# Computer Graphics Software

**Kelly Berger**

# COMPUTER GRAPHICS SOFTWARE

# COMPUTER GRAPHICS SOFTWARE

Kelly Berger

Computer Graphics Software
by Kelly Berger

Copyright© 2022 BIBLIOTEX

www.bibliotex.com

All rights reserved. No part of this book may be reproduced or used in any manner without the prior written permission of the copyright owner, except for the use brief quotations in a book review.

To request permissions, contact the publisher at info@bibliotex.com

Ebook ISBN: 9781984664051

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

# Contents


**Chapter 1** Introduction 1

**Chapter 2** Computer Graphics 31

**Chapter 3** Computer Software Generations 73

**Chapter 4** Software Components 108

**Chapter 5** Computer Graphics System 144

# 1

# Introduction

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.

## Applications of Computer Graphics

Computers have become a powerful tool for the rapid and economical production of pictures. Advances in computer technology have made interactive computer graphics a practical tool. Today, computer graphics is used in the areas as science, engineering, medicine, business, industry, government, art, entertainment, advertising, education, and training.

### Computer Aided Design

A major use of computer graphics is in design processes, particularly for engineering and architectural systems. For some design applications; objects are first displayed in a wireframe outline form that shows the overall sham and internal features of objects.

Software packages for CAD applications typically provide the designer with a multi-window environment. Each window can show enlarged sections or different views of objects. Standard shapes for electrical, electronic, and logic circuits are often supplied

by the design package. The connections between the components have been mad automatically.

- Animations are often used in CAD applications.
- Real-time animations using wire frame displays are useful for testing performance of a vehicle.
- Wire frame models allow the designer to see the interior parts of the vehicle during motion.
- When object designs are complete, realistic lighting models and surface rendering are applied.
- Manufacturing process of object can also be controlled through CAD.
- Interactive graphics methods are used to layout the buildings.
- Three-dimensional interior layouts and lighting also provided.
- With virtual-reality systems, the designers can go for a simulated walk inside the building.

## Presentation Graphics

- It is used to produce illustrations for reports or to generate slide for with projections.
- Examples of presentation graphics are bar charts, line graphs, surface graphs, pie charts and displays showing relationships between parameters.
- 3-D graphics can provide more attraction to the presentation.

## Computer Art

- Computer graphics methods are widely used in both fine are and commercial art applications.

- The artist uses a combination of 3D modelling packages, texture mapping, drawing programmes and CAD software.
- Pen plotter with specially designed software can create "automatic art".
- "Mathematical Art" can be produced using mathematical functions, fractal procedures.
- These methods are also applied in commercial art.
- Photorealistic techniques are used to render images of a product.
- Animations are also used frequently in advertising, and television commercials are produced frame by frame. Film animations require 24 frames for each second in the animation sequence.
- A common graphics method employed in many commercials is morphing, where one object is transformed into another.

## Entertainment

- CG methods are now commonly used in making motion pictures, music videos and television shows.
- Many TV series regularly employ computer graphics method.
- Graphics objects can be combined with a live action.

## Education and Training

- Computer-generated models of physical, financial and economic systems are often used as educational aids.
- For some training applications, special systems are designed.

  Eg. Training of ship captains, aircraft pilots etc.

- Some simulators have no video screens, but most simulators provide graphics screen for visual operation. Some of them provide only the control panel.

## Visualization

- The numerical and scientific data are converted to a visual form for analysis and to study the behaviour called visualization.
- Producing graphical representation for scientific data sets are calls scientific visualization.
- And business visualization is used to represent the data sets related to commerce and industry.
- The visualization can be either 2D or 3D.

## Image Processing

- Computer graphics is used to create a picture.
- Image processing applies techniques to modify or interpret existing pictures.
- To apply image processing methods, the image must be digitized first.
- Medical applications also make extensive use of image processing techniques for picture enhancements, simulations of operations, etc.

## Graphical User Interface

- Nowadays software packages provide graphics user interface (GUI) for the user to work easily.
- A major component in GUI is a window.
- Multiple windows can be opened at a time.

- To activate any one of the window, the user needs just to check on that window.
- Menus and icons are used for fast selection of processing operations.
- Icons are used as shortcut to perform functions. The advantages of icons are which takes less screen space.
- And some other interfaces like text box, buttons, and list are also used.

## 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.

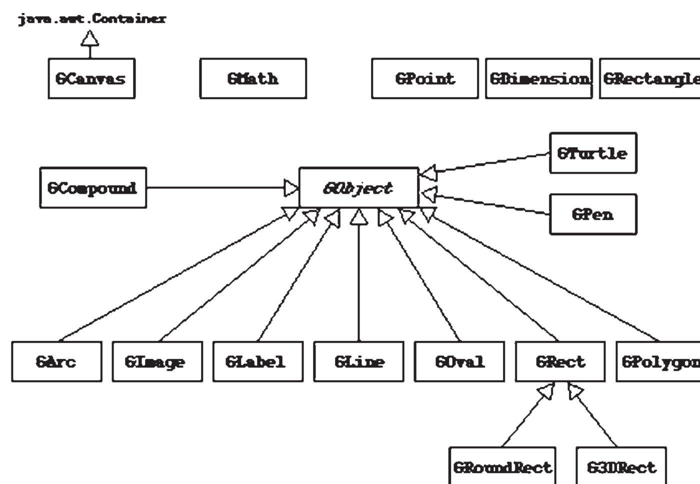



**Fig.** Class Diagram for the acm.graphics Package

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.

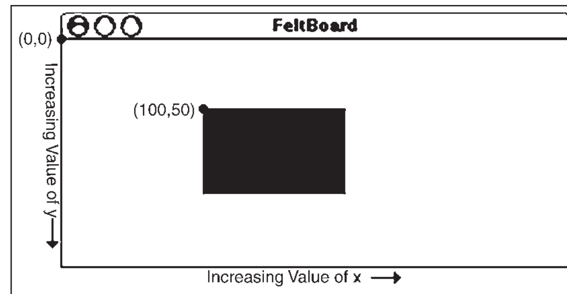



**Fig.** The Java Coordinate System

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

## 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 the GraphicsProgram 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.

## 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

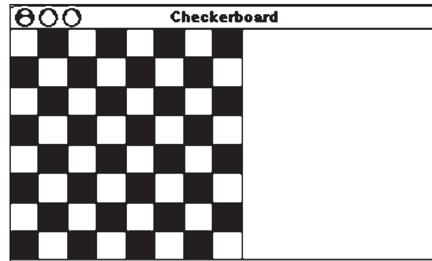
```
new GRect(x, y, width, height)
```

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.

# Modern Computer Graphics

## Motion Capture

Two programmes useful for the realization of a motion capture are MotionBuilder and FaceRobot. FaceRobot is a new computer graphics software for digital acting, intended for professional character animators in the film and game industries. It addresses the technical problems of creating life-like facial animation for realistic human characters with a novel set of algorithms. For motion capture animation, FaceRobot requires a smaller number of markers (only 25 to 30) than traditional high-end approaches.

*FaceRobot provides:*

- Fast organization of the work to create a grid face – animators can quickly import a character and immediately proceed to realistic facial animation,
- Support the processes of creating animations of Autodesk Maya,
- Support for motion capture and key frame animation to achieve optimal results of the work,
- Optimize for game development.

**Fig.** Face Created with the Programme FaceRobot

MotionBuilder is professional 3D character animation software. It is used for motion capture and traditional keyframe animation. It is used in the production of games, films, and other multimedia projects. Functionality includes real-time display and animation

tools, facial and skeletal animation, ragdoll physics, inverse kinematics, story timeline editing etc.

## Simulation of Crowd Behavior

Let us look at three programmes for simulation of collective behaviour of graphical objects. Autodesk Kynaps is the artificial intelligence middleware product. Its purpose is to facilitate the implementation of the mobility function on the scene in real-time characters who are not direct players.

*Kynapse includes:*

- An automatic AI data generation tool,
- A complete 3D pathfinding,
- Spatial reasoning,
- The management of dynamic and destructible terrains,
- Streaming mechanisms to handle very large terrains.

MASSIVE (Multiple Agent Simulation System in Virtual Environment) is a high-end computer animation and artificial intelligence software package used for generating crowd-related visual effects for film and television. Its main feature is the ability to quickly and easily create thousands of agents that all act as individuals. Through the use of fuzzy logic, the software enables every agent to respond individually to its surroundings, including other agents. These reactions affect the agent's behaviour, changing how they act by controlling pre-recorded animation clips, which can come from motion-capture sessions, or can be hand-animated in other 3D animation software packages.

# 2

# Computer Graphics

The development of computer graphics has made computers easier to interact with, and better for understanding and interpreting many types of data. Developments in computer graphics have had a profound impact on many types of media and have revolutionized animation, movies and the video game industry. The term computer graphics has been used in a broad sense to describe "almost everything on computers that is not text or sound". Typically, the term *computer graphics* refers to several different things:

- The representation and manipulation of image data by a computer
- The various technologies used to create and manipulate images
- The images so produced, and

- The sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content, see study of computer graphics

Today, computers and computer-generated images touch many aspects of daily life. Computer imagery is found on television, in newspapers, for example in weather reports, or for example in all kinds of medical investigation and surgical procedures. A well-constructed graph can present complex statistics in a form that is easier to understand and interpret. In the media "such graphs are used to illustrate papers, reports, thesis", and other presentation material. Many powerful tools have been developed to visualize data. Computer generated imagery can be categorized into several different types: 2D, 3D, 4D, 7D, and animated graphics. As technology has improved, 3D computer graphics have become more common, but 2D computer graphics are still widely used.

Computer graphics has emerged as a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Over the past decade, other specialized fields have been developed like information visualization, and scientific visualization more concerned with "the visualization of three dimensional phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources, and so forth, perhaps with a dynamic (time) component". The advance in computer graphics was to come from Ivan Sutherland. In 1961 Sutherland created another computer drawing

programme called Sketchpad. Using a light pen, Sketchpad allowed one to draw simple shapes on the computer screen, save them and even recall them later. The light pen itself had a small photoelectric cell in its tip. This cell emitted an electronic pulse whenever it was placed in front of a computer screen and the screen's electron gun fired directly at it. By simply timing the electronic pulse with the current location of the electron gun, it was easy to pinpoint exactly where the pen was on the screen at any given moment. Once that was determined, the computer could then draw a cursor at that location. Sutherland seemed to find the perfect solution for many of the graphics problems he faced.

Even today, many standards of computer graphics interfaces got their start with this early Sketchpad programme. One example of this is in drawing constraints. If one wants to draw a square for example, s/he doesn't have to worry about drawing four lines perfectly to form the edges of the box. One can simply specify that s/he wants to draw a box, and then specify the location and size of the box. The software will then construct a perfect box, with the right dimensions and at the right location. Another example is that Sutherland's software modeled objects - not just a picture of objects. In other words, with a model of a car, one could change the size of the tires without affecting the rest of the car. It could stretch the body of the car without deforming the tires. These early computer graphics were Vector graphics, composed of thin lines whereas modern day graphics are Raster based using pixels. The difference between vector graphics and raster graphics can be illustrated with a shipwrecked sailor.

He creates an SOS sign in the sand by arranging rocks in the shape of the letters “SOS.” He also has some brightly colored rope, with which he makes a second “SOS” sign by arranging the rope in the shapes of the letters. The rock SOS sign is similar to raster graphics. Every pixel has to be individually accounted for.

The rope SOS sign is equivalent to vector graphics. The computer simply sets the starting point and ending point for the line and perhaps bend it a little between the two end points. The disadvantages to vector files are that they cannot represent continuous tone images and they are limited in the number of colors available. Raster formats on the other hand work well for continuous tone images and can reproduce as many colors as needed. Also in 1961 another student at MIT, Steve Russell, created the first video game, Spacewar. Written for the DEC PDP-1, Spacewar was an instant success and copies started flowing to other PDP-1 owners and eventually even DEC got a copy. The engineers at DEC used it as a diagnostic programme on every new PDP-1 before shipping it. The sales force picked up on this quickly enough and when installing new units, would run the world’s first video game for their new customers.

E. E. Zajac, a scientist at Bell Telephone Laboratory (BTL), created a film called “Simulation of a two-giro gravity attitude control system” in 1963. In this computer generated film, Zajac showed how the attitude of a satellite could be altered as it orbits the Earth. He created the animation on an IBM 7090 mainframe computer. Also at BTL, Ken Knowlton, Frank Sindon and Michael Noll started working

in the computer graphics field. Sindon created a film called Force, Mass and Motion illustrating Newton's laws of motion in operation.

Around the same time, other scientists were creating computer graphics to illustrate their research. At Lawrence Radiation Laboratory, Nelson Max created the films, "Flow of a Viscous Fluid" and "Propagation of Shock Waves in a Solid Form." Boeing Aircraft created a film called "Vibration of an Aircraft." It wasn't long before major corporations started taking an interest in computer graphics. TRW, Lockheed-Georgia, General Electric and Sperry Rand are among the many companies that were getting started in computer graphics by the mid 1960's. IBM was quick to respond to this interest by releasing the IBM 2250 graphics terminal, the first commercially available graphics computer. Ralph Baer, a supervising engineer at Sanders Associates, came up with a home video game in 1966 that was later licensed to Magnavox and called the Odyssey. While very simplistic, and requiring fairly inexpensive electronic parts, it allowed the player to move points of light around on a screen. It was the first consumer computer graphics product.

Also in 1966, Sutherland at MIT invented the first computer controlled head-mounted display (HMD). Called the Sword of Damocles because of the hardware required for support, it displayed two separate wireframe images, one for each eye. This allowed the viewer to see the computer scene in stereoscopic 3D. After receiving his Ph.D. from MIT, Sutherland became Director of Information Processing at ARPA (Advanced Research Projects Agency), and later became a professor at Harvard. Dave Evans was director of engineering at Bendix

Corporation's computer division from 1953 to 1962, after which he worked for the next five years as a visiting professor at Berkeley. There he continued his interest in computers and how they interfaced with people. In 1968 the University of Utah recruited Evans to form a computer science programme, and computer graphics quickly became his primary interest. This new department would become the world's primary research center for computer graphics. In 1967 Sutherland was recruited by Evans to join the computer science programme at the University of Utah. There he perfected his HMD. Twenty years later, NASA would re-discover his techniques in their virtual reality research.

At Utah, Sutherland and Evans were highly sought after consultants by large companies but they were frustrated at the lack of graphics hardware available at the time so they started formulating a plan to start their own company. A student by the name of Edwin Catmull started at the University of Utah in 1970 and signed up for Sutherland's computer graphics class. Catmull had just come from The Boeing Company and had been working on his degree in physics. Growing up on Disney, Catmull loved animation yet quickly discovered that he didn't have the talent for drawing. Now Catmull (along with many others) saw computers as the natural progression of animation and they wanted to be part of the revolution. The first animation that Catmull saw was his own. He created an animation of his hand opening and closing. It became one of his goals to produce a feature length motion picture using computer graphics. In the same class, Fred Parke created an animation of his wife's face.

Because of Evan's and Sutherland's presence, UU was gaining quite a reputation as the place to be for computer graphics research so Catmull went there to learn 3D animation. As the UU computer graphics laboratory was attracting people from all over, John Warnock was one of those early pioneers; he would later found Adobe Systems and create a revolution in the publishing world with his PostScript page description language.

Tom Stockham led the image processing group at UU which worked closely with the computer graphics lab. Jim Clark was also there; he would later found Silicon Graphics, Inc. The first major advance in 3D computer graphics was created at UU by these early pioneers, the hidden-surface algorithm. In order to draw a representation of a 3D object on the screen, the computer must determine which surfaces are "behind" the object from the viewer's perspective, and thus should be "hidden" when the computer creates (or renders) the image.

## Graphic Design

Graphic design is a creative process – most often involving a client and a designer and usually completed in conjunction with producers of form (i.e., printers, programmers, signmakers, etc.) – undertaken in order to convey a specific message (or messages) to a targeted audience. The term "graphic design" can also refer to a number of artistic and professional disciplines that focus on visual communication and presentation. The field as a whole is also often referred to as *Visual Communication* or *Communication Design*. Various methods are used to create and combine words,

symbols, and images to create a visual representation of ideas and messages. A graphic designer may use typography, visual arts and page layout techniques to produce the final result.

Graphic design often refers to both the process (designing) by which the communication is created and the products (designs) which are generated. Common uses of graphic design include identity (logos and branding), web sites, publications (magazines, newspapers, and books), advertisements and product packaging. For example, a product package might include a logo or other artwork, organized text and pure design elements such as shapes and color which unify the piece.

Composition is one of the most important features of graphic design, especially when using pre-existing materials or diverse elements. While Graphic Design as a discipline has a relatively recent history, with the name 'graphic design" first coined by William Addison Dwiggins in 1922, graphic design-like activities span the history of humankind: from the caves of Lascaux, to Rome's Trajan's Column to the illuminated manuscripts of the Middle Ages, to the dazzling neons of Ginza.

In both this lengthy history and in the relatively recent explosion of visual communication in the 20th and 21st centuries, there is sometimes a blurring distinction and over-lapping of advertising art, graphic design and fine art. After all, they share many of the same elements, theories, principles, practices and languages, and sometimes the same benefactor or client. In advertising art the ultimate

objective is the sale of goods and services. In graphic design, "the essence is to give order to information, form to ideas, expression and feeling to artifacts that document human experience."

## Advent of Printing

During the Tang Dynasty (618–907) between the 4th and 7th century AD, wood blocks were cut to print on textiles and later to reproduce Buddhist texts. A Buddhist scripture printed in 868 is the earliest known printed book. Beginning in the 11th century, longer scrolls and books were produced using movable type printing making books widely available during the Song dynasty (960–1279). Sometime around 1450, Johann Gutenberg's printing press made books widely available in Europe. The book design of Aldus Manutius developed the book structure which would become the foundation of western publication design. This era of graphic design is called Humanist or Old Style.

## Emergence of the Design Industry

In late 19th century Europe, especially in the United Kingdom, the movement began to separate graphic design from fine art. In 1849, Henry Cole became one of the major forces in design education in Great Britain, informing the government of the importance of design in his *Journal of Design and Manufactures*. He organized the Great Exhibition as a celebration of modern industrial technology and Victorian design. From 1891 to 1896, William Morris' Kelmscott Press published books that are some of the most significant of the graphic design products of the Arts and Crafts movement, and made a very lucrative business of

creating books of great stylistic refinement and selling them to the wealthy for a premium. Morris proved that a market existed for works of graphic design in their own right and helped pioneer the separation of design from production and from fine art. The work of the Kelmscott Press is characterized by its obsession with historical styles. This historicism was, however, important as it amounted to the first significant reaction to the stale state of nineteenth-century graphic design. Morris' work, along with the rest of the Private Press movement, directly influenced Art Nouveau and is indirectly responsible for developments in early twentieth century graphic design in general.

## Twentieth Century Design

The name "Graphic Design" first appeared in print in the 1922 essay "New Kind of Printing Calls for New Design" by William Addison Dwiggins, an American book designer in the early 20th century. Raffe's *Graphic Design*, published in 1927, is considered to be the first book to use "Graphic Design" in its title. The signage in the London Underground is a classic design example of the modern era and used a font designed by Edward Johnston in 1916. In the 1920s, Soviet constructivism applied 'intellectual production' in different spheres of production. The movement saw individualistic art as useless in revolutionary Russia and thus moved towards creating objects for utilitarian purposes. They designed buildings, theater sets, posters, fabrics, clothing, furniture, logos, menus, etc.

Jan Tschichold codified the principles of modern typography in his 1928 book, *New Typography*. He later repudiated the philosophy he espoused in this book as

being fascistic, but it remained very influential. Tschichold, Bauhaus typographers such as Herbert Bayer and Laszlo Moholy-Nagy, and El Lissitzky have greatly influenced graphic design as we know it today. They pioneered production techniques and stylistic devices used throughout the twentieth century. The following years saw graphic design in the modern style gain widespread acceptance and application. A booming post-World War II American economy established a greater need for graphic design, mainly advertising and packaging. The emigration of the German Bauhaus school of design to Chicago in 1937 brought a "mass-produced" minimalism to America; sparking a wild fire of "modern" architecture and design. Notable names in mid-century modern design include Adrian Frutiger, designer of the typefaces Univers and Frutiger; Paul Rand, who, from the late 1930s until his death in 1996, took the principles of the Bauhaus and applied them to popular advertising and logo design, helping to create a uniquely American approach to European minimalism while becoming one of the principal pioneers of the subset of graphic design known as corporate identity; and Josef Müller-Brockmann, who designed posters in a severe yet accessible manner typical of the 1950s and 1970s era.

The growth of the graphic design industry has grown in parallel with the rise of consumerism. This has raised some concerns and criticisms, notably from within the graphic design community with the First Things First manifesto. First launched by Ken Garland in 1964, it was re-published as the First Things First 2000 manifesto in 1999 in the magazine *Emigre* 51 stating "We propose a reversal of

priorities in favor of more useful, lasting and democratic forms of communication - a mindshift away from product marketing and toward the exploration and production of a new kind of meaning. The scope of debate is shrinking; it must expand. Consumerism is running uncontested; it must be challenged by other perspectives expressed, in part, through the visual languages and resources of design." Both editions attracted signatures from respected design practitioners and thinkers, for example; Rudy VanderLans, Erik Spiekermann, Ellen Lupton and Rick Poynor. The 2000 manifesto was also notably published in Adbusters, known for its strong critiques of visual culture.

## Applications

From road signs to technical schematics, from interoffice memorandums to reference manuals, graphic design enhances transfer of knowledge. Readability is enhanced by improving the visual presentation of text. Design can also aid in selling a product or idea through effective visual communication. It is applied to products and elements of company identity like logos, colors, packaging, and text. Together these are defined as branding. Branding has increasingly become important in the range of services offered by many graphic designers, alongside corporate identity. Whilst the terms are often used interchangeably, branding is more strictly related to the identifying mark or trade name for a product or service, whereas corporate identity can have a broader meaning relating to the structure and ethos of a company, as well as to the company's external image.

Graphic designers will often form part of a team working on corporate identity and branding projects. Other members of that team can include marketing professionals, communications consultants and commercial writers. Textbooks are designed to present subjects such as geography, science, and math. These publications have layouts which illustrate theories and diagrams. A common example of graphics in use to educate is diagrams of human anatomy. Graphic design is also applied to layout and formatting of educational material to make the information more accessible and more readily understandable. Graphic design is applied in the entertainment industry in decoration, scenery, and visual story telling.

Other examples of design for entertainment purposes include novels, comic books, DVD covers, opening credits and closing credits in film, and programmes and props on stage. This could also include artwork used for t-shirts and other items screenprinted for sale. From scientific journals to news reporting, the presentation of opinion and facts is often improved with graphics and thoughtful compositions of visual information - known as information design. Newspapers, magazines, blogs, television and film documentaries may use graphic design to inform and entertain. With the advent of the web, information designers with experience in interactive tools such as Adobe Flash are increasingly being used to illustrate the background to news stories.

## Skills

A graphic design project may involve the stylization and presentation of existing text and either preexisting imagery

or images developed by the graphic designer. For example, a newspaper story begins with the journalists and photojournalists and then becomes the graphic designer's job to organize the page into a reasonable layout and determine if any other graphic elements should be required. In a magazine article or advertisement, often the graphic designer or art director will commission photographers or illustrators to create original pieces just to be incorporated into the design layout. Or the designer may utilize stock imagery or photography. Contemporary design practice has been extended to the modern computer, for example in the use of WYSIWYG user interfaces, often referred to as interactive design, or multimedia design.

## Visual Arts

Before any graphic elements may be applied to a design, the graphic elements must be originated by means of visual art skills. These graphics are often (but not always) developed by a graphic designer. Visual arts include works which are primarily visual in nature using anything from traditional media, to photography or computer generated art. Graphic design principles may be applied to each graphic art element individually as well as to the final composition.

## Typography

Typography is the art, craft and techniques of type design, modifying type glyphs, and arranging type. Type glyphs (characters) are created and modified using a variety of illustration techniques. The arrangement of type is the selection of typefaces, point size, line length, leading (line spacing) and letter spacing. Typography is performed by

typesetters, compositors, typographers, graphic artists, art directors, and clerical workers. Until the Digital Age, typography was a specialized occupation. Digitization opened up typography to new generations of visual designers and lay users.

## Page Layout

The page layout aspect of graphic design deals with the arrangement of elements (content) on a page, such as image placement, and text layout and style. Beginning from early illuminated pages in hand-copied books of the Middle Ages and proceeding down to intricate modern magazine and catalogue layouts, structured page design has long been a consideration in printed material. With print media, elements usually consist of type (text), images (pictures), and occasionally place-holder graphics for elements that are not printed with ink such as die/laser cutting, foil stamping or blind embossing.

## Interface Design

Since the advent of the World Wide Web and computer software development, many graphic designers have become involved in interface design. This has included web design and software design, when end user interactivity is a design consideration of the layout or interface. Combining visual communication skills with the interactive communication skills of user interaction and online branding, graphic designers often work with software developers and web developers to create both the look and feel of a web site or software application and enhance the interactive experience of the user or web site visitor. An important aspect of interface design is icon design.

## Printmaking

Printmaking is the process of making artworks by printing on paper and other materials or surfaces. Except in the case of monotyping, the process is capable of producing multiples of the same piece, which is called a print. Each piece is not a copy but an original since it is not a reproduction of another work of art and is technically known as an impression. Painting or drawing, on the other hand, create a unique original piece of artwork. Prints are created from a single original surface, known technically as a matrix. Common types of matrices include: plates of metal, usually copper or zinc for engraving or etching; stone, used for lithography; blocks of wood for woodcuts, linoleum for linocuts and fabric plates for screen-printing. But there are many other kinds, discussed below. Works printed from a single plate create an edition, in modern times usually each signed and numbered to form a limited edition. Prints may also be published in book form, as artist's books. A single print could be the product of one or multiple techniques.

## Tools

The mind may be the most important graphic design tool. Aside from technology, graphic design requires judgment and creativity. Critical, observational, quantitative and analytic thinking are required for design layouts and rendering. If the executor is merely following a solution (e.g. sketch, script or instructions) provided by another designer (such as an art director), then the executor is not usually considered the designer. The method of presentation (e.g. arrangement, style, medium) may be equally important to the design. The layout is produced using external traditional

or digital image editing tools. The appropriate development and presentation tools can substantially change how an audience perceives a project. In the mid 1980s, the arrival of desktop publishing and graphic art software applications introduced a generation of designers to computer image manipulation and creation that had previously been manually executed. Computer graphic design enabled designers to instantly see the effects of layout or typographic changes, and to simulate the effects of traditional media without requiring a lot of space. However, traditional tools such as pencils or markers are useful even when computers are used for finalization; a designer or art director may hand sketch numerous concepts as part of the creative process.

Some of these sketches may even be shown to a client for early stage approval, before the designer develops the idea further using a computer and graphic design software tools. Computers are considered an indispensable tool in the graphic design industry. Computers and software applications are generally seen by creative professionals as more effective production tools than traditional methods. However, some designers continue to use manual and traditional tools for production, such as Milton Glaser. New ideas can come by way of experimenting with tools and methods. Some designers explore ideas using pencil and paper. Others use many different mark-making tools and resources from computers to sculpture as a means of inspiring creativity. One of the key features of graphic design is that it makes a tool out of appropriate image selection in order to possibly convey meaning.ArtsComputers and the Creative Process

There is some debate whether computers enhance the creative process of graphic design. Rapid production from the computer allows many designers to explore multiple ideas quickly with more detail than what could be achieved by traditional hand-rendering or paste-up on paper, moving the designer through the creative process more quickly. However, being faced with limitless choices does not help isolate the best design solution and can lead to endless iterations with no clear design outcome. A graphic designer may use sketches to explore multiple or complex ideas quickly without the distractions and complications of software. Hand-rendered comps are often used to get approval for an idea execution before a design invests time to produce finished visuals on a computer or in paste-up. The same thumbnail sketches or rough drafts on paper may be used to rapidly refine and produce the idea on the computer in a hybrid process. This hybrid process is especially useful in logo design where a software learning curve may detract from a creative thought process. The traditional-design/computer-production hybrid process may be used for freeing one's creativity in page layout or image development as well. In the early days of computer publishing, many 'traditional' graphic designers relied on computer-savvy production artists to produce their ideas from sketches, without needing to learn the computer skills themselves. However, this practice has been increasingly less common since the advent of desktop publishing over 30 years ago. The use of computers and graphics software is now taught in most graphic design courses.

## Occupations

Graphic design career paths cover all ends of the creative spectrum and often overlap. The main job responsibility of a Graphic Designer is the arrangement of visual elements in some type of media. The main job titles include graphic designer, art director, creative director, and the entry level production artist. Depending on the industry served, the responsibilities may have different titles such as “DTP Associate” or “Graphic Artist”, but despite changes in title, graphic design principles remain consistent. The responsibilities may come from or lead to specialized skills such as illustration, photography or interactive design. Today’s graduating graphic design students are normally exposed to all of these areas of graphic design and urged to become familiar with all of them as well in order to be competitive. Graphic designers can work in a variety of environments. Whilst many will work within companies devoted specifically to the industry, such as design consultancies or branding agencies, others may work within publishing, marketing or other communications companies. Increasingly, especially since the introduction of personal computers to the industry, many graphic designers have found themselves working within non-design oriented organizations, as in-house designers. Graphic designers may also work as free-lance designers, working on their own terms, prices, ideas, etc.

A graphic designer reports to the art director, creative director or senior media creative. As a designer becomes more senior, they may spend less time designing media and more time leading and directing other designers on broader

creative activities, such as brand development and corporate identity development. As graphic designers become more senior, they are often expected to interact more directly with clients.

## Image Types: 2D Computer Graphics

2D computer graphics are the computer-based generation of digital images—mostly from two-dimensional models, such as 2D geometric models, text, and digital images, and by techniques specific to them. 2D computer graphics are mainly used in applications that were originally developed upon traditional printing and drawing technologies, such as typography, cartography, technical drawing, advertising, etc.. In those applications, the two-dimensional image is not just a representation of a real-world object, but an independent artifact with added semantic value; two-dimensional models are therefore preferred, because they give more direct control of the image than 3D computer graphics, whose approach is more akin to photography than to typography.

### Pixel Art

Pixel art is a form of digital art, created through the use of raster graphics software, where images are edited on the pixel level. Graphics in most old (or relatively limited) computer and video games, graphing calculator games, and many mobile phone games are mostly pixel art.

### Vector Graphics

Vector graphics formats are complementary to raster graphics, which is the representation of images as an array

of pixels, as it is typically used for the representation of photographic images Vector graphics consists in encoding information about shapes and colors that comprise the image, which can allow for more flexibility in rendering. There are instances when working with vector tools and formats is best practice, and instances when working with raster tools and formats is best practice. There are times when both formats come together. An understanding of the advantages and limitations of each technology and the relationship between them is most likely to result in efficient and effective use of tools.

## 3D Computer Graphics

3D computer graphics in contrast to 2D computer graphics are graphics that use a three-dimensional representation of geometric data that is stored in the computer for the purposes of performing calculations and rendering 2D images. Such images may be for later display or for real-time viewing. Despite these differences, 3D computer graphics rely on many of the same algorithms as 2D computer vector graphics in the wire frame model and 2D computer raster graphics in the final rendered display. In computer graphics software, the distinction between 2D and 3D is occasionally blurred; 2D applications may use 3D techniques to achieve effects such as lighting, and primarily 3D may use 2D rendering techniques. 3D computer graphics are often referred to as 3D models. Apart from the rendered graphic, the model is contained within the graphical data file. However, there are differences. A 3D model is the mathematical representation of any three-dimensional object. A model is not technically a graphic until it is visually

displayed. Due to 3D printing, 3D models are not confined to virtual space. A model can be displayed visually as a two-dimensional image through a process called *3D rendering*, or used in non-graphical computer simulations and calculations. There are some 3D computer graphics software for users to create 3D images.

## Computer Animation

Computer animation is the art of creating moving images via the use of computers. It is a subfield of computer graphics and animation. Increasingly it is created by means of 3D computer graphics, though 2D computer graphics are still widely used for stylistic, low bandwidth, and faster real-time rendering needs. Sometimes the target of the animation is the computer itself, but sometimes the target is another medium, such as film. It is also referred to as CGI (Computer-generated imagery or computer-generated imaging), especially when used in films. Virtual entities may contain and be controlled by assorted attributes, such as transform values (location, orientation, and scale) stored in an object's transformation matrix. Animation is the change of an attribute over time. Multiple methods of achieving animation exist; the rudimentary form is based on the creation and editing of keyframes, each storing a value at a given time, per attribute to be animated. The 2D/3D graphics software will interpolate between keyframes, creating an editable curve of a value mapped over time, resulting in animation.

Other methods of animation include procedural and expression-based techniques: the former consolidates related elements of animated entities into sets of attributes, useful

for creating particle effects and crowd simulations; the latter allows an evaluated result returned from a user-defined logical expression, coupled with mathematics, to automate animation in a predictable way (convenient for controlling bone behavior beyond what a hierarchy offers in skeletal system set up). To create the illusion of movement, an image is displayed on the computer screen then quickly replaced by a new image that is similar to the previous image, but shifted slightly. This technique is identical to the illusion of movement in television and motion pictures.

## Concepts and Principles

Images are typically produced by optical devices;such as cameras, mirrors, lenses, telescopes, microscopes, etc. and natural objects and phenomena, such as the human eye or water surfaces. A digital image is a representation of a two-dimensional image in binary format as a sequence of ones and zeros. Digital images include both vector images and raster images, but raster images are more commonly used.

## Pixel

In digital imaging, a pixel (or picture element) is a single point in a raster image. Pixels are normally arranged in a regular 2-dimensional grid, and are often represented using dots or squares. Each pixel is a sample of an original image, where more samples typically provide a more accurate representation of the original. The intensity of each pixel is variable; in color systems, each pixel has typically three components such as red, green, and blue.

## Graphics

Graphics are visual presentations on some surface, such as a wall, canvas, computer screen, paper, or stone to brand, inform, illustrate, or entertain. Examples are photographs, drawings, line art, graphs, diagrams, typography, numbers, symbols, geometric designs, maps, engineering drawings, or other images. Graphics often combine text, illustration, and color. Graphic design may consist of the deliberate selection, creation, or arrangement of typography alone, as in a brochure, flier, poster, web site, or book without any other element. Clarity or effective communication may be the objective, association with other cultural elements may be sought, or merely, the creation of a distinctive style.

## Rendering

Rendering is the process of generating an image from a model (or models in what collectively could be called a *scene* file), by means of computer programmes. A scene file contains objects in a strictly defined language or data structure; it would contain geometry, viewpoint, texture, lighting, and shading information as a description of the virtual scene. The data contained in the scene file is then passed to a rendering programme to be processed and output to a digital image or raster graphics image file. The rendering programme is usually built into the computer graphics software, though others are available as plug-ins or entirely separate programmes. The term "rendering" may be by analogy with an "artist's rendering" of a scene. Though the technical details of rendering methods vary, the general

challenges to overcome in producing a 2D image from a 3D representation stored in a scene file are outlined as the graphics pipeline along a rendering device, such as a GPU. A GPU is a purpose-built device able to assist a CPU in performing complex rendering calculations. If a scene is to look relatively realistic and predictable under virtual lighting, the rendering software should solve the rendering equation. The rendering equation doesn't account for all lighting phenomena, but is a general lighting model for computer-generated imagery. 'Rendering' is also used to describe the process of calculating effects in a video editing file to produce final video output.

## 3D Projection

3D projection is a method of mapping three dimensional points to a two dimensional plane. As most current methods for displaying graphical data are based on planar two dimensional media, the use of this type of projection is widespread, especially in computer graphics, engineering and drafting.

## Ray Tracing

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane. The technique is capable of producing a very high degree of photorealism; usually higher than that of typical scanline rendering methods, but at a greater computational cost.

## Shading

Shading refers to depicting depth in 3D models or illustrations by varying levels of darkness. It is a process

used in drawing for depicting levels of darkness on paper by applying media more densely or with a darker shade for darker areas, and less densely or with a lighter shade for lighter areas. There are various techniques of shading including cross hatching where perpendicular lines of varying closeness are drawn in a grid pattern to shade an area. The closer the lines are together, the darker the area appears. Likewise, the farther apart the lines are, the lighter the area appears. The term has been recently generalized to mean that shaders are applied.

## Texture Mapping

Texture mapping is a method for adding detail, surface texture, or colour to a computer-generated graphic or 3D model. Its application to 3D graphics was pioneered by Dr Edwin Catmull in 1974. A texture map is applied (mapped) to the surface of a shape, or polygon. This process is akin to applying patterned paper to a plain white box. Multitexturing is the use of more than one texture at a time on a polygon. Procedural textures (created from adjusting parameters of an underlying algorithm that produces an output texture), and bitmap textures (created in an image editing application) are, generally speaking, common methods of implementing texture definition from a 3D animation programme, while intended placement of textures onto a model's surface often requires a technique known as UV mapping.

## Anti-aliasing

Rendering resolution-independent entities (such as 3D models) for viewing on a raster (pixel-based) device such as

a LCD display or CRT television inevitably causes aliasing artifacts mostly along geometric edges and the boundaries of texture details; these artifacts are informally called "jaggies". Anti-aliasing methods rectify such problems, resulting in imagery more pleasing to the viewer, but can be somewhat computationally expensive. Various anti-aliasing algorithms (such as supersampling) are able to be employed, then customized for the most efficient rendering performance versus quality of the resultant imagery; a graphics artist should consider this trade-off if anti-aliasing methods are to be used. A pre-anti-aliased bitmap texture being displayed on a screen (or screen location) at a resolution different than the resolution of the texture itself (such as a textured model in the distance from the virtual camera) will exhibit aliasing artifacts, while any procedurally-defined texture will always show aliasing artifacts as they are resolution-independent; techniques such as mipmapping and texture filtering help to solve texture-related aliasing problems.

## Volume Rendering

Volume rendering is a technique used to display a 2D projection of a 3D discretely sampled data set. A typical 3D data set is a group of 2D slice images acquired by a CT or MRI scanner. Usually these are acquired in a regular pattern (e.g., one slice every millimeter) and usually have a regular number of image pixels in a regular pattern. This is an example of a regular volumetric grid, with each volume element, or voxel represented by a single value that is obtained by sampling the immediate area surrounding the voxel.

## 3D Modeling

3D modeling is the process of developing a mathematical, wireframe representation of any three-dimensional object, called a "3D model", via specialized software. 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. 3D models may be created using multiple approaches: use of NURBS curves to generate accurate and smooth surface patches, polygonal mesh modeling (manipulation of faceted geometry), or polygonal mesh subdivision (advanced tessellation of polygons, resulting in smooth surfaces similar to NURBS models). A 3D model can be displayed as a two-dimensional image through a process called *3D rendering*, used in a computer simulation of physical phenomena, or animated directly for other purposes. The model can also be physically created using 3D Printing devices.

## Pioneers in Graphic Design

### Charles Csuri

Charles Csuri is a pioneer in computer animation and digital fine art and created the first computer art in 1964. Csuri was recognized by *Smithsonian* as the father of digital art and computer animation, and as a pioneer of computer animation by the Museum of Modern Art (MoMA) and Association for Computing Machinery-SIGGRAPH.

### Donald P. Greenberg

Donald P. Greenberg is a leading innovator in computer graphics. Greenberg has authored hundreds of articles and served as a teacher and mentor to many prominent computer

graphic artists, animators, and researchers such as Robert L. Cook, Marc Levoy, and Wayne Lytle. Many of his former students have won Academy Awards for technical achievements and several have won the SIGGRAPH Achievement Award. Greenberg was the founding director of the NSF Center for Computer Graphics and Scientific Visualization.

## A. Michael Noll

Noll was one of the first researchers to use a digital computer to create artistic patterns and to formalize the use of random processes in the creation of visual arts. He began creating digital computer art in 1962, making him one of the earliest digital computer artists. In 1965, Noll along with Frieder Nake and Georg Nees were the first to publicly exhibit their computer art. During April 1965, the Howard Wise Gallery exhibited Noll's computer art along with random-dot patterns by Bela Julesz.

## Other Pioneers

- Jim Blinn
- Arambilet
- Benoît B. Mandelbrot
- Henri Gouraud
- Bui Tuong Phong
- Pierre Bézier
- Paul de Casteljau
- Daniel J. Sandin
- Alvy Ray Smith
- Ton Roosendaal

- Ivan Sutherland
- Steve Russell

### Computer Graphics Study

The study of computer graphics is a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Although the term often refers to three-dimensional computer graphics, it also encompasses two-dimensional graphics and image processing. As an academic discipline, computer graphics studies the manipulation of visual and geometric information using computational techniques. It focuses on the *mathematical* and *computational* foundations of image generation and processing rather than purely aesthetic issues. Computer graphics is often differentiated from the field of visualization, although the two fields have many similarities.

## Computer-aided Industrial Design

Computer-aided industrial design (CAID) is a subset of computer-aided design (CAD) that includes software that directly helps in product development. Within CAID programmes designers have the freedom of creativity, but typically follow a simple design methodology:

- Creating sketches, using a stylus
- Generating curves directly from the sketch
- Generating surfaces directly from the curves

The end result is a 3D model that projects the main design intent the designer had in mind. The model can then be saved in STL format to send it to a rapid prototyping

machine to create the real-life model. CAID helps the designer to focus on the technical part of the design methodology rather than taking care of sketching and modeling—then contributing to the selection of a better product proposal in less time.

Later, when the requisites and parameters of the product have been defined by means of using CAID software, the designer can import the result of his work into a CAD programme (typically a Solid Modeler) for adjustments prior to production and generation of blueprints and manufacturing processes. What differentiates CAID from CAD is that the former is far more conceptual and less technical than the latter. Within a CAID programme, the designer can express him/herself without extents, whilst in CAD software there is always the manufacturing factor.

## Electronic Design Automation

Electronic design automation (EDA or ECAD) is a category of software tools for designing electronic systems such as printed circuit boards and integrated circuits. The tools work together in a design flow that chip designers use to design and analyze entire semiconductor chips.

### History

### Early Days

Before EDA, integrated circuits were designed by hand, and manually laid out. Some advanced shops used geometric software to generate the tapes for the Gerber photoplotter, but even those copied digital recordings of mechanically-drawn components. The process was fundamentally graphic,

with the translation from electronics to graphics done manually. The best known company from this era was Calma, whose GDSII format survives. By the mid-70s, developers started to automate the design, and not just the drafting. The first placement and routing (Place and route) tools were developed. The proceedings of the Design Automation Conference cover much of this era. The next era began about the time of the publication of “Introduction to VLSI Systems” by Carver Mead and Lynn Conway in 1980. This ground breaking text advocated chip design with programming languages that compiled to silicon.

The immediate result was a considerable increase in the complexity of the chips that could be designed, with improved access to design verification tools that used logic simulation. Often the chips were easier to lay out and more likely to function correctly, since their designs could be simulated more thoroughly prior to construction. Although the languages and tools have evolved, this general approach of specifying the desired behavior in a textual programming language and letting the tools derive the detailed physical design remains the basis of digital IC design today. The earliest EDA tools were produced academically. One of the most famous was the “Berkeley VLSI Tools Tarball”, a set of UNIX utilities used to design early VLSI systems. Still widely used is the Espresso heuristic logic minimizer and Magic. Another crucial development was the formation of MOSIS, a consortium of universities and fabricators that developed an inexpensive way to train student chip designers by producing real integrated circuits. The basic concept was to use reliable, low-cost, relatively low-technology IC

processes, and pack a large number of projects per wafer, with just a few copies of each projects' chips. Cooperating fabricators either donated the processed wafers, or sold them at cost, seeing the programme as helpful to their own long-term growth.

## Birth of Commercial EDA

1981 marks the beginning of EDA as an industry. For many years, the larger electronic companies, such as Hewlett Packard, Tektronix, and Intel, had pursued EDA internally. In 1981, managers and developers spun out of these companies to concentrate on EDA as a business. Daisy Systems, Mentor Graphics, and Valid Logic Systems were all founded around this time, and collectively referred to as DMV. Within a few years there were many companies specializing in EDA, each with a slightly different emphasis. The first trade show for EDA was held at the Design Automation Conference in 1984. In 1986, Verilog, a popular high-level design language, was first introduced as a hardware description language by Gateway Design Automation. In 1987, the U.S. Department of Defense funded creation of VHDL as a specification language. Simulators quickly followed these introductions, permitting direct simulation of chip designs: executable specifications. In a few more years, back-ends were developed to perform logic synthesis.

## Current Status

Current digital flows are extremely modular (see Integrated circuit design, Design closure, and Design flow (EDA)). The front ends produce standardized design descriptions that

compile into invocations of "cells,", without regard to the cell technology. Cells implement logic or other electronic functions using a particular integrated circuit technology. Fabricators generally provide libraries of components for their production processes, with simulation models that fit standard simulation tools. Analog EDA tools are far less modular, since many more functions are required, they interact more strongly, and the components are (in general) less ideal. EDA for electronics has rapidly increased in importance with the continuous scaling of semiconductor technology. Some users are foundry operators, who operate the semiconductor fabrication facilities, or "fabs", and design-service companies who use EDA software to evaluate an incoming design for manufacturing readiness. EDA tools are also used for programming design functionality into FPGAs.

## Software Focuses

## Design

- High-level synthesis(syn. behavioural synthesis, algorithmic synthesis) For digital chips
- Logic synthesis translation of abstract, logical language such as Verilog or VHDL into a discrete netlist of logic-gates
- Schematic Capture For standard cell digital, analog, rf like Capture CIS in Orcad by CADENCE and ISIS in Proteus
- Layout like Layout in Orcad by Cadence, ARES in Proteus

## Design Flows

Design flows are the explicit combination of electronic design automation tools to accomplish the design of an integrated circuit. Moore's law has driven the entire IC implementation RTL to GDSII design flows from one which uses primarily standalone synthesis, placement, and routing algorithms to an integrated construction and analysis flows for design closure. The challenges of rising interconnect delay led to a new way of thinking about and integrating design closure tools. New scaling challenges such as leakage power, variability, and reliability will keep on challenging the current state of the art in design closure. The RTL to GDSII flow underwent significant changes from 1980 through 2005. The continued scaling of CMOS technologies significantly changed the objectives of the various design steps.

The lack of good predictors for delay has led to significant changes in recent design flows. Challenges like leakage power, variability, and reliability will continue to require significant changes to the design closure process in the future. Many factors describe what drove the design flow from a set of separate design steps to a fully integrated approach, and what further changes are coming to address the latest challenges. In his keynote at the 40th Design Automation Conference entitled The Tides of EDA, Alberto Sangiovanni-Vincentelli distinguished three periods of EDA: The Age of the Gods, The Age of the Heroes, and The Age of the Men. These eras were characterized respectively by senses, imagination, and reason. When we limit ourselves to the RTL to GDSII flow of the CAD area, we can distinguish

three main eras in its development: the Age of Invention, the Age of Implementation, and the Age of Integration.

- The Age of Invention: During the invention era, routing, placement, static timing analysis and logic synthesis were invented.
- The Age of Implementation: In the age of implementation, these steps were drastically improved by designing sophisticated data structures and advanced algorithms. This allowed the tools in each of these design steps to keep pace with the rapidly increasing design sizes. However, due to the lack of good predictive cost functions, it became impossible to execute a design flow by a set of discrete steps, no matter how efficiently each of the steps was implemented.
- The Age of Integration: This led to the age of integration where most of the design steps are performed in an integrated environment, driven by a set of incremental cost analyzers.

## Simulation

- Transistor simulation – low-level transistor-simulation of a schematic/layout's behavior, accurate at device-level.
- Logic simulation – digital-simulation of an RTL or gate-netlist's digital (boolean 0/1) behavior, accurate at boolean-level.
- Behavioral Simulation – high-level simulation of a design's architectural operation, accurate at cycle-level or interface-level.

- Hardware emulation – Use of special purpose hardware to emulate the logic of a proposed design. Can sometimes be plugged into a system in place of a yet-to-be-built chip; this is called in-circuit emulation.
- Technology CAD simulate and analyze the underlying process technology. Electrical properties of devices are derived directly from device physics.
- Electromagnetic field solvers, or just field solvers, solve Maxwell's equations directly for cases of interest in IC and PCB design. They are known for being slower but more accurate than the layout extraction above.

## Electronic Circuit Simulation

Electronic circuit simulation uses mathematical models to replicate the behavior of an actual electronic device or circuit. Simulation software allows for modeling of circuit operation and is an invaluable analysis tool. Due to its highly accurate modeling capability, many Colleges and Universities use this type of software for the teaching of electronics technician and electronics engineering programmes. Electronics simulation software engages the user by integrating them into the learning experience. These kinds of interactions actively engage learners to analyze, synthesize, organize, and evaluate content and result in learners constructing their own knowledge. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs.

In particular, for integrated circuits, the tooling (photomasks) is expensive, breadboards are impractical, and probing the behavior of internal signals is extremely difficult. Therefore almost all IC design relies heavily on simulation. The most well known analog simulator is SPICE. Probably the best known digital simulators are those based on Verilog and VHDL. Some electronics simulators integrate a schematic editor, a simulation engine, and on-screen waveforms, and make "what-if" scenarios easy and instant. They also typically contain extensive model and device libraries. These models typically include IC specific transistor models such as BSIM, generic components such as resistors, capacitors, inductors and transformers, user defined models (such as controlled current and voltage sources, or models in Verilog-A or VHDL-AMS). Printed circuit board (PCB) design requires specific models as well, such as transmission lines for the traces and IBIS models for driving and receiving electronics.

## Types

While there are strictly analog electronics circuit simulators, popular simulators often include both analog and event-driven digital simulation capabilities, and are known as mixed-mode simulators. This means that any simulation may contain components that are analog, event driven (digital or sampled-data), or a combination of both. An entire mixed signal analysis can be driven from one integrated schematic. All the digital models in mixed-mode simulators provide accurate specification of propagation time and rise/fall time delays.

The event driven algorithm provided by mixed-mode simulators is general purpose and supports non-digital

types of data. For example, elements can use real or integer values to simulate DSP functions or sampled data filters. Because the event driven algorithm is faster than the standard SPICE matrix solution, simulation time is greatly reduced for circuits that use event driven models in place of analog models. Mixed-mode simulation is handled on three levels; (a) with primitive digital elements that use timing models and the built-in 12 or 16 state digital logic simulator, (b) with subcircuit models that use the actual transistor topology of the integrated circuit, and finally, (c) with In-line Boolean logic expressions.

Exact representations are used mainly in the analysis of transmission line and signal integrity problems where a close inspection of an IC’s I/O characteristics is needed. Boolean logic expressions are delay-less functions that are used to provide efficient logic signal processing in an analog environment. These two modeling techniques use SPICE to solve a problem while the third method, digital primitives, use mixed mode capability.

Each of these methods has its merits and target applications. In fact, many simulations (particularly those which use A/D technology) call for the combination of all three approaches. No one approach alone is sufficient. Another type of simulation used mainly for power electronics represent piecewise linear algorithms. These algorithms use an analog (linear) simulation until a power electronic switch changes its state. At this time a new analog model is calculated to be used for the next simulation period. This methodology both enhances simulation speed and stability significantly.

## Complexities

Often circuit simulators do not take into account the process variations that occur when the design is fabricated into silicon. These variations can be small, but taken together can change the output of a chip significantly. Process variations occur in the manufacture of circuits in silicon. Temperature variation can also be modeled to simulate the circuit's performance through temperature ranges.

## Analysis and Verification

- Functional verification
- Clock Domain Crossing Verification (CDC check): Similar to linting, but these checks/tools specialize in detecting and reporting potential issues like data loss, meta-stability due to use of multiple clock domains in the design.
- Formal verification, also model checking: Attempts to prove, by mathematical methods, that the system has certain desired properties, and that certain undesired effects (such as deadlock) cannot occur.
- Equivalence checking: algorithmic comparison between a chip's RTL-description and synthesized gate-netlist, to ensure functional equivalence at the *logical* level.
- Static timing analysis: Analysis of the timing of a circuit in an input-independent manner, hence finding a worst case over all possible inputs.
- Physical verification, PV: checking if a design is physically manufacturable, and that the resulting chips will not have any function-preventing physical defects, and will meet original specifications.

## Manufacturing preparation

- Mask data preparation, MDP: generation of actual lithography photomask used to physically manufacture the chip.
  - o Resolution enhancement techniques, RET – methods of increasing of quality of final photomask.
  - o Optical proximity correction, OPC – up-front compensation for diffraction and interference effects occurring later when chip is manufactured using this mask.
  - o Mask generation – generation of flat mask image from hierarchical design.
  - o Automatic test pattern generation, ATPG – generates pattern-data to systematically exercise as many logic-gates, and other components, as possible.
  - o Built-in self-test, or BIST – installs self-contained test-controllers to automatically test a logic (or memory) structure in the design

## Companies

For more details on this topic, see List of EDA companies.

## Top Companies

- $3.73 billion - Synopsys
- $2.06 billion - Cadence
- $1.18 billion - Mentor Graphics
- $233 million - Magma Design Automation
- $157 million - Zuken Inc.

Note: Market caps current as of October, 2010. EEsof should likely be on this list, but does not have a market cap as it is the EDA division of Agilent.

## Acquisitions

Many of the EDA companies acquire small companies with software or other technology that can be adapted to their core business. Most of the market leaders are rather incestuous amalgamations of many smaller companies. This trend is helped by the tendency of software companies to design tools as accessories that fit naturally into a larger vendor's suite of programmes ( on digital circuitry, many new tools incorporate analog design, and mixed systems. This is happening because there is now a trend to place entire electronic systems on a single chip.

# 3

# Computer Software Generations

## First Generation

During the 1950's the first computers were programmed by changing the wires and set tens of dials and switches. One for every bit sometimes these settings could be stored on paper tapes that looked like a ticker tape from the telegraph - a punch tape - or punched card. With these tapes and or cards the machine was told what, how and when to do something.

To have a flawless programme a programmer needed to have a very detailed knowledge of the computer where he or she worked on. A small mistake caused the computer to crash.

## Second Generation

Because the first generation "languages" were regarded as very user unfriendly people set out to look for something else, faster and easier to understand. The result was the birth of the second generation languages (2GL) at the mid of the 1950's. These

generation made use of symbols and are called assemblers. An assembler is a programme that translates symbolic instructions to processor instructions. (See above for an example) But deep in the 1950's there was still not a single processor but a whole assembly rack with umpteen tubes and or relays.

A programmer did no longer have to work with one's and zero's when using an assembly language. He or she can use symbols instead. These symbols are called mnemonics because of the mnemonic character these symbols had (STO = store). Each mnemonic stands for one single machine instruction.

But an assembler still works on a very low level with the machine. For each processor a different assembler was written.

## Third Generation

At the end of the 1950's the 'natural language' interpreters and compilers were made. But it took some time before the new languages were accepted by enterprises.

About the oldest 3GL is FORTRAN (Formula Translation) which was developed around 1953 by IBM. This is a language primarily intended for technical and scientific purposes. Standardization of FORTRAN started 10 years later, and a recommendation was finally published by the International Standardization Organization (ISO) in 1968.

Fortran 77 is now standardized

COBOL (= Common Business Oriented Language) was developed around 1959 and is like its name says primarily used, up till now, in the business world.

With a 3GL there was no longer a need to work in symbolics. Instead a programmer could use a programming language what resembled more to natural language. Be it a stripped version with some two or three hundred 'reserved' words. This is the period

(1970's) were the now well known so called 'high level' languages like BASIC,PASCAL, ALGOL, FORTRAN, PL/I, and C have been born.

## Fourth Generation

A 4GL is an aid witch the end user or programmer can use to build an application without using a third generation programming language. Therefore knowledge of a programming language is strictly spoken not needed.

The primary feature is that you do not indicate HOW a computer must perform a task but WHAT it must do. In other words the assignments can be given on a higher functional level.

A few instructions in a 4GL will do the same as hundreds of instructions in a lower generation language like COBOLor BASIC. Applications of 4GL's are concentrating on the daily performed tasks such like screen forms, requests for data, change data, and making hard copies. In most of these cases one deals with Data Base Management Systems (DBMS).

The main advantage of this kind of languages is that a trained user can create an application in a much shorter time for development and debugging than would be possible with older generation programming language. Also a customer can be involved earlier in the project and can actively take part in the development of a system, by means of simulation runs, long before the application is actually finished.

Today the disadvantage of a 4GL lays more in the technological capacities of hardware. Since programs written in a 4GL are quite a bit larger they are needing more disk space and demanding a larger part of the computer's memory capacity than 3GL's. But hardware of technologically high standard is made more available every day, not necessarily cheaper, so in the long run restrictions

will disappear. Considering the arguments one can say that the costs saved in development could now be invested in hardware of higher performance and stimulate the development of the 4GL's.

In the 1990's the expectations of a 4GL language are too high. And the use of it only will be picked up by Oracle and SUN that have enough power to pull it through. However in most cases the 4GL environment is often misused as a documentation tool and a version control implement. In very few cases the use of such programs are increasing productivity. In most cases they only are used to lay the basis for information systems. And programmers use all kinds of libraries and toolkits to give the product its final form.

## Fifth Generation

This term is often misused by software companies that build programming environments. Till today one can only see vague contours. When one sees a nice graphical interface it is tempting to call that a fifth generation. But alas changing the makeup does not make a butterfly into an eagle.

Yes some impressions are communicated from professional circles that are making these environments and sound promising.

But again the Fifth generation only exist in the brains of those trying to design this generation, YET! Many attempts are made but are stranding on the limitations of hardware, and strangely enough on the views and insight of the use of natural language. We need a different speak for this!

But it is a direction that will be taken by these languages: no longer prohibiting for the use of natural language and intuitive approach towards the programme (language) to be developed

The basis of this is laid in the 1990's by using sound, moving images and agents - a kind of advanced macro's of the 1980's.

And it is only natural that neural networks will play an important role.

Software for the end user will be (may be) based on principles of knowbot-agents. An autonomous self changing piece of software that creates new agents based on the interaction of the end user and interface. A living piece of software, as you may say. And were human alike DNA/RNA (intelligent?) algorithms can play a big role.

# Computer Languages

## Introduction

The term computer language includes a wide variety of languages used to communicate with computers. It is broader than the more commonly-used term programming language. Programming languages are a subset of computer languages.

For example, HTML is a markup language and a computer language, but it is not traditionally considered a programming language. Machine code is a computer language. It can technically be used for programming, and has been (e.g. the original bootstrapped for Altair BASIC), though most would not consider it a programming language.

Computer languages can be divided into two groups: high-level languages and low-level languages. High-level languages are designed to be easier to use, more abstract, and more portable than low-level languages.

Syntactically correct programs in some languages are then compiled to low-level language and executed by the computer. Most modern software is written in a high-level language, compiled into object code, and then translated into machine instructions.

Computer languages could also be grouped based on other criteria. Another distinction could be made between human-readable and non-human-readable languages. Human-readable languages are designed to be used directly by humans to communicate with the computer. Non-human-readable languages, though they can often be partially understandable, are designed to be more compact and easily processed, sacrificing readability to meet these ends.

## Types of Computer Languages

Language can be categories broadly into three categories.

## Machine Language

The most elementary and first type of computer, which was invented, was machine language. Machine language was machine dependent. A programme written in machine language cannot be run on another type of computer without significant alterations. Machine language is some times also referred as the binary language i-e, the language of 0 and 1 where 0 stands for the absence of electric pulse and i stands for the presence of electric pulse. Very few computer programs are actually written in machine language.

## Assembly Language

As computer became more popular, it became quite apparent that machine language programming was simply too slow slow tedious for most programmers. Assembly languages are also called as low level language instead of using the string of members programmers began using English like abbreviation to represent the elementary operation. The language provided an opportunity to the programmers to use English like words that were called MNEMONICS.

## High Level Language

The assembly languages started using English like words,m but still it was difficult to learn these languages. High level languages are the computer language in which it is much easier to write a programme than the low level language. A programme written in high level language is just like gibing instruction to person in daily life. It was in 1957 that a high level language called FORTRAN was developed by IBM which was specially developed for scientist and engineers other high level languages are COBOL which is widely used for business data processing task.BASIC language which is developed for the beginners in general purpose programming language. you Can use C language for almost any programming task. PASCAL are other high level languages which has gained widespread acceptance.

# Software Crisis

Indeed, the problem of trying to write an encyclopedia is very much like writing software. Both running code and a hypertext/ encyclopedia are wonderful turn-ons for the brain, and you want more of it the more you see, like a drug. As a user, you want it to do everything, as a customer you don't really want to pay for it, and as a producer you realise how unrealistic the customers are. Requirements will conflict in functionality vs affordability, and in completeness vs timeliness.

## Different Types of Crisis

## Chronic Software Crisis

By today's definition, a "large" software system is a system that contains more than 50,000 lines of high-level language code. It's those large systems that bring the software crisis to light. If

you're familiar with large software development projects, you know that the work is done in teams consisting of project managers, requirements analysts, software engineers, documentation experts, and programmers. With so many professionals collaborating in an organized manner on a project, what's the problem? Why is it that the team produces fewer than 10 lines of code per day over the average lifetime of the project? And why are sixty errors found per every thousand lines of code? Why is one of every three large projects scrapped before ever being completed? And why is only 1 in 8 finished software projects considered "successful?"

- The cost of owning and maintaining software in the 1980's was twice as expensive as developing the software.
- During the 1990's, the cost of ownership and maintenance increased by 30% over the 1980's.
- In 1995, statistics showed that half of surveyed development projects were operational, but were not considered successful.
- The average software project overshoots its schedule by half.
- Three quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer's requirements.

Software projects are notoriously behind schedule and over budget. Over the last twenty years many different paradigms have been created in attempt to make software development more predictable and controllable.

While there is no single solution to the crisis, much has been learned that can directly benefit today's software projects.

*It appears that the Software Crisis can be boiled down to two basic sources*:

1. Software development is seen as a craft, rather than an engineering discipline.
2. The approach to education taken by most higher education institutions encourages that "craft" mentality.

## Software Development

Software development today is more of a craft than a science. Developers are certainly talented and skilled, but work like craftsmen, relying on their talents and skills and using techniques that cannot be measured or reproduced. On the other hand, software engineers place emphasis on reproducible, quantifiable techniques–the marks of science. The software industry is still many years away from becoming a mature engineering discipline. Formal software engineering processes exist, but their use is not widespread. A crisis similar to the software crisis is not seen in the hardware industry, where well documented, formal processes are tried and true, and ad hoc hardware development is unheard of. To make matters worse, software technology is constrained by hardware technology. Since hardware develops at a much faster pace than software, software developers are constantly trying to catch up and take advantage of hardware improvements.

Management often encourages ad hoc software development in an attempt to get products out on time for the new hardware architectures. Design, documentation, and evaluation are of secondary importance and are omitted or completed after the fact. However, as the statistics show, the ad hoc approach just doesn't work. Software developers have classically accepted a certain number of errors in their work as inevitable and part of the job. That mindset becomes increasingly unacceptable as software becomes embedded in more and more consumer electronics. Sixty

errors per thousand lines of code is unacceptable when the code is embedded in a toaster, automobile, ATM machine or razor (let your imagination run free for a moment).

## Computer Science and Product Orientation

Software developers pick up the ad hoc approach to software development early in their computer science education, where they are taught a "product orientation" approach to software development. In the many undergraduate computer science courses I took, the existence of software engineering processes was never even mentioned.

Computer science education does not provide students with the necessary skills to become effective software engineers. They are taught in a way that encourages them to be concerned only with the final outcome of their assignments–whether or not the programme runs, or whether or not it runs efficiently, or whether or not they used the best possible algorithm. Those concerns in themselves are not bad. But on the other hand, they should not be the focus of a project. The focus should be on the complete process from beginning to end and beyond. Product orientation also leads to problems when the student enters the work force–not having seen how processes affect the final outcome, individual programmers tend to think their work from day to day is too "small" to warrant the application of formal methods.

## Fully Supported Software

As we have seen, most software projects do not follow a formal process. The result is a product that is poorly designed and documented. Maintenance becomes problematic because without a design and documentation, it's difficult or impossible to predict what sort of effect a simple change might have on other parts of

the system. Fortunately there is an awareness of the software crisis, and it has inspired a worldwide movement towards process improvement. Software industry leaders are beginning to see that following a formal software process consistently leads to better quality products, more efficient teams and individuals, reduced costs, and better morale.

Ratings range from Maturity Level 1, which is characterized by ad hoc development and lack of a formal software development process, up to Maturity Level 5, at which an organization not only has a formal process, but also continually refines and improves it. Each maturity level is further broken down into key process areas that indicate the areas an organization should focus on to improve its software process (*e.g.* requirement analysis, defect prevention, or change control).

Level 5 is very difficult to attain. In early 1995, only two projects, one at Motorola and another at Loral (the on-board space shuttle software project), had earned Maturity Level 5. Another study showed that only 2% of reviewed projects rated in the top two Maturity Levels, in spite of many of those projects placing an extreme emphasis on software process improvement.

Customers contracting large projects will naturally seek organizations with high CMM ratings, and that has prompted increasingly more organizations to investigate software process improvement. Mature software is also reusable software. Artisans are not concerned with producing standardized products, and that is a reason why there is so little interchangeability in software components.

Ideally, software would be standardized to such an extent that it could be marketed as a “part”, with its own part number and revision, just as though it were a hardware part.

The software component interface would be compatible with any other software system. Though it would seem that nothing less than a software development revolution could make that happen, the National Institute of Standards and Technology (NIST) founded the Advanced Technology Programme (ATP), one purpose of which was to encourage the development of standardized software components.

## Programming Language Generations

Programming languages have been classified into several programming language generations. Historically, this classification was used to indicate increasing power of programming styles. Later writers have somewhat redefined the meanings as distinctions previously seen as important became less significant to current practice.

### Historical View of First Three Generations

The terms "first-generation" and "second-generation" programming language were not used prior to the coining of the term "third-generation." In fact, none of these three terms are mentioned in an early compendium of programming language.

The introduction of a third generation of computer technology coincided with the creation of a new generation of programming languages. The marketing for this generational shift in machines did correlate with several important changes in what were called high level programming languages, discussed below, giving technical content to the second/third-generation distinction among high level programming languages as well, and reflexively renaming assembler languages as first-generation.

## First Generation

As Grace Hopper said about coding in machine language: "We were not programmers in those days. The word had not yet come over from England. We were coders. The task of encoding an algorithm wasn't thought of as writing in a language any more than was the task of wiring a plug-board. But even by the early 1950s, the assembly languages were seen as a distinct "epoch". The distinguishing properties of these first generation programming languages are that:

- The code can be read and written by a programmer. To run on a computer it must be converted into a machine readable form, a process called assembly.
- The language is specific to a particular target machine or family of machines, directly reflecting their characteristics like instruction sets, registers, storage access models, etc., requiring and enabling the programmer to manage their use.
- Some assembler languages provide a macro-facility enabling the development of complex patterns of machine instructions, but these are not considered to change the basic nature of the language.

First-generation languages are sometimes used in kernels and device drivers, but more often find use in extremely intensive processing such as games, video editing, and graphic manipulation/rendering.

## Second Generation

Second-generation programming languages, originally just called high level programming languages, were created to simplify the burden of programming by making its expression

more like the normal mode of expression for thoughts used by the programmer. They were introduced in the late 1950s, with FORTRAN reflecting the needs of scientific programmers, ALGOL reflecting an attempt to produce a European/ American standard view.

The most important issue faced by the developers of second-level languages was convincing customers that the code produced by the compilers performed well-enough to justify abandonment of assembler programming.In view of the widespread skepticism about the possibility of producing efficient programmes with an automatic programming system and the fact that inefficiencies could no longer be hidden, we were convinced that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programmes almost as efficient as hand coded ones and do so on virtually every job.The FORTRAN compiler was seen as a tour-de-force in the production of high-quality code, even including "... a Monte Carlo simulation of its execution ... so as to minimize the transfers of items between the store and the index registers Second-generation programming languages evolved through the decade.

FORTRAN lost some of its machine-dependent features, like access to the lights and switches on the operator console. Most second-generation languages employed a static storage model in which storage for data was allocated only once, when a programme is loaded, making recursion difficult, but Algol evolved to provide block-structured naming constructs and began to expand the set of features made

available to programmers, like concurrency management. In this way (Algol 68) began the movement into a new generation of programming languages.

## Third Generation

The introduction of a third generation of computer technology coincided with the creation of a new generation of programming languages. The third-generation languages emphasized:

- expression of an algorithm in a way that was independent of the characteristics of the machine on which the algorithm would run.
- the rise of strong typing – by which typed languages deprecated or severely controlled access to the underlying storage representation of data. Complete prohibition of such access has never been a feature of major-use programming languages, which generally simply provide barriers to accidental access, e.g. coding them as "native" methods.
- block structure and automated management of storage with a stack – introduced in the Algol family of languages and adopted rapidly by most other major modular languages
- broad-spectrum applicability and greatly extended functionality – which was intended to service the needs of not only the previously separated commercial and scientific domains. The extended functionality often included concurrency features, creation and reference to non-stack data,

## Re-characterization of First Three Generations

Since the 1990s, some authors have recharacterized the development of programming languages in a way that removed the (no longer topical) distinctions between early high-level languages like Fortran or Cobol and later ones, like

## First Generation

In this categorization, a *first-generation programming language* is a machine-level programming language.

Originally, no translator was used to compile or assemble the first-generation language. The first-generation programming instructions were entered through the front panel switches of the computer system.

The main benefit of programming in a first-generation programming language is that the code a user writes can run very fast and efficiently, since it is directly executed by the CPU. However, machine language is a lot more difficult to learn than higher generational programming languages, and it is far more difficult to edit if errors occur. In addition, if instructions need to be added into memory at some location, then all the instructions after the insertion point need to be moved down to make room in memory to accommodate the new instructions. Doing so on a front panel with switches can be very difficult.

## Second Generation

*Second-generation programming language* is a generational way to categorize assembly languages. The term was coined to provide a distinction from higher level third-generation

programming languages (3GL) such as COBOL and earlier machine code languages. Second-generation programming languages have the following properties:

- The code can be read and written by a programmer. To run on a computer it must be converted into a machine readable form, a process called assembly.
- The language is specific to a particular processor family and environment.

Second-generation languages are sometimes used in kernels and device drivers (though C is generally employed for this in modern kernels), but more often find use in extremely intensive processing such as games, video editing, graphic manipulation/rendering. One method for creating such code is by allowing a compiler to generate a machine-optimized assembly language version of a particular function. This code is then hand-tuned, gaining both the brute-force insight of the machine optimizing algorithm and the intuitive abilities of the human optimizer.

## Third Generation

A *third-generation programming language* (*3GL*) is a refinement of a second-generation programming language. Whereas a second generation language is more aimed to fix logical structure to the language, a third generation language aims to refine the usability of the language in such a way to make it more user friendly. This could mean restructuring categories of possible functions to make it more efficient, condensing the overall bulk of code via classes (eg. Visual Basic). A third generation language improves over a second generation language by having more refinement on the

usability of the language itself from the perspective of the user. First introduced in the late 1950s, FORTRAN, ALGOL and COBOL are early examples of this sort of language.

Most "modern" languages (BASIC, C, C++, C#, Pascal, and Java) are also third-generation languages.

Most 3GLs support structured programming.

### Later Generations

"Generational" classification of these languages was abandoned after the third-generation languages, with the natural successors to the third-generation languages being termed object-oriented. C gave rise to C++ and later to Java and C#, Lisp to CLOS, ADA to ADA95, and even COBOL to COBOL2002, and new languages have emerged in that "generation" as well.

But significantly different languages and systems were already being called fourth and fifth generation programming languages by language communities with special interests. The manner in which these generations have been put forward tends to differ in character from those of earlier generations, and they represent software points-of-view leading away from the mainstream.

## Software Developer

A software developer is a person concerned with facets of the software development process. They can be involved in aspects wider than design and coding, a somewhat broader scope of computer programming or a specialty of project managing including some aspects of software product management. This person may contribute to the overview

of the project on the application level rather than component level or individual programming tasks. Software developers are often still guided by lead programmers but also encompasses the class of freelance software developers. A person who develops stand-alone software (that is more than just a simple program) and got involved with all phases of the development (design and code) is a software developer.

Many legendary software people including Peter Norton (developer of *Norton Utilities*), Richard Garriott (Ultima-series creator), Philippe Kahn (Borland key founder), started as entrepreneurial individual or small-team software developers before they became rich and famous. Other names which are often used in the same close context are programmer, software analyst and software engineer. According to developer Eric Sink, the differences between system design, software development and programming are more apparent. Already in the current market place there can be found a segregation between programmers and developers, being that one who actually implements is not the same as the one who designs the class structure or hierarchy. Even more so that developers become systems architects, those who design the multi-leveled architecture or component interactions of a large software system. Aspects of developer's job may include:

- Software design
- Actual core implementation (programming which is often the most important portion of software development)
- Other required implementations (e.g. installation, configuration, customization, integration, data migration)

- Participation in software product definition, including Business case or Gap analysis
- Specification
- Requirements analysis
- Development and refinement of throw-away simulations or prototypes to confirm requirements
- Feasibility and Cost-benefit analysis, including the choice of application architecture and framework, leading to the budget and schedule for the project
- Authoring of documentation needed by users and implementation partners etc.
- Testing, including defining/supporting acceptance testing and gathering feedback from pre-release testers
- Participation in software release and post-release activities, including support for product launch evangelism (e.g. developing demonstrations and/or samples) and competitive analysis for subsequent product build/release cycles
- Maintenance

In a large company, there may be employees whose sole responsibility may consist of only one of the phases above. In smaller development environments, a few, or even a single individual might handle the complete process.

## Separation of Concerns

In more mature engineering disciplines such as mechanical, civil and electrical engineering, the designers are separate from the implementers. That is, the engineers who generate design documents are not the same individuals

who actually build things (such as mechanical parts, circuits, or roads, for instance). In software engineering, it is more common to have the architecture, design, implementation, and test functions performed by a single individual. In particular, the design and implementation of source code is commonly integrated. This resembles the early phases of industrialization in which individuals would both design and built things. More mature organizations have separate test groups, but the architecture, design, implementation, and unit test functions are often performed by the same highly trained individuals.

## Software Engineering

Software engineering (SE) is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build. It is a "systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software." The term *software engineering* first appeared in the 1968 NATO Software Engineering Conference, and was meant to provoke thought regarding the perceived "software crisis" at the time. The IEEE Computer Society's *Software Engineering Body of Knowledge* defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It is the application of Engineering to software because it integrates significant mathematics, computer science and practices whose origins are in Engineering.

*Software development*, a much used and more generic term, does not necessarily subsume the engineering paradigm. Although it is questionable what impact it has had on actual software development over the last more than 40 years, the field's future looks bright according to Money Magazine and Salary.com, which rated "software engineer" as the best job in the United States in 2006.

## History

When the first modern digital computers appeared in the early 1940s, the instructions to make them operate were wired into the machine. Practitioners quickly realized that this design was not flexible and came up with the "stored programme architecture" or von Neumann architecture. Thus the first division between "hardware" and "software" began with abstraction being used to deal with the complexity of computing.

Programming languages started to appear in the 1950s and this was also another major step in abstraction. Major languages such as Fortran, ALGOL, and COBOL were released in the late 1950s to deal with scientific, algorithmic, and business problems respectively. E.W. Dijkstra wrote his seminal paper, "Go To Statement Considered Harmful", in 1968 and David Parnas introduced the key concept of modularity and information hiding in 1972 to help programmers deal with the ever increasing complexity of software systems. A software system for managing the hardware called an operating system was also introduced, most notably by Unix in 1969. In 1967, the Simula language introduced the object-oriented programming paradigm.

These advances in software were met with more advances in computer hardware. In the mid 1970s, the microcomputer was introduced, making it economical for hobbyists to obtain a computer and write software for it. This in turn led to the now famous Personal Computer (PC) and Microsoft Windows. The Software Development Life Cycle or SDLC was also starting to appear as a consensus for centralized construction of software in the mid 1980s. The late 1970s and early 1980s saw the introduction of several new Simula-inspired object-oriented programming languages, including Smalltalk, Objective-C, and C++.

Open-source software started to appear in the early 90s in the form of Linux and other software introducing the "bazaar" or decentralized style of constructing software. Then the World Wide Web and the popularization of the Internet hit in the mid 90s, changing the engineering of software once again. Distributed systems gained sway as a way to design systems, and the Java programming language was introduced with its own virtual machine as another step in abstraction. Programmers collaborated and wrote the Agile Manifesto, which favored more lightweight processes to create cheaper and more timely software.

The current definition of *software engineering* is still being debated by practitioners today as they struggle to come up with ways to produce software that is "cheaper, better, faster". Cost reduction has been a primary focus of the IT industry since the 1990s. Total cost of ownership represents the costs of more than just acquisition. It includes things like productivity impediments, upkeep efforts, and resources needed to support infrastructure.

## Profession

Legal requirements for the licensing or certification of professional software engineers vary around the world. In the UK, the British Computer Society licenses software engineers and members of the society can also become Chartered Engineers (CEng), while in some areas of Canada, such as Alberta, Ontario, and Quebec, software engineers can hold the Professional Engineer (P.Eng)designation and/ or the Information Systems Professional (I.S.P.) designation; however, there is no legal requirement to have these qualifications.

The IEEE Computer Society and the ACM, the two main professional organizations of software engineering, publish guides to the profession of software engineering.

The IEEE's *Guide to the Software Engineering Body of Knowledge - 2004 Version*, or SWEBOK, defines the field and describes the knowledge the IEEE expects a practicing software engineer to have. The IEEE also promulgates a "Software Engineering Code of Ethics".

## Employment

In 2004, the U. S. Bureau of Labor Statistics counted 760,840 software engineers holding jobs in the U.S.; in the same time period there were some 1.4 million practitioners employed in the U.S. in all other engineering disciplines combined.

Due to its relative newness as a field of study, formal education in software engineering is often taught as part of a computer science curriculum, and many software engineers hold computer science degrees.

Many software engineers work as employees or contractors. Software engineers work with businesses, government agencies (civilian or military), and non-profit organizations. Some software engineers work for themselves as freelancers. Some organizations have specialists to perform each of the tasks in the software development process. Other organizations require software engineers to do many or all of them. In large projects, people may specialize in only one role. In small projects, people may fill several or all roles at the same time. Specializations include: in industry (analysts, architects, developers, testers, technical support, middleware analysts, managers) and in academia (educators, researchers).

Most software engineers and programmers work 40 hours a week, but about 15 percent of software engineers and 11 percent of programmers worked more than 50 hours a week in 2008. Injuries in these occupations are rare. However, like other workers who spend long periods in front of a computer terminal typing at a keyboard, engineers and programmers are susceptible to eyestrain, back discomfort, and hand and wrist problems such as carpal tunnel syndrome.

## Certification

The Software Engineering Institute offers certifications on specific topics like Security, Process improvement and Software architecture. Apple, IBM, Microsoft and other companies also sponsor their own certification examinations. Many IT certification programmes are oriented toward specific technologies, and managed by the vendors of these

technologies. These certification programmes are tailored to the institutions that would employ people who use these technologies.

Broader certification of general software engineering skills is available through various professional societies. As of 2006, the IEEE had certified over 575 software professionals as a Certified Software Development Professional (CSDP). In 2008 they added an entry-level certification known as the Certified Software Development Associate (CSDA). In the U.K. the British Computer Society has developed a legally recognized professional certification called *Chartered IT Professional (CITP)*, available to fully qualified Members (*MBCS*). In Canada the Canadian Information Processing Society has developed a legally recognized professional certification called *Information Systems Professional (ISP)*. The ACM had a professional certification programme in the early 1980s, which was discontinued due to lack of interest. The ACM examined the possibility of professional certification of software engineers in the late 1990s, but eventually decided that such certification was inappropriate for the professional industrial practice of software engineering.

## Impact of Globalization

The initial impact of outsourcing, and the relatively lower cost of international human resources in developing third world countries led to the dot com bubble burst of the 1990s. This had a negative impact on many aspects of the software engineering profession. For example, some students in the developed world avoid education related to software engineering because of the fear of offshore outsourcing

(importing software products or services from other countries) and of being displaced by foreign visa workers. Although statistics do not currently show a threat to software engineering itself; a related career, computer programming does appear to have been affected. Nevertheless, the ability to smartly leverage offshore and near-shore resources via the [follow-the-sun] workflow has improved the overall operational capability of many organizations. When North Americans are leaving work, Asians are just arriving to work. When Asians are leaving work, Europeans are arriving to work. This provides a continuous ability to have human oversight on business-critical processes 24 hours per day, without paying overtime compensation or disrupting key human resource sleep patterns.

## Education

A knowledge of programming is a pre-requisite to becoming a software engineer. In 2004 the IEEE Computer Society produced the SWEBOK, which has been published as ISO/IEC Technical Report 19759:2004, describing the body of knowledge that they believe should be mastered by a graduate software engineer with four years of experience. Many software engineers enter the profession by obtaining a university degree or training at a vocational school. One standard international curriculum for undergraduate software engineering degrees was defined by the CCSE, and updated in 2004. A number of universities have Software Engineering degree programmes; as of 2010, there were 244 Campus programmes, 70 Online programmes, 230 Masters-level programmes, 41 Doctorate-level programmes, and 69 Certificate-level programmes in the United States.

In addition to university education, many companies sponsor internships for students wishing to pursue careers in information technology. These internships can introduce the student to interesting real-world tasks that typical software engineers encounter every day. Similar experience can be gained through military service in software engineering.

## Comparison with Other Disciplines

Major differences between software engineering and other engineering disciplines, according to some researchers, result from the costs of fabrication.

## Sub-disciplines

Software engineering can be divided into ten subdisciplines. They are:

- Software requirements: The elicitation, analysis, specification, and validation of requirements for software.
- Software architecture: The elicitation, analysis, specification, definition and design, and validation and control of software architecture requirements.
- Software design: The design of software is usually done with Computer-Aided Software Engineering (CASE) tools and use standards for the format, such as the Unified Modeling Language (UML).
- Software development: The construction of software through the use of programming languages.
- Software testing

- Software maintenance: Software systems often have problems and need enhancements for a long time after they are first completed. This subfield deals with those problems.
- Software configuration management: Since software systems are very complex, their configuration (such as versioning and source control) have to be managed in a standardized and structured method.
- Software engineering management: The management of software systems borrows heavily from project management, but there are nuances encountered in software not seen in other management disciplines.
- Software development process: The process of building software is hotly debated among practitioners; some of the better-known processes are the Waterfall Model, the Spiral Model, Iterative and Incremental Development, and Agile Development.
- Software engineering tools, see Computer Aided Software Engineering
- Software quality

## Related Disciplines

Software engineering is a direct subfield of computer science and has some relations with management science. It is also considered a part of overall systems engineering.

## Systems Engineering

Systems engineers deal primarily with the overall system design, specifically dealing more with physical aspects which include hardware design. Those who choose to specialize

in computer hardware engineering may have some training in software engineering.

### Computer Software Engineers

Computer Software Engineers are usually systems level (software engineering, information systems) computer science or software level computer engineering graduates. This term also includes general computer science graduates with a few years of practical on the job experience involving software engineering.

## Programming Language Specification

In computing, a programming language specification is an artifact that defines a programming language so that users and implementors can agree on what programmes in that language mean. A programming language specification can take several forms, including the following:

- An explicit definition of the syntax and semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., the approach taken for the C language), or a formal semantics (e.g., the Standard MLand Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran). The syntax and semantics of the language has to be inferred from this description, which may be written in natural or a formal language.
- A *model* implementation, sometimes written in the language being specified (e.g., the Prolog). The syntax

and semantics of the language are explicit in the behavior of the model implementation.

## Syntax

Syntax in a programming language is usually described using a combination of

- regular expressions to describe lexemes, and
- context-free grammars to describe how lexemes may be combined to form a valid programme.

## Semantics

Formulating a rigorous semantics of a large, complex, practical programming language is a daunting task even for experienced specialists, and the resulting specification can be difficult for anyone but experts to understand.

The following are some of the ways in which programming language semantics can be described; all languages use at least one of these description methods, and some languages combine more than one:

- Natural language: Description by human natural language.
- Formal semantics: Description by mathematics.
- Reference implementations: Description by computer programme
- Test suites: Description by examples of programmes and their expected behaviors. While few language specifications start off in this form, the evolution of some language specifications has been influenced by the semantics of a test suite (eg, in the past the

specification of Ada has been modified to match the behavior of the Ada Conformity Assessment Test Suite).

## Natural Language

Most widely-used languages are specified using natural language descriptions of their semantics. This description usually takes the form of a *reference manual* for the language. These manuals can run to hundreds of pages. For example, the print version of *The Java Language Specification, 3rd Ed.* is 596 pages long. The imprecision of natural language as a vehicle for describing programming language semantics can lead to problems with interpreting the specification. For example, the semantics of Java threads were specified in English, and it was later discovered that the specification did not provide adequate guidance for implementors.

## Formal Semantics

Formal semantics are grounded in mathematics. As a result, they can be more precise and less ambiguous than semantics given in natural language. However, supplemental natural language descriptions of the semantics are often included to aid understanding of the formal definitions. For example, The ISO Standard for Modula-2 contains both a formal and a natural language definition on opposing pages. Programming languages whose semantics are described formally can reap many benefits. For example:

- Formal semantics enable mathematical proofs of programme correctness;
- Formal semantics facilitate the design of type systems, and proofs about the soundness of those type systems;

- Formal semantics can establish unambiguous and uniform standards for implementations of a language.

Automatic tool support can help to realize many of these benefits. For example, an automated theorem prover or theorem checker can increase a programmer's (or language designer's) confidence in the correctness of proofs about programmes (or the language itself).

The power and scalability of these tools varies widely: full formal verification is computationally intensive, rarely scales beyond programmes containing a few hundred lines and may require considerable manual assistance from a programmer; more lightweight tools such as model checkers require fewer resources and have been used on programmes containing tens of thousands of lines; many compilers apply static type checks to any programme they compile.

## Reference Implementation

A reference implementation is a single implementation of a programming language that is designated as authoritative. The behavior of this implementation is held to define the proper behavior of a programme written in the language.

This approach has several attractive properties. First, it is precise, and requires no human interpretation: disputes as to the meaning of a programme can be settled simply by executing the programme on the reference implementation (provided that the implementation behaves deterministically for that program). On the other hand, defining language semantics through a reference implementation also has several potential drawbacks.

Chief among them is that it conflates limitations of the reference implementation with properties of the language. For example, if the reference implementation has a bug, then that bug must be considered to be an authoritative behavior. Another drawback is that programmes written in this language may rely on quirks in the reference implementation, hindering portability across different implementations. Nevertheless, several languages have successfully used the reference implementation approach. For example, the Perl interpreter is considered to define the authoritative behavior of Perl programmes. In the case of Perl, the Open Source model of software distribution has contributed to the fact that nobody has ever produced another implementation of the language, so the issues involved in using a reference implementation to define the language semantics are moot.

## Test Suite

Defining the semantics of a programming language in terms of a test suite involves writing a number of example programmes in the language, and then describing how those programmes ought to behave — perhaps by writing down their correct outputs. The programmes, plus their outputs, are called the "test suite" of the language. Any correct language implementation must then produce exactly the correct outputs on the test suite programmes.

The chief advantage of this approach to semantic description is that it is easy to determine whether a language implementation passes a test suite. The user can simply execute all the programmes in the test suite, and compare

the outputs to the desired outputs. However, when used by itself, the test suite approach has major drawbacks as well. For example, users want to run their own programmes, which are not part of the test suite; indeed, a language implementation that could *only* run the programmes in its test suite would be largely useless. But a test suite does not, by itself, describe how the language implementation should behave on any programme not in the test suite; determining that behavior requires some extrapolation on the implementor's part, and different implementors may disagree. In addition, it is difficult to use a test suite to test behavior that is intended or allowed to be nondeterministic. Therefore, in common practice, test suites are used only in combination with one of the other language specification techniques, such as a natural language description or a reference implementation.

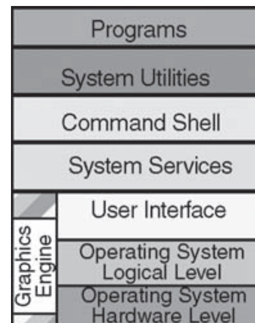
# 4

# Software Components

A computer system consists of three major components: hardware, software, and humans (users, programmers, administrators, operators, etc.). Software can be further divided into seven layers. Firmware can be categorized as part of hardware, part of software, or both.

The seven layers of software are (top to bottom): Programs; System Utilities; Command Shell; System Services; User Interface; Logical Level; and Hardware Level. A Graphics Engine stradles the bottom three layers.

Strictly speaking, only the bottom two levels are the operating system, although even technical persons will often refer to any level other than programs as part of the operating system (and Microsoft tried to convince the Justice Department that their web browser application is actually a part of their operating system). Because this technical analysis concentrates on servers, Internet Facilities are specifically separated out from the layers.




## Examples

*The following are examples of each category*:

- *Programs*: Examples of Programs include your word processor, spreadsheet, graphics programs, music software, games, etc.
- *System Utilities*: Examples of System Utilities include file copy, hard drive repair, and similar items. On the Macintosh, all the Desk Accessories (calculator, key caps, etc.) and all of the Control Panels are examples of System Utilities.
- *Command Shell*: The Command Shell on the Macintosh is the Finder and was the first commercially available graphic command shell. On Windows, the Command Shell is a poorly integrated comination of the File Manager and the Programme Manager. The command line (C:\ prompt) of MS-DOS or Bourne Shell of UNIX are examples of the older style text-based command shells.
- *System Services*: Examples of System Services are built-in data base query languages on mainframes or the QuickTime media layer of the Macintosh.
- *User Interface*: Until the Macintosh introduced Alan Kay's (inventer of the personal computer, graphic user interfaces, object oriented programming, and software agents) ground

breaking ideas on human-computer interfaces, operating systems didn't include support for user interfaces (other than simple text-based shells). The Macintosh user interface is called the Macintosh ToolBox and provides the windows, menus, alert boxes, dialog boxes, scroll bars, buttons, controls, and other user interface elements shared by almost all programs.

- *Logical Level of Operating System*: The Logical Level of the operating system provides high level functions, such as file management, internet and networking facilities, etc.
- *Hardware Level of Operating System*: The Hardware Level of the operating system controls the use of physical system resources, such as the memory manager, process manager, disk drivers, etc.
- *Graphics Engine*: The Graphics Engine includes elements at all three of the lowest levels, from physically displaying things on the monitor to providing high level graphics routines such as fonts and animated sprites.

Human users normally interact with the operating system indirectly, through various programs (application and system) and command shells (text, graphic, etc.), The operating system provides programs with services thrrough system programs and Application Programme Interfaces (APIs).

# Software life cycle models

## Waterfall model

The least flexible of the life cycle models. Still it is well suited to projects which have a well defined architecture and established user interface and performance requirements.

The waterfall model *does* work for certain problem domains, notably those where the requirements are well understood in advance and unlikely to change significantly over the course of development.

Software products are oriented towards customers like any other engineering products. It is either driver by market or it drives the market. Customer Satisfaction was the main aim in the 1980's. Customer Delight is today's logo and Customer Ecstasy is the new buzzword of the new millennium. Products which are not customer oriented have no place in the market although they are designed using the best technology. The front end of the product is as crucial as the internal technology of the product.

A market study is necessary to identify a potential customer's need. This process is also called as market research. The already existing need and the possible future needs that are combined together for study.

A lot of assumptions are made during market study. Assumptions are the very important factors in the development or start of a product's development. The assumptions which are not realistic can cause a nosedive in the entire venture.

Although assumptions are conceptual, there should be a move to develop tangible assumptions to move towards a successful product.

Once the Market study is done, the customer's need is given to the Research and Development Department to develop a cost-effective system that could potentially solve customer's needs better than the competitors.

Once the system is developed and tested in a hypothetical environment, the development team takes control of it. The development team adopts one of the software development models to develop the proposed system and gives it to the customers.

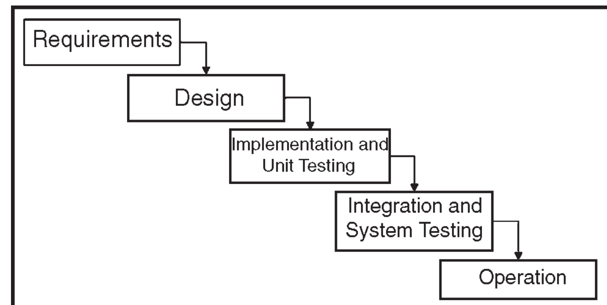



**Fig.** Waterfall Life Cycle Model.

## Advantages

- Simple and easy to use.
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

## Disadvantages

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

## Extreme programming (XP)

Is the latest incarnation of Waterfall model and is the most recent software fad. Most postulates of Extreme programming are pure fantasy. It has been well known for a long time that *big bang* or

waterfall models don't work well on projects with complex or shifting requirements.

The same is true for XP. Too many shops implement XP as an excuse for not understanding the user requirements. XP try improve classic waterfall model by trying to start coding as early as possible but without creating a full-fledged prototype as the first stage. In this sense it can be considered to be variant of evolutionary prototyping (see below). Often catch phase "Emergent design" is used instead of evolutionary prototyping. It also introduces a very questionable idea of pair programming as an attempt to improve extremely poor communication between developers typical for large projects. While communication in large projects is really critical and attempts to improve it usually pay well, "pair programming" is a questionable strategy.

*There are two main problems here:*

1. In a way it can be classified as a hidden attempt to create one good programmer out of two mediocre. But in reality it is creating one mediocre programmer from two or one good. No senior developer is going to put up with some jerk sitting on his lap asking questions about every line. It just prevents the level of concentration needed for high quality coding. Microsoft's idea of having a tester for each programmer is more realistic: one developer writes tests.
2. The actual code to be tested. This forces each of them to communicate and because tester has different priorities then developer such communication brings the developer a new and different perspective on his code, which really improves quality. This combination of different perspectives is a really neat idea as you can see from the stream of Microsoft Office products and operating systems.

## Throwaway prototyping model

Typical implementation language is scripting language and Unix shell (due to availability huge amount of components that can be used for construction of the prototype).

## Spiral model

The spiral model is a variant of "dialectical spiral" and as such provides useful insights into the life cycle of the system. Can be considered as a generalization of the proto-typing model.

That why it is usually implemented as a variant of prototyping model with the first iteration being a prototype. The spiral model is similar to the incremental model, with more emphases placed on risk analysis.

*The spiral model has four phases*: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed.

Each subsequent spirals builds on the baseline spiral. Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions.

A prototype is produced at the end of the risk analysis phase. Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral. In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

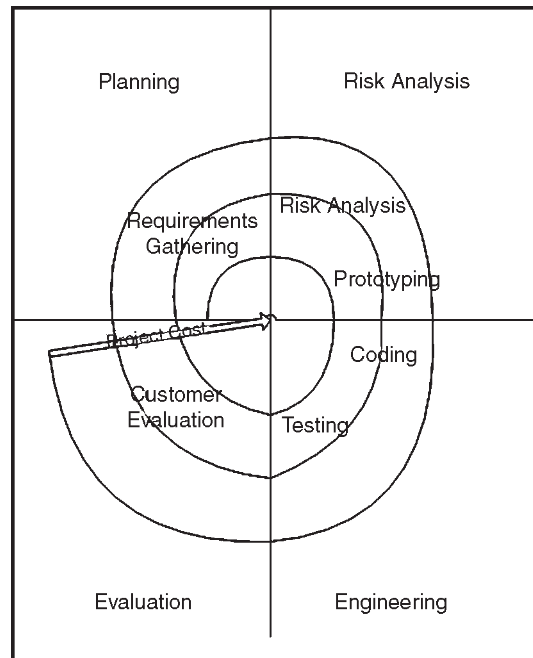



**Fig.** Spiral Life Cycle Model.

## Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

## Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

## Evolutionary prototyping model

This is kind of mix of Waterfall model and prototyping. Presuppose gradual refinement of the prototype until a usable

product emerge. Might be suitable in projects where the main problem is user interface requirements, but internal architecture is relatively well established and static. Can help to cope with organizational sclerosis. One variant involves so called "binary" software implementation model using a scripting language plus statically typed language.

In this case system first is coded in a scripting language and then gradually critical components are rewritten in the lower language.

## OSS development model

It is the latest variant of evolutionary prototype model. The "waterfall model" was probably the first published model and as a specific model for military it was not as naive as some proponents of other models suggest.

The model was developed to help cope with the increasing complexity of aerospace products. The waterfall model followed a documentation driven paradigm.

Prototyping model was probably the first realistic of early models because many aspects of the syst4m are unclear until a working prototype is developed.

A better model, the "spiral model" was suggested by Boehm in 1985. The spiral model is a variant of "dialectical spiral" and as such provides useful insights into the life cycle of the system.

But it also presuppose unlimited resources for the project. No organization can perform more then a couple iterations during the initial development of the system. the first iteration is usually called prototype.

Prototype based development requires more talented managers and good planning while waterfall model works (or does not work)

with bad or stupid managers works just fine as the success in this model is more determined by the nature of the task in hand then any organizational circumstances.

Like always humans are flexible and programmer in waterfall model can use guerilla methods of enforcing a sound architecture as manager is actually a hostage of the model and cannot afford to look back and re-implement anything substantial.

Because the life cycle steps are described in very general terms, the models are adaptable and their implementation details will vary among different organizations.

The spiral model is the most general. Most life cycle models can in fact be derived as special instances of the spiral model. Organizations may mix and match different life cycle models to develop a model more tailored to their products and capabilities.

There is nothing wrong about using waterfall model for some components of the complex project that are relatively well understood and straightforward. But mixing and matching definitely needs a certain level of software management talent.

## V-Shaped Model

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though.

The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation. Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering. The high-level design phase focuses

on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together. The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well. The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.



## Advantages

- Simple and easy to use.
- Each phase has specific deliverables.
- Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
- Works well for small projects where requirements are easily understood.

## Disadvantages

- Very rigid, like the waterfall model.
- Little flexibility and adjusting scope is difficult and expensive.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.

- Model doesn't provide a clear path for problems found during testing phases.

## Incremental Model

The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed iterations.

Each iteration passes through the requirements, design, implementation and testing phases. A working version of software is produced during the first iteration, so you have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

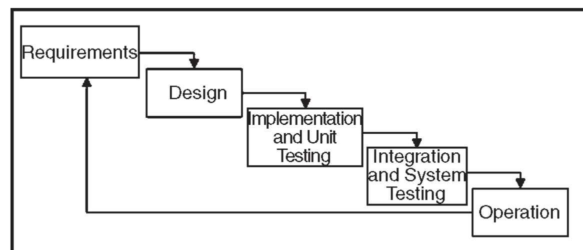



**Fig.** Incremental Life Cycle Model.

## Advantages

- Generates working software quickly and early during the software life cycle.
- *More flexible*: Less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

## Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

# Software Development in Life Cycle Models

The Systems Development Life Cycle (SDLC) is a conceptual model used in project management that describes the stages involved in an information system development project from an initial feasibility study through maintenance of the completed application. Various SDLC methodologies have been developed to guide the processes involved including the waterfall model (the original SDLC method), rapid application development (RAD), joint application development (JAD), the fountain model and the spiral model. Mostly, several models are combined into some sort of hybrid methodology. Documentation is crucial regardless of the type of model chosen or devised for any application, and is usually done in parallel with the development process.

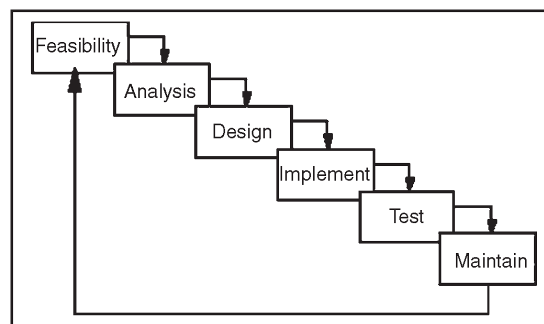



**Fig.** Briefly on different Phases.

Some methods work better for specific types of projects, but in the final analysis, the most important factor for the success of a project may be how closely particular plan was followed. The image above is the classic Waterfall model methodology, which is the first SDLC method and it describes the various phases involved in development.

## Feasibility

The feasibility study is used to determine if the project should get the go-ahead. If the project is to proceed, the feasibility study will produce a project plan and budget estimates for the future stages of development.

## Requirement Analysis and Design

Analysis gathers the requirements for the system. This stage includes a detailed study of the business needs of the organization. Options for changing the business process may be considered. Design focuses on high level design like, what programs are needed and how are they going to interact, low-level design (how the individual programs are going to work), interface design (what are the interfaces going to look like) and data design (what data will be required). During these phases, the software's overall structure is defined. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

## Implementation

In this phase the designs are translated into code. Computer programs are written using a conventional programming language

or an application generator. Progra-mming tools like Compilers, Interpreters, Debuggers are used to generate the code. Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

## Testing

In this phase the system is tested. Normally programs are written as a series of individual modules, these subject to separate and detailed test. The system is then tested as a whole. The separate modules are brought together and tested as a complete system. The system is tested to ensure that interfaces between modules work (integration testing), the system works on the intended platform and with the expected volume of data (volume testing) and that the system does what the user requires (acceptance/beta testing).

## Maintenance

Inevitably the system will need maintenance. Software will definitely undergo change once it is delivered to the customer. There are many reasons for the change. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

## Description

## Curtain Raiser

Like any other set of engineering products, software products are also oriented towards the customer. It is either market driven

or it drives the market. Customer Satisfaction was the buzzword of the 80's. Customer Delight is today's buzzword and Customer Ecstasy is the buzzword of the new millennium.

Products that are not customer or user friendly have no place in the market although they are engineered using the best technology. The interface of the product is as crucial as the internal technology of the product.

## Market Research

A market study is made to identify a potential customer's need. This process is also known as market research. Here, the already existing need and the possible and potential needs that are available in a segment of the society are studied carefully. The market study is done based on a lot of assumptions.

Assumptions are the crucial factors in the development or inception of a product's development. Unrealistic assumptions can cause a nosedive in the entire venture.

Though assumptions are abstract, there should be a move to develop tangible assumptions to come up with a successful product.

## Research and Development

Once the Market Research is carried out, the customer's need is given to the Research & Development division (R&D) to conceptualize a cost-effective system that could potentially solve the customer's needs in a manner that is better than the one adopted by the competitors at present. Once the conceptual system is developed and tested in a hypothetical environment, the development team takes control of it. The development team adopts one of the software development methodologies that is given below, develops the proposed system, and gives it to the

customer. The Sales & Marketing division starts selling the software to the available customers and simultaneously works to develop a niche segment that could potentially buy the software. In addition, the division also passes the feedback from the customers to the developers and the R&D division to make possible value additions to the product.

While developing a software, the company outsources the non-core activities to other companies who specialize in those activities. This accelerates the software development process largely. Some companies work on tie-ups to bring out a highly matured product in a short period.

## Software Development Models

The following are some basic popular models that are adopted by many software development firms

- System Development Life Cycle (SDLC) Model
- Prototyping Model
- Rapid Application Development Model
- Component Assembly Model

### System Development Life Cycle Model

A software life cycle model depicts the significant phases or activities of a software project from conception until the product is retired. It specifies the relationships between project phases, including transition criteria, feedback mechanisms, milestones, baselines, reviews, and deliverables. Typically, a life cycle model addresses the phases of a software project: requirements phase, design phase, implementation, integration, testing, operations and maintenance. Much of the motivation behind utilizing a life cycle model is to provide structure to avoid the problems of the "undisciplined hacker" or

corporate IT bureaucrat (which is probably ten times dangerous then undisciplined hacker). As always, it's a matter of picking the right tool for the job, rather than picking up your hammer and treating everything as a nail.

## System/Information Engineering and Modeling

As software is always of a large system (or business), work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must interface with other elements such as hardware, people and other resources. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, the system should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

## Software Requirement Analysis

This process is also known as feasibility study. In this phase, the development team visits the customer and studies their system. They investigate the need for possible software automation in the given system.

By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc.... The requirement gathering process is intensified and focussed specially on software.

To understand the nature of the programme(s) to be built, the system engineer or "Analyst" must understand the information domain for the software, as well as required function, behaviour,

performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved.

## System Analysis and Design

In this phase, the software development process, the software's overall structure and its nuances are defined. In terms of the client/ server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc... are all defined in this phase.

A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

## Code Generation

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Progra-mming tools like compilers, interpreters, debuggers etc... are used to generate the code.

Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

## Testing

Once the code is generated, the software programme testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases.

Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations.

## Maintenance

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

## Prototyping Model

This is a cyclic version of the linear model. In this model, once the requirement analysis is done and the design for a prototype is made, the development process gets started.

Once the prototype is created, it is given to the customer for evaluation. The customer tests the package and gives his/her feed back to the developer who refines the product according to the customer's exact expectation. After a finite number of iterations, the final software package is given to the customer.

In this methodology, the software is evolved as a result of periodic shuttling of information between the customer and developer. This is the most popular development model in the contemporary IT industry.

Most of the successful software products have been developed using this model - as it is very difficult (even for a whiz kid!) to comprehend all the requirements of a customer in one shot.

There are many variations of this model skewed with respect to the project management styles of the companies. New versions

of a software product evolve as a result of prototyping. The goal of prototyping based development is to counter the first two limitations of the waterfall model discussed earlier. The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding and testing.

But each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.

In such situations letting the client "plan" with the prototype provides invaluable and intangible inputs which helps in determining the requirements for the system. It is also an effective method to demonstrate the feasibility of a certain approach.

This might be needed for novel systems where it is not clear those constraints can be met or that algorithms can be developed to implement the requirements. The process model of the prototyping approach is shown in the figure below.

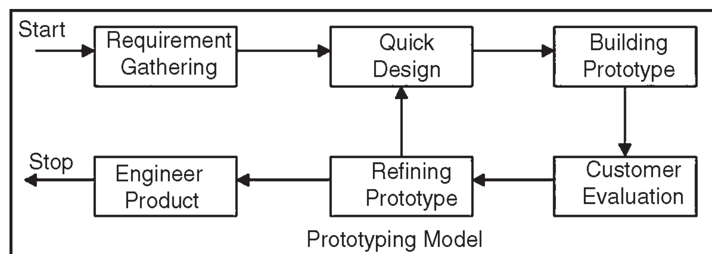



**Fig.** Prototyping Model.

The basic reason for little common use of prototyping is the cost involved in this built-it-twice approach. However, some argue that prototyping need not be very costly and can actually reduce the overall development cost. The prototype are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

In addition, the cost of testing and writing detailed documents are reduced. These factors helps to reduce the cost of developing the prototype. On the other hand, the experience of developing the prototype will very useful for developers when developing the final system. This experience helps to reduce the cost of development of the final system and results in a more reliable and better designed system.

## Advantages of Prototyping

Creating software using the prototype model also has its benefits. One of the key advantages a prototype modeled software has is the time frame of development. Instead of concentrating on documentation, more effort is placed in creating the actual software. This way, the actual software could be released in advance.

The work on prototype models could also be spread to others since there are practically no stages of work in thismodel. Everyone has to work on the same thing and at the same time, reducing man hours in creating a software. The work will even be faster and efficient if developers will collaborate more regarding the status of a specific function and develop the necessary adjustments in time for the integration.

Another advantage of having a prototype modeled software is that the software is created using lots of user feedbacks. In every

prototype created, users could give their honest opinion about the software. If something is unfavorable, it can be changed. Slowly the programme is created with the customer in mind.

- Users are actively involved in the development
- It provides a better system to users, as users have natural tendency to change their mind in specifying requirements and this method of developing systems supports this user tendency.
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier as the system is mode side by side.
- Quicker user feedback is available leading to better solutions.

## Disadvantages

Implementing the prototype model for creating software has disadvantages. Since its being built out of concept, most of the models presented in the early stage are not complete. Usually they lack flaws that developers still need to work on them again and again. Since the prototype changes from time to time, it's a nightmare to create a document for this software. There are many things that are removed, changed and added in a single update of the prototype and documenting each of them has been proven difficult.

There is also a great temptation for most developers to create a prototype and stick to it even though it has flaws. Since prototypes are not yet complete software programs, there is always a possibility of a designer flaw. When flawed software is implemented, it could mean losses of important resources.

Lastly, integration could be very difficult for a prototype model. This often happens when other programs are already stable. The prototype software is released and integrated to the company's suite of software. But if there's something wrong the prototype, changes are required not only with the software. It's also possible that the stable software should be changed in order for them to be integrated properly.

## Prototype Models Types

There are four types of Prototype Models based on their development planning: the Patch-Up Prototype, Nonoperational Prototype, First-of-a-Series Prototype and Selected Features Prototype.

### Patch Up Prototype

This type of Prototype Model encourages cooperation of different developers. Each developer will work on a specific part of the programme. After everyone has done their part, the programme will be integrated with each other resulting in a whole new programme. Since everyone is working on a different field, Patch Up Prototype is a fast development model.

If each developer is highly skilled, there is no need to overlap in a specific function of work.

This type of software development model only needs a strong project manager who can monitor the development of the programme. The manager will control the work flow and ensure there is no overlapping of functions among different developers.

### Non-Operational Prototype

A non-operational prototype model is used when only a certain part of the programme should be updated. Although it's not a fully

operational programme, the specific part of the programme will work or could be tested as planned. The main software or prototype is not affected at all as the dummy programme is applied with the application.

Each developer who is assigned with different stages will have to work with the dummy prototype. This prototype is usually implemented when certain problems in a specific part of the programme arises. Since the software could be in a prototype mode for a very long time, changing and maintenance of specific parts is very important. Slowly it has become a smart way of creating software by introducing small functions of the software.

### First of a Series Prototype

Known as a beta version, this Prototype Model could be very efficient if properly launched. In all beta versions, the software is launched and even introduced to the public for testing. It's fully functional software but the aim of being in beta version is to as for feedbacks, suggestions or even practicing the firewall and security of the software.

It could be very successful if the First of a Series Prototype is properly done. But if the programme is half heartedly done, only aiming for additional concept, it will be susceptible to different hacks, ultimately backfiring and destroying the prototype.

### Selected Features Prototype

This is another form of releasing software in beta version. However, instead of giving the public the full version of the software in beta, only selected features or limited access to some important tools in the programme is introduced.

Selected Features Prototype is applied to software that are part of a bigger suite of programs. Those released are independent of

the suite but the full version should integrate with other software. This is usually done to test the independent feature of the software.

## Rapid Application Development (RAD) Model

The RAD modelis a linear sequential software development process that emphasizes an extremely short development cycle. The RAD model is a "high speed" adaptation of the linear sequential model in which rapid development is achieved by using a component-based construction approach. Used primarily for information systems applications, the RAD approach encompasses the following phases:

### Business Modeling

The information flow among business functions is modeled in a way that answers the following questions:

- What information drives the business process?
- What information is generated?
- Who generates it?
- Where does the information go?
- Who processes it?

### Data Modeling

The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristic (called attributes) of each object is identified and the relationships between these objects are defined.

### Process Modeling

The data objects defined in the data-modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing the descriptions are created for adding, modifying, deleting, or retrieving a data object.

## Application Generation

The RAD model assumes the use of the RAD tools like VB, VC++, Delphi etc... rather than creating software using conventional third generation programming languages. The RAD model works to reuse existing programme components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

## Testing and Turnover

Since the RAD process emphasizes reuse, many of the programme components have already been tested. This minimizes the testing and development time.

## Component Assembly Model

Object technologies provide the technical framework for a component-based process model for software engineering. The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithm that are used to manipulate the data. If properly designed and implemented, object oriented classes are reusable across different applicationsand computer based system architectures. Component Assembly Model leads to software reusability. The integration/assembly of the already existing software components accelerate the development process. Nowadays many component libraries are available on the Internet. If the right components are chosen, the integration aspect is made much simpler.

All these different software development models have their own advantages and disadvantages. Nevertheless, in the contemporary commercial software evelopment world, the fusion

of all these methodologies is incorporated. Timing is very crucial in software development. If a delay happens in the development phase, the market could be taken over by the competitor.

Also if a 'bug' filled product is launched in a short period of time (quicker than the competitors), it may affect the reputation of the company. So, there should be a tradeoff between the development time and the quality of the product. Customers don't expect a bug free product but they expect a user-friendly product.

## Software Measurement and Metrics

The measurement information model is a structure linking information needs to the relevant entities and attri-butes of concern. Entities include processes, products, projects, and resources. The measurement information model describes how the relevant attributes are quantified and converted to indicators that provide a basis for decision-making.

The selection or definition of appropriate measures to address an information need begins with a measurable concept: an idea of which measurable attributes are related to an information need and how they are related. The measurement planner defines measurement constructs that link these attributes to a specifiedinformation need. Each construct may involve several types or levels of measures.

This measurement information model (see Figure) identifies the basic terms and concepts with which the measurement analyst must deal. The measurement modelhelps to determine what the measurement planner needs to specify during measurement planning, performance, and evaluation.

## Entity

An entity is an object (for example, a process, product, project, or resource) that is to be characterized by measuring its attributes. Typical software engineering objects can be classified as products (e.g., design document, source code, and test case), processes (e.g., design process, testing process, requirements analysis process), projects, and resources (e.g., the programmers and the testers).

An entity may have one or more properties that are of interest to meet the information needs. In practice, an entity can be classified into more than one of the above categories.

## Measurable attribute

An attribute is a property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means. An entity may have many attributes, only some of which may be of interest for measurement. The first step in defining a specific instantiation of the measurement information model is to select the attributes that are most relevant to the measurement user's information needs. A given attribute may be incorporated in multiple measurement constructs supporting different information needs.

## Base measure

A base measure is an attribute and the method for quantifying it. A base measure is functionally independent of other measures. A base measure captures information about a single attribute. Data collection involves assigning values to base measures. Specifying the expected range and/or type of values of a base measure helps to verify the quality of the data collected.

## Measurement Method

A measurement method is a logical sequence of operations, described generically, used in quantifying an attribute with respect to a specified scale. The operations may involve activities such as counting occurrences or observing the passage of time. The same measurement method may be applied to multiple attributes.

However, each unique combination of an attribute and a method produces a different base measure. Some measure-ment methods may be implemented in multiple ways. A measurement procedure describes the specific implemen-tation of a measurement method within a given organizatio-nal context.

### Type of Measurement Method

The type of measurement method depends on the nature of the operations used to quantify an attribute. Two types of method may be distinguished:

1. *Subjective*: Quantification involving human judgment
2. *Objective*: Quantification based on numerical rules such as counting. These rules may be implemented via human or automated means.

### Scale

A scale is an ordered set of values, continuous or discrete, or a set of categories to which the attribute is mapped. The measurement method maps the magnitude of the measured attribute to a value on a scale. A unit of measurement often is associated with a scale.

### Type of Scale

The type of scale depends on the nature of the relationship between values on the scale.

*Four types of scales are commonly defined*:

1. *Nominal*: The measurement values are categorical. For example, the classification of defects by their type.
2. *Ordinal*: The measurement values are rankings. For example, the assignment of defects to a severity level.
3. *Interval*: The measurement values have equal distances corresponding to equal quantities of the *attribute*. For example, cyclomatic complexity has the minimum value of one, but each increment represents an additional path.
4. *Ratio:* The measurement values have equal distances corresponding to equal quantities of the *attribute* where the value of zero corresponds to none of the*attribute*. For example, the size of a software component in terms of LOC.

The method of measurement usually affects the type of *scale* that can be used reliably with a given *attribute*.

For example, subjective methods of measurement usually only support ordinal or nominal scales.

### Unit of Measurement

A *unit of measurement* is a particular quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitude relative to that quantity. Only quantities expressed in the same units of measurement are directly comparable. Example of units include the hour and the meter.

## Derived measure

A derived measure is a measure that is defined as a function of two or more base measures. Derived measures capture information about more than oneattribute. Simple transformations

of base measures (for example, taking the square root of a base measure) do not add information, thus do not produce derived measures.

Normalization of data often involves converting base measures into derived measures that can be used to compare different entities.

## Measurement Function

A measurement function is an algorithm or calculation performed to combine two or more base measures. The scale and unit of the derived measure depend on the scales and units of the base measures from which it is composed as well as how they are combined by the function.

## Indicator

An indicator is an estimate or evaluation of specified attributes derived from a model with respect to defined information needs.

Indicators are the basis for analysis and decision-making. These are what should be presented to measurement users.

Measurement is always based on imperfect information, so quantifying the uncertainty, accuracy, or importance of indicators is an essential component of presenting the actual indicator value. Therefore, an interpretation of indicat-ors is performed to provide the desired information product.

## Measurement Model

A measurement model is an algorithm or calculation combining one or more base and/or derived measures with associated decision criteria. It is based on an understanding of, or assumptions about, the expected relationship between the

component measures and/or their behaviour over time. Models produce estimates or evaluations relevant to defined information needs. The scale and measurement method affect the choice of analysis techniques or models used to produceindicators.

### Decision Criteria

Decision criteria are numerical thresholds or targets used to determine the need for action or further investigation, or to describe the level of confidence in a given result.

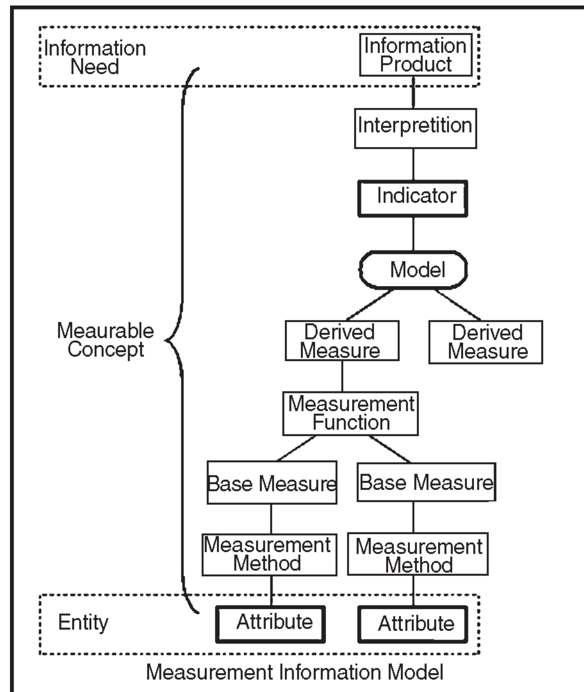
Decision criteria help to interpret the results of measurement. Decision criteria may be calculated or based on a conceptual understanding of expected behaviour. Decision criteria may be derived from historical data, plans, and heuristics, or computed as statistical control limits or statistical confidence limits.

## Measurable concept

A measurable concept is an abstract relationship between attributes of entities and information needs. For example, an Information need may be the need to compare the software development productivity of a project group against a target rate.

The Measurable Concept in this case is "software development productivity rate". To evaluate the concept might require measuring the size of the software products and the amount of resource applied to create the products (depending on the chosen model of productivity).

Additional examples of Measurable Concepts include quality, risk, performance, capability, maturity, and customer value.




## Software Metrics

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality. This module introduces the most commonly used software metrics and reviews their use in constructing models of the software development process.

Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as we gain additional experience with various metrics and oftware metrics are numerical data related to software development. Metrics strongly support software project management activities.

*They relate to the four functions of management as follows*:

1. *Planning*: Metrics serve as a basis of cost estimating, training planning, resource planning, scheduling, and budgeting.
2. *Organizing*: Size and schedule metrics influence a project's organization.
3. *Controlling*: Metrics are used to status and track software development activities for compliance to plans.
4. *Improving*: Metrics are used as a tool for process improvement and to identify where improvement efforts should be concentrated and measure the effects of process improvement efforts.

A metric quantifies a characteristic of a process or product. Metrics can be directly observable quantities or can be derived from one or more directly observable quantities. Examples of raw metrics include the number of source lines of code, number of documentation pages, number of staff-hours, number of tests, number of requirements, etc. Examples of derived metrics include source lines of code per staff-hour, defects per thousand lines of code, or a cost performance index.

The term *indicator* is used to denote a representation of metric data that provides insight into an ongoing software development project or process improvement activity. Indicators are metrics in a form suitable for assessing project behaviour or process improvement. For example, an indicator may be the behaviour of a metric over time or the ratio of two metrics.

Indicators may include the comparison of actual values versus the plan, project stability metrics, or quality metrics. Examples of indicators used on a project include actual versus planned task completions, actual versus planned staffing, number of trouble

reports written and resolved over time, and number of requirements changes over time. Indicators are used in conjunction with one another to provide a more complete picture of project or organization behaviour. For example, a progress indicator is related to requirements and size indicators. All three indicators should be used and interpreted together.

# 5

# Computer Graphics System

Let us consider the organization of a typical graphics system we might use. As our initial emphasis will be on how the applications programmer sees the system, we shall omit details of the hardware.

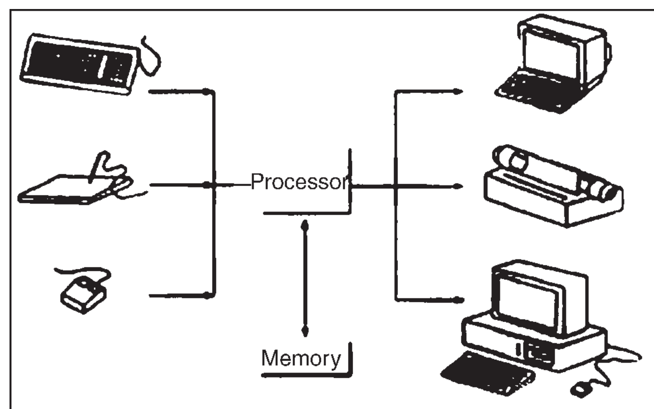



**Fig.** The Graphic System

The model is general enough to include workstations, personal computers, terminals attached to a central time-shared computer, and sophisticated image-generation systems. In most ways, this

block diagram is that of a standard computer. How each element is specialized for computer graphics will characterize this diagram as one of a graphics system, rather than one of a general-purpose computer.

## The Processor

Within the processor box, two types of processing take place. The first is picture formation processing. In this stage, the user programme or commands are processed. The picture is formed from the elements (lines, text) available in the system using the desired attributes. Such as line colour and text font. The user interface is a part of this processing. The picture can be specified in a number of ways, such as through an interactive menu-controlled painting programme or via a C programme using a graphics library. The physical processor used in this stage is often the processor in the workstation or host computer.

The second kind of processing is concerned with the display of the picture. In a raster system, the specified primitives must be scan converted. The screen must be refreshed to avoid flicker. Input from the user might require objects to be repositioned on the display. The kind of processor best suited for these jobs is not the standard type of processor found in most computers. Instead, special boards and chips are often used. As we have already noted, one of the elements that distinguishes real-time graphics systems is their use of display processors. Since we have agreed to stay at the block-diagram level for now, however, we shall not explore these architectures in any detail until later.

## Memory

There are often two distinct types of memory employed in graphics systems. For the processing of the user programme, the

memory is similar to that of a standard computer, as the picture is formed by a standard type of arithmetic processing. Display processing, however, requires high-speed display memory that can be accessed by the display processor, and, in raster systems, memory for the frame buffer.

This display memory usually is different in both its physical characteristics and its organization from what is used by the picture processor. At this point, we need not consider details of how memory can be organized.

You should be aware that the way the internals of our processor and memory boxes are organized distinguishes a slow system from a real-time picture-generating system, such as a flight simulator. However, from our present perspective, we shall emphasize that all implementations have to do the same kinds of tasks to produce output.

## Output Devices

Our basic system has one or more output devices. As raster displays are the dominant type, we shall assume there is a raster-scan CRT on our system. We shall consider the frame buffer to be part of the display memory. In a self-contained system such as a workstation, the display is an integral part of the system, so the transfer of information from the processor to the display will happen rapidly.

When the display is separate, such as with a graphics terminal, the speed of the connection is much slower. Terminals with raster displays usually must have their own frame buffers, so the displays can be refreshed locally. In our simple system, we might also have other displays, such as a plotter, to allow us to produce hardcopy.

### Input Devices

A simple system may have only a keyboard to provide whatever input is necessary. Keyboards provide digital codes corresponding to sequences of keystrokes by a user. These sequences are usually interpreted as codes for characters. If individual keystrokes or groups of keystrokes are interpreted as graphical input, the keyborad can be used as a complex input device. For example, the "arrow" keys available on most keyboards can be used to direct the movement of a cursor on the screen. Most graphics systems will provide at least one other input device. The most common are the mouse, the lightpen, the joystick, and the data tablet. Each can provide positional information to the system and each usually is equipped with one or more buttons to provide signals to the processor. From the programmer's perspective, there are numerous important issues with regard to the input and output devices. We must consider how the programme can communicate with these devices. We must decide what kinds of input and output can be produced. We will be interested in how to control multiple devices, so that we can choose a particular device for our input, and can direct our output to some group of the available output devices.

## Historical Background of Computer Graphics

Today there are very few aspects of our lives not affected by computers. Practically every cash or monetary transaction that takes place daily involves a computer. In many cases, the same is true of computer graphics. Whether you see them on television, in newspapers, in weather reports or while at the doctor's surgery, computer images are all around you.

*"A picture is worth a thousand words"* is a well-known saying, and highlights the advantages and benefits of the visual presentation of our data. We are able to obtain a comprehensive overall view of our data and also study features and areas of particular interest.

A well-chosen graph is able to transform a complex table of numbers into meaningful results. Such graphs are used to illustrate papers, reports, and theses, as well as providing the basis for presentation material in the form of slides and overhead transparencies. A range of tools and facilities are available to enable users to visualise their data, and this document provides a brief summary and overview. Computer graphics are used in many disciplines and subjects but for the purpose of this document, we will split the topic of computer graphics into the following fields:

## Charting

One of the prime uses for graphical software at the University is to produce graphs and charts. Everyone has data of one kind or another, whether on paper, in the computer, or just in the mind. We often need to know the significance and properties of the data, or to be able to compare different parts of it against other data sets.

One of the simplest aspects of data display is the production of charts. This is where you would want to put your data into a graphical form to show relationships and comparisons between sets of values.

*There may be a number of reasons why you would want to put your data into a chart:*

- To illustrate differences between different sets of data,
- To show trends between two variables,
- To show patterns of behaviour in one variable.

*There are basically two broad areas of graphs:*

- Presentation charts and graphs of the kind used to illustrate a few principal points. We see these on news and current affairs programmes on television. A bar chart or a pie chart is used to indicate results of data obtained so far and the general trends. They are often liberally decorated with bright colours to increase their visual appeal and attractiveness to the viewers and to hold their attention. They are used for visual impact and getting a simple point over clearly and effectively.
- Scientific charts and graphs are more concerned with ensuring that the detail in the data is represented accurately and faithfully. We may have some results obtained from experimental measurements and wish to display them. We may want to compare the results from the data measurements with the results we would expect according to a particular theoretical model. We may want to draw a curve through the data points (*i.e.* interpolate the data) and display this along with the original points.

The aims of the two are different, and so the facilities you will want from your charting package will also be different. Presentation charting has more to do with impressive presentation graphics where the aim is to put a salient point across to an audience. As a result the priority with this sort of charting is not always accuracy of representation. You want charts with strong colours, an impressive look and special effects. The effect of a presentation can be enhanced by using 3D graphs, adding pictures to the graph, or using pictograms. These sorts of charts are rarely produced in isolation but as part of a general presentation. Therefore, some presentation packages also have their own

charting module for this purpose. Word and PowerPoint use a module called Microsoft Graph and Excel's charting module has some very powerful presentation graphics features. Origin and Gsharp, both dedicated charting packages, also provide professional presentation charting facilities on the PC systems. Gsharp is also available on the UNIX systems.

In scientific charting you want to display data as accurately as possible in order to analyse it graphically or demonstrate clearly your comparisons and results. As this sort of charting is done mainly for analysis, it is rarely an isolated activity but is often done alongside detailed numerical analysis of your data. Two of the most powerful charting packages available are Origin on the PC network and Gsharp on the PC and UNIX systems. Also, many numerical analysis packages have their own charting modules integrated with the rest of the package. It is clear that your choice of charting programme will depend very much on what purpose you want the chart to fulfil, and also what other programmes you are already using. On the whole, if you are already using a programme that has its own charting module, use that. The table below gives some rough guidelines on your choice of charting PC package, with the packages increasing in facilities and complexity going down the table.

| ***Requirement*** | ***Choice*** |
|---|---|
| Simple bar, column, line or pie charts to integrate in a word processor | Microsoft Graph in Word, Charting Module in Excel |
| Charts for use in a presentation Origin | Microsoft Graph in Word or PowerPoint, Charting Module in Excel, |
| Raw data requiring good quality scientific charting | Origin, Gsharp |
| Data requiring simple mathematical or statistical analysis | Charting Module in Excel, Origin, Gsharp |
| Complicated statistical analysis and good quality scientific charts | Graphics module in SPSS |

## Presentations

Presentation software is used to create material used in presentations, such as OHP transparencies and 35mm slides. The term is also commonly used when a presentation is given using the output from a computer screen. The use of presentation software is becoming of increasing importance as higher standards become expected in courses and presentations. This will often include making use of colour, graphics and the University logo.

Course materials produced using presentation packages can be delivered in a number of ways. The simplest way is to print the material on a laser printer and then use a photocopier to produce overhead projector (OHP) acetates (first making sure that the photocopier can accept acetates). You can also use the output services produced by Information Systems Services and University Media Services to produce colour output or output on 35mm slides.

Alternatively you can give a desktop presentation using OHP projection tablets or projection systems to deliver a presentation using the output from a computer system directly. The simplest presentation software is a word processor. Word processing packages such as Word, which can produce text in a variety of sizes, can be used to create OHP transparencies.

Specialist presentation packages, such as PowerPoint, provide a wider range of facilities than word processors and, in general, are easier to use for the production of presentation materials. PowerPoint is a presentation software programme that helps you quickly and easily create professional quality presentations.

Presentations can be transferred onto paper, overheads or 35mm slides, or they can be shown on a video screen or computer monitor. PowerPoint's printing options include formats ranging from audience handouts to speaker's notes.

# Graphics Pipeline Performance

Over the past few years, the hardware-accelerated rendering pipeline has rapidly increased in complexity, bringing with it increasingly intricate and potentially confusing performance characteristics.

Improving performance used to mean simply reducing the CPU cycles of the inner loops in your renderer; now it has become a cycle of determining bottlenecks and systematically attacking them.

This loop of *identification* and *optimization* is fundamental to tuning a heterogeneous multiprocessor system; the driving idea is that a pipeline, by definition, is only as fast as its slowest stage. Thus, while premature and unfocused optimization in a single-processor system can lead to only minimal performance gains, in a multiprocessor system such optimization very often leads to *zero* gains.

Working hard on graphics optimization and seeing zero performance improvement is no fun. The goal of this chapter is to keep you from doing exactly that.

## The Pipeline

The pipeline, at the very highest level, can be broken into two parts: the CPU and the GPU. Although CPU optimization is a critical part of optimizing your application, it will not be the focus of this chapter, because much of this optimization has little to do with the graphics pipeline.

The GPU, there are a number of functional units operating in parallel, which essentially act as separate special-purpose processors, and a number of spots where a bottleneck can occur. These include vertex and index fetching, vertex shading

(transform and lighting, or T&L), fragment shading, and raster operations (ROP).

## Methodology

Optimization without proper bottleneck identification is the cause of much wasted development effort, and so we formalize the process into the following fundamental identification and optimization loop:

1. Identify the bottleneck. For each stage in the pipeline, vary either its workload or its computational ability (that is, clock speed). If performance varies, you've found a bottleneck.
2. Optimize. Given the bottlenecked stage, reduce its workload until performance stops improving or until you achieve your desired level of performance.
3. Repeat. Do steps 1 and 2 again until the desired performance level is reached.

## Locating the Bottleneck

Locating the bottleneck is half the battle in optimization, because it enables you to make intelligent decisions about focusing your actual optimization efforts. A flow chart depicting the series of steps required to locate the precise bottleneck in your application. Note that we start at the back end of the pipeline, with the frame-buffer operations (also called raster operations) and end at the CPU. Note also that while any single primitive (usually a triangle), by definition, has a single bottleneck, over the course of a frame the bottleneck most likely changes. Thus, modifying the workload on more than one stage in the pipeline often influences performance. For example, a low-polygon skybox is often bound

by fragment shading or frame-buffer access; a skinned mesh that maps to only a few pixels on screen is often bound by CPU or vertex processing. For this reason, it frequently helps to vary workloads on an object-by-object, or material-by-material, basis.

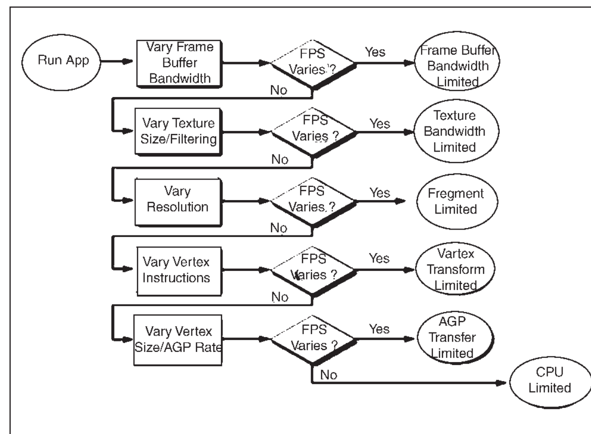



**Fig.** Bottleneck Flowchart

For each pipeline stage, we also mention the GPU clock to which it's tied (that is, core or memory). This information is useful in conjunction with tools such as PowerStrip (EnTech Taiwan 2003), which allows you to reduce the relevant clock speed and observe performance changes in your application.

## Raster Operations

The very back end of the pipeline, raster operations (often called the ROP), is responsible for reading and writing depth and stencil, doing the depth and stencil comparisons, reading and writing colour, and doing alpha blending and testing. As you can see, much of the ROP workload taxes the available frame-buffer bandwidth. The best way to test if your application is frame-buffer-bandwidth bound is to vary the bit depths of the colour or the depth buffers, or both. If reducing your bit depth from 32-bit to 16-bit significantly improves your performance, then you are definitely frame-buffer-bandwidth bound.

Frame-buffer bandwidth is a function of GPU memory clock, so modifying memory clocks is another technique for helping to identify this bottleneck.

## Texture Bandwidth

Texture bandwidth is consumed any time a texture fetch request goes out to memory. Although modern GPUs have texture caches designed to minimize extraneous memory requests, they obviously still occur and consume a fair amount of memory bandwidth.

Modifying texture formats can be trickier than modifying frame-buffer formats as we did when inspecting the ROP; instead, we recommend changing the effective texture size by using a large amount of positive mipmap level-of-detail (LOD) bias. This makes texture fetches access very coarse levels of the mipmap pyramid, which effectively reduces the texture size. If this modification causes performance to improve significantly, you are bound by texture bandwidth.

Texture bandwidth is also a function of GPU memory clock.

## Fragment Shading

Fragment shading refers to the actual cost of generating a fragment, with associated colour and depth values. This is the cost of running the "pixel shader" or "fragment shader." Note that fragment shading and frame-buffer bandwidth are often lumped together under the heading *fill rate*, because both are a function of screen resolution. However, they are two distinct stages in the pipeline, and being able to tell the difference between the two is critical to effective optimization.

Before the advent of highly programmable fragment-processing GPUs, it was rare to be bound by fragment shading. It

was often frame-buffer bandwidth that caused the inevitable correlation between screen resolution and performance. This pendulum is now starting to swing towards fragment shading, however, as the newfound flexibility enables developers to spend oodles of cycles making fancy pixels.

The first step in determining if fragment shading is the bottleneck is simply to change the resolution. Because we've already ruled out frame-buffer bandwidth by trying different frame-buffer bit depths, if adjusting resolution causes performance to change, the culprit is most likely fragment shading. A supplementary approach would be to modify the length of your fragment programmes and see if this influences performance. But be careful not to add instructions that can easily be optimized away by a clever device driver.

Fragment-shading speed is a function of the GPU core clock.

## Vertex Processing

The vertex transformation stage of the rendering pipeline is responsible for taking an input set of vertex attributes (such as model-space positions, vertex normals, texture coordinates, and so on) and producing a set of attributes suitable for clipping and rasterization (such as homogeneous clip-space position, vertex lighting results, texture coordinates, and more). Naturally, performance in this stage is a function of the work done per vertex, along with the number of vertices being processed.

With programmable transformations, determining if vertex processing is your bottleneck is a simple matter of changing the length of your vertex programme. If performance changes, you are vertex-processing bound.

If you're adding instructions, be careful to add ones that actually do meaningful work; otherwise, the instructions may be

optimized away by the compiler or the driver. For example, no-ops that refer to constant registers (such as adding a constant register that has a value of zero) often cannot be optimized away because the driver usually doesn't know the value of a constant at programme-compile time.

If you're using fixed-function transformations, it's a little trickier. Try modifying the load by changing vertex work such as specular lighting or texture-coordinate generation state. Vertex processing speed is a function of the GPU core clock.

## Vertex and Index Transfer

Vertices and indices are fetched by the GPU as the first step in the GPU part of the pipeline. The performance of vertex and index fetching can vary depending on where the actual vertices and indices are placed. They are usually either in system memory—which means they will be transferred to the GPU over a bus such as AGP or PCI Express—or in local frame-buffer memory. Often, on PC platforms especially, this decision is left up to the device driver instead of the application, although modern graphics APIs allow applications to provide usage hints to help the driver choose the correct memory type.

Determining if vertex or index fetching is a bottleneck in your application entails modifying the vertex format size.

Vertex and index fetching performance is a function of the AGP/PCI Express rate if the data is placed in system memory; it's a function of the memory clock if data is placed in local frame-buffer memory.

If none of these tests influences your performance significantly, you are primarily CPU bound. You may verify this fact by underclocking your CPU: if performance varies proportionally, you are CPU bound.

## Optimization

Now that we have identified the bottleneck, we must optimize that particular stage to improve application performance. The following tips are categorized by offending stage.

## Optimizing on the CPU

Many applications are CPU bound—sometimes for good reason, such as complex physics or AI, and sometimes because of poor batching or resource management. If you've found that your application is CPU bound, try the following suggestions to reduce CPU work in the rendering pipeline.

## Reduce Resource Locking

Anytime you perform a synchronous operation that demands access to a GPU resource, there is the potential to massively stall the GPU pipeline, which costs both CPU and GPU cycles. CPU cycles are wasted because the CPU must sit and spin in a loop, waiting for the (very deep) GPU pipeline to idle and return the requested resource. GPU cycles are then wasted as the pipeline sits idle and has to refill.

This locking can occur anytime you

- Lock or read from a surface you were previously rendering to
- Write to a surface the GPU is reading from, such as a texture or a vertex buffer.

In general, you should avoid accessing a resource the GPU is using during rendering.

## Maximize Batch Size

We can also call this tip "Minimize the Number of Batches." A *batch* is a group of primitives rendered with a single API rendering

call (for example, DrawIndexedPrimitive in DirectX 9). The *size* of a batch is the number of primitives it contains.

As a wise man once said, "Batch, Batch, Batch!". Every API function call to draw geometry has an associated CPU cost, so maximizing the number of triangles submitted with every draw call will minimize the CPU work done for a given number of triangles rendered.

*Some tips to maximize the size of your batches:*

- If using triangle strips, use degenerate triangles to stitch together disjoint strips. This will enable you to send multiple strips, provided that they share material, in a single draw call.
- Use texture pages. Batches are frequently broken when different objects use different textures. By arranging many textures into a single 2D texture and setting your texture coordinates appropriately, you can send geometry that uses multiple textures in a single draw call. Note that this technique can have issues with mipmapping and antialiasing. One technique that sidesteps many of these issues is to pack individual 2D textures into each face of a cube map.
- Use GPU shader branching to increase batch size. Modern GPUs have flexible vertex- and fragment-processing pipelines that allow for branching inside the shader. For example, if two batches are separate because one requires a four-bone skinning vertex shader and the other requires a two-bone skinning vertex shader, you could instead write a vertex shader that loops over the number of bones required, accumulating blending weights, and then breaks out of the loop when the weights sum to one. This way,

the two batches could be combined into one. On architectures that don't support shader branching, similar functionality can be implemented, at the cost of shader cycles, by using a four-bone vertex shader on everything and simply zeroing out the bone weights on vertices that have fewer than four bone influences.

- Use the vertex shader constant memory as a lookup table of matrices. Often batches get broken when many small objects share all material properties but differ only in matrix state (for example, a forest of similar trees, or a particle system). In these cases, you can load *n* of the differing matrices into the vertex shader constant memory and store indices into the constant memory in the vertex format for each object. Then you would use this index to look up into the constant memory in the vertex shader and use the correct transformation matrix, thus rendering *n* objects at once.
- Defer decisions as far down in the pipeline as possible. It's faster to use the alpha channel of your texture as a gloss factor, rather than break the batch to set a pixel shader constant for glossiness. Similarly, putting shading data in your textures and vertices can allow for larger batch submissions.

## Reducing the Cost of Vertex Transfer

Vertex transfer is rarely the bottleneck in an application, but it's certainly not impossible for it to happen.

*If the transfer of vertices or, less likely, indices is the bottleneck in your application, try the following:*

- Use the fewest possible bytes in your vertex format. Don't use floats for everything if bytes would suffice (for colours, for example).

- Generate potentially derivable vertex attributes inside the vertex programme instead of storing them inside the input vertex format. For example, there's often no need to store a tangent, binormal, and normal: given any two, the third can be derived using a simple cross product in the vertex programme. This technique trades vertex-processing speed for vertex transfer rate.
- Use 16-bit indices instead of 32-bit indices. 16-bit indices are cheaper to fetch, are cheaper to move around, and take less memory.
- Access vertex data in a relatively sequential manner. Modern GPUs cache memory accesses when fetching vertices. As in any memory hierarchy, spatial locality of reference helps maximize hits in the cache, thus reducing bandwidth requirements.

## Optimizing Vertex Processing

Vertex processing is rarely the bottleneck on modern GPUs, but it may occur, depending on your usage patterns and target hardware. *Try these suggestions if you're finding that vertex processing is the bottleneck in your application:*

- Optimize for the post-T&L vertex cache. Modern GPUs have a small first-in, first-out (FIFO) cache that stores the result of the most recently transformed vertices; a hit in this cache saves all transform and lighting work, along with all work done earlier in the pipeline. To take advantage of this cache, you must use indexed primitives, and you must order your vertices to maximize locality of reference over the mesh. There are tools available—including D3DX and NVTriStrip (NVIDIA 2003)—that can help you with this task.

- Reduce the number of vertices processed. This is rarely the fundamental issue, but using a simple level-of-detail scheme, such as a set of static LODs, certainly helps reduce vertex-processing load.
- Use vertex-processing LOD. Along with using LODs for the number of vertices processed, try LODing the vertex computations themselves. For example, it is likely unnecessary to do full four-bone skinning on distant characters, and you can probably get away with cheaper approximations for the lighting. If your material is multipassed, reducing the number of passes for lower LODs in the distance will also reduce vertex-processing cost.
- Pull out per-object computations onto the CPU. Often, a calculation that changes once per object or per frame is done in the vertex shader for convenience. For example, transforming a directional light vector to eye space is sometimes done in the vertex shader, although the result of the computation changes only once per frame.
- Use the correct coordinate space. Frequently, choice of coordinate space affects the number of instructions required to compute a value in the vertex programme. For example, when doing vertex lighting, if your vertex normals are stored in object space and the light vector is stored in eye space, then you will have to transform one of the two vectors in the vertex shader. If the light vector was instead transformed into object space once per object on the CPU, no per-vertex transformation would be necessary, saving GPU vertex instructions.

- Use vertex branching to "early-out" of computations. If you are looping over a number of lights in the vertex shader and doing normal, low-dynamic-range, [0..1] lighting, you can check for saturation to 1—or if you're facing away from the light—and then break out of further computations. A similar optimization can occur with skinning, where you can break when your weights sum to 1 (and therefore all subsequent weights would be 0). Note that this depends on how the GPU implements vertex branching, and it isn't guaranteed to improve performance on all architectures.

## Speeding Up Fragment Shading

*If you're using long and complex fragment shaders, it is often likely that you're fragment-shading bound. If so, try these suggestions:*

- Render depth first. Rendering a depth-only (no-colour) pass before rendering your primary shading passes can dramatically boost performance, especially in scenes with high depth complexity, by reducing the amount of fragment shading and frame-buffer memory access that needs to be performed. To get the full benefits of a depth-only pass, it's not sufficient to just disable colour writes to the frame buffer; you should also disable all shading on fragments, even shading that affects depth as well as colour (such as alpha test).
- Help early-z optimizations throw away fragment processing. Modern GPUs have silicon designed to avoid shading occluded fragments, but these optimizations rely on knowledge of the scene up to the current point; they can be improved dramatically by rendering in a roughly front-to-back order. Also, laying down depth first in a separate pass

can help substantially speed up subsequent passes (where all the expensive shading is done) by effectively reducing their shaded-depth complexity to 1.

- Store complex functions in textures. Textures can be enormously useful as lookup tables, and their results are filtered for free. The canonical example here is a normalization cube map, which allows you to normalize an arbitrary vector at high precision for the cost of a single texture lookup.
- Move per-fragment work to the vertex shader. Just as per-object work in the vertex shader should be moved to the CPU instead, per-vertex computations (along with computations that can be correctly linearly interpolated in screen space) should be moved to the vertex shader. Common examples include computing vectors and transforming vectors between coordinate systems.
- Use the lowest precision necessary. APIs such as DirectX 9 allow you to specify precision hints in fragment shader code for quantities or calculations that can work with reduced precision. Many GPUs can take advantage of these hints to reduce internal precision and improve performance.
- Avoid excessive normalization. A common mistake is to get "normalization-happy": normalizing every single vector every step of the way when performing a calculation. Recognize which transformations preserve length (such as transformations by an orthonourmal basis) and which computations do not depend on vector length (such as cube-map lookups).

- Consider using fragment shader level of detail. Although it offers less bang for the buck than vertex LOD (simply because objects in the distance naturally LOD themselves with respect to pixel processing, due to perspective), reducing the complexity of the shaders in the distance, and decreasing the number of passes over a surface, can lessen the fragment-processing workload.
- Disable trilinear filtering where unnecessary. Trilinear filtering, even when not consuming extra texture bandwidth, costs extra cycles to compute in the fragment shader on most modern GPU architectures. On textures where mip-level transitions are not readily discernible, turn trilinear filtering off to save fill rate.
- Use the simplest shader type possible. In both Direct3D and OpenGL, there are a number of different ways to shade fragments. For example, in Direct3D 9, you can specify fragment shading using, in order of increasing complexity and power, texture-stage states, pixel shaders version 1.x (ps.1.1 – ps.1.4), pixel shaders version 2.x., or pixel shaders version 3.0. In general, you should use the simplest shader type that allows you to create the intended effect. The simpler shader types offer a number of implicit assumptions that often allow them to be compiled to faster native pixel-processing code by the GPU driver. A nice side effect is that these shaders would then work on a broader range of hardware.

## Reducing Texture Bandwidth

*If you've found that you're memory-bandwidth bound, but mostly when fetching from textures, consider these optimizations:*

- Reduce the size of your textures. Consider your target resolution and texture coordinates. Do your users ever get to see your highest mip level? If not, consider scaling back the size of your textures. This can be especially helpful if overloaded frame-buffer memory has forced texturing to occur from nonlocal memory (such as system memory, over the AGP or PCI Express bus). The NVPerfHUD tool (NVIDIA 2003) can help diagnose this problem, as it shows the amount of memory allocated by the driver in various heaps.
- Compress all colour textures. All textures that are used just as decals or detail textures should be compressed, using DXT1, DXT3, orDXT5, depending on the specific texture's alpha needs. This step will reduce memory usage, reduce texture bandwidth requirements, and improve texture cache efficiency.
- Avoid expensive texture formats if not necessary. Large texture formats, such as 64-bit or 128-bit floating-point formats, obviously cost much more bandwidth to fetch from. Use these only as necessary.
- Always use mipmapping on any surface that may be minified. In addition to improving quality by reducing texture aliasing, mipmapping improves texture cache utilization by localizing texture-memory access patterns for minified textures. If you find that mipmapping on certain surfaces makes them look blurry, avoid the temptation to disable mipmapping or add a large negative LOD bias. Prefer anisotropic filtering instead and adjust the level of anisotropy per batch as appropriate.

## Optimizing Frame-Buffer Bandwidth

The final stage in the pipeline, ROP, interfaces directly with the frame-buffer memory and is the single largest consumer of frame-buffer bandwidth. For this reason, if bandwidth is an issue in your application, it can often be traced to the ROP.

*Here's how to optimize for frame-buffer bandwidth:*

- Render depth first. This step reduces not only fragment-shading cost, but also frame-buffer bandwidth cost.
- Reduce alpha blending. Note that alpha blending, with a destination-blending factor set to anything other than 0, requires both a read and a write to the frame buffer, thus potentially consuming double the bandwidth. Reserve alpha blending for only those situations that require it, and be wary of high levels of alpha-blended depth complexity.
- Turn off depth writes when possible. Writing depth is an additional consumer of bandwidth, and it should be disabled in multipass rendering (where the final depth is already in the depth buffer); when rendering alpha-blended effects, such as particles; and when rendering objects into shadow maps (in fact, for rendering into colour-based shadow maps, you can turn off depth reads as well).
- Avoid extraneous colour-buffer clears. If every pixel is guaranteed to be overwritten in the frame buffer by your application, then avoid clearing colour, because it costs precious bandwidth. Note, however, that you should clear the depth and stencil buffers whenever you can, because many early-z optimizations rely on the deterministic contents of a cleared depth buffer.

- Render roughly front to back. In addition to the fragment-shading advantages mention, there are similar benefits for frame-buffer bandwidth. Early-z hardware optimizations can discard extraneous frame-buffer reads and writes. In fact, even older hardware, which lacks these optimizations, will benefit from this step, because more fragments will fail the depth test, resulting in fewer colour and depth writes to the frame buffer.
- Optimize skybox rendering. Skyboxes are often frame-buffer-bandwidth bound, but you must decide how to optimize them: (1) render them last, reading (but *not* writing) depth, and allow the early-z optimizations along with regular depth buffering to save bandwidth; or (2) render the skybox first, and disable all depth reads and writes. Which option will save you more bandwidth is a function of the target hardware and how much of the skybox is visible in the final frame. If a large portion of the skybox is obscured, the first technique will likely be better; otherwise, the second one may save more bandwidth.
- Use floating-point frame buffers only when necessary. These formats obviously consume much more bandwidth than smaller, integer formats. The same applies for multiple render targets.
- Use a 16-bit depth buffer when possible. Depth transactions are a huge consumer of bandwidth, so using 16-bit instead of 32-bit can be a giant win, and 16-bit is often enough for small-scale, indoor scenes that don't require stencil. A 16-bit depth buffer is also often enough for render-to-texture effects that require depth, such as dynamic cube maps.

- Use 16-bit colour when possible. This advice is especially applicable to render-to-texture effects, because many of these, such as dynamic cube maps and projected-colour shadow maps, work just fine in 16-bit colour.

As power and programmability increase in modern GPUs, so does the complexity of extracting every bit of performance out of the machine. Whether your goal is to improve the performance of a slow application or to look for areas where you can improve image quality "for free," a deep understanding of the inner workings of the graphics pipeline is required. As the GPU pipeline continues to evolve, the fundamental ideas of optimization will still apply: first identify the bottleneck, by varying the load or the computational power of each unit; then systematically attack those bottlenecks, using your understanding of how each pipeline unit behaves.

## Drawing, Painting and Design

Drawing and painting software is available on most platforms at the University. However, there are many differences between software intended primarily for drawing and that intended for painting. Drawing software will provide the user with a set of 'entities' used to construct the drawing (an entity is a drawing element such as a line, circle, or text string). Drawing entities can range from simple lines, points and curves in 2D to their equivalents in 3D and may include 3D surfaces. Advanced versions of drawing packages used for design are referred to as Computer Aided Design (CAD) systems. Painting software tends to work on a conceptually lower layer. Whilst it may provide some entities for constructing geometric shapes (these tend to be 2D geometric

shapes), a painting package will also provide control over individual pixels in the image, *i.e.* it provides direct control over the bitmap. It is worth remembering that opening any image in a painting package causes it to become pixelated.

*The following packages are available on the ISS NT Cluster Desktop:*

- Paint Very basic painting programme. Can create simple pictures and edit bitmaps. Only possible to read in and save files in a BMP format.
- Picture Publisher Painting package used to edit and create pictures. Can read in and save files in a number of different formats.
- Paint Shop Pro Recommended as the main painting package on the desktop. Used to edit and create pictures. Can read in and save files in a number of different formats.
- CorelDraw Recommended as the main drawing package on the desktop. Useful for editing vector graphics.Can read in and save files in both vector and bitmap formats.
- Micrografx Designer Drawing package used for technical drawing.

*The following drawing and painting software is available on the Suns:*

- Island Paint Painting programme that provides tools for creating and editing images formed by monochrome and colour bitmaps. Several painting tools can be used to create geometric and freehand shapes. Scanned images and clip art can also be imported.
- Island Draw 2D drawing package.
- Island Paint General purpose CAD system in use in engineering, and allows 3D solid modelling as well as 2D/ 3D draughting. An extension, AEC, for architectural and construction applications, is also available.

## Computer Aided Design and Drawing

CAD systems provide drawing entities with powerful construction, editing and database techniques. CAD data can also be output and read in by other applications software for analysing the CAD model. For example, a CAD system could be used to generate a 3D model which could then be read into a finite element analysis package. A common requirement in engineering design is to produce a drawing which is a schematic layout of components, and which accurately reflects the relative sizes and relationships of these parts. Engineering drawing and draughting is a specialist area with its own set of procedures and practices which have become de facto standards in the engineering industry. Manual methods are now being replaced by computer-assisted methods, and the software that is used to enable these drawings to be produced embodies the functions and capabilities that are required.

CAD applications are very powerful tools that can be used by a designer. The speed and ease with which a drawing can be prepared and modified using a computer have a tremendous advantage over hand-based drawing techniques. CAD-based drawings can be created very easily using the drawing primitives made available by the software (2D/3D lines, arcs, curves, 3D surfaces, text etc.). The drawing can be shared by a number of designers over a computer network who could all be specialists in particular design areas and located at different sites. CAD also allows drawings to be rapidly edited and modified, any number of times.

Drawings can also be linked into databases that could hold material specifications, material costs etc., thereby providing a comprehensive surveillance from design through to manufacturing. In engineering applications, CAD system

specifications can be passed through to numerically controlled (NC) machines to manufacture parts directly.

For creating three-dimensional objects, most CAD systems will provide 3D primitives (such as boundary representations of spheres, cubes, surfaces of revolution and surface patches). They may also provide a solid modelling facility through Constructive Solid Geometry (CSG). Using CSG, basic 3D solids (usually cubes, spheres, wedges, cones, cylinders and tori) more complex composite solids can be created using three basic operations: joining (union) solids, removing (subtraction) solids and finding the common volume (intersection) of solids. With solid modelling, mass properties of solids (*e.g.* moments of inertia, principal moments etc.) can be quickly calculated.

There is virtually no limit to the kind of drawings and models that can be prepared using a CAD system: if it can be created by hand, a CAD system will allow it to be drawn and modelled. Some of the applications where CAD is used are: architectural and interior design, almost all engineering disciplines (*e.g.* electronic, chemical, civil, mechanical, automotive and aerospace), presentation drawings, topographic maps, musical scores, technical illustration, company logos and line drawing for fine art.

Most CAD models can be enhanced for further understanding and presentation by the use of advanced rendering animation techniques (by adding material specifications, light sources and camera motion paths to the model) to produce realistic images and interactive motion through the model. AutoCad is the primary general purpose CAD system in use in engineering, and allows 3D solid modelling as well as 2D/3D draughting. An extension, AEC, for architectural and construction applications, is also available.

## Scientific Visualisation

Scientific Visualisation is concerned with exploring data and information graphically - as a means of gaining insight into and understanding the data. By displaying multi-dimensional data in an easily-understandable form on a 2D screen, it enables insights into 3D and higher dimensional data and data sets that were not formerly possible. The difference between scientific visualisation and presentation graphics is that the latter is primarily concerned with the communication of information and results that are already understood. In scientific visualisation we are seeking to understand the data.

The recent upsurge of interest in scientific visualisation has been brought about principally by the provision of powerful and high-level tools coupled with the availability of powerful workstations, excellent colour graphics, and access to supercomputers if required. This symbiosis provides a powerful and flexible environment for visualising all kinds and quantities of data.

This was once regarded as the exclusive domain of expert system and application programmers who could write the large programmes required, incorporate the algorithms for the graphics, get rid of the bugs in the resulting programme (a non-trivial and time-consuming task), and then process the data. Most of this now comes already available 'off the shelf' - all the users have to do is activate it and plug in their data sets.

Visualisation tools range from lower-level presentation packages, through turnkey graphics packages and libraries, to higher-level application builders. The former are used for simple and modest requirements on small to medium sized data sets and are often used on PCs. The second take larger and more complex

data sets and have a variety of facilities for analysis and presentation of the data in two and three dimensions. The latter enable users to specify their requirements in terms of their application and 'build' a customised system out of pre-defined components supplied by the software. This can usually be done visually on the screen and then the data can be read in, processed and viewed. You can interact with it by changing parameters or altering values.

## Presentation Packages

Many spreadsheet packages for the PC have the facilities for doing elementary 2D graphics, *i.e.* to take a table of X, Y data and show it in visual form on X, Y axes. This enables us to see the overall form of the data much more easily than looking at the table of numbers.

It also enables us to identify any kinks or unusual features and even missing or incorrect data. These facilities are also available in PC graphics packages such as Origin - this is menu-driven and allows users to read in data and select the options required without any programming knowledge.

## Turnkey Graphics Packages and Libraries

Turnkey graphics packages include the Uniras interactive modules Unigraph, Unimap and Gsharp. Unigraph is used for scientific graphing and charting in two and three dimensions. Unimap is used for mapping, contouring and surface drawing. Gsharp is used for both. All these programmes contain advanced facilities for processing data and for the selection of curve and surface requirements. No programming knowledge or experience is required; the user interacts with the modules via menus on the screen.

## Application Builders

These are large systems which contain a wide variety of pre-defined functions and facilities. Building an application consists of visually selecting the iconised functions on the screen, connecting them together by 'pipes' and then activating the network to read in the data and feed it through the interconnected modules. Many state-of-the-art functions for graphics, imaging, rendering, interfacing and displaying are contained in the system. Users can extend the functions available by writing their own modules and adding them to the system.

Examples of visualisation application builders are AVS/Express and IRIS Explorer. AVS/Express is an advanced interactive visualisation environment for scientists and engineers. AVS/Express supports geometric, image and volume datasets. Modules can be dynamically added, connected and deleted. Modules have control panels for interactive control of input parameters in the form of on-screen sliders, file browsers, dials and buttons. AVS/Express has a wide range of data input, filter, mapper and renderer modules. Examples of mappers include isosurfaces of a 3D field, 2D slices of a 3D data volume and 3D meshes from 2D elevation datasets. Multiple visualisation techniques can be selected to suit the problem being studied. User-written programmes or subroutines in FORTRAN or C can be easily converted into AVS/Express modules which can then be integrated into networks using the network editor.

IRIS Explorer provides similar visualisation and analysis functionality. With IRIS Explorer, users view data and create applications by visually connecting software modules into flow chart configurations called module maps. Modules, the building blocks of IRIS Explorer, perform specific programme functions such

as data reading, data analysis, image processing, geometric and volume rendering and many other tasks.

## Desktop Mapping and GIS

Graphs which are maps, or have a cartographic component, are a special case of a 2D graph which requires some special techniques. Many people who are not geographers require this form of graph. Mapping and GIS are two areas that benefit greatly from computer processing of images. It has been estimated that 85% of all the information used by private and public sector organisations contains some sort of geographic element such as street addresses, cities, states, postcodes or even telephone numbers with area codes. Any of these geographic components can be used to help visualise and summarise the data on a map display, enabling you to see patterns and relationships in the data quickly and easily.

MapInfo Professional is a comprehensive desktop mapping tool, available on the PC network, that enables you to create maps, create thematic maps, integrate tabular data onto maps, as well as perform complex geographic analysis such as redistricting and buffering, linking to your remote data, dragging and dropping map objects into your applications, and much more. A GIS (Geographical Information System) is a system for sorting, manipulating, analysing and displaying information with a significant spatial (map-related) content. ArcView and ArcInfo are the two packages available in this category. ArcView is a leading software package for GIS and mapping. It gives you the power to visualise, explore, query and analyse data geographically. ArcView also has three add-on packages - Spatial, Network and 3D Analyst - for more complicated queries. ArcView is available on the NT Cluster Desktop and on the Sun workstations. ArcInfo is an advanced GIS that gives users of geographic data one of the best

geoprocessing systems available at present. It integrates the modern principles of software engineering, database management and cartographic theory. Users are advised that this is a very comprehensive GIS package and requires familiarity with and understanding of GIS concepts. ArcInfo is available on the Sun workstations.

## Subroutine Libraries for Graphics

Uniras and OpenGL are subroutine libraries which are available at Leeds. The former is available on both the Sun and the Silicon Graphics workstations whilst the latter is only available on the Silicon Graphics workstations. Both libraries have at least FORTRAN and C bindings. This means that users have to embed their graphics requirements into their own application programmes and write their own programme code to do this.

In contrast, the interactive modules of Uniras (*e.g.* Gsharp or Unigraph) work entirely off data sets - you do not need to write a programme. If you have a pre-existing applications programme for which you require graphical output, it may be easier just to produce a data file from the execution of this programme and then read this data file into a software package.

It only becomes necessary to write your own programme (or extend your existing programme to include calls to graphics library routines) if you have to embed your graphics requirements to make them an integral part of your application environment, or (in the case of Uniras) you need the more advanced library functions which are not available in the interactive modules.

## Multimedia

There is joint provision for networked colour printing, graphics, slides and video by Information Systems Services and the Print & Copy Bureau.

## On-line Services: Printers, Slide Makers and Scanners

A4 monochrome (black and white) and colour postscript printers are available on the network. Users can send electronic picture and text information for direct output on to paper or OHP foil. Additional printing facilities are provided by Media Services where users can also discuss converting draft electronic information into pre-designed images with design staff.

## Computer-Based Video Production

Data can be displayed or animated in real-time on a high-powered workstation. However, the audience is clearly limited to those who can sit at the workstation. For research seminars, conference presentations, and grant proposals it is often more useful to be able to record the real-time image sequences on video tape and present them to the audience via a video player or video projector. To ensure such presentations are effective, they have to be at a professional standard of presentation. All of us have become unconsciously accustomed to a high quality of presentation from watching programmes on television. Anything less than this immediately looks inferior and can often reflect on the content of what is being presented.