ray Dream Studio shows how this Programme tessellates spheres into polygonal meshes.

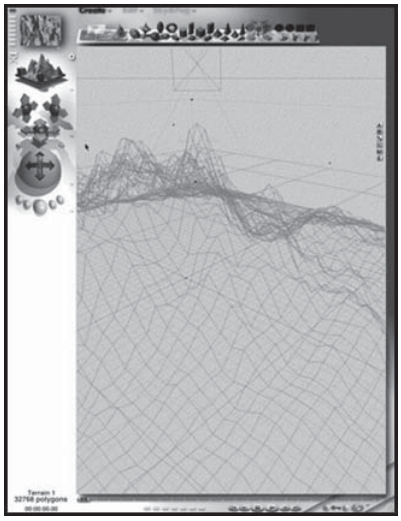
There Differences of Approach

There are two choices the software developers have to make: what primitive objects are to be supported; and what rendering method is to be used? These two questions are interrelated.

The rendering method determines what actual primitive objects the software works with to create images. How we want the user interface to be will determine what primitive objects are available to the user. For a number of reasons that we will examine in the next part of the course, certain

rendering techniques can only actually support polygons, whilst others can actually handle spheres, cylinders, *etc.*

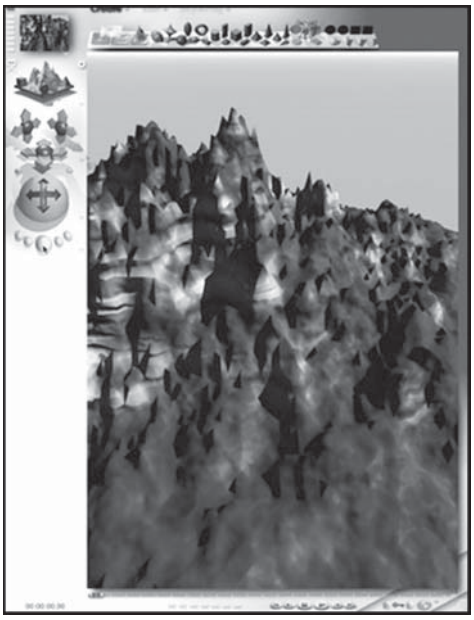
So a Programme that has to use polygons for rendering will convert a sphere into a polygonal approximation, in a process called tessellation, before actually rendering an image. This creates more primitive objects to render but allows the renderer to be highly optimised for the handling of polygons. A Programme which can directly support spheres, say, does not have to do this conversion and thus renders fewer objects in your scene but requires specialised Programme code for each object type it supports.



Another use for polygon meshes is to represent irregular objects, like this landscape in Bryce 3D.

These internal differences in approach are what make some 3D packages good for some types of work and others more suitable for others. Some will handle transparent objects superbly, other handle interior lighting well, for example. Some will make dealing with certain types of objects easy, whereas others make those objects hard but others easy. It is for these reasons that many people working with 3D

software will use a number of packages for different parts of the process. Whilst this is certainly not necessary, it can be a useful approach. It's the same as people using Painter for some things and Photoshop for others, sometimes switching backwards and forwards between the two.



The designers of 3D software have to make a complex set of choices based on their priorities. These choices lead to the differences in single or double sided polygons, whether tessellation is done and what types of rendering options are available, to pick just three.

Some choices will speed up the execution of the Programme whilst others will slow it down. These tradeoffs account for the huge variety that we encounter in 3D Programmes.

Distinction from Photorealistic 2D Graphics

Not all computer graphics that appear 3D are based on a wireframe model. 2D computer graphics with 3D photorealistic effects are often achieved without wireframe

modelling and are sometimes indistinguishable in the final form. Some graphic art software includes filters that can be applied to 2D vector graphics or 2D raster graphics on transparent layers. Visual artists may also copy or visualize 3D effects and manually render photorealistic effects without the use of filters.

3D Modeling

In 3D computer graphics, 3D modeling (also known as meshing) is the process of developing a mathematical representation of any three-dimensional surface of object (either inanimate or living) via specialized software. The product is called a 3D model. It can be displayed as a two-dimensional image through a process called *3D rendering* or used in a computer simulation of physical phenomena. The model can also be physically created using 3D Printing devices. Models may be created automatically or manually. The manual modeling process of preparing geometric data for 3D computer graphics is similar to plastic arts such as sculpting.

Models

3D models represent a 3D object using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc. Being a collection of data (points and other information), 3D models can be created by hand, algorithmically (procedural modeling), or scanned. 3D models are widely used anywhere in 3D graphics. Actually, their use predates the widespread use of 3D graphics on personal computers. Many computer games used pre-rendered images of 3D models as sprites

before computers could render them in real-time. Today, 3D models are used in a wide variety of fields. The medical industry uses detailed models of organs. The movie industry uses them as characters and objects for animated and real-life motion pictures.

The video game industry uses them as assets for computer and video games. The science sector uses them as highly detailed models of chemical compounds. The architecture industry uses them to demonstrate proposed buildings and landscapes through Software Architectural Models. The engineering community uses them as designs of new devices, vehicles and structures as well as a host of other uses. In recent decades the earth science community has started to construct 3D geological models as a standard practice.

Representation

Almost all 3D models can be divided into two categories.

- Solid - These models define the volume of the object they represent (like a rock). These are more realistic, but more difficult to build. Solid models are mostly used for nonvisual simulations such as medical and engineering simulations, for CAD and specialized visual applications such as ray tracing and constructive solid geometry
- Shell/boundary - these models represent the surface, e.g. the boundary of the object, not its volume (like an infinitesimally thin eggshell). These are easier to work with than solid models. Almost all visual models used in games and film are shell models.

Because the appearance of an object depends largely on the exterior of the object, boundary representations are common in computer graphics. Two dimensional surfaces are a good analogy for the objects used in graphics, though quite often these objects are non-manifold. Since surfaces are not finite, a discrete digital approximation is required: polygonal meshes (and to a lesser extent subdivision surfaces) are by far the most common representation, although point-based representations have been gaining some popularity in recent years. Level sets are a useful representation for deforming surfaces which undergo many topological changes such as fluids. The process of transforming representations of objects, such as the middle point coordinate of a sphere and a point on its circumference into a polygon representation of a sphere, is called tessellation. This step is used in polygon-based rendering, where objects are broken down from abstract representations ("primitives") such as spheres, cones etc., to so-called *meshes*, which are nets of interconnected triangles. Meshes of triangles (instead of e.g. squares) are popular as they have proven to be easy to render using scanline rendering. Polygon representations are not used in all rendering techniques, and in these cases the tessellation step is not included in the transition from abstract representation to rendered scene.

Modeling Processes

There are five popular ways to represent a model:

- Polygonal modeling - Points in 3D space, called vertices, are connected by line segments to form a polygonal mesh. Used, for example, by Blender.

The vast majority of 3D models today are built as textured polygonal models, because they are flexible and because computers can render them so quickly. However, polygons are planar and can only approximate curved surfaces using many polygons.

- NURBS modeling - NURBS Surfaces are defined by spline curves, which are influenced by weighted control points. The curve follows (but does not necessarily interpolate) the points. Increasing the weight for a point will pull the curve closer to that point. NURBS are truly smooth surfaces, not approximations using small flat surfaces, and so are particularly suitable for organic modeling. Maya, Rhino 3d and solidThinking are the most well-known commercial programmes which use NURBS natively.
- Splines & Patches modeling - Like NURBS, Splines and Patches depend on curved lines to define the visible surface. Patches fall somewhere between NURBS and polygons in terms of flexibility and ease of use.
- Primitives modeling - This procedure takes geometric primitives like balls, cylinders, cones or cubes as building blocks for more complex models. Benefits are quick and easy construction and that the forms are mathematically defined and thus absolutely precise, also the definition language can be much simpler. Primitives modeling is well suited for technical applications and less for organic shapes. Some 3D software can directly render from primitives (like

POV-Ray), others use primitives only for modeling and convert them to meshes for further operations and rendering.

- Sculpt modeling - Still fairly new method of modeling 3D sculpting has become very popular in the few short years it has been around. There are 2 types of this currently, Displacement which is the most widely used among applications at this moment, and volumetric. Displacement uses a dense model (often generated by Subdivision surfaces of a polygon control mesh) and stores new locations for the vertex positions through use of a 32bit image map that stores the adjusted locations. Volumetric which is based loosely on Voxels has similar capabilities as displacement but does not suffer from polygon stretching when there are not enough polygons in a region to achieve a deformation. Both of these methods allow for very artistic exploration as the model will have a new topology created over it once the models form and possibly details have been sculpted. The new mesh will usually have the original high resolution mesh information transferred into displacement data or normal map data if for a game engine.

The modeling stage consists of shaping individual objects that are later used in the scene. There are a number of modeling techniques, including:

- constructive solid geometry
- implicit surfaces
- subdivision surfaces

Modeling can be performed by means of a dedicated programme (e.g., form•Z, Maya, 3DS Max, Blender, Lightwave, Modo, solidThinking) or an application component (Shaper, Lofter in 3DS Max) or some scene description language (as in POV-Ray). In some cases, there is no strict distinction between these phases; in such cases modeling is just part of the scene creation process (this is the case, for example, with Caligari trueSpace and Realsoft 3D). Complex materials such as blowing sand, clouds, and liquid sprays are modeled with particle systems, and are a mass of 3D coordinates which have either points, polygons, texture splats, or sprites assigned to them. Sculpt

Scene Setup

Scene setup involves arranging virtual objects, lights, cameras and other entities on a scene which will later be used to produce a still image or an animation. Lighting is an important aspect of scene setup. As is the case in real-world scene arrangement, lighting is a significant contributing factor to the resulting aesthetic and visual quality of the finished work. As such, it can be a difficult art to master. Lighting effects can contribute greatly to the mood and emotional response effected by a scene, a fact which is well-known to photographers and theatrical lighting technicians. It is usually desirable to add color to a model's surface in a user controlled way prior to rendering. Most 3D modeling software allows the user to color the model's vertices, and that color is then interpolated across the model's surface during rendering. This is often how models are colored by the modeling software while the model is being created. The most common method of adding color

information to a 3D model is by applying a 2D texture image to the model's surface through a process called texture mapping. Texture images are no different than any other digital image, but during the texture mapping process, special pieces of information (called texture coordinates or UV coordinates) are added to the model that indicate which parts of the texture image map to which parts of the 3D model's surface. Textures allow 3D models to look significantly more detailed and realistic than they would otherwise.

Other effects, beyond texturing and lighting, can be done to 3D models to add to their realism. For example, the surface normals can be tweaked to affect how they are lit, certain surfaces can have bump mapping applied and any other number of 3D rendering tricks can be applied. 3D models are often animated for some uses. They can sometimes be animated from within the 3D modeler that created them or else exported to another programme.

If used for animation, this phase usually makes use of a technique called "keyframing", which facilitates creation of complicated movement in the scene. With the aid of keyframing, one needs only to choose where an object stops or changes its direction of movement, rotation, or scale, between which states in every frame are interpolated. These moments of change are known as keyframes. Often extra data is added to the model to make it easier to animate. For example, some 3D models of humans and animals have entire bone systems so they will look realistic when they move and can be manipulated via joints and bones, in a process known as skeletal animation.

Compared to 2D Methods

3D photorealistic effects are often achieved without wireframe modeling and are sometimes indistinguishable in the final form. Some graphic art software includes filters that can be applied to 2D vector graphics or 2D raster graphics on transparent layers. Advantages of wireframe 3D modeling over exclusively 2D methods include:

- *Flexibility,* ability to change angles or animate images with quicker rendering of the changes;
- *Ease of rendering,* automatic calculation and rendering photorealistic effects rather than mentally visualizing or estimating;
- *Accurate photorealism,* less chance of human error in misplacing, overdoing, or forgetting to include a visual effect.

Disadvantages compare to 2D photorealistic rendering may include a software learning curve and difficulty achieving certain photorealistic effects. Some photorealistic effects may be achieved with special rendering filters included in the 3D modeling software. For the best of both worlds, some artists use a combination of 3D modeling followed by editing the 2D computer-rendered images from the 3D model.

3D Model Market

3CT (3D Catalog Technology) has revolutionized the 3D model market by offering quality 3D model libraries free of charge for professionals using various CAD programmes. Some believe that this uprising technology is gradually eroding the traditional "buy and sell" or "object for object exchange" markets although the quality of the products do

not match those sold on specialized 3d marketplaces. A large market for 3D models (as well as 3D-related content, such as textures, scripts, etc.) still exists - either for individual models or large collections. Online marketplaces for 3D content allow individual artists to sell content that they have created. Often, the artists' goal is to get additional value out of assets they have previously created for projects. By doing so, artists can earn more money out of their old content, and companies can save money by buying pre-made models instead of paying an employee to create one from scratch. These marketplaces typically split the sale between themselves and the artist that created the asset, often in a roughly 50-50 split. In most cases, the artist retains ownership of the 3d model; the customer only buys the right to use and present the model.

Human Models

The first widely available commercial application of human Virtual Models appeared in 1998 on the Lands' End web site. The human Virtual Models were created by the company My Virtual Model Inc. and enabled users to create a model of themselves and try on 3D clothing. There are several modern programmes that allow for the creation of virtual human models (Poser being one example).

6

Graphics Primitives

Introduction

A basic nondivisible graphical element for input or output within a computer-graphics system. Typical output primitives are polyline, polymarker, and fill area. Clipping of an output primitive cannot be guaranteed to produce another output primitive. Output primitives have attributes such as line style and pattern associated with them. Typical input primitives are locator, choice, and valuator. Input primitives often have a style of echoing associated with them. The purpose of graphics system is to make programming easier for the user. Graphics system includes special Hardware for output and input of representating pictures and software routines for performing the basic graphics operations. Some common operations are Moving the Pen (or electron beam), Drawing a line, Writing a character or a string of text, changing the line style.

Display Devices

Computer graphics images are composed of a finite number of picture elements or pixels. Each pixel requires at least one bit of intensity information, light or dark. Many software applications include graphics components. Such programmes are said to support graphics. For example, certain word processors support graphics because they let you draw or import pictures. All CAD/CAM systems support graphics. Some database management systems and spreadsheet programmes support graphics because they let you display data in the form of graphs and charts. Such applications are often referred to as business graphics. If we actually stored the information for each pixel in the computer's memory, a lot of memory may be required. The Portion of the memory which is used to hold the pixels is called "Frame Buffer". The memory is usually scanned and displayed by direct memory access that is special hardware independent of the central processor.

Raster Display

A raster display refresh system includes a charge coupled device (CCD) circulating refresh memory for maintaining a display of information on a cathode ray tube (CRT) screen. The X and Y address of picture elements to be changed are stored in raster scan sequence in a small random access memory (RAM). Whenever a picture element address in the data register of the RAM equals the X and Y screen address of the scanning beam of the CRT, a corresponding new picture element signal stored in the RAM is substituted for the old picture element signal previously circulating the CCD refresh memory.

- In raster display, the frame buffer may be examined to determine what is currently being displayed.
- Surface lines are displayed on raster display devices.
- Images may be displayed on television style picture tubes.
- The raster terminal can also display colour images.

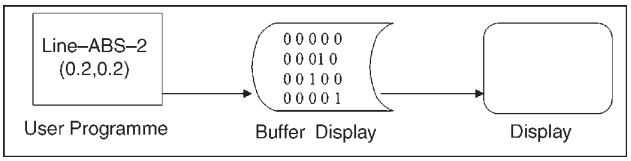


Fig. Raster Display System

Disadvantages

- Cost of the required memory
- Time which may be required to alter every pixel whenever the image is changed.

Plotting System

- A pen is lowered on to paper and moved under the direction of a vector generation algorithm.
- Once the line is drawn, the ink on the paper remembers it. And the computer need not consider it further.

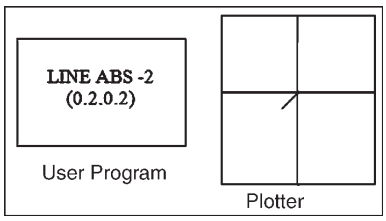


Fig. Plotting system

Disadvantages

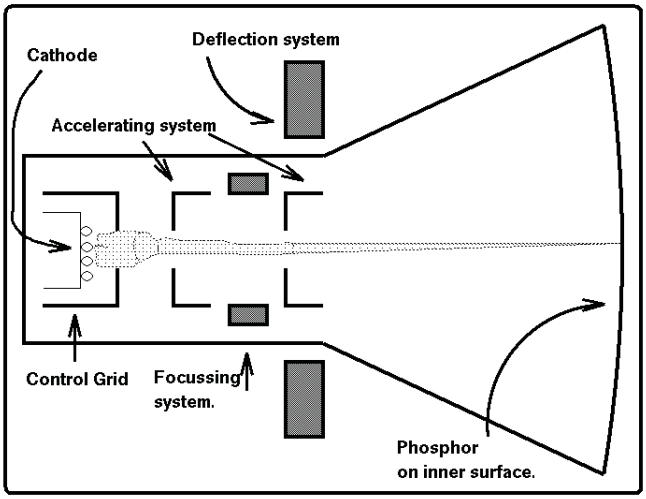
1. Once a line drawn, it cannot be easily removed. If we wish to change the picture we must get a fresh paper and redraw the picture.

2. This can be time consuming and use lot of paper.

DVST (Direct View Storage Tubes)

A cathode-ray tube in which secondary emission of electrons from a storage grid is used to provide an intensely bright display for long and controllable periods of time. Also known as display storage tube; viewing storage tube. This is the first CRT display produced by Tektronix.

The terminal use special cathode ray tubes called DVST, behave same way as plotter. An electron beam is directed at the surface of the screen. It has good resolution and low cost.



The position of the beam is controlled by electronic or magnetic fields with in the tube. Once the screen phosphors of this special tube have been illuminated by the electron beam, they stay lit. It has good resolution and low cost.

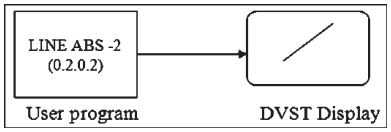


Fig. Direct View Storage Tube System

Disadvantages

One cannot alter a DVST image except by erasing the entire screen and drawing it again. This can be done faster than on a plotter. But the process is still time-consuming making interaction difficult.

Apart from this, there are many other disadvantage such as:

1. Ordinary do not display colour
2. Selected part of the picture can't be erased
3 To eliminate a picture section, the entire screen must be erased

Plasma Panel

A display device which stores the image, but allows selective erasing is the Plasma Panel. The Plasma panel contains a gas at low pressure sandwiched between horizontal and vertical grids of find wires. A large voltage between horizontal and vertical wire will cause the gas to glow as it does in a neon street sign. A lower voltage will not start a glow but will maintain aglow once started. Plasma panels are very durable and are often used for military applications and PLATO educational system.

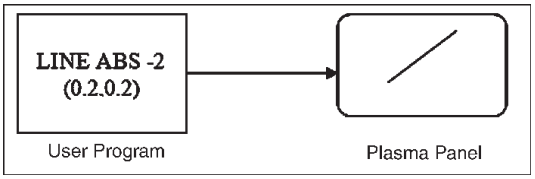
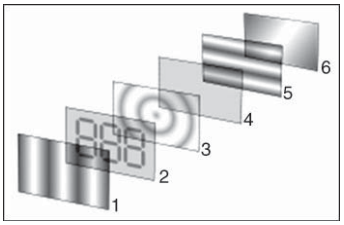


Fig. Plasma Panel System

Liquid Crystal Display

It is less bulky than CRTs, because of its low voltage and power requirements, it is lighter in weight. In a liquid crystal

display, light is either transmitted or blocked depending upon the orientation of molecules in the liquid crystal.



An electrical signal can be used to change the molecular orientation, turning a pixel ON or OFF. The material is sandwiched between horizontal and vertical grids of electrodes which are used to select the pixel.

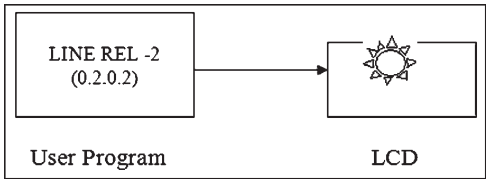


Fig. Liquid Crystal Display System

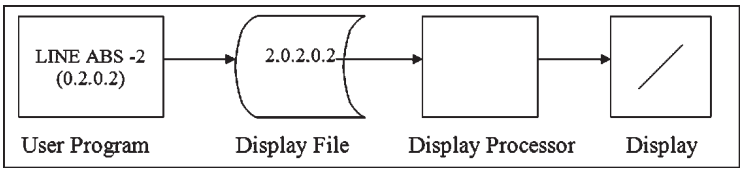
Liquid Crystal Displays (LCDs) are used to in flat-panel monitors and high-end flat-panel television. Recently, the market for LCDs has exploded as flat screen monitors and televisions have become popular consumer electronics devices.

Vector Refresh Display

The vector refresh display stores the image in the computer's memory, but it tries to be much more efficient about than a raster display. To specify a line segment, all that is required is the coordinates of its endpoints. The vector refresh display stores only the commands necessary for drawing the line segments. The input to the vector generator is saved instead of the output. These commands are saved in separate file called as "Display file". They are examined

and the lines are drawn using a vector generation algorithm. They are usually implemented in hardware.

Refresh Displays allow real-time alteration of the image. The disadvantage is the images formed are composed of line segments, not surfaces and a complex display may flicker because it will take a long time to analyse and draw it.



DISPLAY FILE

The concept of display file provides an interface between the image specification process and the image display process. It also defines a compact description of the image. The display-file idea may be applied to devices other than refresh displays. The *proc* file system is sometimes referred to as a process information pseudo-file system. It does not contain "real" files but rather runtime system information (*e.g.* system memory, devices mounted, hardware configuration, etc). For this reason it can be regarded as a control and information center for the kernel.

In fact, quite a lot of system utilities are simply calls to files in this directory. For example, the command lsmod, which lists the modules loaded by the kernel, is basically the same as 'cat/proc/modules' while lspci, which lists devices connected to the PCI bus of the system, is the same as 'cat/proc/pci'. By altering files located in this directory you can change kernel parameters while the system is running. Such files are sometimes called pseudo display files or metafiles.

Primitive Operations

Most graphics system offer set of graphics primitive operations. The first primitive command is drawing a line segment. The final point of the last segment becomes the first point of the next segment. To avoid specifying this point twice, the system can keep track of the correct pen or electron bean position.

LINE- ABS-2(x, y) is called an absolute line command because the actual coordinates of the final position are passed.

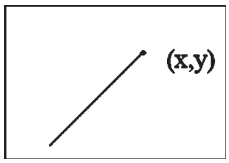


Fig. The Absolute Line Command

There is also a relative line command. Here we can specify how far to move from the current position.

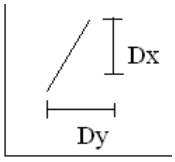


Fig. The Relative Line Command

LINE-REL-2(DX, DY)

Suppose (XC,YC) denote current position.

LINE-REL-2(DX, DY) is same as

LINE-ABS-2(DX + XC, DY+ YC)

The above procedures are fine for producing a connected string of line segments. We can also draw two disconnected segment by the same mechanism if we picture these two segments as connected by a middle segment which happens to be invisible. We can construct a line drawing by a series of

line and move commands. Path relative to working directory does not start with a slash.

Example. If 'ls' shows "terosdir", you can use a relative path. To change directory to "terosdir", under current directory:

```
$ cd terosdir
```

Absolute path starts with a slash "/". The first slash in path refers to topmost directory, the root directory, whose name is simply "/". You can go up to root directory "/" by commanding 'cd..' many times, then finally checking path with 'pwd'.

Change directory to etc, that is under the root dir "/". With absolute path, working directory does not matter.

```
$ cd/etc
```

Final slash after path does not mean anything, and these two commands mean exactly the same:

```
$ cd terosdir/
$ cd terosdir
```

The Display File Interpreter

The Display file will contain the information necessary to construct the picture. The information will be in the form of "*Draw a line*" or "*Move the pen*". Saving instruction is less storage than saving the picture itself. Display file interpreter is used to convert these instructions into actual images. These instructions can be thought of as a programme for creating the image. In some graphics system there is a separate computer called display processor, which is located in the graphics terminal. In other systems, the behaviour of a display processor is simulated.

Our display-file interpreter serves as an interface between our graphics programme and the display device. The Dis-

play file instruction may actually be saved in a file either for display later or for transfer to another machine. Such files of imaging instructions are sometimes called Meta files.

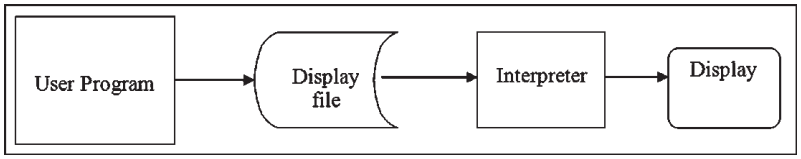


Fig. Display File and Interpreter

Normalized Device Co-ordinates

The device independence units are called the normalized device co-ordinates. In this, the screen measures one unit wide and one high. The lower left corner of the screen is the origin and the upper left corner is the point (1.1). The Point (.5,.5) is in the corner of the screen.

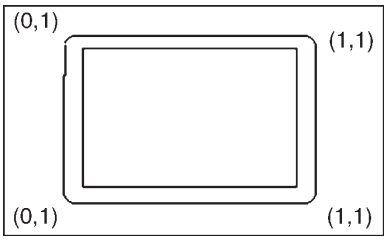


Fig. Normalized Device Coordinates.

- Suppose that of the actual display the index of the leftmost pixel is WIDTH-START and that there are WIDTH pixels in the horizontal direction.
- Suppose also that the bottom most pixel is HEIGHT-START and the member of pixels in the vertical direction is HEIGHT.
- In the normalized coordinates the screen is one unit wide, but in the actual co-ordinates it is width units wide. So the normalized x position should be multiplied by WIDTH/1 to convert to actual screen units.

- At position X_n= 0 in normalized we should get X_s= WIDTH, in actual screen co-ordinates, so the conversion formula should be

$$X_s = WIDTH * X_n + WIDTH–START$$

Similarly for vertical direction,

$$Y_s = HEIGHT * Y_n + HEIGHT–START$$

- If we have a display which is not square, we can either use the displays full height or width in the conversion formula. If we use full dimension, the image will be stretched or squashed. If we use a square area of the display the image is correctly proportioned.

Display file structure

Each display file command contains two parts:

- Operation code which indicates what kind of command.
- Operands which are the coordinates of a point (x, y).

The display file is made up of a series of instructions. Three arrays are used for representing display file commands.

They are:

DF-OP—one of the operation code

DF-X–one for the x-coordinate

DY-Y–one for the y-coordinate

We must assign meaning to the possible operation codes before we can proceed to interpret them. Only two possible instructions are available. They are, MOVE and LINE commands.

Opcode 1–Move command

Opcode 2–Line command

Example

A command to move to position x = 0.3 and y = 0.7

DF–OP[3] ←1;

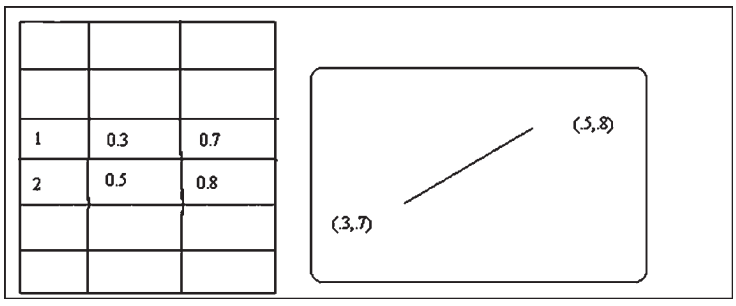
DF–X[3] ← 0.3;

DF–Y[3] ← 0.7;

Let us develop the algorithm for inserting display file instruction. Line arguments require two endpoints for their specification, but we shall enter only one end point and assume that the other endpoint is the current pen position. We will therefore need variables DF-PEN-Y to keep track of the current pen position.

We will need to know this position for the conversion of relative commands to absolute commands. We shall also need a variable FREE to indicate where the next free cell of the display file is located. These variables, together with the display file itself, are used by several different routines and must be maintained between accesses. They are therefore presented as global variables.

The first algorithm we will consider actually puts an instruction into display file.



Algorithm PUT_POINT (OP, X, Y)

Arguments OP, X, Y the instruction to be entered.

Global DF-OP, DF-X, DF-Y the three display file

arrays.
FREE the position of the next free cell.

Constant DFSIZE the length of the display-file arrays.

```
BEGIN
    IF FREE > DFSIZE THEN RETURN ERROR
    "DISPLAY FILEFULL";
    DF-OP [FREE]← OP;
    DF-X [FREE] ←     X;
    DF-Y [FREE] ←     Y;
    FREE ← FREE + 1;
    RETURN;
END;
```

This algorithm stores the operation code and the coordinates of the specified position in the display file. The pointer FREE to the next free cell is incremented so that it will be in the correct position for the next entry. We also wish to access in the display file. We isolate the accessing mechanism in a separate routine so that any changes in the data structure used for the display file will not affect the rest of our graphics package.

Algorithm

GET-POINT (NTH, OP, X, Y) retrieve the NTH instruction from the display file.

Arguments NTH the number of the desired instruction
OP, X, Y the instruction to be returned

Global arrays DF-OP, DF-X, DF-Y the display file

```
BEGIN
    OP← DF–OP[NTH];
```

```
        X← DF–X[NTH];
        Y← DF–Y[NTH];
        RETURN;
    END;
```

Our MOVE and LINE instructions must update the current pen position and enter a command into the display file. If the update of the pen position is done first then the new pen position will serve as the operand for the display file instruction.

IT will prove convenient to have a separate routine which takes the operation code and the pen position and enters them onto the display file as an instruction.

Algorithm

DISPLAY-FILE-ENTER (OP) combines operation and position to form an instruction and save it in the display file.

```
    Argument    OP the operation to be entered
    Global      DF-PEN–X, DF=PEN–Y the current pen
                position
    BEGIN
        PUT-POINT (OP,DF-PEN-X,DF-PEN-Y);
        RETURN;
    END;
```

Using DISPLAY-FILE-ENTER to place instructions in the displayfile, the absolute MOVE routine becomes the following:

```
    Algorithm   MOVE-ABS-2(X,Y)
    Arguments   X, Y the point to which to move the pen
                GlobalDF_PEN-X, DF-PEN-Y the current
                pen position
```

```
BEGIN
        DF-PEN-X ←X;
        DF-PEN-Y ←Y;
        DISPLAY-FILE-ENTER(1);
        RETURN;
END;
```

The point (DF-PEN-X, DF-PEN-Y) is keeping track of where we wish the pen to go. By setting (DF-PEN-X, DF-PEN-Y) to (X,Y), we are saying the pen is to be at position (X,Y). The algorithm for entering a LINE command is similar.

Algorithm

MOVE-ABS-2(X,Y)

Arguments (X,Y) user routine to save a command to draw a line.

Arguments X,Y the points where to draw the line

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

```
BEGIN
        DF-PEN-X←X;
        DF-PEN-Y← Y;
        DISPLAY-FILE-ENTER (2);
RETURN;
END;
```

Again by changing DF-PEN-X and DF-PEN-Y we indicate that the pen will be placed at (X,Y) but by entering an operation code of 2 instead of 1, we instruct the interpreter to draw a line as the pen is moved. We can also write algorithm for the relative commands.

Algorithm MOVE-REL-2(DX, DY) user routine to save a command to move the pen

Arguments DX, DY the change in the pen position
Global DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
DF-PEN-X ← DF-PEN-X + DX;
DF-PEN-Y ← DF-PEN-Y + Y;
DISPLAY-FILE-ENTER (1);
RETURN;
END;

Algorithm

LINE-REL-2(DX, DY) user routine to save a command to draw a line
Arguments DX, DY the change over which to draw a line
Global DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
DF-PEN-X ← DF-PEN-X + DX;
DF-PEN-Y ← DF-PEN-Y + DY;
DISPLAY-FILE-ENTER (2);
RETURN;
END;

The relative LINE and MOVE routines act like the absolute routines in that they will tell where the pen is to be placed and how it is to get there. They differ in that the new pen position is calculated as an offset to the old pen position.

7

Computers and Java

Central Processing Unit, or CPU. In a modern desktop computer, the CPU is a single “chip” on the order of one square inch in size. The job of the CPU is to execute programmess.

A programme is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called machine language. Each type of computer has its own machine language, and the computer can directly execute a programme only if the programme is expressed in that language. (It can execute programmes written in other languages if they are first translated into machine language.)

When the CPU executes a programme, that programme is stored in the computer’s main memory (also called the RAM or random access memory). In addition to the programme, memory can also hold data that is being used or processed by the programme. Main memory consists of a sequence of locations.

These locations are numbered, and the sequence number of a location is called its address.

An address provides a way of picking out one particular piece of information from among the millions stored in memory. When the CPU needs to access the programme instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the data contained in the specified location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (although it is very complicated in detail). The CPU executes a programme that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or fetching, an instruction from memory and then carrying out, or executing, that instruction. This process-fetch an instruction, execute it, fetch another instruction, execute it, and so on forever is called the fetch-and-execute cycle.

The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal registers, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers—the programme counter, or PC—to keep track of where it is in the programme it is executing.

The PC stores the address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that is

to be executed in the next cycle. (Usually, but not always, this is just the instruction that sequentially follows the current instruction in the programme.)

A computer executes machine language programmes mechanically—that is without understanding them or thinking about them—simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called transistors, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off.

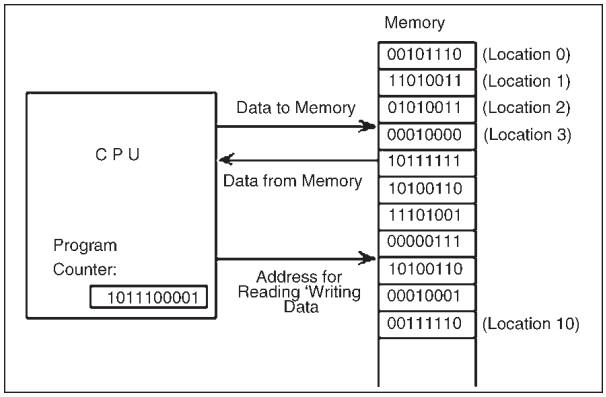
As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the programme that the computer is executing.

Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that particular instruction.

The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together. So, you should understand this much about how computers work: Main memory

holds machine language programmes and data. These are encoded as binary numbers. The CPU fetches machine language instructions from memory one after another and executes them.

It does this mechanically, without thinking about or understanding what it does—and therefore the programme it executes must be perfect, complete in all details, and unambiguous because the CPU can do nothing but execute it exactly as written. Here is a schematic view of this first-stage understanding of the computer:



Computer Architecture

To understand digital signal processing systems, we must understand a little about how computers compute. The modern definition of a *computer* is an electronic device that performs calculations on data, presenting the results to humans or other computers in a variety of (hopefully useful) ways.

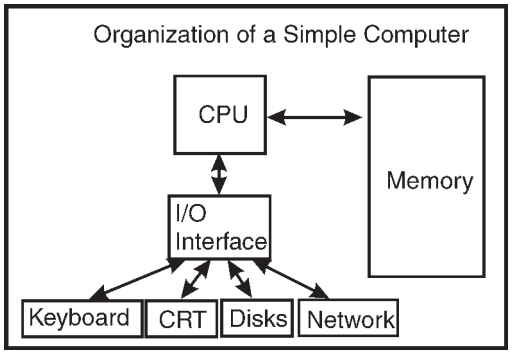


Fig. Generic Computer Hardware Organization.

The generic computer contains *input* devices (keyboard, mouse, A/D (analog-to-digital) converter, etc.), a *computational unit*, and output devices (monitors, printers, D/A converters). The computational unit is the computer's heart, and usually consists of a *central processing unit* (CPU), a *memory*, and an input/output (I/O) interface. What I/O devices might be present on a given computer vary greatly.

A simple computer operates fundamentally in discrete time: Computers are *clocked* devices, in which computational steps occur periodically according to ticks of a clock. This description belies clock speed: When you say "I have a 1 GHz computer, " you mean that your computer takes 1 nanosecond to perform each step. That is incredibly fast! A "step" does not, unfortunately, necessarily mean a computation like an addition; computers break such computations down into several stages, which means that the clock speed need not express the computational speed. Computational speed is expressed in units of millions of instructions/second (Mips). Your 1 GHz computer (clock speed) may have a computational speed of 200 Mips.

Computers perform integer (discrete-valued) computations: Computer calculations can be numeric (obeying the laws of arithmetic), logical (obeying the laws of an algebra), or symbolic (obeying any law you like). Each computer instruction that performs an elementary numeric calculation — an addition, a multiplication, or a division — does so only for integers. The sum or product of two integers is also an integer, but the quotient of two integers is likely to not be an integer. How does a computer deal with numbers that have digits to the right of the decimal point? This problem is addressed by using the so-called *floating-point* representation of real numbers. At its heart, however, this representation relies on integer-valued computations.

Representing Numbers

Focusing on numbers, all numbers can represented by the *positional notation system*. 2 The *b*-ary positional representation system uses the position of digits ranging from 0 to *b*-1 to denote a number. The quantity *b* is known as the *base* of the number system. Mathematically, positional systems represent the positive integer *n* as

$$\forall d_k, d_k \in <\text{apply}> \{0,\ldots,b-1\}. </\text{apply}>:$$
$$($$
$$n = \sum_{k=0}^{\infty} \left(d_k b^k \right)$$
$$)$$

and we succinctly express *n* in base-*b* as $n_b = d_N d_{N''1} \ldots d_0$. The number 25 in base 10 equals $2 \times 10^1 + 5 \times 10^0$, so that the*digits* representing this number are d_0=5, d_1=2, and all other d_k equal zero. This same number in *binary* (base 2) equals 11001 ($1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$) and 19 in hexadecimal (base 16). Fractions between zero and one are represented the same way.

$$\forall d_k, d_k \in <\text{apply}> \{0,\ldots,b-1\}. </\text{apply}>:$$
$$($$
$$f = \sum_{k=-\infty}^{-1} \left(d_k b^k \right)$$
$$)$$

All numbers can be represented by their sign, integer and fractional parts. Complex numbers can be thought of as two real numbers that obey special rules to manipulate them.

Humans use base 10, commonly assumed to be due to us having ten fingers. Digital computers use the base 2 or *binary* number representation, each digit of which is known as a *bit* (*b*inary dig*it*).

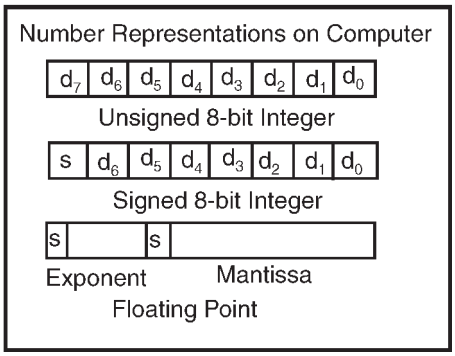


Fig. The Various ways Numbers are Represented in binary are Illustrated. The Number of Bytes for the Exponent and Mantissa Components of floating Point Numbers Varies.

Here, each bit is represented as a voltage that is either "high" or "low, " thereby representing "1" or "0, " respectively. To represent signed values, we tack on a special bit—the *sign bit*—to express the sign. The computer's memory consists of an ordered sequence of *bytes*, a collection of eight bits.

A byte can therefore represent an unsigned number ranging from 0 to255. If we take one of the bits and make it the sign bit, we can make the same byte to represent numbers ranging from "128to 127. But a computer cannot represent *all* possible real numbers. The fault is not with the binary number system; rather having only a finite number of bytes is the problem. While a gigabyte of memory may seem to be a lot, it takes an infinite number of bits to represent ∂.

Since we want to store many numbers in a computer's memory, we are restricted to those that have a *finite*binary representation. Large integers can be represented by an ordered sequence of bytes. Common lengths, usually expressed in terms of the number of bits, are 16, 32, and 64. Thus, an unsigned 32-bit number can represent integers ranging between 0 and2^{32}"1 (4, 294, 967, 295), a number almost big enough to enumerate every human in the world.

Computer Arithmetic and Logic

The binary addition and multiplication tables are:

(

0 + 0 = 0

0 + 1= 1

1 + 1 = 10

1 + 0 = 1

0 × 0 = 0

0 × 1 = 0

1 × 1 = 1

1 × 0 = 0

)

Note that if carries are ignored, subtraction of two single-digit binary numbers yields the same bit as addition. Computers use high and low voltage values to express a bit, and an array of such voltages express numbers akin to positional notation. Logic circuits perform arithmetic operations.

Processors and Memory

There are two main components which have been part of nearly all computer systems ever designed and built. The first is called the processor (known also as the Central Processing Unit or CPU)and the second is called the memory.

The processor is the part of a computer system which does the actual computing. That is, the part which adds, subtracts, multiplies and divides.

Most processors can also compare values and perform conditional actions as a result of such comparisons. Many processors have instructions which perform various types of conversions between different representations of data.

The processor itself is divided into three components that carry out the various functions that the computer is capable of performing. The first component of the processor is the controller. The controller acts as a foreman that oversees the tasks of the processor. The controller looks at the next instruction to be executed and assigns the sub-tasks that must be accomplished to carry out that instruction to the other components of the processor.

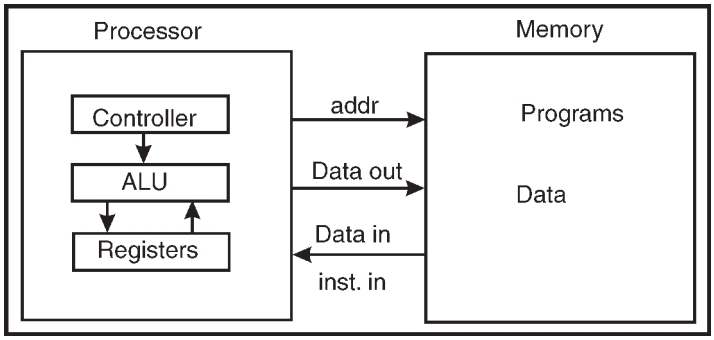
Another component of the processor is the Arithmetic-Logic Unit (ALU). This unit is the part of the processor which performs the mathematical computations and logical tasks that we expect a computer to be able to do. Addition, subtraction, comparison, etc. , are all carried out by the ALU.

The last component of the processor is a collection of one or more registers. Registers are special named memory cells in the processor where information is temporarily stored during various stages of a computation. The currently executing instruction, for example, resides in a register called the Instruction Register. Since modern processors can execute millions of instructions per second it is expected that information would not stay in a register for more than a few millionths of a second.

A computer memory system is accessed (or read) by specifying the location (called an address) of the memory cell. The memory system then responds by producing a copy of the contents of that cell. The original value of the cell is not changed by this process. This is sometimes called a non-destructive read.

A typical computer might be organized as indicated in the following diagram:

A computer memory system is changed (or written) by specifying the location of the memory cell together with the new value for that cell. The previous value stored in the cell is replaced by the new value. This is sometimes called a destructive write.



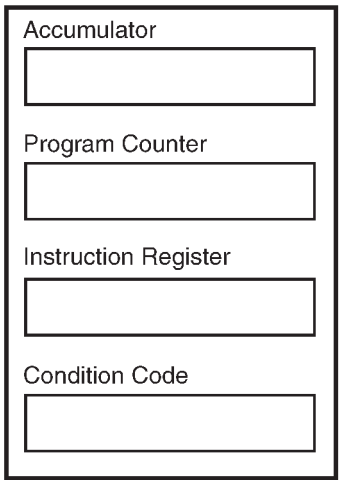
The values stored in a computer memory are simply numbers. Numbers are used to represent both data and instructions. In fact, one cannot distinguish instructions from data when examining the contents of a memory system. It is up to the programmer or operating system to keep track of which memory cells hold data and which are programme instructions. Programs have been written which manipulate and produce programs. Such programs treat instructions as data.

The processor and memory unit are wired together wiring connections called buses. A *bus* is a low resistance connection consisting of 1 or more wires. There are three such connections. The first, called the address bus, is used by the processor to tell the memory system the number or location of the memory cell the processor wishes to access. The second bus is used to send data out to the memory. The third bus is used to transmit data and instructions to the processor. The processor contains a few memory cells, called registers, which are used for efficient temporary storage:

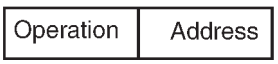
Processor Registers

The Contents of a memory cell or a register is simply a number. For purposes of illustration, suppose that each memory cell is large enough to hold numbers up to 4 decimal digits in size. If a memory cell holds an an instruction, use the

first two decimal digits represent the operation and the remaining digits to represent the adderess or location of the instruction operand.



Instruction format



Using this scheme, we could set up the following operation codes:

```
Operation Code Function
add  01c(acc)=c(acc)+c(addr)
sub  02c(acc)=c(acc)-c(addr)
load  03c(acc)=c(addr)
store  04c(addr)=c(acc)
```

These codes do not correspond to any known real computer, but rather, they are the operation codes for a hypothetical model computer which we will use to illustrate important aspects of computer organization.

Simple Programs

Perhaps the shortest useful programme we might write would move a value from one memory cell to another.

This could be modeled by the J expression:

```
q =: r
```

To accomplish this task, it is necessary to first load the value of r into the accumulator and then store the accumulator in the memory cell q.

```
load r
store q
```

Consider a programme which computes the following expression:

```
c =: a + b
load a
add b
store c
```

It is possible to write a programme which will automatically translate from the J version of this programme to the symbolic form of the machine instructions shown above.

Such a programme, which translates from the J notation, c =: a + b to the machine language:

```
load a
add b
store c
```

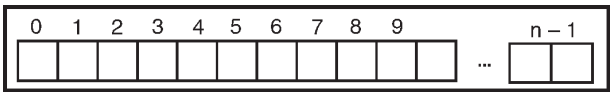
is called a J compiler.

The programme must be stored in memory before it may be executed. Since we don't know exactly where in memory we wish to put either the programme or the data items a, b or c, we can only partially translate the programme into machine readable form as:

```
op location
03 a
01 b
04 c
```

The memory system is a collection of memory cells, each having a number or address.

The following diagram illustrates the organization of memory:



Suppose we decide to locate the above programme beginning in memory cell 4. Since the programme takes 3 cells we might put

the data in the first free cell after the last instruction in the programme which means that we could define:

```
Symbol Location
a7
b8
c9
```

The final translated programme then looks like:

Machine Language Programme

```
location operation address
04  03  07
05  01  08
06  04  09
```

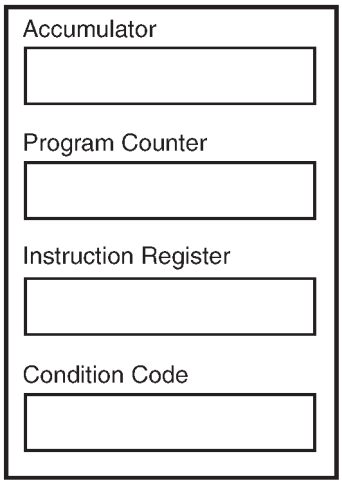
A programme which translates from the symbolic form for a programme:

```
loada
addb
storec
```

to the machine language form is called a symbolic assembler, or simply an assembler.

How the Computer Executes a Programme

We next focus on how a computer executes such a programme. *Recall the organization of the processor*:



The programme counter register always contains the location of the next instruction to be processed.

To get started executing this programme we somehow have to set the contents of the programme counter (pc) to 4 because this is the location of the first instruction of our programme. The following flow chart best explains how the processor executes a programme.

Executing a Programme

To illustrate the various steps in the processor flow chart we give the following trace of the execution of Programme.

Step 0 Set initial value for the pc:

```
pc04
ir?
acc?
```

Step 1 *(fetch)*:

```
pc04
ir0307
ac?
```

Step 2 *(increment pc)*:

```
pc05
ir0307
ac?
```

Step 3 *(execute)*:

```
pc05
ir0307
ac22(here we are supposing     that C(07) has been set
  to 22 somehow and also     suppose C(08) has been
  set to 3)
```

Step 4 *(fetch)*:

```
pc05
ir0108
acc22
```

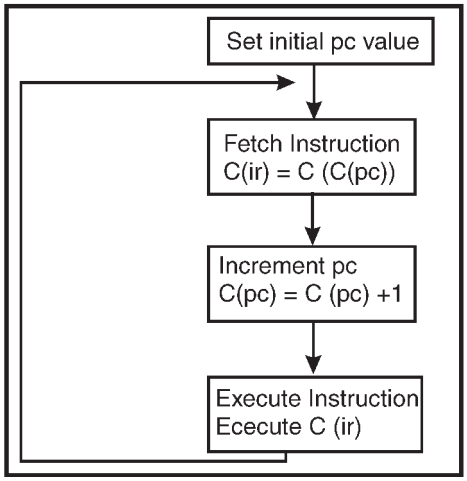
Step 5 *(increment pc)*:

```
pc06
ir0108
```

```
acc22
```
Step 6 *(execute)*:
```
pc06
ir0108
acc25
```
Step 7 *(fetch)*:
```
pc06
ir0409
acc25
```
Step 8 *(increment pc)*:
```
pc07
ir0409
acc25
```
Step 9 *(execute)*:
```
pc07
ir0409
acc25(contents of 09 changed to 25)
```
Step 10 *(fetch)*:
```
pc07
ir0022
acc25
```
Step 11 *(increment pc)*:
```
pc08
ir0022
acc25
```
Step 12 *(execute)*:
```
pc08
ir0022
acc25
```
Of course, at this point we have a problem since the ir does not contain an instruction.

This means we need some kind of more elaborate setup for placing programs in memory and starting and stopping programs. These are some of the tasks performed by a computer operating system.

Processor Flow Chart



Asynchronous Eve nts: Polling Loops and Interrupts

The CPU spends almost all of its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system.

A complete system contains other devices such as:

- A hard disk for storing programmes and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk is used for permanent storage of larger amounts of information, but programmes have to be loaded from disk into main memory before they can actually be executed.).

- A keyboard and mouse for user input.
- A monitor and printer which can be used to display the computer's output.
- An audio output device that allows the computer to play sounds.
- A network interface that allows the computer to communicate with other computers that are connected to it on a network, either wirelessly or by wire.
- A scanner that converts images into coded binary numbers that can be stored and manipulated on the computer.

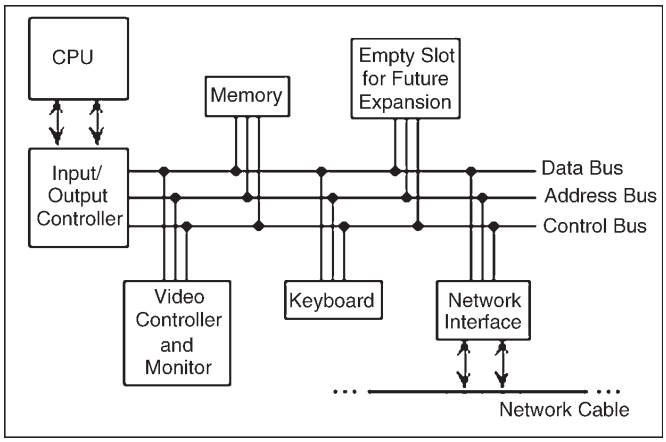
The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). The way this works is that for each device in a system, there is a device driver, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since, the CPU would not be able to communicate with it.

A computer system consisting of many devices is typically organised by connecting those devices to one or more busses. A bus is a set of wires that carry various sorts of information between the devices connected to those wires. The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available

for it on the data bus. A fairly simple computer system might be organised like this:

Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called polling, since, the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, interrupts are often used instead of polling. An interrupt is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt. Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what it is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.



Again, you should understand that this is a purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, the CPU saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the programme counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an interrupt handler that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signalled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

Interrupts allow the CPU to deal with asynchronous events. In the regular fetch-and-execute cycle, things happen in a predetermined order; everything that happens is "synchronised" with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen "asynchronously, " that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on the hard disk. The CPU can access data directly only if it is in main memory. Data on the disk has to be copied into memory before it can be accessed. Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular programme.) Then, instead of just waiting the long and unpredictable amount of time

that the disk drive will take to do this, the CPU goes on with some other task. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

Now, you might have noticed that all this only makes sense if the CPU actually has several tasks to perform. If it has nothing better to do, it might as well spend its time polling for input or waiting for disk drive operations to complete. All modern computers use multitasking to perform several tasks at once. Some computers can be used by several people at once. Since, the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called timesharing. But a modern personal computer with just a single user also uses multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being downloaded over the network.

Each of the individual tasks that the CPU is working on is called a thread. (Or a process; there are technical differences between threads and processes, but they are not important here, since, it is threads that are used in Java.) Many CPUs can literally execute more than one thread simultaneously—such CPUs contain multiple "cores, " each of which can run a thread—but there is always a limit on the number of threads that can be executed at the same time. Since, there are often more threads than can be executed simultaneously, the computer has to be able switch its attention from one thread to another, just as a timesharing computer switches its attention from one user to another.

In general, a thread that is being executed will continue to run until one of several things happens:

- The thread might voluntarily yield control, to give other threads a chance to run.
- The thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be blocked, and other threads, if any, have a chance to run. When the event occurs, an interrupt will "wake up" the thread so that it can continue running.
- The thread might use up its allotted slice of time and be suspended to allow other threads to run. Not all computers can "forcibly" suspend a thread in this way; those that can are said to use preemptive multitasking. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not. All modern desktop and laptop computers use preemptive multitasking.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks or threads that they want the computer to perform; the details of how the computer manages to get all those tasks done are not important to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether.

However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking and multiprocessing. In fact, the

ability to work with threads is fast becoming an essential job skill for programmers. Fortunately, Java has good support for threads, which are built into the Java programming language as a fundamental programming concept.

Just as important in Java and in modern programming in general is the basic concept of asynchronous events. While programmers don't actually deal with interrupts directly, they do often find themselves writing event handlers, which, like interrupt handlers, are called asynchronously when specific events occur. Such "event-driven programming" has a very different feel from the more traditional straight-through, synchronous programming.

By the way, the software that does all the interrupt handling, handles communication with the user and with hardware devices, and controls which thread is allowed to run is called the operating system. The operating system is the basic, essential software without which a computer would not be able to function. Other programmes, such as word processors and World Wide Web browsers, are dependent upon the operating system. Common operating systems include Linux, Windows XP, Windows Vista, and Mac OS.

JDBC Product Components

JDBC includes four components:

- *The JDBC API :* The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4. 0 API is divided into two packages: java. sql and javax. sql. Both packages are included in the Java SE and Java EE platforms.

- *JDBC Driver Manager:* The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

  The Standard Extension packages javax. naming and javax. sql let you use a DataSource object registered with a *Java Naming and Directory Interface™* (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a DataSource object is recommended whenever possible.
- *JDBC Test Suite:* The JDBC driver test suite helps you to determine that JDBC drivers will run your programme. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.
- *JDBC-ODBC Bridge:* The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

This Trail uses the first two of these four JDBC components to connect to a database and then build a java programme that uses

SQL commands to communicate with a test Relational Database. The last two components are used in specialised environments to test web applications, or to communicate with ODBC-aware DBMSs.

JDBC Architecture

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network.

This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

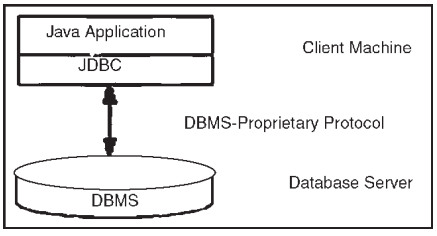


Fig. Two-tier Architecture for Data Access.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS

directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

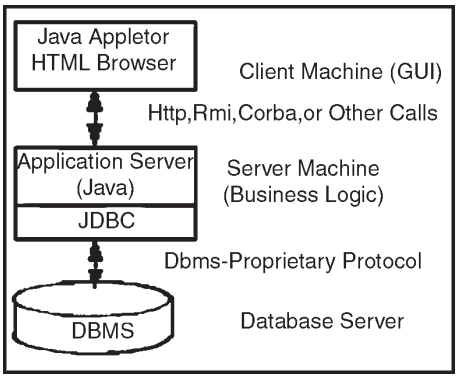


Fig. Three-tier Architecture for Data Access.

Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimising compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

The Employees table illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee. The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use First_Name and Last_Name because the combination of the two also identifies just one row in our sample database.

Table. Employees Table

Employee_Number	First_Name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

Using the last name alone would not work because there are two employees with the last name of "Washington. " In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Yamaguchi gets a job at this company and the primary key is First_Name, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row.

Note that although using First_Name and Last_Name is a unique composite key for this example, it might not be unique in a larger database. Note also that the Employee table assumes that there can be only one car per employee.

Select Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements.

A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First_Name and Last_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

FIirst_Name	Last_Name
Axel	Washington
Florence	Wojokowski

The following code produces a result set that includes the whole table because it asks for all of the columns in the table

Employees with no restrictions (no WHERE clause). Note that SELECT * means "SELECT all columns. "

```
SELECT *
FROM Employees
```

Where Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in LIKE clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions.

This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of WHERE clause involves a join, which is explained in the next section.

Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car.

This information is stored in another table, Cars:

Cars Table

Car_Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car_Number, which is the primary key for the table Cars and the foreign key in the table Employees.

If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car_Number 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car_Number) in the Employees table would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car_Number column in the table Employees because it is possible for an employee not to have a company car. The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the FROM clause lists both Employees and Cars because the requested data is contained in both tables.

Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees. First_Name, Employees. Last_Name,
Cars. Make, Cars. Model, Cars. Year
FROM Employees, Cars
WHERE Employees. Car_Number = Cars. Car_Number
```

This returns a result set that will look similar to the following:

First_Name	Last_Name	Make	Model	Year
Axel	Washington	Honda	Civic DX	1996
Florence	Wojokowski	Toyota	Corolla	1999

Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

A list of the more common DML commands follows:

- Select—used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.
- Insert—adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- Delete—removes a specified row or set of rows from a table
- Update—changes an existing value in a column or group of columns in a table.

The more common DDL commands follow:

- *Create Table:* Creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- *Drop Table:* Deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional. In addition, the behaviour of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.

- *Alter Table:* Adds or removes a column from a table. It also adds or drops table constraints and alters column attributes.

Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many.

A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

Transactions

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data.

For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

Stored Procedures

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-programme, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language.

The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to writ tcode it following.

```
import java. sql. *;
public class UpdateCar {
public static void UpdateCarNum(int carNo, int empNo)
throws SQLException {
Connection con = null;
PreparedStatement pstmt = null;
try {
con = DriverManager. getConnection(
"jdbc:default:connection");
pstmt = con. prepareStatement(
"UPDATE EMPLOYEES " +
"SET CAR_NUMBER = ? " +
"WHERE EMPLOYEE_NUMBER = ?");
pstmt. setInt(1, carNo);
pstmt. setInt(2, empNo);
pstmt. executeUpdate();
}
finally {
if (pstmt != null) pstmt. close();
}
}
}
```

Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth.

Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface DatabaseMetaData, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardised way to get metadata.

In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.