Transformations in Computer Graphics

Dana Burris

TRANSFORMATIONS IN COMPUTER GRAPHICS

TRANSFORMATIONS IN COMPUTER GRAPHICS

Dana Burris

BIBLIOTEX
Digital Library

Transformations in Computer Graphics
by Dana Burris

Ebook ISBN: 9781984664143

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

Contents

Chapter 1 Introduction to Transformation 1

Chapter 2 The Evolution of Computer Graphics 15

Chapter 3 Graphics System Device 31

Chapter 4 Computer Graphics Software 86

Chapter 5 Projection Transformations in Graphics 111

Chapter 6 Computer-Generated Imagery 132

Chapter 7 Workspace Management in Computer Graphics 148

Chapter 8 Engineering Drawing 167

1

Introduction to Transformation

A transformation is the process of mapping points to other locations.

Common transformations are Translation, Scaling and Rotation.

3D ROTATIONS

3D rotation sequence:

- Translate the object so that the rotation axis passes through the coordinate origin
- Rotate the object so that the axis of rotation coincides with one of the coordinate axes
- Perform the specified rotation about the selected axis
- Apply inverse rotations to bring the rotation axis back to its original orientation
- Apply the inverse translation to bring the rotation axis back to its original spatial position.

Process of Clipping

Clipping is the process of determining the portions of a primitive lying within a region called the 'clipping region'. Types of clipping are Point clipping, Line clipping, Polygon clipping.

Purpose of Clipping

It is for preventing:

- Activity in one window from affecting pixels in other windows
- Mathematical overflow and underflow from primitives passing behind the eye point or at great distances (in 3D).

Area Clipping

- Clipping a line segment yields at most one line segment
- Clipping a polygon can yield multiple polygons (However, clipping a convex polygon can yield at most one other polygon).

Rasterization

Rasterization (scan conversion) is to determine which pixels that are inside a primitive specified by a set of vertices.

- To convert continuous geometry, inside viewing region, into discrete pixels
- Fragments have a location (pixel location) and other attributes such colour and texture coordinates that are determined by interpolating values at vertices
- Pixel colours determined later using colour, texture, and other vertex properties.

LINE-DRAWING ALGORITHMS

Bresenham's line algorithm properties:

- Only uses incremental integer calculations
- Can be adapted to display circles and other curves
- Basic idea find next pixel from current one.

AREA FILL BOUNDARY FILL

Start at an inside position and 'paint' the interior, pixel by pixel, with the desired colour until the boundary colour is encountered.

Flood fill

- Start at an inside position and 'repaint' all pixels that are currently set to a certain colour with the desired colour.

TRANSFORMATIONS IN AFFINE TRANSFORMS

Transformations are central in computer graphics. They are used to map from one space to another along the graphics pipeline.

AFFINE TRANSFORMS

A very good source for affine maps in Gerald Farin's book, "Curves and Surfaces for CAGD: A Practical Approach." Some of this introductory material comes from Farin's text.

Let $a,b \in E^3$ be two *points* in three dimensional Euclidean space E^3. Their *difference*;

$$\vec{v} = b - a \in R^3$$

is the *vector from a to b* in the three dimensional linear space R^3. Vectors can be added, subtracted and multiplied by constants.

Points be subtracted, but addition and scalar multiplication of points is not defined. Points can have a vector added to them to form another point:

$$b = a + \vec{v}.$$

This is a *translation* of point *a* along vector $\vec{v}$ to point *b*.

Points determine position; vectors determine direction and magnitude.

For any two points *a* and *b* there is but one vector $\vec{v} = b - a$ from *a* to *b*. However, given there are infinitely many pairs of points that determine $\vec{v}$. Indeed, if $\vec{v} = b - a$ and $\vec{w}$ is any vector, then

$$\vec{v} = (b + \vec{w}) - (a + \vec{w}) = b - a.$$

Now let $a_0, a_1, \ldots, a_n$ be *n* + 1 points in E^3. And let $\alpha_0, \alpha_1, \ldots, \alpha_n$ be *n* + 1 real numbers (*weights*) that sum to 1. We define the *barycentric (or affine) combination* of these points to be

$$a = \sum_{j=0}^{n} \alpha_j a_j, a_j \in E^3, \sum_{j=0}^{n} \alpha_j = 1.$$

This looks like we've invalidated our statement that points can not be added, but the fact that the weights add to one allow us to write the barycentric combination as a point plus the sum of vectors. That is,

$$a = a_0 + \sum_{j=1}^{n} \alpha_j (a_j - a_0).$$

An important special case of barycentric combinations are *convex combinations*. Here we require that the weighs be non-negative (≥ 0) as well as sum to 1.

Note that a weighted sum of points is a vector when the weights add up to zero.

Affine Maps

A map A that maps E^3 into itself is called *affine* if it leaves barycentric combinations invariant. That is, pretend

$$p = \sum_{j=0}^{n} \alpha_j a_j, p, a_j \in E^3, \sum_{j=-}^{n} \alpha_j = 1.$$

is a barycentric combination of points

$$a_0, a_1, \ldots, a_n,$$

and A is an affine map. Then

$$A = \sum_{j=0}^{n} \alpha_j a_j A, pA, a_j A \in \mathbf{E}^3$$

is a barycentric combination of points.

To be more specific, let's think of point p with coordinates $(x\ y\ z)$. An affine map can be represented in the familiar form

$$A = pM + \vec{v},$$

where M is a 3×3 matrix and is a (translation) vector in R^3. Note that we write point-matrix multiplication with the point on the left of the matrix: This seems common practice in the computer graphics literature. Placing the point on the right is more common in mathematical writing. It is easy to change from one form to the other via the *transpose* operation. We will write

$$pM = (x\ \ y\ \ z)\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

This is equivalent to

$$M^T p^T = \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Thus the major difference is: we write points (and vectors) as rows, others write them as columns. We will see that the useful tranformations: translations, scale, rotation, shear, and parallel projection are all affine maps.

Linear Interpolation

A particularly useful barycentric combination is *linear interpolation*. Let $a,b \in E^3$ be two points and let the weights be 1-t and t for some real number (*parameter*) t. Then the points

$$L = L(t) = (1-t)a + tb, t \in R$$

is called the *straight line* through a and b. The line $L(t)$ is a barycentric combination. If we restrict the parameter t to lie between zero and one ($0 \le t \le 1$), $L(t)$ is a convex combination: It is the *line segment from a tob*. Note that there is a direction of travel implied along the line.

MATRICES

Matrices are the basic tool that transform (map) points from E^3 into E^3. A matrix is an $n \times m$ array with n rows and m columns.

You need to know how to perform matrix multiplication. Most of our matrices will be 4 × 4, but they'll start out as 3 × 3.

Pretending that matrix multiplication is a collection inner products is useful since it provides a geometric interpretation. That is, the (i, j) element in the product AB is the inner product of the i-th row of A with the j-th column of B. If you are uncertain about inner products, you'll want to read about them.

Rows and Columns

A row

$$P = [x\,y\,z\,w]$$

should be thought of as a point (using our notational conventions). A column

$$E = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

should be thought of as a plane. The inner (or dot, scalar, matrix) product of them

$$P \cdot E = [x\,y\,z\,w] \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = ax + by + cz + dw$$

is a scalar (real number). If the value is zero, the point lies in the plane.

Scales

Scaling alters the size of an object. Pretend you are given a point $p = (x\,y\,z)$ which is an object vertex, and let $(s_x\,s_y\,s_z)$ be *scale factors* in x y z, respectively.

Then the point can be scaled to a new point by the matrix

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}.$$

In particular,

$$pS = (x\ y\ z)\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = (s_x x \quad s_y y \quad s_z z).$$

To scale (enlarge or shrink) the size of an object, each object vertex is multiplied by the scale matrix S as shown above.

The Fixed Point of a Scale

Note that the origin O = [0 0 0] is unchanged by a scale (it is still the origin). There is always one *fixed point* for any scaling operation. By default the *fixed point* is the origin O = (0 0 0), but we can select an arbitrary fixed point $F = [x_f\ y_f\ z_f]$ by the following three step process, which will be more completely defined below.

1. Translate $F = [x_f\ y_f\ z_f]$ to O = [0 0 0]
2. Scale by $[s_x\ s_y\ s_z]$
3. Translate O = [0 0 0] to $F = [x_f\ y_f\ z_f]$

The Inverse of a Scale

As long as we do not scale by zero, a scale can always be inverted (undone) by the matrix

$$S^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & \frac{1}{s_z} \end{bmatrix}.$$

The product $SS^{-1} = S^{-1}S = I$, the 3 × 3 indentity matrix.

ROTATIONS

Rotations alter the orientation of an object: They are a little more complex than scales. Starting in two dimensional rotations is easiest.

Rotations in Two Dimensions

A rotation moves a point along a circular path centered at the origin (the pivot). It is a simple trigonometry problem to show that rotating $P=[x\,y]$ counter-clockwise by θ radians produces a new point $P'=[x'\,y']$ given by

$$x' = x\cos\theta - y\sin\theta$$
$$y' = y\cos\theta + x\sin\theta$$

For example, pretend P=[1, 1] and $\theta=\pi/2$. Then $P'=[-11]$, which you should agree correctly matches the description.

Of course, we can express the rotation in matrix form

$$[x'\ y'\ 1]=[x\ y\ 1]\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 0\end{bmatrix}$$

The Pivot of a Rotation

By default the *pivot point* is the origin *o* = [0 0 0], but we can arrange for an arbitrary pivot $P=[x_p\,y_p]$ by using a three step process similar to the one for scaling about an arbitrary fixed point described about.

1. Translate $P=[x_p\,y_p]$ to $O=[00]$
2. Rotate by θ
3. Translate $0=[00]$ to $P=[x_p\,y_p]$.

Rotations in Three Dimensions

In three dimensions points are rotated about an *axis*, which is a line in three dimensional space. There are three *principle* axes: the *x*, *y*, and *z* axes. We assume a right-handed coordinate system, with the convention that positive rotation is counter-clockwise.

Rotating $P=[x\,y\,z]$ about the *z*-axis by θ radians produces a new point $P'=[x'\,y'\,z']$ where :

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$
$$z' = z$$

or in matrix notation

$$[x'\,y'\,z'] = [x\,y\,z]\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 \end{bmatrix} = [x\,y\,z]R_z = [x\,y\,z]R_{xy}.$$

The notations R_z and R_{xy} are meant to be mneumonics for "rotate about *z*" and "rotate from *x* towards *y*."

Rotating $P = [x\,y\,z]$ about the *x*-axis by θ radians produces a new point $P' = [x'\,y'\,z']$ where :

$$x' = x$$
$$y' = y\cos\theta - z\sin\theta$$
$$z' = y\sin\theta + z\cos\theta$$

or in matrix notation

$$[x'\,y'\,z'] = [x\,y\,z]\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} = [x\,y\,z]R_x = [x\,y\,z]R_{yz}.$$

Rotating $P = [x\,y\,z]$ about the *y*-axis by θ radians produces a new point $P' = [x'\,y'\,z']$ where

$$x' = x\cos\theta + z\sin\theta$$
$$y' = y$$
$$z' = -x\sin\theta + z\cos\theta$$

or in matrix notation

$$[x'\,y'\,z'] = [x\,y\,z]\begin{bmatrix} \cos\theta & 1 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} = [x\,y\,z]R_y = [x\,y\,z]R_{zx}.$$

Angles of rotation about the principle axes are called *Euler* angles.

Rotation About an Arbitrary Axis

Consider an axis through the origin determined by a unit length direction vector $\vec{D} = \langle d_x \, d_y \, d_z \rangle$, (the completely arbitrary case will be easily handled after translations are introduced). We can arrange to rotate by *theta* radians about this axis using a five step process.

1. Rotate $\vec{D} = \langle d_x \, d_y \, d_z \rangle$ into the *xz* plane, call the result $\vec{D}' = \langle d'_x \, 0 \, d'_z \rangle$
2. Rotate $\vec{D}' = \langle d'_x \, 0 \, d'_z \rangle$ into the *z* axis
3. Rotate about the *z* axis by θ radians
4. Invert the rotation of into $\vec{D}' = \langle d'_x \, 0 \, d'_z \rangle$ the *z* axis
5. Invert the rotation of into $\vec{D} = \langle d_x \, d_y \, d_z \rangle$ $\vec{D}' = \langle d'_x \, 0 \, d'_z \rangle$.

This is messy and error-prone when using hand calculation, however, if you carry it out, the result is:

$$R = \begin{bmatrix} d_x^2 + \cos\theta\left(1 - d_x^2\right) & d_x d_y (1 - \cos\theta) - d_z \sin\theta & d_z d_x (1 - \cos\theta) + d_y \sin\theta \\ d_x d_y (1 - \cos\theta) + d_z \sin\theta & d_y^2 + \cos\theta\left(1 - d_y^2\right) & d_y d_z (1 - \cos\theta) - d_x \sin\theta \\ d_z d_x (1 - \cos\theta) - d_y \sin\theta & d_y d_z (1 - \cos\theta) + d_x \sin\theta & d_z^2 + \cos\theta\left(1 - d_z^2\right) \end{bmatrix}. \quad (1)$$

You should verify that this matrix reduces to rotations about *z*, *x*, and *y* for appropriate choices of the direction vector $\vec{D}$. Below, we'll see that there are better ways to derive this matrix.

The Inverse of a Rotation

The inverse of a rotation by θ radians can be created by $-\theta$ rotating by radians, but this is not the best way to view it. Consider the trigonometic identities:

$$\cos(-\theta) = \cos\theta$$
$$\sin(-\theta) = -\sin\theta$$

If you plug these into the arbitrary rotation from equation (1), you'll see that the inverse of *R* is the *transpose* of *R*. This is an important observation.

TRANSLATIONS

Translations change the position of an object. A pure (three dimensional) translation can not be implemented using a 3× matrix: It is an *affine* map. We must alter our notion of a point P = [x y z] to accommodate translations. A three dimensional point will be embedded in three dimensional *homogeneous* space and represented as a 4-tuple P_h = [x y z w]. For now, the homogeneous coordinate w will have the fixed value 1. This allows us to implement translations using 4 × 4 matrices, in particular, the matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

translated the point $P_h = [x\,y\,z\,1]$ into the point

$$P'_h = [x+t_x\;y+t_y\;z+t_z\;1].$$

The Inverse of a Translation

To undo a translation by t_x, t_y, t_z use the matrix

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & & \\ 0 & 1 & 0\ldots\ldots & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 & & \end{bmatrix}.$$

We can now complete scaling about an arbitrary fixed point and rotation about an arbitrary pivot. To scale about $F = [x_f\; y_f\; z_f]$ use the composition of matrices

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_f & -y_f & -z_f & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_f & y_f & z_f & 1 \end{bmatrix}$$

which when multiplied out yields

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ x_f(1-s_x) & y_f(1-s_y) & z_f(1-s_z) & 1 \end{bmatrix}.$$

So a scaled point $[x\,y\,z\,1]$ becomes

$$[x'\,y'\,z'\,1] = [x\,y\,z\,1]\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ x_f(1-s_x) & y_f(1-s_y) & z_f(1-s_z) & 1 \end{bmatrix} \qquad (2)$$

$$= [xs_x + x_f(1-s_x) \quad ys_y + y_f(1-s_y) \quad zs_z + z_f(1-s_z)1] \qquad (3)$$

In a similar manner you can determine that rotation about a pivot $R = [x_r\ y_r]$ results in

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (y - y_r)\cos\theta + (x - x_r)\sin\theta$$

EFFICIENCY OF MATRIX MULTIPLICATION

Now is a good time to mention the fact that it is more efficient, in general, to form one composite transform than to pass a sequence of points through one transform, then another, and another, and so on.

Multiplying one point (a 4-tuple) by a transformation (4 × 4 matrix) costs 16 multiplies and 12 additions. Therefore, transforming an object with *n* vertices by one transform costs 16*n* multiplies and 12*n* additions.

On the other hand, multiplying two 4 × 4 matrices costs 64 multiplies and 48 additions. So compositing *m* 4 × 4 matrices together costs 64(*m*-1) multiplies and 48(*m*-1) additions.

So consider the alternatives:

- Multiply *n* vertices through a sequence of *m* transformations at a cost of 16*n* multiplies and

12*n* adds per transform. The total cost will be 16 nm multiplies, 12nm additions.

- Form on composite matrix and pass *n* vertices through it. The total cost will be 64 (m – 1) + 16n multiplies, 48(m – 1) + 12n additions.

TRANSFORMATIONS

Objects defined in model space can be scaled, translated, and rotated into world space and then viewed from any position. The map from model to world to view is most often concatenated into one single tranform so we map from a model directly into view space without every stopping in the world.

Next we map objects from view space into perspective space. This involves projections: either *parallel* or *perspective*. Note that we must stop in view space to compute the illumination that lights bring to our view.

From perspective to clip space and from clip space through normalized space to device space are fairly straight forward scales and translations — we just need to be careful not to introduce distortions by our scaling of the objects.

Non-linear Transforms

Here we want to describe perspective transforms. They are non-linear, that is, lines do not map into lines.

2

The Evolution of Computer Graphics

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

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

1960s

- 1963: Ivan Sutherland presented his Ph.D. dissertation, an interactive design on a vector graphics display monitor with a light pen input device called *Sketchpad*. This instance is often credited as the event that marks the beginning of computer graphics.
- Jack Bresenham develops a system of drawing lines and circles on a raster device, and Steve Coons introduces parametric surfaces and computer-aided geometric design concepts.
- Arthur Appel at IBM introduces hidden surface and shadow algorithms.
- The fast Fourier transform was discovered by J. W. Cooley and John Tukey, allowing computer engineers to better understand signals to develop antialiasing techniques.
- Doug Englebart develops the mouse at Xerox PARC.
- Evans & Sutherland Corps. and GE start building flight simulators with raster graphics.

1970s

- Rendering and a reflection model were discovered and developed by H. Gouraud and Bui Tuong Phong at the University of Utah.
- Xerox PARC develops a “paint programme.”
- Edward Catmull introduces parametric patch rendering, the z-buffer algorithm and texture mapping.

- Turner Whitted develops recursive ray tracing that would become the standard for photorealism.
- Apple I and Apple II computers were the first commercially successful options for personal computing.
- Arcade games Pong and Pac Man become popular.

1980s

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

1990s

- Shaded raster graphics appear in films.
- Computers have 24-bit raster display and hardware support for Gouraud shading.
- Laser printers and single-frame video recorders become standard.
- Mosaic, the first graphical internet browser is created.
- Dynamical systems that allowed programmers to animate collisions, friction and cause and effects are introduced.
- Handheld computers are invented at Hewlett-Packard and zip drives invented at Iomega.
- Nintendo 64 game console arrives on the market.

- Linux and open source software emerges.
- Pixar is first studio to fully embrace an entirely computer-generated film with *Toy Story.*

2000s

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

THE COMPUTER GRAPHICS PIPELINE

The process that goes into the production of a fully realised 3D movie character or environment is known by industry professionals as the "computer graphics pipeline." Even though the process is quite complex from a technical standpoint, it's actually very easy to understand when illustrated sequentially. Think of your favourite 3D movie character.

It could be Wall-E or Buzz Lightyear, or maybe you were a fan of Po in *Kung Fu Panda*. Even though these three characters look very different, their basic production sequence is the same.

In order to take an animated movie character from an idea or storyboard drawing to a fully polished 3D rendering, the character passes through six major phases:

PRE-PRODUCTION

In pre-production, the overall look of a character or environment is conceived. At the end of pre-production, finalized design sheets will be sent to the modeling team to be developed.

- Every Idea Counts: Dozens, or even hundreds of drawings & paintings are created and reviewed on a daily basis by the director, producers, and art leads.
- Colour Palette: A character's colour scheme, or palette, is developed in this phase, but usually not finalized until later in the process.
- Concept Artists may work with digital sculptors to produce preliminary digital mock-ups for promising designs.
- Character Details are finalized, and special challenges (like fur and cloth) are sent off to research and development.

3D MODELLING

With the look of the character finalized, the project is now passed into the hands of 3D modellers. The job of a modeller is to take a two dimensional piece of concept art and translate it into a 3D model that can be given to animators later on down the road.

In today's production pipelines, there are two major techniques in the modeller's toolset: polygonal modelling & digital sculpting.

- Each has its own unique strengths and weaknesses, and despite being vastly different, the two approaches are quite complementary.

- Sculpting lends itself more to organic (character) models, while polygonal modelling is more suited for mechanical/architectural models.

The subject of 3D modelling is far too extensive to cover in three or four bullet points, but its something we'll continue covering in depth in both the blog, and in the Maya Training series.

SHADING AND TEXTURING

The next step in the visual effects pipeline is known as shading and texturing. In this phase, materials, textures, and colours are added to the 3D model.

- Every component of the model receives a different shader-material to give it an appropriate look.
- Realistic materials: If the object is made of plastic, it will be given a reflective, glossy shader. If it is made of glass, the material will be partially transparent and refract light like real-world glass.
- Textures and colours are added by either projecting a two dimensional image onto the model, or by painting directly on the surface of the model as if it were a canvas. This is accomplished with special software (like ZBrush) and a graphics tablet.

LIGHTING

In order for 3D scenes to come to life, digital lights must be placed in the scene to illuminate models, exactly as lighting rigs on a movie set would illuminate actors and actresses. This is probably the second most technical phase of the production pipeline (after rendering), but there's still a good deal of artistry involved.

- Proper lighting must be realistic enough to be believable, but dramatic enough to convey the director's intended mood.
- Mood Matters: Believe it or not, lighting specialists have as much, or even more control than the texture painters when it comes to a shot's colour scheme, mood, and overall atmosphere.
- Back-and-Forth: There is a great amount of communication between lighting and texture artists. The two departments work closely together to ensure that materials and lights fit together properly, and that shadows and reflections look as convincing as possible.

ANIMATION

Animation, as most of you already know, is the production phase where artists breathe life and motion into their characters.

Animation technique for 3D films is quite different than traditional hand drawn animation, sharing much more common ground with stop-motion techniques:

- Rigged for Motion: 3D characters are controlled by means of a virtual skeleton or "rig" that allows an animator to control the model's arms, legs, facial expressions, and posture.
- Pose-to-Pose: Animation is typically completed pose-to-pose. In other words, an animator will set a "key-frame" for both the starting and finishing pose of an action, and then tweak everything in between so that the motion is fluid and properly timed.

Jump over to our computer animation companion site for extensive coverage of the topic.

RENDERING AND POST-PRODUCTION

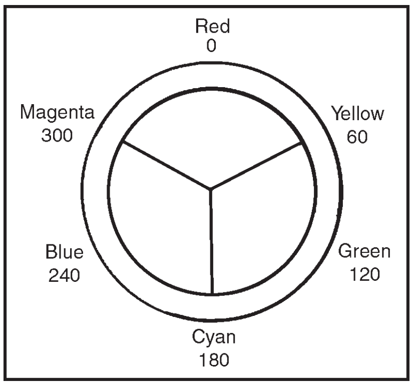
The final production phase for a 3D scene is known as rendering, which essentially refers to the translation of a 3D scene to a finalized two dimensional image. Rendering is quite technical, so we won't spend too much time on it here. In the rendering phase, all the computations that cannot be done by your computer in real-time must be performed.

This includes, but is hardly limited to the following:

- Finalizing Lighting: Shadows and reflections must be computed.
- Special Effects: This is typically when effects like depth-of-field blurring, fog, smoke, and explosions would be integrated into the scene.
- Post-processing: If brightness, colour, or contrast needs to be tweaked, these changes would be completed in an image manipulation software following render time.

COMPARISON OF COLOUR MODELS IN COMPUTER GRAPHICS

COLOUR BASICS



PRIMARY COLORS AND HUE

First, “colour” refers to the human brain’s subjective interpretation of combinations of a narrow band of wavelengths of light. For this reason, the definition of “colour” is not based on a strict set of physical phenomena. Therefore, even basic concepts like “primary colours” are not clearly defined.

For example, traditional “Painter’s Colours” use red, blue, and yellow as the primary colours, “Printer’s Colours” use cyan, yellow, and magenta, and “Light Colours” use red, green, and blue.· “Light colours”, more formally known as additive colours, are formed by combining red, green, and blue light. This section refers to additive colours and refers to red, green, and blue as the primary colours.

Hue is a term describing a pure colour, that is, a colour not modified by tinting or shading. In additive colours, hues are formed by combining two primary colours. When two primary colours are combined in equal intensities, the result is a “secondary colour”.

Colour Wheel

A colour wheel is a tool that provides a visual representation of the relationships between all possible hues. The primary colours are arranged around a circle at equal (120 degree) intervals. (Warning: Colour wheels frequently depict “Painter’s Colours” primary colours, which leads to a different set of hues than additive colours.) The illustration shows a simple colour wheel based on the additive colours.

Note that the position (top, right) of the starting colour, typically red, is arbitrary, as is the order of green and blue (clockwise, counter-clockwise). The illustration also shows the secondary colours, yellow, cyan, and magenta, located half-way between (60 degrees) the primary colours.

Complementary Colour

The complement of a hue is the hue that is opposite it (180 degrees) on the colour wheel. Using additive colours, mixing a hue and its complement in equal amounts produces white.

Tints, Shades

An illustration involving three projectors pointing to the same spot on a screen. Each projector is capable of generating one hue. The "intensities" of each projector are "matched" and can be equally adjusted from zero to full. (Note: "Intensity" is used here in the same sense as the RGB colour model. The subject of matching, or "gamma correction", is beyond the level of this article.)

A *shade* is produced by "dimming" a hue. Painters refer to this as "adding black". In our illustration, one project is set to full intensity, a second is set to some intensity between zero and full, and third is set to zero. "Dimming" is accomplished by decreasing each projector's intensity setting to the same fraction of its start setting.

In the shade example, with any fully shaded hue, that all three projectors are set to zero intensity, resulting in black. A *tint* is produced by "lightening" a hue. Painters refer to this as "adding white". In our illustration, one project is set to full intensity, a second is set to some intensity between zero and full, and third is set to zero. "Lightening" is accomplished by increasing each projector's intensity setting by the same fraction from its start setting to full.

In the tinting example, note that the third projector is now contributing. When the hue is fully lightened, all three projectors are each at full intensity, and the result is white.

Note an attribute of the total intensity in the additive model. If full intensity for one projector is 1, then a primary colour has a combined intensity of 1. A secondary colour has a total intensity of 2. White has a total intensity of 3. Tinting, or "adding white", increases the total intensity of the hue. While this is simply a fact, the HSL model will take this fact into account in its design.

Tones

Tone is a general term, typically used by painters, to refer to the effects of reducing the "colourfulness" of a hue. ; painters refer to it as "adding gray". Note that gray is not a colour or even a single concept but refers to all the range of values between black and white where all three primary colours are equally represented. The general term is provided as more specific terms have conflicting definitions in different colour models. Thus, shading takes a hue towards black, tinting takes a hue towards white, and tones cover the range between.

Choosing a Colour Model

No one colour model is necessarily "better" than another. Typically, the choice of a colour model is dictated by external factors, such as a graphics tool or the need to specify colours according to the CSS2 or CSS3 standard. The models function, centred on the concepts of hue, shade, tint, and tone.

RGB

The RGB model's approach to colours is important because:

- It directly reflects the physical properties of "Truecolour" displays.
- As of 2011, most graphics tools support it, even if they prefer another colour model.
- It is the only means of specifying a specific colour in the CSS2 standard for Web pages.

In the model, a colour is described by specifying the intensity levels of the colours red, green, and blue. The typical range of intensity values for each colour, 0 - 255, is based on taking a binary number with 32 bits and breaking it up into four bytes of 8 bits each. 8 bits can hold a value from 0 to 255. The fourth byte is used to specify the "alpha", or the opacity, of the colour. Opacity comes into play when layers with different colours are stacked. If the colour in the top layer is less than fully opaque (alpha < 255), the colour from underlying layers "shows through".

In the RGB model, hues are represented by specifying one colour as full intensity (255), a second colour with a variable intensity, and the third colour with no intensity (0).

The following provides some examples using red as the full-intensity and green as the partial-intensity colours; blue is always zero:

Red	Green	Result
255	0	Red (255, 0, 0)
255	128	Orange (255, 128, 0)
255	255	Yellow (255, 255, 0)

Shades are created by multiplying the intensity of each primary colour by 1 minus the shade factor, in the range 0 to 1.

A shade factor of 0 does nothing to the hue, a shade factor of 1 produces black:

```
new intensity = current intensity * (1 - shade factor)
```

The following provides examples using orange:

0	.25	.5	.75	1.0
(255, 128, 0)	(192, 96, 0)	(128, 64, 0)	(64, 32, 0)	(0, 0, 0)

Tints are created by modifying each primary colour as follows: the intensity is increased so that the *difference* between the intensity and full intensity (255) is *decreased* by the tint factor, in the range 0 to 1.

A tint factor of 0 does nothing, a tint factor of 1 produces white:

```
new intensity = current intensity + (255 - current intensity)
* tint factor
```

The following provides examples using orange:

0	.25	.5	.75	1.0
(255, 128, 0)	(255, 160, 64)	(255, 192, 128)	(255, 224, 192)	(255, 255, 255)

Tones are created by applying both a shade and a tint. The order in which the two operations are performed does not matter, with the following restriction: when a tint operation is performed on a shade, the intensity of the dominant colour becomes the “full intensity”; that is, the intensity value of the dominant colour must be used in place of 255.

The following provides examples using orange:

	0	.25	.5	.75	1.0
0	(255, 128, 0)	(192, 96, 0)	(128, 64, 0)	(64, 32, 0)	(0, 0, 0)
.25	(255, 160, 64)	(192, 144, 96)	(128, 80, 32)	(64, 40, 16)	(0, 0, 0)
.5	(255, 192, 128)	(192, 144, 96)	(128, 96, 64)	(64, 48, 32)	(0, 0, 0)
.75	(255, 240, 192)	(192, 168, 144)	(128, 112, 96)	(64, 56, 48)	(0, 0, 0)
1.0	(255, 255, 255)	(192, 192, 192)	(128, 128, 128)	(64, 64, 64)	(0, 0, 0)

HSV

The HSV, or HSB, model describes colours in terms of hue, saturation, and value (brightness). Note that the range of values for each attribute is arbitrarily defined by various tools or standards. Be sure to determine the value ranges before attempting to interpret a value.

Hue corresponds directly to the concept of hue in the Colour Basics section. The advantages of using hue are

- The angular relationship between tones around the colour circle is easily identified.
- Shades, tints, and tones can be generated easily without affecting the hue.

Saturation corresponds directly to the concept of tint in the Colour Basics section, except that full saturation produces no tint, while zero saturation produces white, a shade of gray, or black.

Value corresponds directly to the concept of intensity in the Colour Basics section.

- Pure colours are produced by specifying a hue with full saturation and value.
- Shades are produced by specifying a hue with full saturation and less than full value.
- Tints are produced by specifying a hue with less than full saturation and full value.
- Tones are produced by specifying a hue and both less than full saturation and value.
- White is produced by specifying zero saturation and full value, regardless of hue.
- Black is produced by specifying zero value, regardless of hue or saturation.
- Shades of gray are produced by specifying zero saturation and between zero and full value.

The advantage of HSV is that each of its attributes corresponds directly to the basic colour concepts, which makes it conceptually simple.

The perceived disadvantage of HSV is that the saturation attribute corresponds to tinting, so desaturated colours have increasing total intensity. For this reason, the CSS3 standard plans to support RGB and HSL but not HSV.

HSL

The HSL model describes colours in terms of hue, saturation, and lightness (also called luminance). (Note: the definition of saturation in HSL is substantially different from HSV, and lightness is not intensity.)

- The transition from black to a hue to white is symmetric and is controlled solely by increasing lightness.
- Decreasing saturation transitions to a shade of gray dependent on the lightness, thus keeping the overall intensity relatively constant.

The properties have led to the wide use of HSL, in particular, in the CSS3 colour model.

As in HSV, hue corresponds directly to the concept of hue in the Colour Basics section. The advantages of using hue are

- The angular relationship between tones around the colour circle is easily identified.
- Shades, tints, and tones can be generated easily without affecting the hue.

Lightness combines the concepts of shading and tinting from the Colour Basics section. Assuming full saturation, lightness is neutral at the midpoint value, for example 50%, and the hue displays unaltered. The midpoint, it has the effect of shading. Zero lightness produces black. As the value increases above 50%, it has the effect of tinting, and full lightness produces white.

At zero saturation, lightness controls the resulting shade of gray. A value of zero still produces black, and full lightness still produces white. The midpoint value results in the "middle" shade of gray, with an RGB value of.

Saturation, or the lack of it, produces tones of the reference hue that converge on the zero-saturation shade of gray, which is determined by the lightness. The following examples uses the hues red, orange, and yellow at midpoint lightness with decreasing saturation. The resulting RGB value and the total intensity is shown.

1.0	**.75**	**.5**	**.25**	**0**
(255, 0, 0), 256	(224, 32, 32), 288	(192, 64, 64), 320	(160, 96, 96), 352	(128, 128, 128), 384
(255, 128, 0), 384	(224, 128, 32), 384	(192, 128, 64), 384	(160, 128, 96), 384	(128, 128, 128), 384
(255, 255, 0), 512	(224, 224, 32), 480	(192, 192, 64), 448	(160, 160, 96), 416	(128, 128, 128), 384

Note that the physical nature of additive colour prevents the scheme from working exactly except for hues halfway between the primary and secondary colours. However, the total intensity of the tones resulting from decreasing saturation are much closer than tinting alone, as in HSV.

3

Graphics System Device

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.

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.

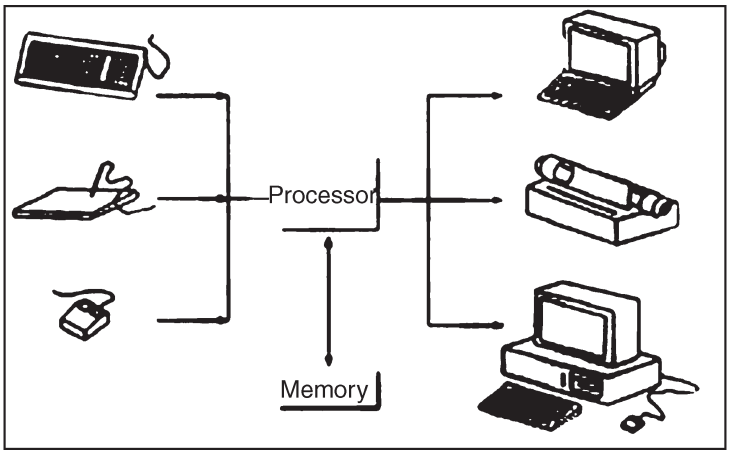


Fig. The Graphic System

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.

SOFTWARE THAT CREATES GRAPHIC ORGANIZERS

You probably have a lot of software on programmes on the computer that you use that can create Graphic Organizers.

These include the Office Productivity Suite applications (Word Processing, Spreadsheet, and Presentation Programs). If you use Microsoft(TM) Windows, you probably have a low end drawing programme called, “Paint.” All these programmes can create Graphics Organizers.

If you do not have this Office Suite, we have included an Open Source (Free) Office Suite called “Open Office.” This programme is free to use and to share with others. Open Office applications also can save your Graphic Organizer files in the PDF file format. If you save Graphic Organizer files in the PDF format, you can share them with everyone, and the file will print exactly as you created it.

OPEN OFFICE (OPEN SOURCE)

The catch with sharing Graphic Organizers that are saved in the PDF file format is that you cannot make changes to them without expensive software. However, the viewer programme that opens and prints the files is free and most people who connect to the Internet have the Acrobat Reader programme. We have included the latest version to save you from having to download it from the Internet.

SOFTWARE THAT IS A GRAPHIC ORGANIZER

There are a lot of software products on the market that are Graphic Organizers.

The majority of these products call themselves, “Mind Mapping” software.

The competition in this market is very strong, so all vendors seem to offer free trials of their products. It is possible that a teacher could use a different trial version of these products each month, and never purchase a copy.

The only catch is that the formats of the various products are proprietary. This means that you cannot open the files you create with another company's product.

Inspiration(TM) and Kidspiration(TM) are products that fall into this category, and these products are often available in school districts. Inspiration and Kidspiration are easy to use, but low-end products.

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.

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.

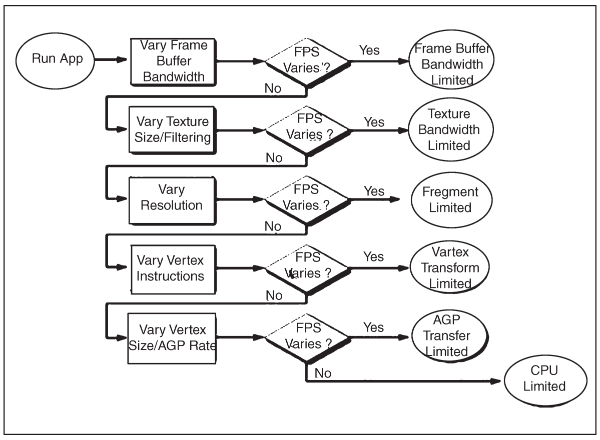


Fig. Bottleneck Flowchart

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.

RASTERIZATION

Rasterization is the process of converting a vertex representation to a pixel representation; rasterization is also called *scan conversion*. Included in this definition are geometric objects such as circles where you are given a centre and radius.

In these notes we will cover:

- The digital differential analyzer (DDA) which introduces the basic concepts for rasterization.
- Bresenham's algorithm which improves on the DDA.
- The scan line fill for polygons.
- And, time permitting, flood and boundary fill algorithms.

Scan conversion algorithms use *incremental* methods that exploit *coherence*. An incremental method computes a new value quickly from an old value, rather than computing the new value from scratch, which can often be slow. Coherence in space or time is the term used to denote that nearby objects (*e.g.*, pixels) have qualities similar to the current object.

THE DDA ALGORITHM

Let $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$ be two endpoints of a line segment. We will assume that these points are in device space so that the coordinates x_0, y_0, x_1, y_1 are integers. The point-intercept form of the equation of the line from p_0 to p_1 is

$$y = mx + b$$

where the slope is

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

and the y intercept is

$$b = y_1 - mx_1,$$

although it is not necessary to compute the y intercept.

Notice that if x is incremented to $x + 1$, then y changes to $y + m$. Similarly, if y is incremented to $y + 1$, then x changes to $x + \frac{1}{m}$. Of course, the slope (or its reciprocal) often won't be integers. Let's consider the two cases where these increments occur.

IMAGE RASTERIZATION

Understanding Vectors and Bitmaps

Images in Flash come in two flavours: vector images and bitmap images. Each format has advantages and drawbacks. You'll learn about each format, the pros and cons of each format, and when it is appropriate to use each format.

Vector Graphic

A vector graphic is a shape drawn with a series of points and lines connecting the points. For example, a square consists of four corner points with lines connecting each point. A circle contains the same four points, but the lines between them are curved instead of straight. A vector shape has a fill colour and an outline (stroke) colour. Usually a vector graphic is composed of dozens or more vector shapes that overlap to form a picture.

Vector graphics have the advantage of being light-weight and scalable. Under the hood, vector graphics are defined entirely by the math describing their points and lines, so they are not composed of resolution. As a result, vector graphics are light-weight and can be scaled up and down in size without losing their quality. And vector graphics can be edited and changed at any time.

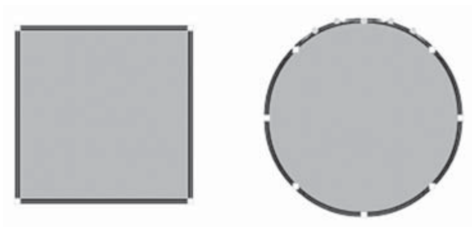


Fig. The Points and Lines that Define a Square and a Circle Vector Image

Vector graphics have the disadvantage of being processor-intensive in some situations. Vector graphics are rendered by the CPU at runtime and have to be re-rendered whenever a change in the graphic occurs. For example, when a vector graphic is used in a tween animation, or if the vector graphic on Stage is overlapped by a tween animation, the shape is rendered again in each frame to display the changes on the screen. Most modern desktop and laptop computers don't have any problem rendering complex groups of vector graphics, but be aware that mobile devices may display visible performance issues.

Use vectors whenever you need to create scalable graphics, work with editable text and shapes, or when flexible content is required for animations.

Tip: The drawing tools in Flash Professional natively draw vector graphics, but in many cases, it is a best practice to publish the graphics as bitmaps in order to improve performance.

Bitmap Graphic

A bitmap graphic is an image composed of a grid of dots called pixels. Each pixel contains a colour. Collectively, the grid of coloured pixels forms the image. The number of pixels per inch defines the resolution of an image. The common screen resolution for computer monitors is 72 dpi (dots per inch).

Bitmap graphics have the advantage of displaying highly detailed photographic content without the use of CPU rendering. Once the bitmap has downloaded to the display, it does not need to be rendered again.

Bitmap graphics have the disadvantage of producing larger file sizes. The resolution, number of colours, compression scheme, and dimensions of the bitmap all contribute to the file size of the image. Also, since bitmaps are defined in resolution, they cannot be scaled to larger sizes without incurring a loss of quality. Bitmaps are not editable in Flash; you can use a tool such as Photoshop or Fireworks to edit your bitmap images prior to importing them in Flash.

Use bitmaps for backgrounds and static images that don't need to be edited or scaled. Also, for performance reasons and portability to mobile devices, consider using bitmaps or rasterization techniques whenever possible.

Typical Performance Issues and their Workarounds

Depending on the complexity of your Flash movie, you may find that some animations and page transitions seem sluggish or fragmented. This scenario can occur in projects when too many overlapping vector images are redrawn in every frame. The result can be inconsistent frame rates and intermittent pauses in the vector rendering.

Here are a few things to consider that may improve a project's performance:

1. Use bitmap images for background graphics. Animations often appear on top of larger background graphics. Using bitmaps for the backgrounds will help reduce the resources required to render the graphics and minimize CPU usage.
2. Large areas of animation are more likely produce performance issues. Try to limit the area an animation plays to the smallest size possible.
3. Animations and complex graphics that use transparency (alpha) require more CPU usage than opaque graphics.

4. Animations and complex graphics that use filters and blend modes require more CPU usage than graphics that do not. Apply filters and blend modes sparingly. Also, try to use lower quality settings when working with filters.
5. Avoid displaying animations that constantly idle, if possible.
6. Use the rasterization features in Flash to convert static graphics into bitmaps at author-time or runtime.

Benefits of Rasterizing Vector Graphics

Flash Professional and ActionScript 3 provide a handful of options for converting vector graphics to bitmaps. The benefit of using these features is that you can often avoid the performance pitfalls described previously while leaving your artwork editable at author-time. The result can produce projects that require less CPU usage, smoother animation performance and frame rate playback, and improved performance—especially for applications being ported to mobile devices.

GRAPHICAL I/O DEVICES

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

INPUT DEVICES

Various hardware devices have been developed to enable the user to interact in the more natural manner. These devices can be separated into two classes. They are Locators and Selectors.

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

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

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

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

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

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

Tablet: A Tablet composed of a flat surface and a pen like stylus or window like tablet cursor. The tablet is able to sense the position of the stylus or tablet cursor on the surface. A number of different physical principles have been employed for the sensing of the stylus. Most do not require actual contact between the stylus and the tablet surface, so that a drawing or blueprint might be placed upon the surface and the stylus used to trace it. A feedback mechanism on the screen is not as necessary for a graphics tablet as it is for a joystick because the user can look at the tablet to see what position he is indicating. If tablet entries are to be coordinated with items already on the screen, then some form of feedback, such as a screen cursor, is useful.

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

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

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

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

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

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

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

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

Image Scanners: Drawings, graphs, colour and black and white photos or text can be stored for computer processing with an image scanner by passing an optical scanning mechanism over the information to be stored. The gradations of gray scale or colour are then recorded and stored in an array. Once we have the internal representation of a picture, we can apply transformations to rotate, scale or crop the picture to a particular screen area. We can also apply various image processing methods to modify the array representation of the picture. For scanned text input, various editing operations can be performed on the stored documents. Some scanners are able to scan either graphical representations or text and they come in a variety of sizes and capabilities.

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

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

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

Electrical Touch Panels: It is constructed with two transparent plates separated by a small distance. One of the plates is coated with a conducting material and the other plate is coated with a resistive material. When the outer plate is touched, it is forced into contact with the inner plate. This contact creates a voltage drop across the resistive plate that is converted to the coordinate values of the selected screen position.

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

Voice Systems

Speech recognizers are used in some graphics workstations as input devices to accept voice command. The voice system input can be used to initiate graphics operations or to enter data. These systems operate by matching an input against a predefined dictionary of words and phrases.

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

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

OUTPUT DEVICES

Printers

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

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

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

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

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

Plotters

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

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

Display Devices

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

The CRT

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

Electron Gun

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

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

- It must be accurately focused so that it produces a sharp spot of light where it strikes the phosphor.
- It must have high velocity, since, the brightness of the image depends on the velocity of the electron beam.
- Means must be provided to control the flow of electrons so that the intensity of the trace of the beam can be controlled.

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

The Deflection System

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

Phosphors

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

In attempts to improve performance in one or another of these respects, many different phosphors have been produced, using various compounds of calcium, cadmium and zinc together with traces of rare earth elements. These phosphors are identified by a numbering system like P1, P4, P7 *etc.*

Raster-scan Displays

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

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

Intensity range for pixel positions depends on the capability of the raster system. In a simple black and white system, each screen point is either on or off, so only one bit per pixel is needed to control the intensity of screen positions. Here 1 indicates that the electron beam is to be turned on at that position, and value 0 indicates that the electron beam intensity is to be off. Additional bits are needed when colour and intensity variations can be displayed. Up to 24 bits per pixel are included in high quality systems, which can require several megabytes of storage for the frame buffer, depending on the resolution of 1024 by 1024 requires 3 megabytes of storage for the frame buffer. On a black and white system with one bit per pixel, the frame buffer is commonly called a bitmap. For systems with multiple bits per pixel, the frame buffer is often referred to as a pixmap.

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

Random-scan Displays

When operated as a random-scan display unit, a CRT has the electron beam directed only to the parts of the screen where a picture is to be drawn. Random-scan monitors draw a picture one line at a time and for this reason are also referred to as vector displays. A pen plotter operates in a similar way and is an example of a random-scan, hard copy device. Refresh rate on a random-scan system depends on the number of lines to be displayed. Picture definition is now stored as a set of line drawing commands in an area of memory referred to as the refresh display file. Sometime the refresh display file is called the display list or display Programme or refresh buffer. To display a specified picture, the system cycles through the set of commands in the display file, drawing each component line in turn. After all line drawing commands have been processed, the system cycles back to the first line command in the list.

Colour CRT Monitors

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

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

Shadow-mask methods are commonly used in raster-scan systems because they produce a much wider range of colours than the beam-penetration method. A shadow-mask CRT has three phosphor colour dots at each pixel position. One phosphor dot emits a red light, another emits a green light, and the third emits a blue light. This type of CRT has three electron guns, one for each colour dot and a shadow-mask grid just behind the phosphor-coated screen. We obtain colour variations in a shadow-mask CRT by varying the intensity levels of the three electron beams. By turning off the red and green guns, we get only the colour coming from the blue phosphor. A white area is the result of activating all three dots with equal intensity.

Direct-View StorageTubes

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

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

Flat-Panel Displays

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

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

LCD Technology

Borrowing technology from laptop manufacturers, some companies provide LCD (Liquid Crystal Display) displays. LCDs have low glare flat screens and low power requirements. The colour quality of an active matrix LCD panel actually exceeds that of most CRT displays. At this point, how ever, LCD screens usually are more limited in resolution than typical CRTs and are much more expensive. There are three basic LCD choices.

They are.......

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

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

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

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

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

Raster-scan Systems

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

Video Controller

Frame buffer locations, and the corresponding screen positions, are referenced in Cartesian co-ordinates. For many graphics monitors, the co-ordinate origin is defined at the lower left screen corner. The screen surface is then represented as the first quadrant of a two-dimensional system, with positive x values increasing to the right and positive y values increasing from bottom to top. Scan lines are then labelled from y_{max} at the top of the screen to 0 at the bottom. Along each scan line, screen pixel positions are labelled from 0 to x_{max}. Two registers are used to store the co-ordinates of the screen pixels. Initially, the x register is set to 0 and the y register is set to ymax. The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam. Then the x register is incremented by 1, and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the scan line. After the last pixel on the top scan line has been processed, the x register is reset to 0 and the y register is decremented by 1. Pixels along this scan line are then processed in turn, and the procedure is repeated for each successive scan line. After cycling through all pixels along the bottom scan line ($y = 0$), the video controller resets the registers to the first pixel position on the top scan line and the refresh process starts over.

Raster-scan Display Processor

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

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

In an effort to reduce memory requirements in raster systems, methods have been devised for organizing the frame buffer as a linked list and encoding the intensity information. One way to do this is to store each scan line as a set of integer pairs. One number of each pair indicates an intensity value, and the second number specifies the number of adjacent pixels on the scan line that are to have that intensity. This technique called run-length encoding. A similar approach can be taken when pixel intensities change linearly. Another approach is to encode the raster as a set of rectangular areas (cell encoding).

Random-scan Systems

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

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

4

Computer Graphics Software

The graphics software is the collection of programmes written to make it convenient for a user to operate the computer graphics system. It includes programmes to generate images on the CRT screen, to manipulate the images, and to accomplish various types of interaction between the user and the system. In addition to the graphics software, there may be additional programmes for implementing certain specialized functions related to CAD/ CAM. These include design analysis programmes (*e.g.*, finite-element analysis and kinematic simulation) and manufacturing planning programmes (*e.g.*, automated process planning and numerical control part programming).

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

The software configuration of a graphics system.

The graphics software can be divided into three modules:

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

Functions of a Graphics Package

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

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

USING THE ACM.GRAPHICS PACKAGE

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

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

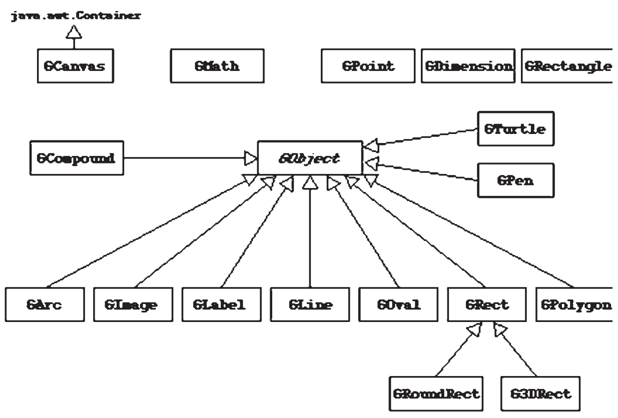


Fig. Class Diagram for the acm.graphics Package

THE ACM.GRAPHICS MODEL

When you create a picture using the acm.graphics package, you do so by arranging graphical objects at various positions on a background called a canvas. The underlying model is similar to that of a collage in which an artist creates a composition by taking various objects and assembling them on a background canvas. In the world of the collage artist, those objects might be geometrical shapes, words clipped from newspapers, lines formed from bits of string, or images taken from magazines. In the acm.graphics package, there are counterparts for each of these graphical objects.

The "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.

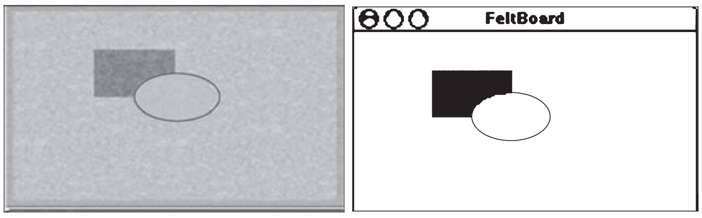


Fig. Physical FeltBoard and its Virtual Equivalent

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.

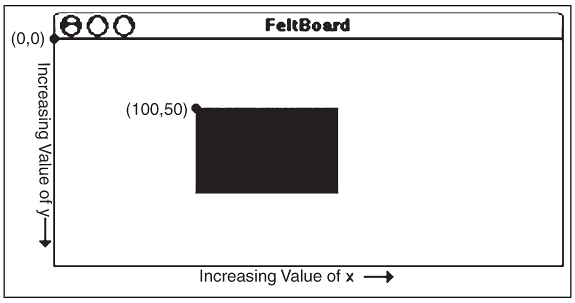


Fig. The Java Coordinate System

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

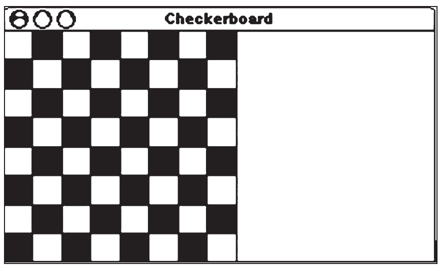
```
new GRect(x, y, width, height)
```

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.

5

Projection Transformations in Graphics

The desired modelview matrix so that the correct modelling and viewing transformations are applied. The desired projection matrix, which is also used to transform the vertices in your scene. Before you issue any of the transformation commands, remember to call;

glMatrixMode(GL_PROJECTION);

glLoadIdentity();

so that the commands affect the projection matrix rather than the modelview matrix and so that you avoid compound projection transformations. Since each projection transformation command completely describes a particular transformation, typically you don't want to combine a projection transformation with another transformation.

The purpose of the projection transformation is to define a *viewing volume*, which is used in two ways. The viewing volume determines how an object is projected onto the screen (that is, by using a perspective or an orthographic projection), and it defines which objects or portions of objects are clipped out of the final image. You can think of the viewpoint we've been talking about as existing at one end of the viewing volume. At this point, you might want to reread "A Simple Example: Drawing a Cube" for its overview of all the transformations, including projection transformations.

PERSPECTIVE PROJECTION

The most unmistakable characteristic of perspective projection is foreshortening: the farther an object is from the camera, the smaller it appears in the final image. This occurs because the viewing volume for a perspective projection is a frustum of a pyramid (a truncated pyramid whose top has been cut off by a plane parallel to its base). Objects that fall within the viewing volume are projected towards the apex of the pyramid, where the camera or viewpoint is. Objects that are closer to the viewpoint appear larger because they occupy a proportionally larger amount of the viewing volume than those that are farther away, in the larger part of the frustum. This method of projection is commonly used for animation, visual simulation, and any other applications that strive for some degree of realism because it's similar to how our eye (or a camera) works.

The command to define a frustum, glFrustum(), calculates a matrix that accomplishes perspective projection and multiplies the current projection matrix (typically the identity matrix) by it. Recall that the viewing volume is used to clip objects that lie outside of it; the four sides of the frustum, its top, and its base correspond to the six clipping planes of the viewing. Objects or parts of objects outside these planes are clipped from the final image. Note that glFrustum() doesn't require you to define a symmetric viewing volume.

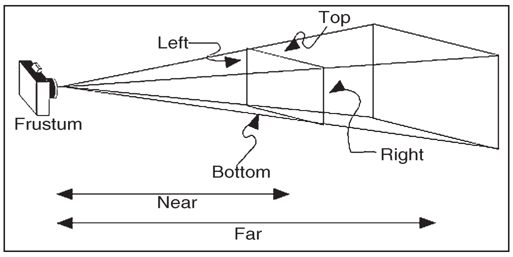


Fig. Perspective Viewing Volume Specified by glFrustum()

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,

GLdouble top, GLdouble near, GLdouble far);

Creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters: (left, bottom, -near) and (right, top, -near) specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; near and far give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

The frustum has a default orientation in three-dimensional space. You can perform rotations or translations on the projection matrix to alter this orientation, but this is tricky and nearly always avoidable.

Advanced

Also, the frustum doesn't have to be symmetrical, and its axis isn't necessarily aligned with the *z*-axis. For example, you can use glFrustum() to draw a picture as if you were looking through a rectangular window of a house, where the window was above and to the right of you. Photographers use such a viewing volume to create false perspectives. You might use it to have the hardware calculate images at much higher than normal resolutions, perhaps for use on a printer. For example, if you want an image that has twice the resolution of your screen, draw the same picture four times, each time using the frustum to cover the entire screen with one-quarter of the image. After each quarter of the image is rendered, you can read the pixels back to collect the data for the higher-resolution image.

Although it's easy to understand conceptually, glFrustum() isn't intuitive to use. Instead, you might try the Utility Library routine gluPerspective(). This routine creates a viewing volume of the same shape asglFrustum() does, but you specify it in a different way. Rather than specifying corners of the near clipping plane, you specify the angle of the field of view in the *y* direction and the aspect ratio of the width to height (*x*/*y*). These two parameters are enough to determine an untruncated pyramid along the line of sight. You also specify the distance between the viewpoint and the near and far clipping planes, thereby truncating the pyramid. Note that gluPerspective() is limited to creating frustums that are symmetric in both the *x*- and *y*-axes along the line of sight, but this is usually what you want.

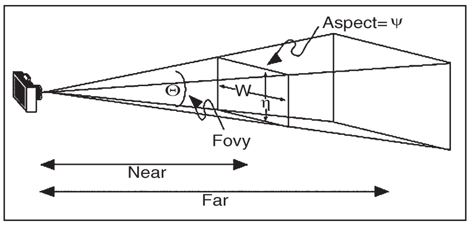


Fig. Perspective Viewing Volume Specified by gluPerspective()

void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. Fovy is the angle of the field of view in the x-z plane; its value must be in the range [0.0,180.0]. Aspect is the aspect ratio of the frustum, its width divided by its height. Near and far values the distances between the viewpoint and the clipping planes, along the negative z-axis. They should always be positive.

Just as with glFrustum(), you can apply rotations or translations to change the default orientation of the viewing volume created by gluPerspective(). With no such transformations, the viewpoint remains at the origin, and the line of sight points down the negative *z*-axis.

With gluPerspective(), you need to pick appropriate values for the field of view, or the image may look distorted. For example, suppose you're drawing to the entire screen, which happens to be 11 inches high. If you choose a field of view of 90 degrees, your eye has to be about 7.8 inches from the screen for the image to appear undistorted. (This is the distance that makes the screen subtend 90 degrees.) If your eye is farther from the screen, as it usually is, the perspective doesn't look right. If your drawing area occupies less than the full screen, your eye has to be even closer. To get a perfect field of view, figure out how far your eye normally is from the screen and how big the window is, and calculate the angle the window subtends at that size and distance. It's probably smaller than you would guess. Another way to think about it is that a 94-degree field of view with a 35-millimeter camera requires a 20-millimeter lens, which is a very wide-angle lens.

The preceding paragraph mentions inches and millimeters - do these really have anything to do with OpenGL? The answer is, in a word, no. The projection and other transformations are inherently unitless. If you want to think of the near and far clipping planes as located at 1.0 and 20.0 meters, inches, kilometres, or leagues, it's up to you. The only rule is that you have to use a consistent unit of measurement. Then the resulting image is drawn to scale.

Orthographic Projection

With an orthographic projection, the viewing volume is a rectangular parallelepiped, or more informally. Unlike perspective projection, the size of the viewing volume doesn't change from one end to the other, so distance from the camera doesn't affect how large an object appears. This type of projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them as they're projected.

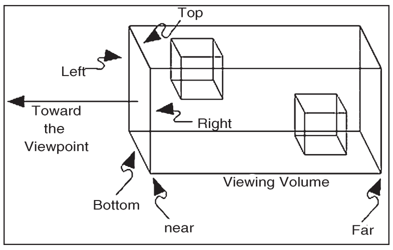


Fig. Orthographic Viewing Volume

The command glOrtho() creates an orthographic parallel viewing volume. As with glFrustum(), you specify the corners of the near clipping plane and the distance to the far clipping plane.

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, -near) and (right, top, -near) are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. (left, bottom, -far) and (right, top, -far) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both near and far can be positive or negative.

With no other transformations, the direction of projection is parallel to the *z*-axis, and the viewpoint faces towards the negative *z*-axis. Note that this means that the values passed in for *far* and *near* are used as negative *z* values if these planes are in front of the viewpoint, and positive if they're behind the viewpoint.

For the special case of projecting a two-dimensional image onto a two-dimensional screen, use the Utility Library routine gluOrtho2D(). This routine is identical to the three-dimensional version, glOrtho(), except that all the *z* coordinates for objects in the scene are assumed to lie between -1.0 and 1.0. If you're drawing two-dimensional objects using the two-dimensional vertex commands, all the *z* coordinates are zero; thus, none of the objects are clipped because of their *z* values.

void gluOrtho2D(GLdouble left, GLdouble right,

GLdouble bottom, GLdouble top);

Creates a matrix for projecting two-dimensional coordinates onto the screen and multiplies the current projection matrix by it. The clipping region is a rectangle with the lower-left corner at (left, bottom) and the upper-right corner at (right, top).

Viewing Volume Clipping

After the vertices of the objects in the scene have been transformed by the modelview and projection matrices, any primitives that lie outside the viewing volume are clipped.

The six clipping planes used are those that define the sides and ends of the viewing volume.

You can specify additional clipping planes and locate them wherever you choose. Keep in mind that OpenGL reconstructs the edges of polygons that get clipped.

2D TRANSFORMATIONS

Transformaions are a fundamental part of computer graphics. Transformations are used to position objects, to shape objects, to change viewing positions, and even to change how something is viewed (*e.g.* the type of perspective that is used).

In 3D graphics, we must use 3D transformations. However, 3D transformations can be quite confusing so it helps to first start with 2D.

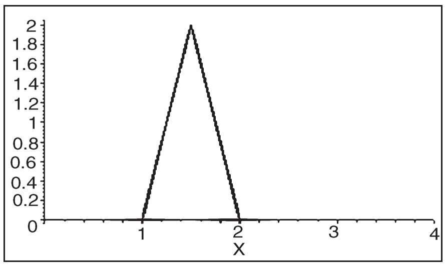
There are 4 main types of transformations that one can perform in 2 dimensions:

- Translations
- Scaling
- Rotation
- Shearing.

These basic transformations can also be combined to obtain more complex transformations. In order to make the representation of these complex transformations easier to understand and more efficient, we introduce the idea of homogeneous coordinates.

REPRESENTATION OF POINTS/OBJECTS

A point p in 2D is represented as a pair of numbers: p = (*x*, *y*) where *x* is the x-coordinate of the point p and *y* is the y-coordinate of p. 2D objects are often represented as a set of points (vertices), {p_1, p_2,..., p_n}, and an associated set of edges {e_1, e_2,..., e_m}. An edge is defined as a pair of points e = {p_i, p_j}. What are the points and edges of the triangle below?



We can also write points in vector/matrix notation as

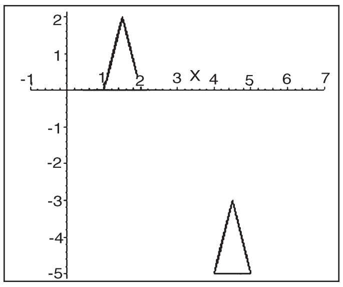
$$p = \begin{bmatrix} x \\ y \end{bmatrix}$$

TRANSLATIONS

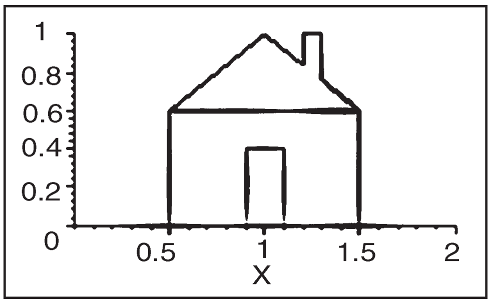
Assume you are given a point at (x,y)=(2,1). Where will the point be if you move it 3 units to the right and 1 unit up? Ans: (x',y') = (5,2). How was this obtained? (x', y') = (x+3,y+1). That is, to move a point by some amount *dx* to the right and *dy up*, you must add *dx* to the x-coordinate and add *dy* to the y-coordinate.

What was the required transformation to move the green triangle to the red triangle? Here the green triangle is represented by 3 points

triangle = { p1=(1, 0), p2=(2, 0), p3=(1.5, 2)}



What are the points and edges in this picture of a house? What are the transformation is required to move this house so that the peak of the roof is at the origin? What is required to move the house as shown in animation?

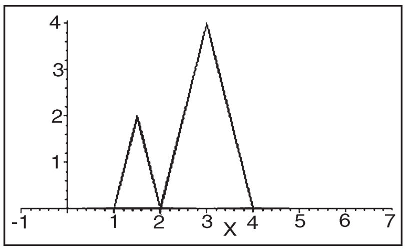


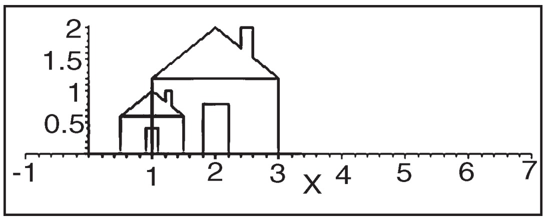
Matrix/Vector Representation of Translations

A translation can also be represented by a pair of numbers, $t = (t_x, t_y)$ where t_x is the change in the x-coordinate and t_y is the change in y coordinate. To translate the point p by t, we simply add to obtain the new (translated) point $q = p + t$.

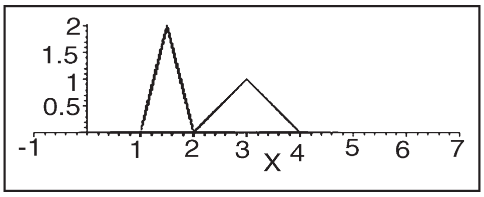
$$q = p + t = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

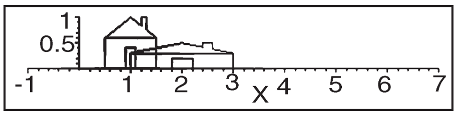
SCALING





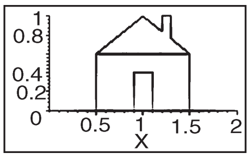
The scaling for the x dimension does not have to be the same as the y dimension. If these are different, then the object is distorted. What is the scaling in each dimension of the pictures below?





And if we double the size, where is the resulting object? In the pictures above, the scaled object is always shifted to the right.

This is because it is scaled with *respect to the origin*. That is, the point at the origin is left fixed. Thus scaling by more than 1 moves the object away from the origin and scaling of less than 1 moves the object towards the origin. This can be seen in the animation below.



This is because of how basic scaling is done. The above objects have been scaled simply by multiplying each of its points by the appropriate scaling factor. For example, the point p=(1.5,2) has been scaled by 2 along x and .5 along y. Thus, the new point is q = (2×1.5, .5×2) = (1, 1).

Matrix/Vector Representation of Scaling

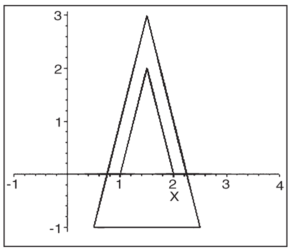
Scaling transformations are represented by matrices. For example, the above scaling of 2 and .5 is represented as a matrix:

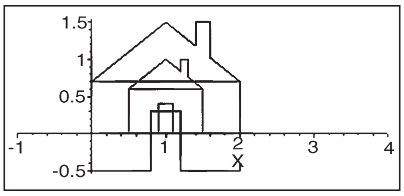
Scale matrix: $s = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & .5 \end{bmatrix}$

New point: $q = s \times p = \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx & x \\ sy & y \end{bmatrix}$

Scaling about a Particular Point

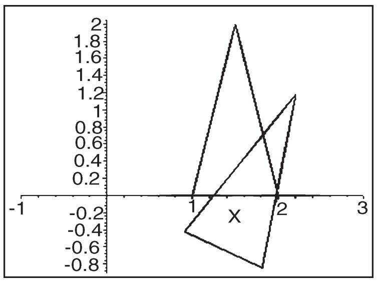
What do we do if we want to scale the objects about their centre as show below?

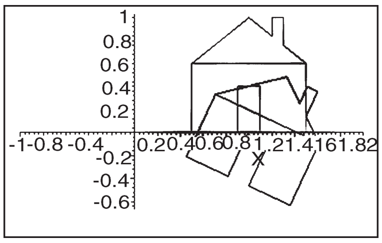




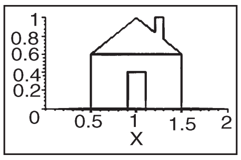
ROTATION

Below, we see objects that have been rotate by 25 degrees.





Again, we see that basic rotations are with respect to the origin:



Matrix/Vector Representation of Rotation

Counterclockwise rotation of *a* degrees

$$= \begin{bmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{bmatrix}$$

SHEAR

Matrix/Vector Representation of Shear

shear along x axis = $\begin{bmatrix} 1 & shearx \\ 0 & 1 \end{bmatrix}$

shear along y axis = $\begin{bmatrix} 1 & 0 \\ shearx & 1 \end{bmatrix}$

Combining Transformations

We saw that the basic scaling and rotating transformations are always with respect to the origin. To scale or rotate about a particular point (the fixed point) we must first translate the object so that the fixed point is at the origin. We then perform the scaling or rotation and then the inverse of the original translation to move the fixed point back to its original position.

For example, if we want to scale the triangle by 2 in each direction about the point fp = (1.5, 1), we first translate all the points of the triangle by T = (–1.5, 1), scale by 2 (S), and then translate back by -T=(1.5, 1). Mathematically this looks like

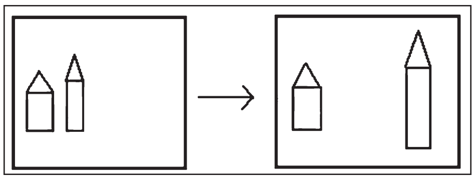
$$q = \begin{bmatrix} x_2 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \left(\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} -1.5 \\ -1 \end{bmatrix} \right) + \begin{bmatrix} 1.5 \\ 1 \end{bmatrix}$$

2D MODELLING TRANSFORMATIONS

Modelling transformations are the mechanism used to compose an image from modelling primitives. The modelling proimitives are defined in their own modelling coordinate system and then placed in the final scene by using modelling transformations.

We can change the Camera position or World Coordinate Window to scale or move an entire image, but we may want to only change a particular part of the image.

Example: We might want to translate and scale the tall house but not change the short house. We can’t do this by modifying the window, so we want to apply transformations to the individual objects in the scene.

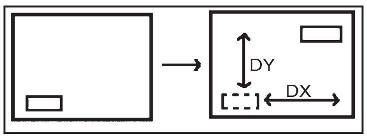


Possible modelling transformations include the following:

- Change size of object: Scaling
- Move object: Translation
- Rotate object: Rotation.

2D TRANSLATION

Translation is a simple straight line movement of the object.



(old coordinates are (x, y) and the new coordinates are (x', y'))

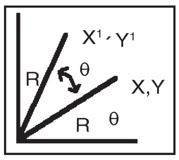
$$x' = x + Tx$$
$$y' = y + Ty$$

For a complex object such as a polygon, add the translation distances (Tx, Ty) for each endpoint of the object. For a circle or ellipse:

$$xc' = xc + Tx$$
$$yc' = yc + Ty$$

2D ROTATION

Example of a 2D rotation through an angle w where the coordinates x, y go into x', y'. Note that w is positive for a counterclockwise rotation and that that rotation is about the origin (0, 0).



Derive the Formula for Rotation

(old coordinates are (x, y) and the new coordinates are (x', y'))

θ = initial angle,

ϕ = angle of rotation.

$x = r \cos \theta$

$y = r \sin \theta$

$x' = r \cos (\theta + \phi) = r \cos \theta \cos \phi - r \sin \theta \sin \phi$

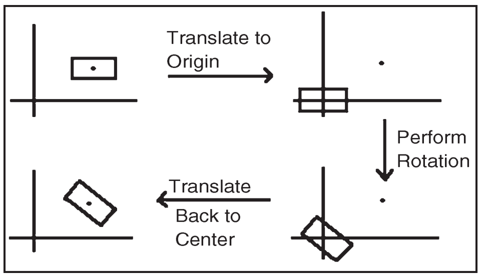
$y' = r \sin (\theta + w) = r \sin \theta \cos \phi + r \cos \theta \sin \phi$

Hence:

$x' = x \cos \theta - y \sin \theta$

$y' = y \cos \phi + x \sin \phi$

What if we want to rotate about another point rather than the origin, *e.g.*, the centre of an object? Then we have the same problem as with scaling. A solution to this problem is to perform several transformations rather tnan just one.



Now we could apply the 3 transformations to the object one at a time. But this is inefficient, especially when the object has many points. It would be nice to be able to compose the transformations into one and then apply this total transformation to the object.

2D SCALING

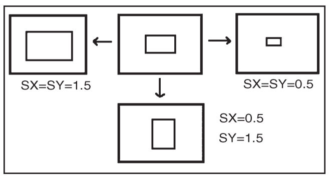
Scaling alters the size of the object. Sx and Sy are the scaling factors:

(old coordinates are (x, y) and the new coordinates are (x’, y’))

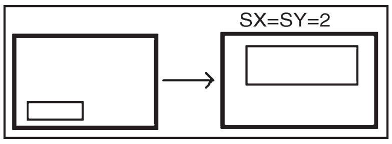
x’ = x * Sx

y’ = y * Sy

If Si > 1.0 then the object becomes larger. If Si < 1.0, then the object becomes smaller. If Sx = Sy then we have uniform scaling and it maintains the relative proportions of the object. If Sx <> Sy then we have differential scaling and it deforms the object.



Examples of Scaling with the object symmetrical about origin (0, 0). Example of Scaling with the object not symmetrical about the origin (0, 0). Note that this produces both a translation and size change.

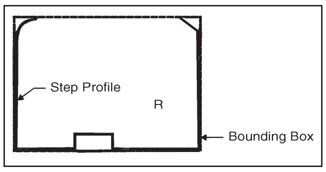


WINDOWING AND TRANSFORMATION

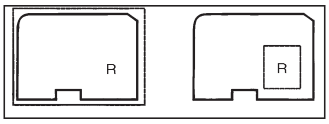
The ODB_Rip has the ability to apply both windowing and transformations to the input data.

WINDOWING

When the step to rasterize has been selected, the library generates a computes a bounding box for the input data based on the profile associated with that step. Note that when opening the ODB++ file, the function returns the extents box for each of the steps found in the ODB++ file.



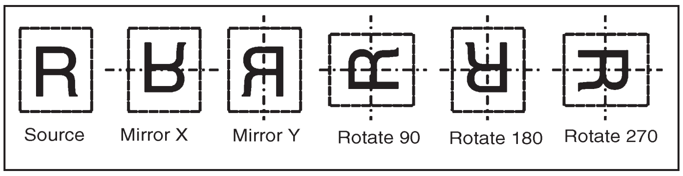
However using the -extents argument in the setup function allows the calling programme to define a different data window as shown below:



TRANSFORMATIONS

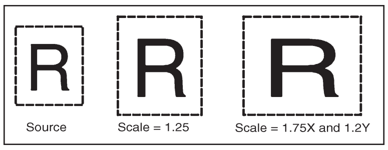
Mirroring and Rotation

Once the extents are known (whether computed from the profile or specified in the setup function) it is then possible to apply scaling, rotation and mirroring to the selected window. Note that all transformations are applied using the centre of the window as a reference point.



Scaling

It is possible to perform scaling independently in X and Y if this is required.



Only after all the transformations are applied are the bands defined.

3D TRANSFORMATION

In 3D graphics, transformation is often used to operate on vertices and vectors. It is also used to convert them in one space to another. Transformation is performed via multiplication with a matrix. There are typically three types of primitive transformation that can be performed on vertices: translation (where it lies in space relative to the origin), rotation (its direction in relation to the x, y, z frame), and scaling (its distance from origin). In addition to those, projection transformation is used to go from view space to projection space. The D3DX library contains APIs that can conveniently construct a matrix for many purposes such as translation, rotation, scaling, world-to-view transformation, view-to-projection transformation, etc. An application can then use these matrices to transform vertices in its scene. A basic understanding of matrix transformations is required. We will briefly look at some examples below.

TRANSLATION

Translation refers to moving or displacing for a certain distance in space. In 3D, the matrix used for translation has the form:

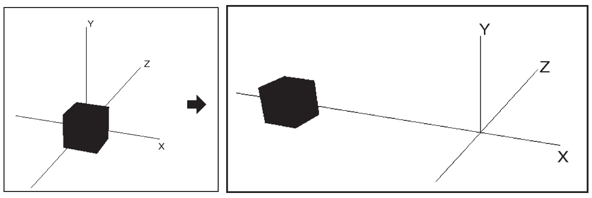
$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{matrix}$$

where (a, b, c) is the vector that defines the direction and distance to move. For example, to move a vertex -5 unit along the X axis (negative X direction), we can multiply it with this matrix:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -5 & 0 & 0 & 1 \end{matrix}$$

If we apply this to a cube object centred at origin, the result is that the box is moved 5 units towards the negative X axis, after translation is applied.

The Effect of Translation



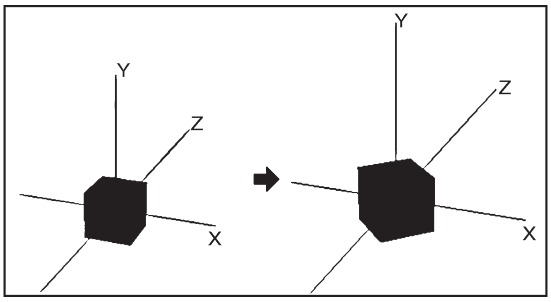
In 3D, a space is typically defined by an origin and three unique axes from the origin: X, Y and Z. There are several spaces commonly used in computer graphics: object space, world space, view space, projection space, and screen space.

ROTATION

Rotation refers to rotating vertices about an axis going through the origin. Three such axes are the X, Y, and Z axes in the space. An example in 2D would be rotating the vector [1 0] 90 degrees counter-clockwise. The result from the rotation is the vector [0 1]. The matrix used for rotating ? degrees clockwise about the Y axis looks like this:

$$\begin{matrix} \cos? & 0 & -\sin? & 0 \\ 0 & 1 & 0 & 0 \\ \sin? & 0 & \cos? & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Figure shows the effect of rotating a cube centred at origin for 45 degrees about the Y axis.



SCALING

Scaling refers to enlarging or shrinking the size of vector components along axis directions. For example, a vector can be scaled up along all directions or scaled down along the X axis only. To scale, we usually apply the scaling matrix below:

$$\begin{matrix} p & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & r & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Where p, q, and r are the scaling factor along the X, Y, and Z direction, respectively. The effect of scaling by 2 along the X axis and scaling by 0.5 along the Y axis.

6

Computer-Generated Imagery

Computer-generated imagery (CGI) is the application of the field of computer graphics or, more specifically, 3D computer graphics to special effects in art, films, television programmes, commercials, simulators and simulation generally, and printed media. The visual scenes may be either dynamic or static.

The term *computer animation* refers to dynamic CGI rendered as a movie. The term *virtual world* refers to agent-based, interactive environments. 3D computer graphics software is used to make computer-generated imagery for movies, etc. Recent availability of CGI software and increased computer speeds have allowed individual artists and small companies to produce professional grade films, games, and fine art from their home computers. This has brought about an Internet subculture with its own set of global celebrities, clichés, and technical vocabulary.

STATIC IMAGES AND LANDSCAPES

Not only do animated images form part of computer-generated imagery, natural looking landscapes, such as fractal landscapes are also generated via computer algorithms. A simple way to generate fractal surfaces is to use an extension of the triangular mesh method, relying on the construction of some special case of a de Rham curve, e.g. midpoint displacement. For instance, the algorithm may start with a large triangle, then recursively zoom in by dividing it into 4 smaller Sierpinski triangles, then interpolate the height of each point from its nearest neighbors. The creation of a Brownian surface may be achieved not only by adding noise as new nodes are created, but by adding additional noise at multiple levels of the mesh. Thus a topographical map with varying levels of height can be created using relatively straightforward fractal algorithms. Some typical, and easy to programme fractals used in CGI are the *plasma fractal* and the more dramatic *fault fractal*. A large number of specific techniques have been researched and developed to produce highly focused computer-generated effects, e.g. the use of specific models to represent the chemical weathering of stones to model erosion and produce an “aged appearance” for a given stone-based surface.

ARCHITECTURAL SCENES

Modern architects use services from computer graphic firms to create 3-dimensional models for both customers and builders. These computer generated models can be more accurate than traditional drawings. Architectural animation (which provides animated movies of buildings, rather than interactive images) can also be used to see the possible relationship a building will have in relation to the environment and its surrounding buildings. The rendering of architectural spaces without the use of paper and pencil tools is now a widely accepted practice with a number of computer-assisted architectural design systems. Architectural modeling tools allow an architect to visualize a space and perform "walk throughs" in an interactive manner, thus providing "interactive environments" both at the urban and building levels. Specific applications in architecture not only include the specification of building structures such as walls and windows, and walk-throughs, but the effects of light and how sunlight will affect a specific design at different times of the day. Architectural modeling tools have now become increasingly internet-based. However, the quality of internet-based systems still lags those of sophisticated inhouse modeling systems. In some applications, computer-generated images are used to "reverse engineer" historical buildings. For instance, a computer-generated reconstruction of the monastery at Georgenthal in Germany was derived from the ruins of the monastery, yet provides the viewer with a "look and feel" of what the building would have looked like in its day.

ANATOMICAL MODELS

Computer generated models used in skeletal animation are not always anatomically correct, however, organizations such as the Scientific Computing and Imaging Institute have developed anatomically correct computer-based models.

Computer generated anatomical models can be used both for instructional and operational purposes. To date, a large body of artist produced medical images continue to be used by medical students, such as images by Frank Netter, e.g. Cardiac images. However, a number of online anatomical models are becoming available. A single patient X-ray is not a computer generated image, even in the case of digitized x-rays. However, in applications which involve CT scans a three dimensional model is automatically produced from a large number of single slice x-rays, producing "computer generated image". Applications involving magnetic resonance imaging also bring together a number of "snapshots" (in this case via magnetic pulses) to produce a composite, internal image. In modern medical applications, patient specific models are constructed in "computer assisted surgery". For instance, in total knee replacement, the construction of a detailed patient specific model can be used to carefully plan the surgery. These three dimensional models are usually extracted from multiple CT scans of the appropriate parts of the patient's own anatomy. Such models can also be used for planning aortic valve implantations, one of the common procedures for treating heart disease. Given that the shape, diameter and position of the coronary openings can vary greatly from patient to patient, the extraction (from CT scans) of a model that closely resembles a patient's valve anatomy can be highly beneficial in planning the procedure.

GENERATING CLOTH AND SKIN IMAGES

Models of cloth generally fall into three groups: the geometric-mechanical structure at yarn crossings, secondly the mechanics of continuous elastic sheets and thirdly the geometric macroscopic features of cloth. To date, making the clothing of a digital character automatically fold in a natural way remains a challenge for many animators. In addition to their use in film, advertising and other modes of public display, computer generated images of clothing are now routinely used by top fashion design firms. The challenge in rendering human skin images involves three levels of realism: *photo realism* in that it should look like real skin at the static level; *physical realism* in that it should closely simulate real skin's movements and *functional realism* in that it should act like real skin in response to actions.

INTERACTIVE SIMULATION AND VISUALIZATION

Interactive visualization is a general term that applies to the rendering of data that may vary dynamically and allowing a user to view the data from multiple perspectives. The applications areas may vary significantly, ranging from the visualization of the flow patterns in fluid dynamics to specific computer aided design applications. The data rendered may correspond to specific visual scenes that change as the user interacts with the system, e.g. simulators such as flight simulators make extensive use of CGI techniques for representing the world. At the abstract level an interactive visualization process involves a '*data pipeline* in which the raw data is managed and filtered to a form that makes it suitable for rendering. This is often called the "visualization data". The visualization data is then mapped to a "visualization representation" that can be fed to a rendering system. This is usually called a "renderable representation". This representation is then rendered as a displayable image. As the user interacts with the system, e.g. by using joystick controls to change their position within the virtual world, the raw data is fed through the pipeline to create a new rendered image, often making real-time computational efficiency a key consideration in such applications.

COMPUTER ANIMATION

While computer generated images of landscapes may be static, the term computer animation only applies to dynamic images that resemble a movie. However, in general the term computer animation refers to dynamic images that do not allow user interaction, and the term virtual world is used for the interactive animated environments. Computer animation is essentially a digital successor to the art of stop motion animation of 3D models and frame-by-frame animation of 2D illustrations. Computer generated animations are more controllable than other more physically based processes, such as constructing miniatures for effects shots or hiring extras for crowd scenes, and because it allows the creation of images that would not be feasible using any other technology. It can also allow a single graphic artist to produce such content without the use of actors, expensive set pieces, or props. To create the illusion of movement, an image is displayed on the computer screen and repeatedly replaced by a new image that is similar to the previous image, but advanced slightly in the time domain (usually at a rate of 24 or 30 frames/second). This technique is identical to how the illusion of movement is achieved with television and motion pictures.

VIRTUAL WORLDS

A *virtual world* is a simulated environment, which allows user to interact with animated characters, or interact with other users through the use of animated characters known as avatars. Virtual worlds are intended for its users to inhabit and interact, and the term today has become largely synonymous with interactive 3D virtual environments, where the users take the form of avatars visible to others graphically. These avatars are usually depicted as textual, two-dimensional, or three-dimensional graphical representations, although other forms are possible (auditory and touch sensations for example). Some, but not all, virtual worlds allow for multiple users.

3D Modeling

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

MODELS

3D models represent a 3D object using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc. Being a collection of data (points and other information), 3D models can be created by hand, algorithmically (procedural modeling), or scanned. 3D models are widely used anywhere in 3D graphics. Actually, their use predates the widespread use of 3D graphics on personal computers. Many computer games used pre-rendered images of 3D models as sprites before computers could render them in real-time. Today, 3D models are used in a wide variety of fields. The medical industry uses detailed models of organs. The movie industry uses them as characters and objects for animated and real-life motion pictures. The video game industry uses them as assets for computer and video games. The science sector uses them as highly detailed models of chemical compounds. The architecture industry uses them to demonstrate proposed buildings and landscapes through Software Architectural Models. The engineering community uses them as designs of new devices, vehicles and structures as well as a host of other uses. In recent decades the earth science community has started to construct 3D geological models as a standard practice.

REPRESENTATION

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

- Solid - These models define the volume of the object they represent (like a rock). These are more realistic, but more difficult to build. Solid models are mostly used for nonvisual simulations such as medical and engineering simulations, for CAD and specialized visual applications such as ray tracing and constructive solid geometry

- Shell/boundary - these models represent the surface, e.g. the boundary of the object, not its volume (like an infinitesimally thin eggshell). These are easier to work with than solid models. Almost all visual models used in games and film are shell models.

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

MODELING PROCESSES

There are five popular ways to represent a model:

- Polygonal modeling - Points in 3D space, called vertices, are connected by line segments to form a polygonal mesh. Used, for example, by Blender. The vast majority of 3D models today are built as textured polygonal models, because they are flexible and because computers can render them so quickly. However, polygons are planar and can only approximate curved surfaces using many polygons.
- NURBS modeling - NURBS Surfaces are defined by spline curves, which are influenced by weighted control points. The curve follows (but does not necessarily interpolate) the points. Increasing the weight for a point will pull the curve closer to that point. NURBS are truly smooth surfaces, not approximations using small flat surfaces, and so are particularly suitable for organic modeling. Maya, Rhino 3d and solidThinking are the most well-known commercial programmes which use NURBS natively.
- Splines & Patches modeling - Like NURBS, Splines and Patches depend on curved lines to define the visible surface. Patches fall somewhere between NURBS and polygons in terms of flexibility and ease of use.

- Primitives modeling - This procedure takes geometric primitives like balls, cylinders, cones or cubes as building blocks for more complex models. Benefits are quick and easy construction and that the forms are mathematically defined and thus absolutely precise, also the definition language can be much simpler. Primitives modeling is well suited for technical applications and less for organic shapes. Some 3D software can directly render from primitives (like POV-Ray), others use primitives only for modeling and convert them to meshes for further operations and rendering.
- Sculpt modeling - Still fairly new method of modeling 3D sculpting has become very popular in the few short years it has been around. There are 2 types of this currently, Displacement which is the most widely used among applications at this moment, and volumetric. Displacement uses a dense model (often generated by Subdivision surfaces of a polygon control mesh) and stores new locations for the vertex positions through use of a 32bit image map that stores the adjusted locations. Volumetric which is based loosely on Voxels has similar capabilities as displacement but does not suffer from polygon stretching when there are not enough polygons in a region to achieve a deformation. Both of these methods allow for very artistic exploration as the model will have a new topology created over it once the models form and possibly details have been sculpted. The new mesh will usually have the original high resolution mesh

information transferred into displacement data or normal map data if for a game engine.

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

- constructive solid geometry
- implicit surfaces
- subdivision surfaces

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

SCENE SETUP

Scene setup involves arranging virtual objects, lights, cameras and other entities on a scene which will later be used to produce a still image or an animation. Lighting is an important aspect of scene setup. As is the case in real-world scene arrangement, lighting is a significant contributing factor to the resulting aesthetic and visual quality of the finished work. As such, it can be a difficult art to master. Lighting effects can contribute greatly to the mood and emotional response effected by a scene, a fact which is well-known to photographers and theatrical lighting technicians. It is usually desirable to add color to a model's surface in a user controlled way prior to rendering. Most 3D modeling software allows the user to color the model's vertices, and that color is then interpolated across the model's surface during rendering. This is often how models are colored by the modeling software while the model is being created. The most common method of adding color information to a 3D model is by applying a 2D texture image to the model's surface through a process called texture mapping. Texture images are no different than any other digital image, but during the texture mapping process, special pieces of information (called texture coordinates or UV coordinates) are added to the model that indicate which parts of the texture image map to which parts of the 3D model's surface. Textures allow 3D models to look significantly more detailed and realistic than they would otherwise.

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

If used for animation, this phase usually makes use of a technique called "keyframing", which facilitates creation of complicated movement in the scene. With the aid of keyframing, one needs only to choose where an object stops or changes its direction of movement, rotation, or scale, between which states in every frame are interpolated. These moments of change are known as keyframes.

Often extra data is added to the model to make it easier to animate. For example, some 3D models of humans and animals have entire bone systems so they will look realistic when they move and can be manipulated via joints and bones, in a process known as skeletal animation.

COMPARED TO 2D METHODS

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

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

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

3D MODEL MARKET

3CT (3D Catalog Technology) has revolutionized the 3D model market by offering quality 3D model libraries free of charge for professionals using various CAD programmes. Some believe that this uprising technology is gradually eroding the traditional "buy and sell" or "object for object exchange" markets although the quality of the products do not match those sold on specialized 3d marketplaces. A large market for 3D models (as well as 3D-related content, such as textures, scripts, etc.) still exists - either for individual models or large collections. Online marketplaces for 3D content allow individual artists to sell content that they have created.

Often, the artists' goal is to get additional value out of assets they have previously created for projects. By doing so, artists can earn more money out of their old content, and companies can save money by buying pre-made models instead of paying an employee to create one from scratch. These marketplaces typically split the sale between themselves and the artist that created the asset, often in a roughly 50-50 split. In most cases, the artist retains ownership of the 3d model; the customer only buys the right to use and present the model.

HUMAN MODELS

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

7

Workspace Management in Computer Graphics

The Blender GUI is made up of one or more screens, each of which can be divided into sections and subsections that can be of any type of Blender's views or window-types. Each window-type's own GUI elements can be controlled with the same tools that manipulate 3D view. For example, one can zoom in and out of GUI-buttons in the same way one zooms in and out in the 3D viewport. The GUI viewport and screen layout is fully user-customizable. It is possible to set up the interface for specific tasks such as video editing or UV mapping or texturing by hiding features not utilized for the task.

HARDWARE REQUIREMENTS

Blender has very low hardware requirements compared to other 3D suites. However, for advanced effects and high-poly models, a powerful system is needed.

FILE FORMAT

Blender features an internal file system that allows one to pack multiple scenes into a single file (called a ".blend" file).

- All of Blender's ".blend" files are forward, backward, and cross-platform compatible with other versions of Blender.
- Snapshot ".blend" files can be auto-saved periodically by the program, making it easier to survive a programme crash.
- All scenes, objects, materials, textures, sounds, images, post-production effects for an entire animation can be stored in a single ".blend" file. Data loaded from external sources, such as images and sounds, can also be stored externally and referenced through either an absolute or relative pathname. Likewise, ".blend" files themselves can also be used as libraries of Blender assets.
- Interface configurations are retained in the ".blend" files, such that what you save is what you get upon load. This file can be stored as "user defaults" so this screen configuration, as well as all the objects stored in it, is used every time you load Blender.

The actual ".blend" file is similar to the EA Interchange File Format, starting with its own header (for example BLENDER_v248) that specifies the version, endianness and pointer size, followed by a collection of binary chunks storing the data blocks, and all the type and struct definitions also known as DNA. Although it is hard to read and convert a

".blend" file to another format using external tools, the readblend utility can do this. Dozens of import/export scripts that run inside Blender itself, accessing the object data via API, make it possible to inter-operate with other 3D tools.

Jeroen Bakker documented the Blender file format to allow inter-operation with other tooling. The document can be found at mystery of the blend. A DNA structure browser is also available on this site.

Blender organizes data as various kinds of "data blocks", such as Objects, Meshes, Lamps, Scenes, Materials, Images and so on. An object in Blender consists of multiple data blocks - for example, a polygon mesh has at least an Object and Mesh data block, and usually also a Material. This allows various data blocks to refer to each other; there may be, for example, multiple Objects that refer to the same Mesh, allowing the mesh to be duplicated while only keeping one copy of the mesh data in memory, and allowing subsequent editing of all duplicated meshes at the same time. Data block relationships can also be changed manually. Data blocks can also be referred to in other .blend files, allowing the use of .blend files as reusable object libraries.

COMPARISON WITH OTHER 3D SOFTWARE

Blender is a dominant open source product with a range of features comparable to mid- to high-range commercial, proprietary software. In 2010, CGenie classed Blender as a fledgling product with the majority of its users being "hobbyists" rather than students or professionals but with its high standards rising year on year. They also reported that users thought Blender needed more development and required more compatibility with other programmes.

In 2007, TDT3D considered that Blender's interface was not up to industry standards but was nevertheless suited to fast workflow and was sometimes more intuitive. Poor documentation was also criticized although there is community support through an online wiki, and a range of books published both by the Blender Foundation and independently.

In 2010, Blender 2.5 Beta was released for open-testing. Featuring a completely redesigned and greatly simplified user interface, it aims to improve work flow and ease of use. Although not yet fully featured, Blender 2.5 is in its final stages of development and its animation system is considered by early users to be as good or better than some professional packages.

DEVELOPMENT

Since the opening of the source, Blender has experienced significant refactoring of the initial codebase and major additions to its feature set. Recent improvements include an animation system refresh; a stack-based modifier system; an updated particle system (which can also be used to simulate hair and fur); fluid dynamics; soft-body dynamics; GLSL shaders support in the game engine; advanced UV unwrapping; a fully-recoded render pipeline, allowing separate render passes and "render to texture"; node-based material editing and compositing; Projection painting.

Part of these developments were fostered by Google's Summer of Code program, in which the Blender Foundation has participated since 2005. The current release version is 2.49b. Primarily, the last release, 2.48a was an update to reflect many of the Blender Game Engine changes made throughout the Yo Frankie! project; including real-time shading, many real-time GLSL materials, and updates to the physics components. Version 2.48a also made changes to the animation systems, adds wind simulation, and fixes a number of backlogged bugs.

Blender 2.5 is currently in the test version release cycle, beginning with the release of Alpha 0 version on 24 November 2009, and currently 2.56 Beta as of the 30th of December. New features currently in 2.56 include:

- New user interface
- New animation system, which allows almost any value to be animated
- Re-written, Python 3.x scripting API
- Smoke simulation
- Ocean simulation
- Updated toolset, with improved implementation
- Approximate indirect lighting
- Volume rendering
- Ray tracing optimizations, rendering some scenes "up to 10x faster"
- Solidify modifier
- Sculpt brush and stroke upgrade
- Add-on system

- Custom keyboard shortcuts
- Spline IK
- Color management
- Fluid particles (smoothed-particle hydrodynamics)
- Network render
- Deep shadow maps
- Dynamic paint system

SUPPORT

In the month following the release of Blender v2.44, it was downloaded 800,000 times; this worldwide user base forms the core of the support mechanisms for the program. Most users learn Blender through community tutorials and discussion forums on the internet such as Blender Artists (previously known as elYsiun); however, another learning method is to download and inspect ready-made Blender models. Numerous other sites, for example BlenderArt Magazine—a free, downloadable magazine with each issue handling a particular area in 3D development—and BlenderNation, provide information on everything surrounding Blender, showcase new techniques and features, and provide tutorials and other guides.

USE IN THE MEDIA INDUSTRY

Blender started out as an inhouse tool for a Dutch commercial animation company, NeoGeo. Blender has been used for television commercials in several parts of the world including Australia, Iceland, Brazil, Russia and Sweden.

The first large professional project that used Blender was *Spider-Man 2*, where it was primarily used to create animatics and pre-visualizations for the storyboard department.

“As an animatic artist working in the storyboard department of Spider-Man 2, I used Blender’s 3D modeling and character animation tools to enhance the storyboards, re-creating sets and props, and putting into motion action and camera moves in 3D space to help make Sam Raimi’s vision as clear to other departments as possible.” - Anthony Zierhut, Animatic Artist, Los Angeles.

The French-language film *Friday or Another Day* (*Vendredi ou un autre jour*) was the first 35 mm feature film to use Blender for all the special effects, made on GNU/Linux workstations. It won a prize at the Locarno International Film Festival. The special effects were by Digital Graphics of Belgium. Blender has also been used for shows on the History Channel, alongside many other professional 3D graphics programmes. Tomm Moore’s *The Secret of Kells*, which was partly produced in Blender by the Belgian studio Digital Graphics, has been nominated for an Oscar in the category ‘Best Animated Feature Film’.

ELEPHANTS DREAM (OPEN MOVIE PROJECT: ORANGE)

In September 2005, some of the most notable Blender artists and developers began working on a short film using primarily free software, in an initiative known as the Orange Movie Project hosted by the Netherlands Media Art Institute (NIMk). The resulting film, *Elephants Dream*, premiered on

March 24, 2006. In response to the success of *Elephants Dream*, the Blender Foundation founded the Blender Institute to do additional projects with two announced projects: Big Buck Bunny, also known as “Project Peach” (a ‘furry and funny’ short open animated film project) and Yo Frankie, also known as Project Apricot (an open game in collaboration with CrystalSpace which reused some of the assets created during Project Peach).

BIG BUCK BUNNY (OPEN MOVIE PROJECT: PEACH)

On October 1, 2007, a new team started working on a second open project, “Peach”, for the production of the short movie *Big Buck Bunny*. This time, however, the creative concept was totally different. Instead of the deep and mystical style of Elephants Dream, things are more “funny and furry” according to the official site[The movie had its premiere on April 10, 2008.

YO FRANKIE! (OPEN GAME PROJECT: APRICOT)

Apricot is a project for production of a game based on the universe and characters of the Peach movie (Big Buck Bunny) using free software. The game is titled *Yo Frankie*. The project started February 1, 2008, and development was completed at the end of July 2008. A finalized product was expected at the end of August; however, the release was delayed. The game was released on December 9, 2008, under either the GNU GPL or LGPL, with all content being licensed under Creative Commons Attribution 3.0.

PLUMÍFEROS

Plumíferos, a commercial animated feature film created entirely in Blender, was premiered in February 2010 in Argentina. Its main characters are anthropomorphic talking animals.

SINTEL (OPEN MOVIE PROJECT: DURIAN)

The Blender Foundation announced its newest Open Movie, codenamed Project Durian (in keeping with the tradition of fruits as code names). It was this time chosen to make a fantasy action epic of about twelve minutes in length, starring a female teenager and a young dragon as the main characters. The film premiered online on September 30 2010.

Geist3D

Geist3D is a free software programme for real time modelling and rendering three-dimensional graphics and animations. At this time Geist3D is only available for the Microsoft Windows operating system. Geist3D began as a Ph.D. research project at the University of Victoria in 2001. It was originally designed as a robotics simulation tool, but over the past five years it has grown into a more general graphics engine that computes rigid body physics, generates planetary-sized terrain and renders skeletal based characters. Geist3D also provides a combination of Petri nets and Lua scripts as a programming language to interpret virtual sensor input and control the parameters of a simulation. The editor has a comprehensive user interface to construct 3D models, Petri nets, Lua scripts and OpenGL 2.0 shading language programmes using drag-and-drop operations and integrated source code editors.

K-3D

K-3D is a free 3D modelling and animation software. Despite its name it is not a KDE application. It features a plug-in-oriented procedural engine for all of its content. K-3D supports polygonal modelling, and includes basic tools for NURBS, patches, curves and animation.

MAIN FEATURES

K-3D's interface uses platform's look-and-feel, and it's consistent with other applications that already exists. Because of this new artists will find K-3D easy to understand, and professionals feel right at home. K-3D is intuitive, consistent, and discoverable. K-3D features procedural and parametric work-flows. Properties can be adjusted interactively and results appear immediately. The powerful, node-based visualization pipeline allows more possibilities than traditional modifier stacks or histories. Selection flows from one modifier to the next. Industrial-strength standards form the foundation on which K-3D builds - including native RenderMan(TM) support that integrates tightly with the K-3D user interface. K-3D supports a node-based visualization pipeline, thus allowing the connection of multiple bodies. Work on one side of a model, show the other side mirrored, and see the end result welded together as a subdivision surface in real-time. Using K-3D, complex work-flows are easy to create and understand. Go back, modify the beginning of a work-flow, and watch as changes propagate automatically to the end.

MeshLab

MeshLab, is a free 3D mesh processing software program; MeshLab, started in late 2005, is an open-source general-purpose system aimed to help the processing of the typical not-so-small unstructured 3D models that arise in the pipeline of processing of the data coming from 3D scanning. MeshLab is oriented to the management and processing of unstructured large meshes and provides a set of tools for editing, cleaning, healing, inspecting, rendering and converting these kinds of meshes.

The automatic mesh cleaning filters includes removal of duplicated, unreferenced vertices, non manifold edges, vertices and null faces. Remeshing tools support high quality simplification based on quadric error measure, various kinds of subdivision surfaces and two surface reconstruction algorithms from point clouds based on the *ball-pivoting* technique and on the Poisson surface reconstruction approach. For the removal of noise, usually present in acquired surfaces, MeshLab supports various kinds of smoothing filters and tools for curvature analysis and visualization. It includes a tool for the registration of multiple range maps based on the Iterative Closest Point algorithm. MeshLab also includes an interactive direct paint-on-mesh system that allows to interactively change the color of a mesh, to define selections and to directly smooth out noise and small features. MeshLab is available for most platforms, including Windows, Linux and Mac OS X. The system supports input/output in the following formats: PLY, STL, OFF, OBJ, 3DS, VRML 2.0, U3D, X3D and COLLADA. MeshLab allows also to directly import the point clouds reconstructed using Photosynth to further process and reconstruction. MeshLab is used in various academic and research contexts, like microbiology, Cultural heritage, surface reconstruction and desktop manufacturing.

Misfit Model 3d

Misfit Model 3d is a 3D computer graphics editor that works with triangle-based models. It is designed to be easy to use and easy to extend with plugins and scripts. Misfit Model 3d is free software and distributed under GNU General Public License.

FEATURES

- Multi-level undo
- Skeletal animation
- Simple texturing
- Scripting
- Command-line batch processing
- Plugin system for adding new model and image filters

SUPPORTED FILE FORMATS

- MilkShape 3D (ms3d)
- Wavefront (.obj)

- LightWave 3D Object (lwo)
- Quake II model (md2)
- Quake III Arena model (md3)
- Caligari trueSpace (cob)
- AutoCAD (dxf)

OpenFX

OpenFX is also the name of an 2D image plugin standard used by several image processing packages in motion picture effects. http://openfx.sourceforge.net. Possibly this page needs to be split and disambiguated.

OpenFX is an Open-Source, free modeling and animation studio, distributed under the GNU General Public License, created by Dr. Stuart Ferguson. He made the decision to release the source code to the public in the middle of 1999 and released a stable version a year and a half later. The product, formerly named SoftF/X, was renamed to OpenFX. The OpenFX featureset includes a full renderer and raytracing engine, NURBS support, kinematics-based animation, morphing, and an extensive plugin API. Plugin capabilities include image post processor effects such as lens flare, fog and depth of field. Animation effects such as explosions, waves and dissolves add to the flexibility of the program. Version 2.0 also features support for modern graphics cards with hardware GPU acceleration.

OpenFX supports the Win32 platform, including Windows 95, NT, 98, ME, 2000 and XP. It can run under Unix-based platforms by using the Wine compatibility layer.

Seamless3d

Seamless3d is open source 3D modeling software free and available for all under the MIT license. The models for the virtual reality world, Techuelife Island were created using Seamless3d technology. Techuelife Island is showcased by Blaxxun as an example of what is possible when using the interactive multi-user Blaxxun platform.

Many Seamless3d tutorials have been translated to French.

HISTORY

In 2001 Seamless3d was made freely available online as a C++ library. The library centered around the creation of animated single mesh avatars for the Blaxxun 3d multi-user platform. It allowed the user to create smooth shaped triangle meshes and join different meshes together with tangent matching surfaces at the joining edges using a C++ compiler. By February 2003 Seamless3d had been transformed into a GUI based 3d modelling application with a file format designed around VRML format. This allowed Seamless3d files to be edited using VrmlPad utilising its syntax checking.

In 2005 a script compiler was developed and in May 2006 Seamless3d was able to act as a web browser for seamless3d files containing complex scripted animations.

In 2006 a set of specialised nodes for creating simple shapes such as: Sphere, Cylinder, Cone, Torus, Box and Bezier Lathe were added to make Seamless3d easier for the novice to quickly make simple models.

In 2007 the animation interface was greatly simplified by the introduction of a specialised control panel called the Anim bar. Towards the end of 2007 NURBS were introduced for making shapes and for synthesizing sounds.

In 2010 NURBS control point animation, NURBS stitching and a number of other features to aid making movies were introduced.

BUILD NODE TECHNOLOGY

Seamless3d can be used as a mesh editor and an animator, however its key feature is its build node technology. Build nodes allow the user to perform a sequence of complex operations in real time whenever a control point in the 3d window is dragged.

NURBS SURFACE POLY EDITING (NSPE)

NSPE allows the user to hand edit the polygons on NURBS surfaces. This includes being able to drag the vertices anywhere along the NURBS surface as well as join the vertices together, break the vertices apart and color them. NSPE has a significant advantage over simply converting a NURBS surface to a polygon mesh for editing because NSPE lets the user be able continue to modify the NURBS surface for the hand edited polygon structure.

Because NSPE ensures that when a polygon's vertex is dragged it will always be on the NURBS surface, NSPE greatly helps the user to avoid unintentionally changing the shape of the model when optimizing for real time animation.

FUSING NURBS SURFACES

By including a FuseSurface feature designed for fusing 2 NURBS surfaces together, Seamless3d allows for the creation of smooth continuous curvy models made from multiple NURBS surfaces.

SEAMLESSSCRIPT

Seamless3d has its own built in script compiler which compiles SeamlessScript (a very fast light weight scripting language) into native machine code. SeamlessScript is designed to look and feel a lot like JavaScript while being able to be compiled by a standard C++ compiler. This allows the user to develop complex animation sequences using a C++ IDE which gives the user access to professional debugging aids such as single step execution.

SEAMLESS3D CHAT

The Multi-User Seamless3d chat server designed for 3D World Wide Web browsing is open source under the MIT license and can be compiled for both Linux and Windows. Currently the Seamless3d modeller is used as the 3D chat client.

An online Seamless3d chat server has been in continuous service since April 2009. The general public can freely use it for their own custom made worlds and avatars.

FEATURES

- Exports to VRML, X3D (including H-Anim), OBJ and POV-Ray formats

- Imports VRML and X3D VRML Classic formats
- Imports Canal/Blaxxun Avatar Studio avatars
- Imports H-Anim
- Imports and Exports Biovision Hierarchy Motion Capture (BVH) files
- Support for FFmpeg which allows for the creation of AVI, MPG, MP4 and FLV movie formats
- Transform hierarchies
- Morphing
- Skinned animation
- Texture mapping
- JPEG and PNG texture formats (and BMP when using DirectX)
- Béziers & NURBS lathes and NURBS patches
- Tangent matched NURBS Surface Fusion
- Nurbs Surface Poly Modeling (NSPE)
- Software robot demonstration help
- Scripting
- Key-frame based and Script based animation
- Sound synthesis using NURBS
- Seamless3d files are a compact human readable text format
- Multi-User 3D chat web browsing

Wings 3D

Wings 3D is a free, open source, subdivision modeler inspired by Nendo and Mirai from Izware. Wings 3D is named after the winged-edge data structure it uses internally to store coordinate and adjacency data, and is commonly referred to by its users simply as Wings. Wings 3D is available for most platforms, including Windows, Linux and Mac OS X, using the Erlang environment.

Wings 3D can be used to model and texture low to mid-range polygon models. Wings does not support animations and has only basic OpenGL rendering facilities, although it can export to external rendering software such as POV-Ray and YafRay. Still, Wings is often used in combination with other software, whereby models made in Wings are exported to applications more specialized in rendering and animation such as Blender.

INTERFACE

Wings 3D uses context sensitive menus as opposed to a highly graphical, icon oriented interface. Modeling is done using the mouse and keyboard to select and modify different aspects of a model's geometry in four different selection modes: Vertex, Edge, Face and Body. Because of Wings' context sensitive design, each selection mode has its own set of mesh tools. Many of these tools offer both basic and advanced uses, allowing users to specify vectors and points to change how a tool will affect their model. Wings also allows you to add textures and materials to your models, and has built-in AutoUV mapping facilities.

FEATURES

- A wide variety of Selection and Modeling Tools
- Modeling Tool support for Magnets and Vector Operations

- Customizable Hotkeys and Interface
- Tweak Mode lets you make quick adjustments to a mesh
- Assign and edit Lighting, Materials, Textures, and Vertex Colours
- AutoUV Mapping
- Ngon mesh support
- A Plugin Manager for adding and removing plugins
- Import and Export in many popular formats

SUPPORTED FILE FORMATS

Wings loads and saves models in its own format (.wings), but also supports several standard 3D formats as well.

8

Engineering Drawing

An engineering drawing, a type of technical drawing, is created within the technical drawing discipline, and used to fully and clearly define requirements for engineered items.

OVERVIEW

Engineering drawings are usually created in accordance with standardized conventions for layout, nomenclature, interpretation, appearance (such as typefaces and line styles), size, etc. One such standardized convention is called GD&T. Each field in the Fields of engineering will have its own set of requirements for the producing drawings in terms line weight, symbols, and technical jargon. Some fields of engineering have no GD&T requirements. The purpose of such a drawing is to accurately and unambiguously capture all the geometric features of a product or a component. The end goal of an engineering drawing is to convey all the required information that will allow a manufacturer to produce that component. Engineering drawings used to be created by hand using tools such as pencils, ink, straightedges, T-squares, French curves, triangles, rulers, scales, and erasers. Today they are usually done electronically with computer-aided design (CAD).

The drawings are still often referred to as "blueprints" or "bluelines", although those terms are anachronistic from a literal perspective, since most copies of engineering drawings that were formerly made using a chemical-printing process that yielded graphics on blue-colored paper or, alternatively, of blue-lines on white paper, have been superseded by more modern reproduction processes that yield black or multicolour lines on white paper.

The more generic term "print" is now in common usage in the U.S. to mean any paper copy of an engineering drawing.

The process of producing engineering drawings, and the skill of producing them, is often referred to as technical drawing or drafting, although technical drawings are also required for disciplines that would not ordinarily be thought of as parts of engineering.

ENGINEERING DRAWINGS: COMMON FEATURES

Drawings convey the following critical information:

- *Geometry* – the shape of the object; represented as views; how the object will look when it is viewed from various angles, such as front, top, side, etc.

- *Dimensions* – the size of the object is captured in accepted units.
- tolerances – the allowable variations for each dimension.
- *Material* – represents what the item is made of.
- *Finish* – specifies the surface quality of the item, functional or cosmetic. For example, a mass-marketed product usually requires a much higher surface quality than, say, a component that goes inside industrial machinery.

LINE STYLES AND TYPES

A variety of line styles graphically represent physical objects. Types of *lines* include the following:

- *visible* – are continuous lines used to depict edges directly visible from a particular angle.
- *hidden* – are short-dashed lines that may be used to represent edges that are not directly visible.
- *center* – are alternately long- and short-dashed lines that may be used to represent the axes of circular features.
- *cutting plane* – are thin, medium-dashed lines, or thick alternately long- and double short-dashed that may be used to define sections for section views.
- *section* – are thin lines in a pattern (pattern determined by the material being "cut" or "sectioned") used to indicate surfaces in section views resulting from "cutting." Section lines are commonly referred to as "cross-hatching."

- *phantom* - (not shown) are alternately long- and double short-dashed thin lines used to represent a feature or component that is not part of the specified part or assembly. E.g. billet ends that may be used for testing, or the machined product that is the focus of a tooling drawing.

Lines can also be classified by a letter classification in which each line is given a letter.

- Type A lines show the outline of the feature of an object. They are the thickest lines on a drawing and done with a pencil softer than HB.
- Type B lines are dimension lines and are used for dimensioning, projecting, extending, or leaders. A harder pencil should be used, such as a 2H.
- Type C lines are used for breaks when the whole object is not shown. They are freehand drawn and only for short breaks. 2H pencil
- Type D lines are similar to Type C, except they are zigzagged and only for longer breaks. 2H pencil
- Type E lines indicate hidden outlines of internal features of an object. They are dotted lines. 2H pencil
- Type F lines are Type F*[typo]* lines, except they are used for drawings in electrotechnology. 2H pencil
- Type G lines are used for centre lines. They are dotted lines, but a long line of 10–20 mm, then a gap, then a small line of 2 mm. 2H pencil
- Type H lines are the same as Type G, except that every second long line is thicker. They indicate the cutting plane of an object. 2H pencil

- Type K lines indicate the alternate positions of an object and the line taken by that object. They are drawn with a long line of 10–20 mm, then a small gap, then a small line of 2 mm, then a gap, then another small line. 2H pencil.

MULTIPLE VIEWS AND PROJECTIONS

In most cases, a single view is not sufficient to show all necessary features, and several views are used. Types of *views* include the following:

Orthographic Projection

The orthographic projection shows the object as it looks from the front, right, left, top, bottom, or back, and are typically positioned relative to each other according to the rules of either first-angle or third-angle projection.

- First angle projection is the ISO standard and is primarily used in Europe. The 3D object is projected into 2D "paper" space as if you were looking at an X-ray of the object: the top view is under the front view, the right view is at the left of the front view.
- Third angle projection is primarily used in the United States and Canada, where it is the default projection system according to BS 8888:2006, the left view is placed on the left and the top view on the top.

Not all views are necessarily used, and determination of what surface constitutes the front, back, top and bottom varies depending on the projection used.

Auxiliary Projection

An auxiliary view is an orthographic view that is projected into any plane other than one of the six principal views. These views are typically used when an object contains some sort of inclined plane. Using the auxiliary view allows for that inclined plane (and any other significant features) to be projected in their true size and shape. The true size and shape of any feature in an engineering drawing can only be known when the Line of Sight (LOS) is perpendicular to the plane being referenced.

Isometric Projection

The isometric projection show the object from angles in which the scales along each axis of the object are equal. Isometric projection corresponds to rotation of the object by ± 45° about the vertical axis, followed by rotation of approximately ± 35.264° [= arcsin(tan(30°))] about the horizontal axis starting from an orthographic projection view. "Isometric" comes from the Greek for "same measure". One of the things that makes isometric drawings so attractive is the ease with which 60 degree angles can be constructed with only a compass and straightedge. Isometric projection is a type of axonometric projection. The other two types of axonometric projection are:

- Dimetric projection
- Trimetric projection

Oblique Projection

An oblique projection is a simple type of graphical projection used for producing pictorial, two-dimensional images of three-dimensional objects:

- it projects an image by intersecting parallel rays (projectors)
- from the three-dimensional source object with the drawing surface (projection plan).

In both oblique projection and orthographic projection, parallel lines of the source object produce parallel lines in the projected image.

Perspective

Perspective is an approximate representation on a flat surface, of an image as it is perceived by the eye. The two most characteristic features of perspective are that objects are drawn:

- Smaller as their distance from the observer increases
- Foreshortened: the size of an object's dimensions along the line of sight are relatively shorter than dimensions across the line of sight.

SECTION VIEWS

Projected views (either Auxiliary or Orthographic) which show a cross section of the source object along the specified cut plane. These views are commonly used to show internal features with more clarity than may be available using regular projections or hidden lines. In assembly drawings, hardware components (e.g. nuts, screws, washers) are typically not sectioned.

SCALE

Plans are usually "scale drawings", meaning that the plans are drawn at specific ratio relative to the actual size of the place or object. Various scales may be used for different drawings in a set. For example, a floor plan may be drawn at 1:50 (1:48 or 1/4"=1'-0") whereas a detailed view may be drawn at 1:25 (1:24 or 1/2"=1'-0"). Site plans are often drawn at 1:200 or 1:100.

SHOWING DIMENSIONS

The required sizes of features are conveyed through use of *dimensions.* Distances may be indicated with either of two standardized forms of dimension: linear and ordinate.

- With *linear* dimensions, two parallel lines, called "extension lines," spaced at the distance between two features, are shown at each of the features. A line perpendicular to the extension lines, called a "dimension line," with arrows at its endpoints, is shown between, and terminating at, the extension lines. The distance is indicated numerically at the midpoint of the dimension line, either adjacent to it, or in a gap provided for it.
- With *ordinate* dimensions, one horizontal and one vertical extension line establish an origin for the entire view. The origin is identified with zeroes placed at the ends of these extension lines. Distances along the x- and y-axes to other features are specified using other extension lines, with the distances indicated numerically at their ends.

Sizes of circular features are indicated using either diametral or radial dimensions. Radial dimensions use an "R" followed by the value for the radius; Diametral dimensions use a circle with forward-leaning diagonal line through it, called the *diameter symbol*, followed by the value for the diameter. A radially-aligned line with arrowhead pointing to the circular feature, called a *leader*, is used in conjunction with both diametral and radial dimensions. All types of dimensions are typically composed of two parts: the *nominal* value, which is the "ideal" size of the feature, and the *tolerance*, which specifies the amount that the value may vary above and below the nominal.

- Geometric dimensioning and tolerancing is a method of specifying the functional geometry of an object.

SIZES OF DRAWINGS

Sizes of drawings typically comply with either of two different standards, ISO (World Standard) or ANSI/ASME Y14 (American), according to the following tables:

ISO A Drawing Sizes (mm)

A4	210 X 297
A3	297 X 420
A2	420 X 594
A1	594 X 841
A0	841 X 1189

ANSI/ASME Drawing Sizes (Inches)

A	8.5" X 11"
B	11" X 17"
C	17" X 22"
D	22" X 34"
E	34" X 44"

Other U.S. Drawing Sizes

D1	24" X 36"
E1	30" X 42"

The metric drawing sizes correspond to international paper sizes. These developed further refinements in the second half of the twentieth century, when photocopying became cheap. Engineering drawings could be readily doubled (or halved) in size and put on the next larger (or, respectively, smaller) size of paper with no waste of space. And the metric technical pens were chosen in sizes so that one could add detail or drafting changes with a pen width changing by approximately a factor of the square root of 2. A full set of pens would have the following nib sizes: 0.13, 0.18, 0.25, 0.35, 0.5, 0.7, 1.0, 1.5, and 2.0 mm. However, the International Organization for Standardization (ISO) called for four pen widths and set a colour code for each: 0.25 (white), 0.35 (yellow), 0.5 (brown), 0.7 (blue); these nibs produced lines that related to various text character heights and the ISO paper sizes. All ISO paper sizes have the same aspect ratio, one to the square root of 2, meaning that a document designed for any given size can be enlarged or reduced to any other size and will fit perfectly. Given this ease of changing sizes, it is of course common to copy or print a given document on different sizes of paper, especially within a series, e.g. a drawing on A3 may be enlarged to A2 or reduced to A4. The U.S. customary “A-size” corresponds to “letter” size, and “B-size” corresponds to “ledger” or “tabloid” size. There were also once British paper sizes, which went by names rather than alphanumeric designations. American National Standards Institute (ANSI) Y14.2, Y14.3, and Y14.5 are standards that are commonly used in the U.S.

TECHNICAL LETTERING

Technical lettering is the process of forming letters, numerals, and other characters in technical drawing. It is used to describe, or provide detailed specifications for, an object. With the goals of legibility and uniformity, styles are standardized and lettering ability has little relationship to normal writing ability. Engineering drawings use a Gothic sans-serif script, formed by a series of short strokes. Lower case letters are rare in most drawings of machines.

EXAMPLE OF AN ENGINEERING DRAWING

Here is an example of an engineering drawing (an isometric view of the same object is shown above). The different line types are colored for clarity.

- Black = object line and hatching
- Red = hidden line
- Blue = center line of piece or opening
- Magenta = phantom line or cutting plane line

Sectional views are indicated by the direction of arrows, as in the example above.

Computational Geometry

Computational geometry is a branch of computer science devoted to the study of algorithms which can be stated in terms of geometry. Some purely geometrical problems arise out of the study of computational geometric algorithms, and such problems are also considered to be part of computational geometry. The main impetus for the development of computational geometry as a discipline was progress in computer graphics and computer-aided design and manufacturing (CAD/CAM), but many problems in computational geometry are classical in nature, and may come from mathematical visualization. Other important applications of computational geometry include robotics (motion planning and visibility problems), geographic information systems (GIS) (geometrical location and search, route planning), integrated circuit design (IC geometry design and verification), computer-aided engineering (CAE) (programming of numerically controlled (NC) machines).

The main branches of computational geometry are:

- *Combinatorial computational geometry*, also called *algorithmic geometry*, which deals with geometric objects as discrete entities. A groundlaying book in the subject by Preparata and Shamos dates the first use of the term “computational geometry” in this sense by 1975.
- *Numerical computational geometry*, also called *machine geometry*, *computer-aided geometric design* (CAGD), or *geometric modeling*, which deals primarily with representing real-world objects in forms suitable for computer computations in CAD/CAM systems. This branch may be seen as a further development of descriptive geometry and is often considered a branch of computer graphics or CAD. The term “computational geometry” in this meaning has been in use since 1971.

COMBINATORIAL COMPUTATIONAL GEOMETRY

The primary goal of research in combinatorial computational geometry is to develop efficient algorithms and data structures for solving problems stated in terms of basic geometrical objects: points, line segments, polygons, polyhedra, etc.

Some of these problems seem so simple that they were not regarded as problems at all until the advent of computers. Consider, for example, the *Closest pair problem*:

- Given n points in the plane, find the two with the smallest distance from each other.

One could compute the distances between all the pairs of points, of which there are $n(n-1)/2$, then pick the pair with the smallest distance. This brute-force algorithm takes $O(n^2)$ time; i.e. its execution time is proportional to the square of the number of points. A classic result in computational geometry was the formulation of an algorithm that takes $O(n \log n)$. Randomized algorithms that take $O(n)$ expected time, as well as a deterministic algorithm that takes $O(n \log \log n)$ time, have also been discovered. Computational geometry focuses heavily on computational complexity since the algorithms are meant to be used on very large datasets containing tens or hundreds of millions of points.

For large data sets, the difference between $O(n^2)$ and $O(n \log n)$ can be the difference between days and seconds of computation.

PROBLEM CLASSES

The core problems in computational geometry may be classified in different ways, according to various criteria. The following general classes may be distinguished.

Static Problems

In the problems of this category, some input is given and the corresponding output needs to be constructed or found. Some fundamental problems of this type are:

- Convex hull: Given a set of points, find the smallest convex polyhedron/polygon containing all the points.
- Line segment intersection: Find the intersections between a given set of line segments.
- Delaunay triangulation
- Voronoi diagram: Given a set of points, partition the space according to which points is closest to the given points.
- Linear programming
- Closest pair of points: Given a set of points, find the two with the smallest distance from each other.
- Euclidean shortest path: Connect two points in a Euclidean space (with polyhedral obstacles) by a shortest path.
- Polygon triangulation: Given a polygon, partition its interior into triangles
- Mesh generation

The computational complexity for this class of problems is estimated by the time and space (computer memory) required to solve a given problem instance.

Geometric Query Problems

In geometric query problems, commonly known as geometric search problems, the input consists of two parts: the search space part and the query part, which varies over the problem instances. The search space typically needs to be preprocessed, in a way that multiple queries can be answered efficiently.

Some fundamental geometric query problems are:

- Range searching: Preprocess a set of points, in order to efficiently count the number of points inside a query region.
- Point location: Given a partitioning of the space into cells, produce a data structure that efficiently tells in which cell a query point is located.
- Nearest neighbor: Preprocess a set of points, in order to efficiently find which point is closest to a query point.
- Ray tracing: Given a set of objects in space, produce a data structure that efficiently tells which object a query ray intersects first.

If the search space is fixed, the computational complexity for this class of problems is usually estimated by:

- the time and space required to construct the data structure to be searched in
- the time (and sometimes an extra space) to answer queries.

For the case when the search space is allowed to vary, see “Dynamic problems”.

Dynamic Problems

Yet another major class is the dynamic problems, in which the goal is to find an efficient algorithm for finding a solution repeatedly after each incremental modification of the input data (addition or deletion input geometric elements). Algorithms for problems of this type typically involve dynamic data structures. Any of the computational geometric problems may be converted into a dynamic one, at the cost of increased processing time. For example, the range searching problem may be converted into the dynamic range searching problem by providing for addition and/or deletion of the points. The dynamic convex hull problem is to keep track of the convex hull, e.g., for the dynamically changing set of points, i.e., while the input points are inserted or deleted.

The computational complexity for this class of problems is estimated by:

- the time and space required to construct the data structure to be searched in
- the time and space to modify the searched data structure after an incremental change in the search space
- the time (and sometimes an extra space) to answer a query.

Variations

Some problems may be treated as belonging to either of the categories, depending on the context. For example, consider the following problem.

- Point in polygon: Decide whether a point is inside or outside a given polygon.

In many applications this problem is treated as a single-shot one, i.e., belonging to the first class. For example, in many applications of computer graphics a common problem is to find which area on the screen is clicked by a mouse cursor. However in some applications the polygon in question is invariant, while the point represents a query. For example, the input polygon may represent a border of a country and a point is a position of an aircraft, and the problem is to determine whether the aircraft violated the border. Finally, in the previously mentioned example of computer graphics, in CAD applications the changing input data are often stored in dynamic data structures, which may be exploited to speed-up the point-in-polygon queries. In some contexts of query problems there are reasonable expectations on the sequence of the queries, which may be exploited either for efficient data structures or for tighter computational complexity estimates. For example, in some cases it is important to know the worst case for the total time for the whole sequence of N queries, rather than for a single query.

NUMERICAL COMPUTATIONAL GEOMETRY

This branch is also known as geometric modelling and computer-aided geometric design (CAGD). Core problems are curve and surface modelling and representation. The most important instruments here are parametric curves and parametric surfaces, such as Bezier curves, spline curves and surfaces. An important non-parametric approach is the level set method.

Application areas include shipbuilding, aircraft, and automotive industries. The modern ubiquity and power of computers means that even perfume bottles and shampoo dispensers are designed using techniques unheard of by shipbuilders of 1960s.