Course- Computer Graphics (Third Year Computer Graphics)

Chapter 03- Introduction to the rendering process with OpenGL

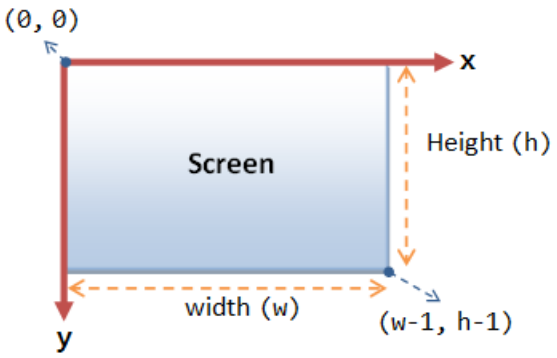
In the last tutorial we learned how we can use matrices to our advantage by transforming all vertices with transformation matrices. OpenGL expects all the vertices that we want to become visible, to be in normalized device coordinates after each vertex shadier run. That is, the x, y and z coordinates of each vertex should be between -1.0 and 1.0; coordinates outside this range will not be visible. What we usually do, is specify the coordinates in a range we configure ourselves and in the vertex shader transform these coordinates to NDC. These NDC coordinates are then given to the rasterizer to transform them to 2D coordinates/pixels on your screen.

Transforming coordinates to NDC and then to screen coordinates is usually accomplished in a step-by-step fashion where we transform an object's vertices to several coordinate systems before finally transforming them to screen coordinates. The advantage of transforming them to several *intermediate* coordinate systems is that some operations/calculations are easier in certain coordinate systems as will soon become apparent. There are a total of 5 different coordinate systems that are of importance to us:

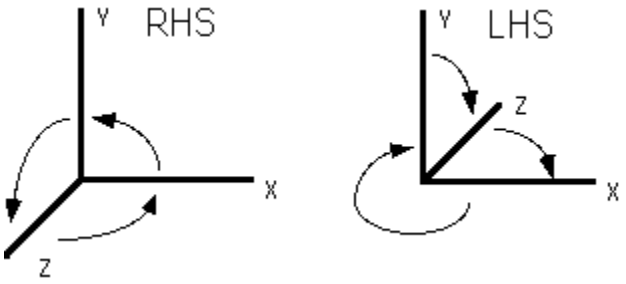
- Local space (or Object space)
- World space
- View space (or Eye space)
- Clip space
- Screen space

Those are all a different state at which our vertices will be transformed in before finally ending up as fragments.

You're probably quite confused by now by what a space or coordinate system actually is so we'll explain them in a more understandable fashion by showing the total picture and what each specific space actually does.



The 2D Screen Coordinates: The origin is located at the top-left corner, with x-axis pointing left and y-axis pointing down.



In a 2-D coordinate system the X axis generally points from left to right, and the Y axis generally points from bottom to top. (Although some windowing systems will have their Y coordinates going from top to bottom.)

When we add the third coordinate, Z, we have a choice as to whether the Z-axis points into the screen or out of the screen:

Right Hand Coordinate System (RHS)

Z is coming out of the page

Counterclockwise rotations are positive

if we rotate about the X axis: the rotation Y->Z is positive

if we rotate about the Y axis: the rotation Z->X is positive

if we rotate about the Z axis : the rotation X->Y is positive

Left Hand Coordinate System (LHS)

Z is going into the page

Clockwise rotations are positive

if we rotate about the X axis: the rotation Y->Z is positive

if we rotate about the Y axis: the rotation Z->X is positive

if we rotate about the Z axis : the rotation X->Y is positive

So basically its the same thing ...

The important thing to note is what coordinate system is being used by the package you are working with, both for the creation of models and the displaying of them. Also note that if the two packages use different coordinate systems, then the model(s) may need to be inverted in some fashion when they are loaded in for viewing.

OpenGL generally uses a right-hand coordinate system.

3.2.1 3D Coordinates

A coordinate system is a way of assigning numbers to points. In two dimensions, you need a pair of numbers to specify a point. The coordinates are often referred to as *x* and *y*, although of course, the names are arbitrary. More than that, the assignment of pairs of numbers to points is itself arbitrary to a large extent. Points and objects are real things, but coordinates are just numbers that we assign to them so that we can refer to them easily and work with them mathematically. We have seen the power of this when we discussed transforms, which are defined mathematically in terms of coordinates but which have real, useful physical meanings.

In three dimensions, you need three numbers to specify a point. (That's essentially what it means to be three dimensional.) The third coordinate is often called *z*. The *z*-axis is perpendicular to both the *x*-axis and the *y*-axis.

This demo illustrates a 3D coordinate system. The positive directions of the *x*, *y*, and *z* axes are shown as big arrows. The *x*-axis is green, the *y*-axis is blue, and the *z*-axis is red. You can drag on the axes to rotate the image.

This example is a 2D image, but it has a 3D look. (The illusion is much stronger if you rotate the image.) Several things contribute to the effect. For one thing, objects that are farther away from the viewer in 3D look smaller in the 2D image. This is due to the way that the 3D scene is "projected" onto 2D. We will discuss projection in the next section. Another factor is the "shading" of the objects. The objects are shaded in a way that imitates the interaction of objects with the light that illuminates them.

OpenGL programmers usually think in terms of a coordinate system in which the *x*- and *y*-axes lie in the plane of the screen, and the *z*-axis is perpendicular to the screen with the positive direction of the *z*-axis pointing **out of** the screen towards the viewer. Now, the default coordinate system in OpenGL, the one that you are using if you apply no transformations at all, is similar but has the positive direction of the *z*-axis pointing **into** the screen. This is not a contradiction: The coordinate system that is actually used is arbitrary. It is set up by a transformation. The convention in OpenGL is to work with a coordinate system in which the positive *z*-direction points toward the viewer and the negative *z*-direction points away from the viewer. The transformation into default coordinates reverses the direction of the *z*-axis.

This conventional arrangement of the axes produces a right-handed coordinate system. This means that if you point the thumb of your right hand in the direction of the positive *z*-axis, then when you curl the fingers of that hand, they will curl in the direction from the positive *x*-axis towards the positive *y*-axis. If you are looking at the tip of your thumb, the curl will be in the

counterclockwise direction. Another way to think about it is that if you curl the figures of your right hand from the positive *x* to the positive *y*-axis, then your thumb will point in the direction of the positive *z*-axis. The default OpenGL coordinate system (which, again, is hardly ever used) is a left-handed system.

All of that describes the natural coordinate system from the viewer's point of view, the so-called "eye" or "viewing" coordinate system. However, these eye coordinates are not necessarily the natural coordinates on the world. The coordinate system on the world—the coordinate system in which the scene is assembled—is referred to as world coordinates.

Recall that objects are not usually specified directly in world coordinates. Instead, objects are specified in their own coordinate system, known as object coordinates, and then modeling transforms are applied to place the objects into the world, or into more complex objects. In OpenGL, object coordinates are the numbers that are used in the *glVertex** function to specify the vertices of the object. However, before the objects appear on the screen, they are usually subject to a sequence of transformations, starting with a modeling transform.

So far we've been working in the 2-D coordinate system.…With positive X increasing to the right of the screen.…And positive Y increasing to the top of the screen.…The Z-axis has been left out so far.…OpenGL is a right-handed coordinate system.…So if you take your right-hand,…and point your thumb to the right and your index finger up.…Your middle finger will point out at you.…That's positive Z.…Thus, positive Z comes out of the screen…in the right-handed coordinate system.…Many a programmer has been tricked…by the right-left hand foolery.…

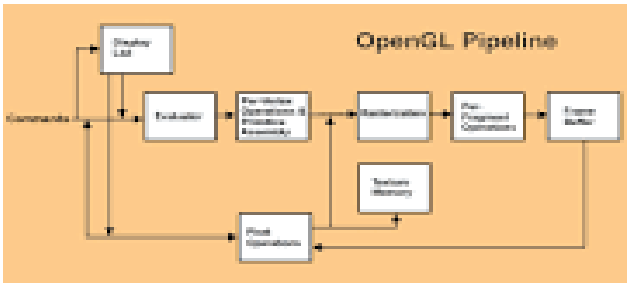
So it's best to keep in mind,…positive Z comes out towards you.…Away from you is negative.…This means that if you wanna move forward into the screen…you need to move in the negative Z direction.…This can be counter-intuitive…so it's good to keep in mind.…So before we get started on talking about transformations,…and navigation in our scene.…Keep in mind, this central fact.…Positive Z comes out towards you.…

What is OpenGL?

It is a window system independent, operating system independent graphics rendering API which is capable of rendering high-quality color images composed of geometric and image primitives. OpenGL is a library for doing computer graphics. By using it, you can create interactive applications which render high-quality color images composed of 3D geometric objects and

images. As OpenGL is window and operating system independent. As such, the part of application which does rendering is platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform working on.

OpenGL Architecture



This important diagram represents the flow of graphical information, as it is processed from CPU to the frame buffer. There are two pipelines of data flow. The upper pipeline is for geometric, vertex-based primitives. The lower pipeline is for pixel-based, image primitives. Texturing combines the two types of primitives together.

OpenGL as a Renderer

OpenGL is a library for rendering computer graphics. Generally, there are two operations that is done with OpenGL:

• draw something

• change the state of how OpenGL draws

OpenGL has two types of things that it can render: geometric primitives and image primitives. *Geometric primitives* are points, lines and polygons. *Image primitives* are bitmaps and graphics images. Additionally, OpenGL links image and geometric primitives together using *texture mapping*.

The other common operation that you do with OpenGL is *setting state.* "Setting state" is the process of initializing the internal data that OpenGL uses to render your primitives. It can be as simple as setting up the size of points and color that you want a vertex to be, to initializing multiple mipmap levels for texture mapping.

OpenGL and Related APIs

OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this.

Some examples are:

• GLX for the X Windows system, common on Unix platforms

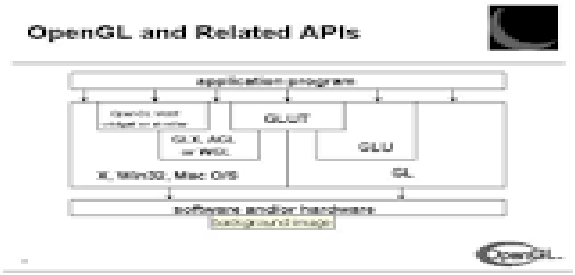
• AGL for the Apple Macintosh

• WGL for Microsoft Windows

OpenGL also includes a utility library, GLU, to simplify common tasks such as:

rendering quadric surfaces (i.e. spheres, cones, cylinders, etc.), working with NURBS and curves, and concave polygon tessellation.

Finally to simplify programming and window system dependence, we'll be using the freeware library, GLUT. GLUT, written by Mark Kilgard, is a public domain window system independent toolkit for making simple OpenGL applications. It simplifies the process of creating windows, working with events in the window system and handling animation.



2. The Synthetic Camera

Often in computer graphics it is useful to view the geometric model in a specific way, that mimics the image captured by a physical still or motion picture camera. While the equations describing the exact path of light through the lens and to the film for physical cameras is complex, they can be simplified to a set of equations that work quite well.

In creating **photorealistic** images, half of the problem is to correctly reproduce camera properties such as *field of view*, *depth of field*, *exposure*, *image aspect ratio*; the other half is to correctly reproduce the physics of light on materials comprising the objects in the scene. OpenGL and Direct3D offer some support to implement these two tasks. We will return to these

topics later, but for now we must start with the basics, such as determining the **lens** characteristics, **camera position**, and its **orientation**.

2.1 Camera Lens

The purpose of the lens is to gather light and focus it on an image plane, which might have film, an electronic CCD sensor for video, or be an abstract entity like a **pixel** A pixel stands for Picture Element (IBM calls them PELs).

The object of the exercise is to generate a rendering of a 3D scene, just as you might take a photograph. The DATA within the computer program act as objects in the real world. What, then, takes the place of our EYE or CAMERA? What about the camera Lens? and film?

The rendering process, or graphics pipeline, takes the place of the optics and sensory nerves of the eye, or of the lens, shutter, and film of the camera. However, as with a real camera, the image produced by the graphics pipeline depends of several factors, including:

- The Position of the camera in space,
- The Orientation of the camera (the direction in which it is facing),
- The Projection type (analagous to the lens of the camera), and
- The kind of 'film' that stores the image (in this case, a product of the rendering process)

"View Vector" (aka "Line of Sight") ≈ position + orientation

Imagine a vector from the eye or camera (the "eye point") to a point in the scene (the "focal point" or "center of view"). With one assumption (that the camera's "up" direction aligns with the world's "up" direction), this vector completely specifies the *Position* and *Orientation* for a camera. The focal point will turn out to be in the center of the image, and the view-vector's relationship to the model geometry determines whether the program produces a 1-point, 2-point or 3-point perspective (see below).

"Projection" ≈ Lens Length

A camera, mounted on a rigid tripod, pointed at a fixed scene, can still record radically different images if we change the lens attached to the camera. The lens changes the way in which light is focused on the film. That is, it changes the way light is **projected**. In the same way, a given view-vector can be used to produce different images of your model geometry though changes to the projection used. The common projections are **perspective** and **parallel**.

2.2 Cameras v. Synthetic Cameras v. Camera Analogies

Photographers may have noticed that several traditional concerns are missing from this discussion.

- The depth of field (with only a few exceptions, renderings have infinite depth of field, and it's all in focus). However, check out the "frustum" discussion.
- Shutter-speed and f-stop (since the digital world doesn't change while the rendering is in progress "motion blur" has to be added back in where needed (rare), and light levels are controlled almost exclusively through controlling the number and intensity of light sources).
- Lens-flare and chromatic aberration don't appear as issues.

These issues arise due to the chemistry of film or the physics of lens optics. A few true "synthetic camera" rendering systems do take such things into account, or simulate them through lightening/darkening of the image. The discussion here might be better understood as a "camera analogy" but that feels awkward.

3.3.1 ATTRIBUTES

LINE ATTRIBUTES

Basic attributes of a straight line segment are its type, its width, and its color. In some graphics packages, lines can also be displayed **using** selected pen or brush options.

Line Type

line-type attribute - solid lines, dashed lines, and dotted lines.

We modify a line drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path.

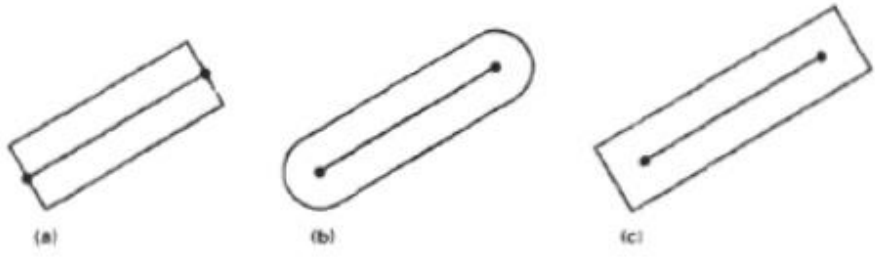


Figure 4-5
Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

A miter join is accomplished by extending the outer boundaries of each of the two lines until they meet.
A round join is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line width.
And a bevel join is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet.

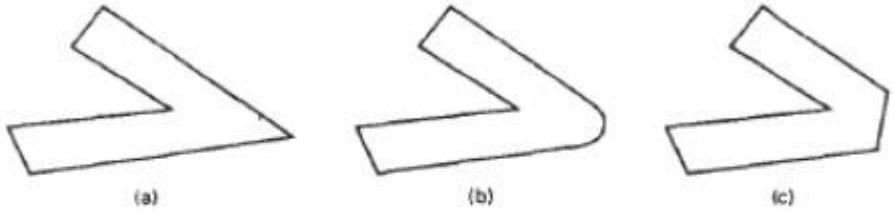


Figure 4-6
Thick line segments connected with (a) miter join, (b) round join, and (c) bevel join.

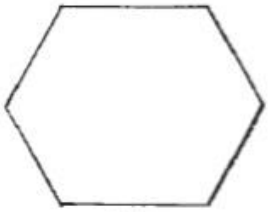
Line Color

When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A poly line routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the set pixel procedure.

The number of color choices depends on the number of bits available per pixel in the frame buffer. We set the line color value in **PHlCS** with the function Set Polyline ColourIndex (le)
CURVE ATTRIBUTES

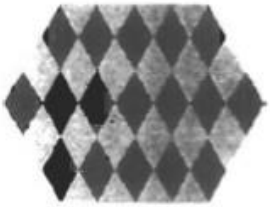
Parameters for curve attributes are the **same** as those for line segments. We can display curves with varying colors, widths, dotdash patterns, and available pen or brush options.

AREA FILL ATTRIBUTES



Hollow
(a)

Solid
(b)



Patterned
(c)

Figure 4-18

AREA-FILL ATTRIBUTES

Options for filling a defined region include a choice between a solid color or a patterned fill and choices for the particular colors and patterns. These fill options can be applied to polygon regions or to areas defined with curved boundaries, depending on the capabilities of the available package. In addition, areas can be painted using various brush styles, colors, and transparency parameters.

Fill Styles

Areas are displayed with three basic fill styles: hollow with a color border, filled with a solid color, or filled with a specified pattern or design. A basic fill style is selected in a PHIGS program with the function

```
setInteriorStyle (fs)
```

Values for the fill-style parameter `fs` include *hollow*, *solid*, and *pattern* (Fig. 4-18). Another value for fill style is *hatch*, which is used to fill an area with selected hatching patterns—parallel lines or crossed lines—as in Fig. 4-19. As with line attributes, a selected fill-style value is recorded in the list of system attributes and applied to fill the interiors of subsequently specified areas. Fill selections for parameter `fs` are normally applied to polygon areas, but they can also be implemented to fill regions with curved boundaries.

Hollow areas are displayed using only the boundary outline, with the interior color the same as the background color. A solid fill is displayed in a single color up to and including the borders of the region. The color for a solid interior or for a hollow area outline is chosen with

```
setInteriorColourIndex (fc)
```

where fill-color parameter `fc` is set to the desired color code. A polygon hollow fill

The width only of text can be set with the function

```
setCharacterExpansionFactor (cw)
```

where the character-width parameter cw is set to a positive real value that scales the body width of characters. Text height is unaffected by this attribute setting. Examples of text displayed with different character expansions is given in Fig. 4-27.

Spacing between characters is controlled separately with

```
setCharacterSpacing (cs)
```

where the character-spacing parameter cs can be assigned any real value. The value assigned to cs determines the spacing between character bodies along print lines. Negative values for cs overlap character bodies; positive values insert space to spread out the displayed characters. Assigning the value 0 to cs causes text to be displayed with no space between character bodies. The amount of spacing to be applied is determined by multiplying the value of cs by the character height (distance between baseline and capline). In Fig. 4-28, a character string is displayed with three different settings for the character-spacing parameter.

width 0.5

width 1.0

width 2.0

Figure 4-27
The effect of different character-width settings on displayed text.

Pattern Fill

We select fill patterns with

```
setInteriorStyleIndex (pi)
```

where pattern index parameter `pi` specifies a table position. For example, the following set of statements would fill the area defined in the `fillArea` command with the second pattern type stored in the pattern table:

```
setInteriorStyle (pattern);
setInteriorStyleIndex (2);
fillArea (n, points);
```

Separate tables are set up for hatch patterns. If we had selected *hatch* fill for the interior style in this program segment, then the value assigned to parameter `pi` is an index to the stored patterns in the hatch table.

For fill style *pattern*, table entries can be created on individual output devices with

```
setPatternRepresentation (ws, p.. nx, ny, cp)
```

TABLE 4-3
A WORKSTATION PATTERN TABLE WITH TWO ENTRIES, USING THE COLOR CODES OF TABLE 4-1

Index (pi)	*Pattern (cp)*
1	$\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

CHARACTER ATTRIBUTES

The appearance of displayed characters is controlled by attributes such as font, size, color, and orientation. Attributes can be set for entire character strings (text) and for individual characters defined as marker symbols.

TEXT ATTRIBUTES

There are a great many text options that can be made available to graphics programmers. First of **all,** there is the choice of font (or typeface), which is a set of characters with a particular design style such as New York, Courier, Helvetica, London, 'Times Roman, and various special symbol groups.

ANTIALIASING

- Displayed primitives generated by the raster algorithms have a jagged, or stair step, appearance because the sampling process digitizes coordinate points on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called aliasing.

We can improve the appearance of displayed raster lines by applying antialiasing methods that compensate for the under sampling process.To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the Nyquist sampling frequency (or Nyquist sampling rate) fs:

$$f_s = 2f_{max}$$

Another way to state this is that the sampling interval should be no larger than one-half the cycle interval (called the Nyquist sampling interval). For x-interval sampling, the Nyquist sampling interval Ax, is

3.3.2 Output Primitives

Here describes how an output primitive is drawn. For example, line type and line width scale factor are aspects of the polyline primitive. Entries in the PHIGS traversal state list define the *attributes* of the output primitive. The attributes define how the aspects of the output primitive are linked or bound to the output primitive. In the simplest case described so far, the attribute controlled by SET LINETYPE directly controls the line type aspect. More complex control is possible than in this simple example.

The coordinates used in the definition of structure elements are called *modelling coordinates*. The application can choose the coordinate system most appropriate to its needs. On structure traversal, modelling transformations may be applied to the coordinates so that the coordinates of the output primitive created are different from the coordinates in the structure element. PHIGS calls this coordinate system *world coordinates*.

Structure traversal occurs when a structure is posted to a workstation. The structure traversal creates a graphical scene in world coordinates which is viewed by the specified workstation. The viewing process itself is quite complex (particularly in 3D). Suffice it to say that the world coordinate scene is mapped to a *picture* in Normalized Projection Coordinates (NPC) and that picture or some part of it is displayed on the workstation.

The basic elements constituting a graphic are called output primitives. GRPH1 has the following output primitives.

- Polyline
- Polymarker
- Text
- Tone

There are other secondary primitives besides these.

- Line:
- Arrow

Five of these six primitives (excluding the tone primitive) all consist of line segments. Line segments have two attributes: the line index and line type. A tone has the attribute of tone pattern index. These attributes are defined as follows.

Coordinates handled by GRPH1 include curvilinear coordinates, so it is not appropriate in some cases to connect coordinates specifying line segments and boundaries of tone patterns with straight lines. Therefore, a function is available that interpolates between the coordinates by dividing the space between the coordinates into several sections. Furthermore, all primitives are subject to clipping at a certain boundary

1.7.1 Line Indexes

The thickness and color of a line segment drawn by GRPH1 is specified with a 3-digit integer (nnm) called the line index. Even when outputting to a device that can only change either the thickness or color, two lines with different line indexes can be distinguished because the line indexes follow the rules below for thickness and color.

In a system where both the thickness and color can be changed, the first 2 digits (nn=0- 99) indicate the color index, and the last digit (m=0- 9) represents thickness. The standard color indexes from 1 to 5 indicate

1: white or black (foreground)

2: red

3: green

4: blue

5: yellow

Colors specified by larger indexes are defined in the color map file. The line thickness is specified by 1(fine) - 9(thick). With an output device in which only the line thickness can be changed, nn is read as m only when m=0. With an output device in which only the color can be

changed, m is read as nn only when nn=0. Therefore, except in cases where the line thickness and color needs to be specified, specification with a single-digit index will make the lines distinguishable in a device-dependent method.

1.7.2 Line Types

Line types indicate solid, dashed, and doted lines. The following line types are allotted to numbers 1 through 4.

1: solid

2: dashed

3: dotted

4: alternate long and short dash

Structures Modelling Coordinates Scene World Coordinates Device Independent Picture Normalized Projection Coordinates Display Device Coordinates Device Dependent.

In the following sections, the structure elements associated with output primitives will be described giving the meaning of the output primitive created from the structure element on traversal. The description will be informal in that the function that creates the structure element will be described as though it created the output primitive directly whereas this only happens on traversal.

The output primitives are:

1. *polyline*: which draws a sequence of connected line segments;
2. *polymarker*: which marks a sequence of points with a symbol;
3. *fill area*: which defines the boundary of an area to be displayed;
4. *fill area set*: which defines the boundaries of a set of areas to be displayed as one;
5. *text*: which draws a sequence of characters;
6. *annotation text*: which draws a sequence of characters to annotate a drawing;
7. *cell array*: which displays an image;
8. *generalized drawing primitive*: which provides non-standard facilities in a standard way.

For the following examples, the boundary of the figures extends from 0 to 12 in the X-direction and 0 to 8 in the Y -direction.

3.2.3 Line types

The complete table of linetypes for PHIGS is given below:

1 solid

2 dashed

3 dotted

4 dashed-dotted

3.2.4 Line width scale factor

Due to the large variability of output devices, PHIGS has taken the decision that it is not feasible to precisely define the width of a line in the modelling coordinates that define the points of the polyline. One problem is that the aspect ratio may be changed which makes it difficult to decide which of the 3 dimensions should be used to specify width. Secondly, several instances of the same set of polylines may be required at different sizes but with the same line thickness. In consequence, line thickness is defined relative to some notional line thickness that is regarded as normal on the device. For a plotter, this is likely to be the line thickness produced by drawing with the standard pens. For a vector refresh display, it would be the standard intensity line. For a raster device, it will be the number of pixels that produce a clear line on the display.

Line width is specified relative to this nominal line width. The value of line width scale factor can be less than or greater than 1 and is a real number. For example, line width scale factor can be set to 0.5 of the nominal line width or 2.5 times the nominal line width. The device is expected to take the nominal line width and multiply by the line width scale factor and attempt to draw the lines as close to that width as possible. Figure 3.4 gives some examples of line widths that might be available where the second from the bottom might be the nominal line width.

Figure 3.4: Line width scale factors

3.2.5 Colour index

The third aspect of a polyline is its colour. In PHIGS, colour is specified indirectly by defining a *colour index* that points to a colour description. The *polyline colour index* is the aspect that is associated with polyline output primitives.

Because colour is quite closely connected to the device that will display the output, the table containing the colour descriptions is stored on the workstation. The number of entries in the table will depend on the workstation. Entries 0 and I always exist. Entry 0 is defined as the

background colour of the display and entry 1 is initially set to the default foreground colour of the display. Individual colours can be specified using a number of different colour models.

As the contents of the workstation colour table may be different on different workstations, a polyline colour index value may produce different colours on different workstations depending on the entries in the workstation colour tables. For the same colour to appear on each workstation for a specified index value, the colour description associated with that polyline colour index value must be the same on each workstation. Even then it will require the workstations to have similar colour displays.

Other primitives have their own colour aspect. For example, polymarkers have a polymarker colour index aspect. As there is a single colour table on a workstation, if the polyline and polymarker colour index aspects are the same, the same colour will be produced on a workstation.

3.3.3 Marker type

The complete table of marker types for PHIGS is given below:

1 dot

2 plus

3 star

4 circle

5 cross

Marker type 1 is the smallest available dot and may well be difficult to see without looking carefully. We have deliberately increased its size in Figure 3.6 so that it is visible! Registration and implementation dependent marker types are handled in the same way.

3.3.4 Marker size scale factor

PHIGS does not treat marker size as a geometric value that is settable in modelling coordinates. Instead, it is assumed that the device has some notional size for markers. The aspect markersize scale factor defines the size of the marker relative to this notional size. Consequently. These markers are 3, 5, 7, 9, and 11 times the nominal marker size for this device.

The scale factor can be less than 1. If the implementation has defined the notional marker size greater than the minimum that it can draw, the smaller size marker will be output. Because an

implementation only has to approximate to the marker size (some devices may only be capable of drawing markers at certain sizes), the aspect should not be used when precise sizing is required. In this case. defining the glyph as a structure using polylines, fill area or fill area set is more appropriate.

3.4 FILL AREA AND FILL AREA SET

3.4.1 The functions

PHIGS provides four functions:

FILL AREA 3(N, XA, YA, ZA)

FILL AREA(N, XA, Y A)

FILL AREA SET 3(N, IA, XA, Y A, ZA)

FILL AREA SET(N, IA, XA, YA)

There are two output primitives, fill area and fill area set, each having a shorthand form for the case on the Z=0 plane. Both define an area to be filled with some rendering or pattern. Fill area specifies a single boundary to be filled whereas fill area set specifies a set of boundaries. The major reason for including fill area is for compatibility with GKS. However, it is a simpler primitive to use. The two primitives share a set of aspects but fill area set has some additional ones that control the appearance of the boundary edges.

The parameters for fill area are similar to those for polyline and poly marker defining a sequence of points where the last point is normally the same as the first point. If that is not the case, an additional point is added to the sequence at the end which is the same as the first point. This ensures that the sequence of points *always* defines a *closed* boundary.

Whereas polyline and poly marker are true 3-dimensional primitives with no constraints on the points, this is not the case for fill area or fill area set. Both area primitives define boundaries which are in a single plane. If by accident, the application defines a fill area or fill area set where the points are not all in a single plane, the implementation is allowed to interpret it how it likes. For example, it could let the first few points define the plane and project all the other points onto that plane or it could be more sophisticated and attempt to make the best fit by reasoning which points are incorrect.

If insufficient points are given to define a plane (less than 3), no error will occur but nothing will be visible on the display either. A simple example of fill area is:

Fill area set has the points (XA(I),YA(I),ZA(I)) for the first area defined by the values 1=1 to IA(1). The second area is defined by the points from IA(1)+1 to IA(2) and so on until the last area is defined by the sequence of points from IA(N-1)+1 to IA(N). For example:

```
    DATA XFAS /3,  6,  6,  3,  3,  2, 7,  7, 2, 2,  4, 5, 5, 4, 4/
    DATA YFAS /23,23, 18, 18, 23, 16, 16, 9, 9, 16, 7, 7, 4, 4, 7/
    DATA IA /5,10,15/
    DO 10 I=1,5
    XFAS(I)=0.333*XFAS(I)
    YFAS(I)=0.333*YFAS(I)
10  CONTINUE
    FILL AREA SET(3, IA, XFAS, YFAS)
```

Figure 3.9 shows, on the left, the three rectangles defined as a fill area set in the program above. The remaining parts of the diagram are described later.

3.4.2 Definition of interior

So far, the description has assumed that it is straightforward to decide what is inside the boundary and that is true when the boundary does not intersect itself. However, as soon as that is allowed, and it is in PHIGS, it is necessary to precisely define the inside and outside of the boundary as only the inside will be filled.

PHIGS defines the inside by a rule called the *even-odd* rule. Note that this is not the only rule that could be used and, for example, the X window system has two rules that can be used for specifying the inside.

The even-odd rule states that a point is inside the boundary if a line drawn from infinity to the point hits the boundary an odd number of times.

If the line happens to touch the boundary or intersects with the boundary at a point where it crosses itself, care must be taken to make sure the count remains correct. The best solution is to choose a line from infinity that only intersects with the boundary at straightforward positions!

3.4.3 Area aspects

Fill area and fill area set have six common aspects that control the appearance of the interior of the boundary. The main aspect is *interior style* which defines the type of rendering required. The remaining aspects are appropriate for some subset of the styles.

The area can be rendered in basically three ways, a *solid* colour, a *hatch* style or a *pattern*. The *interior colour index* aspect is used by the first two to define the colour of the solid area or the hatch lines. As both hatching and patterning can be done in a variety of ways, an *interior style index* aspect indicates which style has been chosen. Hatch styles are implementation dependent and the interior style index does not mandate which style is associated with which index. Much more control is defined for patterns. The interior style index for patterns points to a *pattern array* of colour index values to be mapped onto the area to be filled. Thus it does not use the interior colour index value. The three aspects *pattern size*, *pattern reference point* and *pattern vectors* define the size and orientation of the pattern. These apply to all pattern styles but are not used by solid or hatch styles.

The interior colour index aspect and the entries in the pattern array are defined in the same way as the colour index values for polylines and polymarkers. Each workstation provides a colour description for each index value.

3.4.4 Interior style

The interior style aspect has the values HATCH and PATTERN to specify which filling is required. The hatching used is specified by the interior style index. of possible hatch styles. None are mandated and the style available in an implementation for a specific interior style index is not defined.

The solid colour is specified by the interior style aspect having one of the values HOLLOW, SOLID or EMPTY. SOLID fills the interior with a uniform colour defined by the colour index value. HOLLOW is a quick and dirty rendering which just draws the outline of the fill area. It is provided as a quick debug aid. the way the boundary is represented is up to the workstation or implementation to define. There is no necessity for it to take account of, say, the polyline aspects like linetype. If the aspect is defined as EMPTY, no filling occurs and no boundary line is drawn. The most complex fill area style is interior style PATTERN. In this case, the type of pattern is defined by the interior style index which points to an array of colour indices.

3.4.5 Pattern aspects

For interior style PATTERN, the interior style index, sometimes called the pattern index, is a pointer into a pattern table which defines the set of patterns available to render the area to be filled.

PRP PW PH (2,1) (2,2) (1,1) (1,2)

The pattern is a 2-dimensional array of colour index values which define the repetitive pattern with which to render the area. Clearly, this does not completely define the form of the rendering as it is necessary to specify the origin from which the pattern is to start and the size of the pattern cells.

Even if the same pattern is used, the same area can have significantly different rendering depending on the size of the pattern cell and how its origin relates to the area to be filled. The main problem in the definition occurs if the origin or vectors do not occur in the plane of the fill area or fill area set. In this case, the points are projected onto the fill area or fill area set plane along a normal to that plane. Again, this ensures that it is difficult in PHIGS to generate an error in the definition of the pattern.

3.4.6 Edge aspects

The major difference between fill area and fill area set is that fill area set has aspects which define how the boundary of the fill area set is rendered. For fill area, no boundary is drawn unless fill area style HOLLOW is specified.

For fill area set, additional aspects provided are *edge flag, edgetype, edgewidth scale factor* and *edge colour index*. The edge flag can be set to ON and OFF. If set ON, the edge is rendered according to the other aspects. The meaning of these is similar to the meaning of the polyline aspects. Edge type corresponds to line type, edge width scale factor to line width scale factor and edge colour index to polyline colour index.

Edges are drawn on top of the interior. Conceptually, the two are disjoint. Thus, an edge drawn as a dashed line may be displayed with the boundary of a HOLLOW fill area being visible between dashes. If a fill area set is clipped with the edge flag set to ON, the new edges generated by the clipping will not be rendered. It is only those unclipped parts of the original boundary that will be rendered.

While it is quite easy to draw a polyline around a fill area by just one extra function invocation, it may require a series of polyline functions to do the same for fill area set. This is one of the reasons for adding the edge boundary to fill area set.

3.5 TEXT

3.5.1 The functions

Each defines output consisting of a sequence of characters, specified by CHARS, located with reference to a point called the *text position*. The second function is a 2D shorthand form of the first where output is with reference to the text position (PX,PY,0) in the Z=0 plane.

3.5.2 Text aspects

Text is the most complex of the output primitives in PHIGS with nine aspects to control the appearance on the display. Five aspects, *character height*, *character expansion factor*, *character spacing*, *text path* and *text font* define a *text extent rectangle* that encloses the character string in the text plane. The coordinates of the text extent rectangle are defined in the text local coordinate system. The aspect *character up vector* specifies the orientation of this text extent rectangle relative to the text local coordinate system axes. The aspect *text alignment* positions the text extent rectangle relative to the text position. Finally, *text colour index* defines the colour in the same way as polyline colour. The *text precision* aspect allows some of the above aspects to be ignored.

The text extent rectangle in the text plane will be transformed by any modelling transformations defined during traversal and will be clipped as required. In consequence, the most general case will be that the text extent rectangle is transformed to a parallelogram by the modelling transformation. This may result in the text being sheared.

3.5.3 Font specification

Any PHIGS implementation will provide the user with a number of differentiable character fonts. The font designer specifies the shape of each character relative to some local 2D coordinate system. Fonts can either be monospaced (all characters have the same width) or proportional (width depends on the character form).

The height of a character is defined as the distance from the base line to the cap line. The width of a character is defined as the distance from the left line to the right line. Normally, the font will be defined such that characters abutted together will have sufficient space between them for comfortable reading. Thus the width is usually greater than the width of the character form itself.
top cap half base bottom left centre right

The text font aspect is a number which specifies the font to be used on a workstation to draw the characters. Fonts numbered 1 and 2 have to be provided. Fonts numbered greater than 2 are subject to the Registration scheme. An implementation can define any number of fonts using

zero or negative numbers that are local to the implementation. Implementations frequently have a large number of fonts available (40 or more) so it is a point to look at when choosing an implementation. Font number 1 has to be a monospaced font and font number 2 must be differentiable from it. Both must define the characters in the ISO 646 standard. The requirements for both fonts 1 and 2 to be available was a late addition to the standard and earlier implementations of PHIGS sometimes do not comply with this requirement.

3.5.4 Defining the text extent rectangle

The TEXT or TEXT 3 function defines the sequence of characters to be output on the display. The text font aspect defines the font to be used. Different fonts define the relative widths of characters differently so it is necessary to define the text font to be used in order to identify the precise character bodies of the characters in the text string to be displayed.

The character height aspect defines the height of the character bodies to be displayed. The character body in the font description is expanded equally in both the X and Y directions until the specified character height is obtained.

The character expansion factor aspect provides additional expansion in the X-direction. If a value of 2.0 is specified, the width of the character body is doubled while leaving the height the same.

The character spacing aspect specifies how much additional spacing is required between two adjacent character bodies. A positive value will insert additional spacing as a fraction of the character height. A negative value of character spacing will cause the adjacent character bodies to overlap. Whether the spacing is applied in the top-bottom or left-right direction of the characters depends on the setting of the text path aspect. 0 E F 0.25 E F 0.5 E F 0.75 E F

If the text path aspect is set to RIGHT, the second character body in the sequence of characters to be displayed is placed to the right of the first character body with the additional spacing in between if defined. The third character is placed to the right of the second and so on. If the text path aspect is set to LEFT, the second character body is placed to the left of the first character body and so on. In both cases, the height of the text extent rectangle is defined by the character height and extends from the top line to the bottom line. The width of the text extent rectangle extends from the left line of the left-most character to the right line of the right-most character.

If the text path aspect is set to DOWN, the second character body is placed below the first character with the additional spacing, if defined, in between. The third character body is placed

below the second and so on. If the text path aspect is set to UP, the second character body is placed above the first character body and so on. In both cases, the height of the text extent rectangle is from the top-line of the top-most character to the bottom-line of the bottom-most character. The width is the same as the width of the widest character to be displayed.

As the fonts are defined separately on each workstation, the size of the text extent rectangle can be different on each workstation. This clearly causes problems if the application, say, is trying to surround the text with a rectangle drawn as a polyline. To ensure that this is possible, PHIGS insists that font number 1 is a monospaced font and that the aspect ratio for all the characters is the same for every workstation in the implementation. Consequently, if font number 1 is used, the rectangle around the text will appear the same on every workstation. If any other font is used, there is no guarantee, say, that the output on the display is the same as the output on an associated plotter which could have a different set of fonts defined.

3.5.5 Orientation

The character up vector aspect defines the orientation of the text extent rectangle (and the enclosed text) to the local text coordinate system in the plane by specifying the direction of the rectangle's height with respect to the local text coordinate system. For example, if the character up vector is set to (0,1), the vector is along the Y-axis of the local text coordinate system and so the text extent rectangle is aligned with the axes of the local text coordinate system. If the character up vector is set to (1,0), the up direction is along the X-axis of the local text coordinate system so the text extent rectangle is rotated by 90° clockwise.

It should be remembered that just by setting the character up vector aspect to (0,1) does not guarantee that the text will be in its normal orientation. The text plane defined may be at an angle to the horizontal and a modelling transformation applied at traversal may also rotate the text.

3.5.6 Alignment

The text primitive defines the text position. The text alignment aspect defines the position in the text extent rectangle that coincides with the text position. The alignment is relative to the coordinate system of the text extent rectangle, that is the Y -direction is the direction of the character up vector.

The text alignment aspect has a pair of values that control the positioning in the X and Y-directions. The X-component has four possible values, LEFT, CENTRE, RIGHT and NORMAL. The Y-component has six possible values, TOP, CAP, HALF, BASE, BOTTOM and NORMAL.

LEFT CENTRE RIGHT E F L E F L E F L

If the X-component has value LEFT, the left side of the text extent rectangle coincides with the text position. For RIGHT, the right side of the text extent rectangle coincides with the text position while for CENTRE the text position lies midway between the left and right sides of the text extent rectangle. Examples of alignment horizontally are given in Figure 3.20.

TOP CAP HALF BASE BOTTOM E j E j E j E j E j

If the Y -component has value TOP, the top of the text extent rectangle coincides with the text position and similarly if it has value BOTTOM, the bottom of the text extent rectangle coincides with the text position. The values CAP, HALF and BASE are really only useful for text defined with text path LEFT or RIGHT. However, they are defined in such a way that they are also well defined for text path UP and DOWN. If CAP is defined, the text position coincides with the cap-line of the top-most character (for LEFT and RIGHT paths this is all the characters). If BASE is defined, the text position coincides with the base-line of the bottom-most character. If HALF is defined, the text position coincides with the line half way between the half-lines of the top and bottom-most characters. In consequence, for text defined with text path set to LEFT or RIGHT, CAP, HALF, and BASE have their obvious meanings. Examples of vertical alignment are given in Figure 3.21. The NORMAL X and Y-alignment values define the most appropriate text alignment depending on the text path defined. The table below gives the values of NORMAL for the possible text paths.

TEXT	**NORMAL Alignment**	
PATH	**HORIZONTAL**	**VERTICAL**
RIGHT	**LEFT**	**BASE**
LEFT	**RIGHT**	**BASE**
UP	**CENTRE**	**BASE**
DOWN	**CENTRE**	**TOP**

The effect of a modelling transformation which does an expansion in the X-direction and a contraction in the Y-direction on a text primitive when the character up vector is set to (-1,1). Note that this can distort the character forms even in 2D. It also places quite a demand on the workstation which has to render the text primitive on the display.

W I F W I F Modelling Coordinates World Coordinates

Wife Wife Wife efiW

Figure 3.24: Different text paths and aspects

Figure 3.24 shows the same text string *Wife* output with text position set to the four corners of a central square with the letter W at the corners. The character up vector aspect is set to (-1,1) throughout. The top left text string is defined with text path set to UP. The top right is with text path set to RIGHT and character spacing set to 2.0. The bottom right has the text path set to DOWN and character expansion set to 3.0 while the bottom left has the text path set to LEFT, the character height double the others and a different font selected.

3.5.7 Text precision

It is possible that a workstation will have difficulty in reproducing all the text aspects described above. For example, hardware fonts on workstations very rarely allow character shearing. On some simple workstations, it is only possible to output text horizontally.

The text precision aspect in conjunction with the text font aspect specifies how closely the workstation can represent the text output. If the text precision aspect is set to STROKE, the workstation is expected to use all the aspects correctly. In some cases, this may mean that the characters have to be output by software line drawing or area filling.

If the text precision aspect is set to CHAR, the text extent rectangle and the individual character bodies are calculated precisely. However, the individual characters may not be precisely defined within the character body. For example, the correct character height may be used for the character but the character will be vertical irrespective of the value of the character up vector. If the text precision is set to STRING, a text string of approximately the right height and width is output but its alignment, orientation and path direction may be incorrect. This quick and dirty text can be used in debugging when the STROKE precision text is expensive to produce. STRING precision text is frequently used for error messages and operator prompts that do not fom1 a major part of the graphical output and where precision is less necessary as long as it is readable.

Every workstation must support at least two STROKE precision fonts (numbers I and 2). It is also assumed that CHAR and STRING precision is available for these fonts (not difficult as drawing at STROKE precision is an allowable definition). For other fonts, there is no guarantee that all precisions are supported. If a requested font is not available, text font I will be used at STRING precision.