Chapter 2

Drawing

This chapter describes how to draw graphics *primitives*. Primitives are basic geometric elements such as *points*, *lines*, and *polygons*. You can draw primitives with different colors and techniques. You can draw points with different sizes, lines with different widths and styles, and polygons with different patterns and filling methods.

- Section 2.1, "Drawing with the GL," describes how to draw geometric figures with GL subroutines.
- Section 2.2, "Old-Style Drawing," describes GL subroutines that were used for drawing in early releases of the GL, and is included for compatibility only.

2.1 Drawing with the GL

When you provide the specifications of a geometric figure, also called a *geometry*, the GL draws it on the screen right away. This is called *immediate mode*—the GL is drawing things as the drawing subroutines are called.

You describe a geometry in GL terms by specifying its *edges* and *corners*. Each corner is a *vertex* (a point in space). You specify the coordinates (position in space) of the vertices and the order in which the vertices are connected to form *edges*. Edges then connect to form the geometry. Edges connected as lines make a *wireframe* geometry; edges connected as polygons make a geometry with solid faces.

When you draw a geometry, you use one of the GL primitives to draw and connect the vertices. You mark the beginning and end of the vertex list with special bgn* and end* subroutines, which signify not only the beginning and end of the list of vertices, but also the type of primitive, as indicated by the asterisk (*). Which bgn* and end* subroutines you use depends on what kind of geometry you are drawing. You will see the bgn*/end* programming structure throughout the GL. It is analogous to the begin-end paradigm of modular programming languages.

Note: You need to call gflush() after the end* statement to complete the drawing process. If you have used the GL before, you may remember that this step was previously necessary only when you were using the DGL. Because the GL is network-transparent, all programs need to call gflush() after the last drawing command.

You tell the GL to begin drawing with a bgn*geometry* statement, where *geometry* denotes the primitive to use. You then specify a list of vertices, whose coordinates are of the appropriate type of vertex data, to connect in order to form the edges. Finally, you tell the GL to close the figure with an end*geometry* statement.

2.1.1 Vertex Subroutines

Specify a vertex list by calling the vertex() subroutine for each vertex between the bgn* and end* statements.

The content of the bgn*/end* modules have the format illustrated by this pseudocode:

```
bgngeometry();
   vtype(vertex 1);
   vtype(vertex 2);
   vtype(vertex .);
   vtype(vertex .);
   vtype(vertex n);
endgeometry();
gflush();
```

Note: No drawing is guaranteed to happen until gflush() is called.

The GL contains 12 forms of the `vertex()` subroutine—one for each possible data type of a vertex coordinate. This group of subroutines is known collectively as the `v()`, for vertex, subroutine.

You can specify vertex coordinates as short integers (16 bits), long integers (32 bits), single-precision–floating-point values (32 bits), and double-precision–floating-point values (64 bits). For each of these types, there is a `v()` subroutine for defining vertices in 2-D, 3-D, and 4-D, also called *homogeneous coordinates*.

Homogeneous coordinates are referred to as 4D because they use a fourth parameter in addition to the 3-D coordinates. Homogeneous coordinates are useful for matrix manipulations and other operations common to graphics.

All forms of the vertex subroutine begin with the letter v. The second character is 2, 3, or 4, indicating the number of dimensions, and the final character indicates the data type: *s* for short integer (16 bits), `i` for long integer (32 bits), `f` for single-precision floating point (32 bits), `d` for double-precision floating point (64 bits).

Table 2-1 lists the vertex subroutines.

Argument Type	2-D	3-D	4-D
16-bit integer	`v2s()`	`v3s()`	`v4s()`
32-bit integer	`v2i()`	`v3i()`	`v4i()`
32-bit floating point	`v2f()`	`v3f()`	`v4f()`
64-bit floating point	`v2d()`	`v3d()`	`v4d()`

Table 2-1 Vertex Subroutines

This sample program, *greensquare2.c*, demonstrates the use of some of the different vertex subroutines. This program draws a square with green lines.

```
#include <gl/gl.h>

short vert1[3] = {200, 200, 0};      /* lower left corner */
long vert2[2] = {200, 400};          /* upper left corner */
float vert3[2] = {400.0, 400.0};     /* upper right corner */
double vert4[3] = {400.0, 200.0, 0.0}; /* lower right corner */

main()
{
   prefsize(400, 400);
   winopen("greensquare2");
   ortho2(100.5, 500.5, 100.5, 500.5);
   color(WHITE);
   clear();
   color(GREEN);
   bgnline();
      v3s(vert1);
      v2i(vert2);
      v2f(vert3);
      v3d(vert4);
      v3s(vert1);
   endline();
   sleep(10);
   gexit();
   return 0;
}
```

You have seen the `prefsize()` and `winopen()` commands before. The `ortho2()` command sets up a coordinate system inside your window to allow you to see the output of the program at a reasonable size and perspective. See Chapter 7, “Coordinate Transformations,” for more information on `ortho2()`. The values have .5 added to them so that pixels are centered on whole numbers. This eliminates the potential problem of *roundoff error* from non-integer pixel locations causing the display to be shifted by one or more pixel.

Although it is unlikely that you would write a program like the one above, it does illustrate two things:

- Within one geometric figure, you can mix different types of vertices. In a typical application, all the vertices tend to have the same dimension and the same form.
- In the GL, all geometric figures are 3-D and the hardware treats them as such. 2-D versions of the vertex subroutines are actually shorthand for an equivalent 3-D subroutine with the *z* coordinate set to zero.

This sample program, *crisscross.c*, clears a window to white, and then draws a pair of intersecting red lines connecting its opposite corners.

```
#include <gl/gl.h>

long vert1[2] = {101, 101};       /* lower left corner */
long vert2[2] = {101, 500};       /* upper left corner */
long vert3[2] = {500, 500};       /* upper right corner */
long vert4[2] = {500, 101};       /* lower right corner */

main()
{
   prefsize(400, 400);
   winopen("crisscross");
   ortho2(100.5, 500.5, 100.5, 500.5);
   color(WHITE);
   clear();
   color(RED);
   bgnline();
      v2i(vert1);
      v2i(vert3);
   endline();
   bgnline();
      v2i(vert2);
      v2i(vert4);
   endline();
   sleep(10);
   gexit();
   return 0;
}
```

This example declares four long arrays *(vert1*, *vert2*, *vert3*, and *vert4*) and assigns values to all the elements of each array. The size of the next window to be opened is established by `prefsize()` as 400 pixels by 400 pixels. Next, `winopen()` opens a window with the title *crisscross*.

The `ortho2()` call sets up the coordinate system so that a point with coordinates *(x, y)* maps exactly to the point on the screen that has the same coordinates. The `ortho2()` command is discussed in Chapter 7. The `color()` call sets the current color to white and paints the window with the current color, which is white.

The next four lines of code draw a line from (101, 101) to (500, 500)—the lower-left corner to the upper-right corner. `bgnline()` tells the system to prepare to draw a line using the list of vertices immediately following it. `v2i()` takes an array of coordinates as its argument and creates vertices at the specified coordinates.

The first `v2i()` subroutine call creates the first endpoint of the line segment. The second `v2i()` subroutine call creates the other endpoint of the line segment and the system draws a line.

The `endline()` subroutine call tells the system that it has all the vertices for the line. The next four lines draw a line from (101, 500) to (500, 101)—the lower-right corner to the upper-left corner. The IRIX call *sleep(10)* delays the program from exiting until 10 seconds have elapsed.

You can use any of the five primitives described in the next five sections—*points*, *lines*, *closed lines*, *polygons*, and *meshes*—with the vertex subroutines.

2.1.2 Points

Use `bgnpoint()` and `endpoint()` to specify a vertex list that is drawn as a group of disconnected points.

This sample program, *pointpatch.c*, draws a set of points arranged in a square pattern. The square is 20 pixels wide by 20 pixels high, and the points are spaced 10 pixels apart.

```
#include <gl/gl.h>

main()
{
    long vert[2];
    int i, j;

    prefsize(400, 400);
    winopen("pointpatch");
    color(BLACK);
    clear();
    color(WHITE);
    for (i = 0; i < 20; i++) {
        vert[0] = 100 + 10 * i;    /* load the x coordinate */
        bgnpoint();
        for (j = 0; j < 20; j++) {
            vert[1] = 100 + 10 * j;/* load the y coordinate */
            v2i(vert);             /* draw the point */
        }
        endpoint();
    }
    sleep(10);
    gexit();
    return 0;
}
```

Mathematical points have no size, but a point must be assigned a size to be displayed. The system draws a point as a 1-pixel point on the screen. On some systems, you can define the size of the points that are drawn using the `pntsize()` or `pntsizef()` subroutines. Specify the size of the point in pixels as a short for `pntsize()` and as a float for `pntsizef()`. See the *pntsize*(3G) and *pntsizef*(3G) man pages for information about point size limits and the capabilities of different systems.

2.1.3 Lines

The GL has two types of line primitives: *polylines* and *closed lines*. A polyline is a series of connected line segments. A closed line automatically connects the last vertex in a polyline to the first vertex in the polyline.

Polylines

Use `bgnline()` and `endline()` to specify a vertex list that is drawn as a polyline. Line segments can cross each other, and vertices can be reused. If the vertices are specified with 3-D or 4D coordinates, you can place them anywhere in 3-D space; they need not all lie in the same plane.

This sample program, *greensquare.c*, draws lines that form a green square. The `bgnline()/endline()` pair is used to select the polyline primitive for drawing. The first vertex, `v2i(vert1)`, is repeated at the end of the vertex list to connect the first and last line segments. It is better to draw such a sequence using the `closedline()` primitive, as described next in "Closed Lines".

```
#include <gl/gl.h>

long vert1[2] = {200, 200};
long vert2[2] = {200, 400};
long vert3[2] = {400, 400};
long vert4[2] = {400, 200};

main()
{
   prefsize(400, 400);
   winopen("greensquare");
   ortho2(100.5, 500.5, 100.5, 500.5);
   color(WHITE);
   clear();
   color(GREEN);
   bgnline();
      v2i(vert1);
      v2i(vert2);
      v2i(vert3);
      v2i(vert4);
      v2i(vert1);
   endline();
   sleep(10);
   gexit();
   return 0;
}
```

Closed Lines

Use `bgnclosedline()` and `endclosedline()` to specify a vertex list that is drawn as a series of line segments, in which the last vertex in the sequence is automatically connected to the first vertex.

This sample program, *n-gon.c*, draws a regular *n*-gon, an *n*-sided polygon with sides of equal length:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#define X    0
#define Y    1
#define XY   2

main(argc, argv)
int argc;
char *argv[];
{
   int n, i;
   float vert[XY];

/* Tell user to enter number of sides if no number is typed */
   if (argc != 2) {
      fprintf(stderr, "Usage: n-gon <number of sides>\n");
      return 1;
   }
   n = atoi(argv[1]); /* Convert character entered to integer */
   prefsize(400, 400);
   winopen("n-gon");
   color(WHITE);
   clear();
   color(RED);
   bgnclosedline();
   for (i = 0; i < n; i++) {
      vert[X] = 200.0 + 100.0 * fcos(i * 2.0 * M_PI / n);
      vert[Y] = 200.0 + 100.0 * fsin(i * 2.0 * M_PI / n);
      v2f(vert);
   }
   endclosedline();
   sleep(10);
   gexit();
   return 0;
}
```

Run the program by typing **ngon *n***, where *n* is the number of sides you want in the *n*-gon. If you do not specify a number, the program exits and displays a usage message telling you how to run it.

This example draws the *n*-gon only once, so there is no real penalty for computing the coordinates of the vertices. If it were necessary to draw the polygon over and over again, the calculated vertices should probably be saved in an array. In applications that draw the same geometry repeatedly with different viewing parameters, it is usually more efficient to save the coordinates in arrays. The sample program does not save the coordinates because the *n*-gon is drawn only once.

Linestyles

A *linestyle* describes the way the system draws lines on the screen. The linestyle represents a 16-bit pattern on the screen. The least significant bit of the pattern is the *mask* for the first pixel of the line and every sixteenth pixel thereafter.

The mask determines which pixels are turned on and which pixels are left off. Pixels corresponding to 1 in the linestyle are drawn; those corresponding to 0 are not drawn. For example, the linestyle 0xFFFF draws a solid line; 0xF0F0 draws a dashed line; and 0x8888 draws a dotted line. The system runs this pattern repeatedly to determine which pixels in a 16-pixel line segment to color. There is no concept of an opaque linestyle in the GL.

Defining a Linestyle

Use the `deflinestyle()` subroutine to define a line style to save in an indexed table. When you want that style to be used, you retrieve it by its index. There are 2^{16} possible linestyle patterns. By default, index 0 contains linestyle 0xFFFF, which draws solid lines. You cannot redefine linestyle 0.

To replace a linestyle in the table, specify the index of the new linestyle in place of the old one. Use `getlstyle()` to query the index of the current linestyle.

The system uses the current linestyle to draw lines and to outline rectangles, polygons, circles, and arcs. Linestyle 0 (solid line) is the default linestyle. Use `setlinestyle()` to select another linestyle. Its argument is an index into the linestyle table built by calls to `deflinestyle()`.

Modifying the Linestyle Pattern

Two routines modify the application of the linestyle pattern: `lsrepeat()` and `linewidth()`. You can get the current values for these attributes using `getstyle()`, `getlsrepeat()`, and `getlwidth()`.

Use `lsrepeat()` to create linestyles that are longer than 16 bits. `lsrepeat()` multiplies each bit in the pattern by *factor*. Consequently, each 0 in the linestyle pattern becomes a series of factor×0, and each 1 becomes a series of factor×1. For example, if the pattern is 0xFE00 and factor=1, the linestyle is 9 bits off followed by 7 bits on. If factor=3, the linestyle is 27 bits off followed by 21 bits on.

Use `getlsrepeat()` to query the factor (integer) by which the linestyle is multiplied for patterns that are longer than 16 bits.

Use `linewidth()` or `linewidthf()` to specify the width of a line. Specify the width of the line in pixels as a short for `linewidth()`, and as a float for `linewidthf()`. The system measures the width in pixels along the *x* axis or along the *y* axis. It defines the width of a line as the number of pixels along the axis having the smallest difference between the endpoints of the line.

Use `getlwidth()` to query the current linewidth in pixels.

The ANSI C specifications for the line style and line width subroutines are:

```
void deflinestyle(short n, Linestyle ls);

long getlsrepeat(void);

long getlstyle(void);

long getlwidth(void);

void linewidth(short n);

void linewidthf(float n);
```

2.1.4 Polygons

A *polygon* is specified by a connected sequence of vertices that all lie in a plane. You define the boundary of the polygon by connecting the vertices in order: *v1* to *v2*, *v2* to *v3*, and so on, finally connecting *vn* back to *v1*. These connecting segments are called *edges*. The interior of a polygon is the area inside its edges. The GL lets you specify up to 255 vertices per polygon.

Note: Even though the GL supports polygons with up to 255 vertices, performance is usually optimized only for polygons of 3 or 4 vertices. Polygons with more than 4 vertices are typically better represented as *meshes*. See Section 2.1.6, "Meshes," for the definition of meshes and for information about how they are used.

Figure 2-1 through Figure 2-6 contain some examples of polygons. The heavy black dots represent vertices and the lines represent edges.

A polygon is *simple* if its edges intersect only at their common vertices. In other words, the edges cannot cross or touch each other. A polygon is *convex* if a line segment joining any two points in its interior is completely contained within the polygon.

Figure 2-1 is a convex and simple polygon.

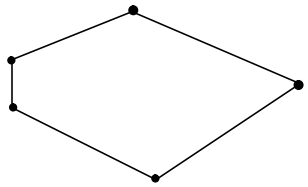

Figure 2-1 Simple Convex Polygon

Figure 2-2 and Figure 2-3 are both simple, but not convex, polygons. They are not convex because you can draw a line connecting two interior points (shown as a dashed line) that appears outside of the polygon.

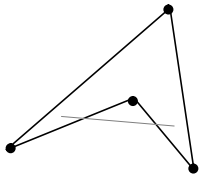

Figure 2-2 Simple Concave Polygon

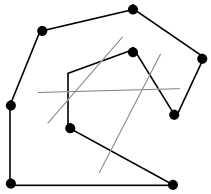

Figure 2-3 Another Simple Concave Polygon

Non-convex polygons are also called *concave.* Algorithms that render only convex polygons are much simpler than those that can render both convex and concave polygons.

Some versions of the hardware automatically check for and draw concave polygons correctly, but others do not. The function `concave()` guarantees that the system renders concave polygons correctly. On some hardware there is a performance penalty when you use `concave()`.

Note: If you intend to draw concave polygons, use `concave()`, even if your code is running on a machine that automatically does the correct thing. There is a minor performance penalty for setting the concave flag, but it makes the code portable to other Silicon Graphics machines. If you do not want to pay the performance penalty of using `concave()` on any machine, break up the concave polygons into smaller, convex polygons yourself.

The GL and the Silicon Graphics hardware can correctly render any polygon if it is simple, or if it consists of exactly four points.

Figure 2-4, Figure 2-5, and Figure 2-6 are not simple polygons.

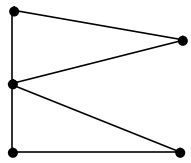

Figure 2-4 Nonsimple Polygon

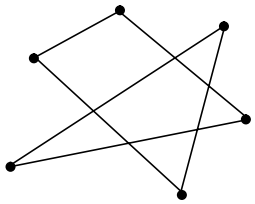

Figure 2-5 Another Nonsimple Polygon

Figure 2-6 shows a special type of polygon called a *bowtie, a* 4-vertex nonsimple polygon.

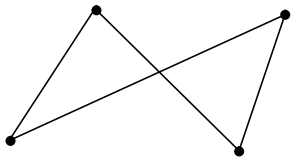

Figure 2-6 Bowtie Polygon

The GL can render the simple polygons in Figure 2-1, Figure 2-2, Figure 2-3, and the bowtie polygon in Figure 2-6. Bowtie polygons are handled in a hardware-specific manner. The polygon appears either as a bowtie, or as a quadrilateral with a segment missing from one edge. The results of rendering the type of non-simple polygons shown in Figure 2-4 and Figure 2-5 are unpredictable.

Draw polygons using the same basic syntax as the other primitives, that is, specify a list of vertex subroutines between a `bgnpolygon()`/`endpolygon()` pair. The system draws a polygon as a filled area on the screen. As it does with closed lines, the GL automatically connects the first and the last point. You do not need to repeat the first point at the end of the sequence.

This sample program, *bluehex.c*, draws a filled blue hexagon on the screen:

```
#include <gl/gl.h>

float hexdata[6][2] ={
   {20.0, 10.0},
   {10.0, 30.0},
   {20.0, 50.0},
   {40.0, 50.0},
   {50.0, 30.0},
   {40.0, 10.0}
};

main()
{
   int i;

   prefsize(400, 400);
   winopen("bluehex");
   ortho2(0.0, 60.0, 0.0, 60.0);
   color(BLACK);
   clear();
   color(BLUE);
   bgnpolygon();
   for (i = 0; i < 6; i++)
      v2f(hexdata[i]);
   endpolygon();
   sleep(10);
   gexit();
   return 0;
}
```

If you approximate a surface with 4-sided polygons, some of them may not lie in a plane. If the vertices of a polygon do not lie in a plane, it is likely that in certain orientations the polygon on the screen may look like its edges cross. This is especially true at the *silhouette edges* of a mesh, where the mesh wraps around to the back of the shape. However, with 4 vertices, these distorted polygons will all be interpreted as bowtie polygons that are correctly rendered by the GL.

The GL can render the bowties that arise from surface-approximating meshes. In most other circumstances, however, the GL routines generate only true (planar) polygons.

If a polygon lies in a plane, the only way distortion can occur is from floating point inaccuracies.

IRIS-4D Series systems convert all arguments to 32-bit floating point for hardware calculations. Consequently, only long integers in the range of -2^{23} and $2^{23}-1$ retain full precision after conversion. Integers outside this range retain 24 bits of precision after conversion.

Patterns

You can fill polygons (as well as rectangles and arcs) with *patterns*. A pattern is an array of short integers that defines a rectangular pixel array. The pattern controls which pixels the system colors when it draws filled polygons. The system aligns the pattern to the lower-left corner of the screen, rather than to the filled shape, so that the pattern appears continuous over large areas.

Use `defpattern()` to define a pattern to be saved in an indexed table. Specify an index into a table of patterns (*n*), the length of the array (*size*), and an array of short integers (*mask*). The *size* argument selects either a 16×16 (*size*=32) or a 32×32 (*size*=64) pattern.

The origin of the pattern is the lower-left corner of the screen. Define the bottom row of the pattern first. Specify each row of a 16×16 pattern with a single short. Specify each row of a 32×32 pattern with two shorts—first the left 16 bits, then the right 16 bits. Bit 0 of each short is the right-most bit of its respective position in the row. There is no concept of an opaque pattern in the GL.

Pattern 0 is the default pattern, which is solid. You cannot redefine the pattern at index 0.

Use `setpattern()` to select which defined pattern the system uses. The argument for `setpattern()` is the index you defined with `defpattern()`. Use `getpattern()` to query the index of the current pattern.

The ANSI C specifications for the pattern subroutines are:

```
void defpattern(short n, short size, unsigned short mask[]);
long getpattern(void);
void setpattern(short index);
```

2.1.5 Point-Sampled Polygons

This section tells you exactly which pixels are turned on when the system displays a polygon. It is important to know which pixels are turned on for display accuracy and drawing performance reasons, but you may want to skip this section on your first reading, because it contains advanced concepts.

To represent a polygon on the screen, the system must turn on a group of pixels. Given a set of coordinates for the vertices of a polygon, there is more than one way to decide which pixels ought to be turned on.

To illustrate the problem, consider drawing the two rectangles shown in Figure 2-7. The numbered grid represents pixels and the black dots represent pixels that are on.

The rectangle on the left has sides of ($2 \leq x \leq 5$ and $1 \leq y \leq 4$).
The rectangle on the right has sides of ($2 \leq x \leq 5$ and $4 \leq y \leq 6$).

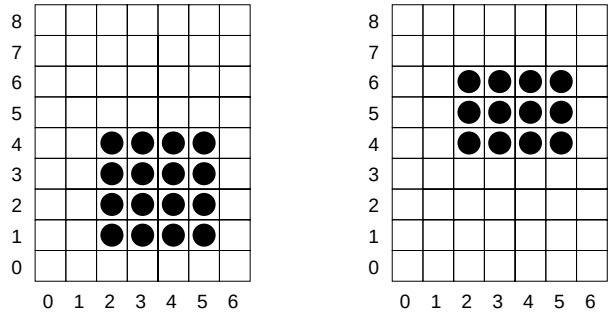

Figure 2-7 Non–Point-Sampled Polygons

What pixels should the system turn on in both cases? The most obvious answer is shown in Figure 2-7. If you draw a figure consisting of the two polygons in Figure 2-7, you would expect them to fit together. Unfortunately, if you draw both rectangles the way Figure 2-7 shows, the pixels on the line $y = 4$ are drawn twice—once for each polygon.

Drawing every overlapping edge twice is not a very efficient way to draw a large number of polygons. Drawing the polygons in this manner also gives the effect of outlining the polygon, which may not be the way you want the polygons to look. For example, if the polygons represent a transparent surface, the duplicated edge is twice as dense as the interior of the polygon, giving the entire surface an outlined appearance.

Even if the surface is not transparent, filling the polygons in this way can still create undesirable visual effects. If you draw a checkerboard pattern with edges that overlap by exactly one pixel, then redraw it in single buffer mode, the redrawing is visible because the edges of the squares flicker from one color to the other, even though both images are identical. See Chapter 6, "Animation," for more information about single buffer mode.

The GL resolves these problems by using *point-sampled polygons*. Point-sampled polygons assume that *ideal mathematical lines* (lines with no thickness) connect the vertices. The system draws any pixel whose center lies inside the mathematically precise polygon. Pixels whose centers lie outside the polygon do not get drawn. Pixels whose centers lie exactly on a mathematical line segment or vertex are filled in a hardware-dependent manner that attempts to avoid both multiple fills and gaps at the boundaries of adjacent polygons. All that is guaranteed about this algorithm is that pixels on the left and bottom of a screen-aligned rectangle that is drawn on the exact pixel centers are filled, whereas the pixels on the right and top of such a rectangle are not filled.

Figure 2-8 shows point-sampled versions of the two rectangles in Figure 2-7. Point-sampling effectively eliminates the duplication of pixels from the right and top edges of the polygon, but adjacent polygons can fill those pixels.

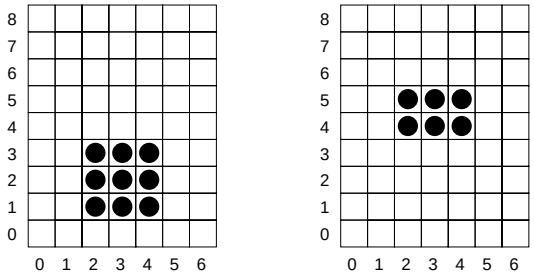

Figure 2-8 Point-Sampled Polygons

Another advantage of a point-sampled polygon without an outline is that the drawn area of the polygon is much closer to the actual mathematical area of the polygon. In the examples above, the drawn areas correspond exactly to the true areas of the polygons. In non-rectangular polygons, the drawn area of the polygon cannot be exact, but the drawn area of the point-sampled polygon is closer to the true area of the polygon than is the area of an outlined polygon. Outlined polygons that have increased area are sometimes called *fat polygons*.

Figure 2-9 illustrates the pixels that are turned on in a point-sampled representation of the polygon that connects the vertices (1,1) (1,4) (5,6) and (5,1). The pixels at (1, 4), (3, 5), (5,6), (5,5), (5,4), (5,3), (5,2), and (5,1) all lie mathematically on the boundary of the polygon but are not drawn because they are on the upper or right edge.

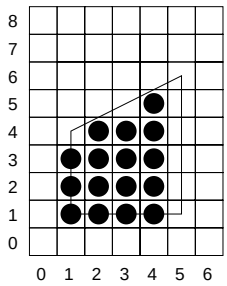

Figure 2-9 Point-Sampled Polygon with Edges Exactly on Pixel Centers

As mathematical entities, lines have no thickness. However, to represent a line on the screen, the system assumes a thickness of exactly one pixel (or whatever line width you have assigned). When you scale an object composed of lines, the lines behave differently from polygons. No matter how much a transformation magnifies or reduces an object composed of lines, the representation of the line remains one pixel thick. If you draw a line around a point-sampled polygon, it fills in the pixels at the upper and right-hand edges.

The default for the older subroutines such as `polf()`, `rect()`, `circle()`, is to draw a line around the point-sampled polygon. However, if you use the old-style subroutines, it is recommended that you use `glcompat()` as described in Section 2.2.2, “Nonrecommended Old-Style Subroutines,” to specify point-sampled polygons. You don’t have to do this for the vertex subroutines because they draw point-sampled polygons by default.

Anomalies can occur in the display of very thin filled polygons. Figure 2-10 shows a thin point-sampled triangle connecting the points (1,1), (2,3), and (12,7). The polygon looks like it has holes, but if you draw adjacent polygons that share the same vertices, all the pixels are eventually filled.

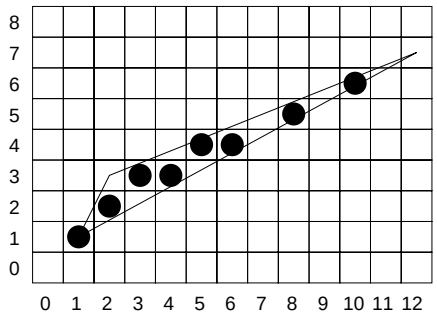

Figure 2-10 Point Sampling Anomaly

2.1.6 Meshes

This section covers an advanced topic. You may want to skip it on the first reading, because it mentions topics that are not fully covered until later in this guide.

This section describes how to draw geometric figures that are constructed entirely of adjacent 3-sided (triangular) or 4-sided (quadrilateral) polygons. When you draw connecting polygons, a *mesh* is formed. A mesh made of triangles is called a *triangular mesh (t-mesh)*, and a mesh made of quadrilaterals is called a *quadrilateral strip (q-strip)*.

Figure 2-11 shows an example of a simple t-mesh.

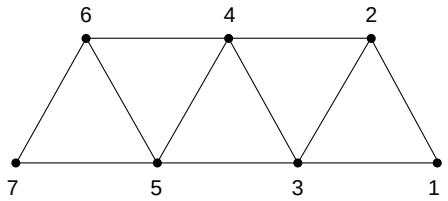

Figure 2-11 Simple Triangle Mesh

In this example, the seven vertices form five triangles (123, 324, 345, 546, 567). Vertex 1 and vertex 7 appear in one triangle, vertices 2 and 6 appear in two triangles, and all the rest of the vertices each appear in three different triangles. In a larger mesh, a higher percentage of the points would be used three times. Drawing the geometry in Figure 2-11 as a t-mesh is more efficient than drawing it as five separate triangles, because the t-mesh draws the shared vertices only once.

To draw a t-mesh, you specify a sequence of vertices between a `bgntmesh()/endtmesh()` pair.

The `bgntmesh()` call signifies that the vertices following it are connected as triangles until an `endtmesh()` call is received. The system uses two memory locations to remember the last two vertices and a pointer to keep track of what it is doing. The pointer alternates from one retained vertex to the other while it is drawing the mesh.

Refer to Table 2-2 and the discussion following it to see the sequence of events that happens internally when a t-mesh is drawn.

Sequence	Vertex	P→	R1	R2	Next Vertex
1	`bgntmesh()`	R1	~~~	~~~	v1
2	v1	R2	v1	~~~	v2
3	v2	R1	v1	v2	v3
4	v3	R2	v3	v2	v4
5	v4	R1	v3	v4	v5
6	v5	R2	v5	v4	v6
7	v6	R1	v5	v6	v7

Table 2-2 Sequence of Vertices in a Mesh

Let P→ represent the pointer and let R1 and R2 represent the two vertices that are retained:

1. The `bgntmesh()` call initializes the pointer to R1.
2. The first vertex (v1) is stored in the location pointed to by P, which is R1. The pointer is now changed to point to R2.
3. The next vertex (v2) is stored in the location pointed to by P, which is R2. The pointer is changed to point to R1.
4. The next vertex (v3) is the third vertex, so a triangle is drawn. Triangles are drawn R1, R2, next vertex, so triangle 123 is drawn. v3 is stored in the location pointed to by P, which is R1. The pointer is changed to point to R2.
5. The next vertex (v4) is now connected to R1 and R2. R1 has v3 stored in it, and R2 has v2 stored in it, so the triangle drawn (R1,R2,new) is triangle 324. v4 stored in the location pointed to by P, which is R2. The pointer is changed to point to R1.
6. This process continues until the `endtmesh()` call is encountered.

This process draws the triangles 123, 324, 345, 546, and 567 for the mesh in Figure 2-11.

Figure 2-12 illustrates a more complex situation. The first six triangles (123, 324, 345, 546, 567, 768) could be drawn as before, but if nothing is done, the arrival of point 9 would cause triangle 789 to be drawn, not triangle 689 as desired.

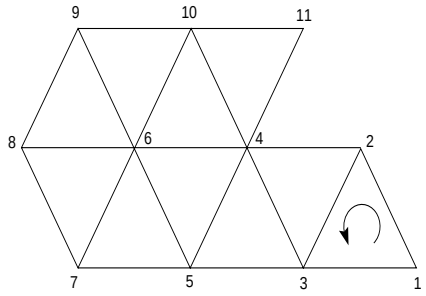

Figure 2-12 Example of `swaptmesh()` Construction

To draw meshes like the one in Figure 2-12, you must exchange the order of the two stored vertices. Use the `swaptmesh()` subroutine to exchange the pointer to the other retained vertex, as shown in the following code fragment:

```
bgntmesh();
   v3f(vert1);
   v3f(vert2);
   v3f(vert3);
   v3f(vert4);
   v3f(vert5);
   v3f(vert6);
   v3f(vert7);
swaptmesh();
   v3f(vert8);
swaptmesh();
   v3f(vert9);
swaptmesh();
   v3f(vert10);
   v3f(vert4);
   v3f(vert11);
endtmesh();
```

Figure 2-13 shows another example of how to use `swaptmesh()`.

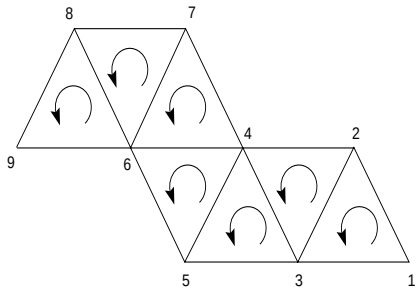

Figure 2-13 Another `swaptmesh()` Example

This sequence draws the mesh in Figure 2-13:

```
bgntmesh();
   v3f(vert1);
   v3f(vert2);
   v3f(vert3);
   v3f(vert4);
   v3f(vert5);
swaptmesh();
   v3f(vert6);
   v3f(vert7);
swaptmesh();
   v3f(vert8);
   v3f(vert9);
endtmesh();
```

The arrows show that all the faces in this t-mesh specify their vertices in counter-clockwise order. This is important if you want to do *hidden-surface removal*, described in Chapter 8, or *two-sided lighting*, described in Chapter 9, later. Because the faces are counter-clockwise, they specify *front-facing* polygons, which display when backface removal is turned on. For lighting, the counter-clockwise faces specify normals that follow the right-hand rule—that is, point out of the paper toward you. If you take your right hand and curl your fingers in the direction pointed to by the arrows, with your thumb sticking out, your thumb points straight out, away from the paper. Your thumb represents the vertex normals for the counter-clockwise faces.

This sample program, *octahedron.c*, draws a three-dimensional octahedron (8-sided regular polyhedron) using the t-mesh primitive. Because meshes in two dimensions are of little use, the example is three-dimensional. Knowing how to create a three-dimensional mesh is quite useful. This program also uses a number of routines that are covered in later chapters, including three-dimensional rotations, hidden surface removal, smooth (double-buffered) motion, and `cpack()`, so ignore the subroutines that do not apply to the specification of the geometry if you are studying the program to understand the logic of mesh drawing. The calculations of rotation angles simply cause the octahedron to tumble in an interesting way.

```
#include <stdio.h>
#include <gl/gl.h>

float octdata[6][3] = {
    { 1.0, 0.0, 0.0},
    { 0.0, 1.0, 0.0},
    { 0.0, 0.0, 1.0},
    {-1.0, 0.0, 0.0},
    { 0.0, -1.0, 0.0},
    { 0.0, 0.0, -1.0}
};
unsigned long octcolor[6] = {
    0xff0000,                       /* [0] = blue */
    0x00ff00,                       /* [1] = green */
    0x0000ff,                       /* [2] = red */
    0xff00ff,                       /* [3] = magenta */
    0xffff00,                       /* [4] = cyan */
    0xffffff,                       /* [5] = white */
};

void vertex(i)
int i;
{
    cpack(octcolor[i]);
    v3f(octdata[i]);
}

void drawoctahedron()
{
    bgntmesh();
        shademodel(GOURAUD);
        vertex(0);
        vertex(1);
    swaptmesh();
        vertex(2);
```

```
    swaptmesh();
       vertex(4);
    swaptmesh();
       vertex(5);
    swaptmesh();
       vertex(1);
       vertex(3);
    shademodel(FLAT);
       vertex(2);
    swaptmesh();
       vertex(4);
    swaptmesh();
       vertex(5);
    swaptmesh();
       vertex(1);
    endtmesh();
}
main()
{
    Angle xang, yang, zang;
    long zval;
    int cnt;
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
       fprintf(stderr, "Double buffered RGB not available\n");
       return 1;
    }
    if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
       fprintf(stderr, "Z-buffer not available\n");
       return 1;
    }
    prefsize(400, 400);
    winopen("octahedron");
    doublebuffer();
    RGBmode();
    gconfig();
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    zbuffer(TRUE);   /* hidden surfaces removed with z-buffer */
    zval = getgdesc(GD_ZMAX);
    xang = yang = zang = 0;
    for (cnt = 0; cnt < 1000; cnt++) {
       czclear(0x000000, zval);
       pushmatrix();/* save viewing transformation */
            rotate(xang, 'x'); /* rotate by xang about x axis */
            rotate(yang, 'y'); /* rotate by yang about y axis */
            rotate(zang, 'z'); /* rotate by zang about z axis */
            drawoctahedron();
```

```
        popmatrix();        /* restore viewing transformation */
        swapbuffers();              /* show completed drawing */

        xang += 10;
        yang += 13;
        if (xang + yang > 3000)
            zang += 17;
        if (xang > 3600)
            xang -= 3600;
        if (yang > 3600)
            yang -= 3600;
        if (zang > 3600)
            zang -= 3600;
    }
    gexit();
    return 0;
}
```

Quadrilateral Strips

In addition to the t-mesh, the GL supports quadrilateral strips (q-strips). Q-strips are similar in many ways to t-meshes, but might be better suited for the representation of shapes that are fundamentally quadrilateral rather than triangular in nature.

Use `bgnqstrip()` and `endqstrip()` to specify vertex list that forms quadrilateral strips.

The `bgnqstrip()` and `endqstrip()` commands must surround an even number of vertex commands that is four or greater, and is unbounded. Filling results are undefined if these conditions are not met. There is no maximum to the number of vertices that can be specified between `bgnqstrip()` and `endqstrip()`. If the number is odd, however, the result is undefined.

Vertices specified after `bgnqstrip()` and before `endqstrip()` form a sequence of quadrilaterals. You cannot alter the replacement algorithm, because there is no quadrilateral equivalent to the `swaptmesh()` command.

For example, this sequence draws the three quadrilaterals: (1,2,4,3), (3,4,6,5), and (5,6,8,7) in Figure 2-14:

```
bgnqstrip();
   v3f(vert1);
   v3f(vert2);
   v3f(vert3);
   v3f(vert4);
   v3f(vert5);
   v3f(vert6);
   v3f(vert7);
   v3f(vert8);
endqstrip();
```

Figure 2-14 Mesh of Quadrilateral Strips

Note: The quadrilaterals are drawn as though each quadrilateral were an independent polygon with vertex order (*n*, *n*+1, *n*+3, *n*+2).

Note that the vertex order required by q-strips matches the order required for "equivalent" triangle meshes. The example vertex sequence that produces the mesh in Figure 2-14 produces triangles (123, 324, 345, 546, 567, 768), as shown in Figure 2-15 when bounded by `bgntmesh()` and `endtmesh()` calls.

Figure 2-15 Equivalent T-mesh

In general, quadrilateral data looks better when drawn with q-strips than with a t-mesh. This is because Gouraud shading calculations operate on the original quadrilateral data, rather than on the decomposed triangles.

Note: IRIS-4D VGX, VGXT, SkyWriter, and RealityEngine graphics use vertex normals to determine how to decompose quadrilaterals into triangles during scan conversion. If you do not specify vertex normals, or, equivalently, if the four vertices share the same normal, the selected decomposition matches that of the equivalent triangle mesh.

2.1.7 Controlling Polygon Rendering

Use the `polymode()` subroutine to specify how the system renders polygons. This statement controls polygons created with triangular mesh or quadrilateral strips as well as explicit polygons (that is, polygons created inside a `bgn*/end*` loop).

The ANSI C specification for `polymode()` is:

```
void polymode(long mode);
```

where *mode* is defined by one of the following symbols:

`PYM_POINT`	draw only points at each vertex.
`PYM_LINE`	draw lines from vertex to vertex.
`PYM_FILL`	fill the polygon interior.
`PYM_HOLLOW`	fill only interior pixels at the boundaries.

`PYM_POINT` and `PYM_LINE` draw points and lines consistent with all applicable point and line modes. Therefore the antialiasing mode, `pntsmooth()`, or `linesmooth()` as well as linewidth and linestipple are significant. `PYM_FILL` is the standard fill operation that was previously the only option. See Chapter 15, "Antialiasing," for information on antialiasing.

Figure 2-16 shows how `PYM_LINE` affects clipping. Polygons drawn in `PYM_LINE` mode clip differently from closed lines. The `PYM_LINE` polygon always clips to a closed line, with line segments generated along the edges of the clipping planes, which are usually the borders of the viewport.

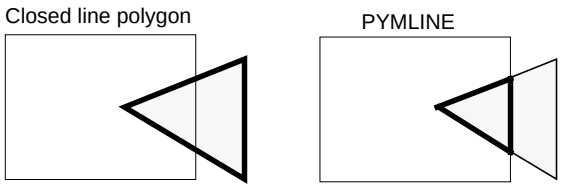

Figure 2-16 Clipping Behavior of `PYM_LINE`

PYM_HOLLOW supports a special kind of polygon fill with the following properties:

- Only pixels on the polygon edge are filled. These pixels form a single-width line (regardless of the current linewidth) around the inner perimeter of the polygon.
- Only pixels that would have been filled (PYM_FILL) are changed (that is, the outline does not extend beyond the exact polygon boundaries).
- Changed pixels take the exact color and depth values they would have, had the polygon been filled.

Because their pixel depth values are exact, hollow polygons can be accurately composed of filled polygons. Both hidden-line and scribed-surface renderings can be done taking advantage of this fact. See Chapter 8, "Hidden-Surface Removal," for more information on hidden surfaces.

Note: Not all systems support `polymode()`. Use `getgdesc()` with the `GD_POLYMODE` argument to determine whether polymode is supported. The IRIS-4D VGX requires special setup to support `PYM_HOLLOW`. See the *polymode*(3G) man page for details.

2.2 Old-Style Drawing

This section describes drawing methods that were used in previous releases of the GL. Skip this section if you are developing new GL applications.

The vertex drawing subroutines described in the last section are the preferred way to draw any geometry except curves and surfaces, which are covered in Chapter 14, "Curves and Surfaces." All new development should use the vertex method. It is OK to use the subroutines described under Section 2.2.1, "Basic Shapes," but the subroutines described in Section 2.2.2, "Nonrecommended Old-Style Subroutines," are not recommended.

The architecture of earlier Silicon Graphics systems was tuned to a different set of subroutines for drawing points, lines, and polygons. For compatibility, all of the earlier subroutines are still in the GL.

In most cases, the internals of these earlier subroutines have been rewritten to use the vertex subroutines. However, to guarantee that you get the optimal

performance of the new programs, use the vertex subroutines described at the beginning of this chapter.

Except for polygons, the figures drawn by the old-style subroutines are the same as those drawn by the vertex subroutines. For example, points are drawn as a single pixel. However, the earlier subroutines did not draw point-sampled polygons. They effectively drew point-sampled polygons with lines connecting the vertices. For compatibility, the old polygon subroutines draw point-sampled polygons with an outline, so they appear exactly the way they did before. For many polygons, the drawing time is increased when both the polygon and its outline are drawn.

In most cases, absolute compatibility with the old polygon filling style is not required, so there is a subroutine, `glcompat()`, that you can use to turn off outlining for the old-style subroutines. You can significantly increase polygon drawing performance for old code by turning off the compatibility mode. `glcompat()` takes two arguments. The first is the compatibility mode, and the second is the value to which it is set.

The default `GLC_OLDPOLYGON` value is 1, in which outlining is turned on. To turn off polygon outlining, use:

```
glcompat(GLC_OLDPOLYGON, 0);
```

Performance for the basic shapes subroutines can also be improved significantly by calling `glcompat(GLC_OLDPOLYGON, 0)` to draw point-sampled polygons and defeat polygon outlining.

2.2.1 Basic Shapes

In this section and the next, the subroutine names follow a pattern. The root name of the subroutine indicates the shape that is drawn. Letters appended to the root name indicate whether the shape is filled or drawn as an outline and also indicate its data type. The default data type is floating point (32 bits):

f — indicates that the figure is filled rather than drawn as an outline (32 bits).

s — indicates that the data type is a short integer (16 bits).

i — indicates that the data type is a long integer (32 bits).

Rectangles

The GL provides two types of rectangle subroutines—filled and unfilled. Filled rectangles are rectangular polygons, and unfilled rectangles are rectangular outlines. `rect()` draws a rectangular outline, while `rectf()` draws a filled (solid) rectangle. Only the *x* and *y* coordinates of the corners of the rectangle are given, and the *z* coordinate is forced to zero. The rectangle is aligned with the *x* and *y* axes.

Table 2-3 lists the six different forms of the rectangle subroutine.

Argument Type	Filled	Unfilled
16-bit integer	`rectfs()`	`rects()`
32-bit integer	`rectfi()`	`recti()`
32-bit float	`rectf()`	`rect()`

Table 2-3 Rectangle Subroutines

The arguments to the rectangle subroutines are the coordinates of the corners (*x1, y1, x2, y2*). The point *(x1, y1)* is one corner of the rectangle, and *(x2, y2)* is the opposite corner. Because the rectangle is aligned with the axes, the coordinates of the other corners would be *(x1, y2)* and *(y1, x2)*.

Rectangles can undergo transformations as described in Chapter 7, "Coordinate Transformations," and the resulting figure need not appear to be a rectangle. For example, imagine rotating the rectangle about the *x* axis so that one end is farther from you, then viewing it in perspective. On the screen, the rotated rectangle appears to be a trapezoid.

It is important to understand that although rectangles created with `rectf()` or its variants can be transformed by the statements described in Chapter 7, primitives such as rectangles, circles, arcs, and character strings are planar, and the apparent rotation or translation takes place because of manipulations to the underlying transformation matrix.

If you wish to build a composite of different rectangular shapes (for instance, a 3-D cube) that is to be part of a 3-D model, the correct way to proceed is to use the 3-D drawing functions `bgnpolygon()` and `endpolygon()`.

The following sample program, *chessboard.c* draws a chess board with black and white squares and red outlines on a green background.

```
#include <gl/gl.h>

#define ODD(n)   ((n) % 2)

main()
{
    int i, j;

    prefsize(400, 400);
    winopen("chessboard");
    color(GREEN);
    clear();
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 8; j++) {
            if (ODD(i + j))
                color(WHITE);
            else
                color(BLACK);
            sboxfi(100 + i*25, 100 + j*25, 124 + i*25, 124 + j*25);
        }
    }
    color(RED);
    recti(97, 97, 302, 302);
    sleep(10);
    gexit();
    return 0;
}
```

Screen Boxes

Screen boxes are a subclass of rectangles. They are always 2-D and are always aligned with the screen coordinates. Draw screen boxes with `sbox()` and `sboxf()`. As with `rect()` and `rectf()`, the f signifies that the screen box is filled with the current color and pattern.

All screen box commands use four arguments:

x1	*x* coordinate of one corner of the box
y1	*y* coordinate of one corner of the box
x2	*x* coordinate of the opposite corner of the box
y2	*y* coordinate of the opposite corner of the box

The screen box drawing commands fill in the rectangle given these diagonal corner coordinates. The `sbox()` statements draw 2-D, screen-aligned rectangles using the current color, writemask, and linestyle. The `sboxf()` statements draw filled 2-D, screen-aligned rectangles using the current color, writemask, and pattern.

Table 2-4 lists the screen box subroutines.

Argument Type	Filled	Unfilled
16-bit integer	`sboxfs()`	`sboxs()`
32-bit integer	`sboxfi()`	`sboxi()`
32-bit floating point	`sboxf()`	`sbox()`

Table 2-4 Screen Box Subroutines

You cannot use lighting, backfacing, depth-cueing, *z*-buffering, Gouraud shading, or alpha blending with the `sbox()` or `sboxf()` command.

Circles

Like rectangles, circles are 2-D figures that lie in the *x*-*y* plane, with *z* coordinates equal to zero. All six circle subroutines have the same parameters: *x*, *y*, and *radius*. Like rectangles, circles are either filled or unfilled, and the center coordinates and radius are specified in integers, short integers, or floats.

Table 2-5 lists the circle subroutines.

Argument Type	Filled	Unfilled
16-bit integer	`circfs()`	`circs()`
32-bit integer	`circfi()`	`circi()`
32-bit floating point	`circf()`	`circ()`

Table 2-5 Circle Subroutines

Circles are drawn with 80 equally spaced vertices, either as a closed line (unfilled circles) or as a polygon (filled circles). If your program draws many circles, you can write a circle primitive that uses fewer line segments to speed up the drawing. Circles drawn with 80 segments look reasonably good over a

wide range of sizes, but on large circles, you can easily see the straight line segments. You can also draw circles with NURBS, as explained in Chapter 14.

This sample program, *bullseye.c*, draws an archery target using filled circles:

```
#include <gl/gl.h>
main()
{
   prefsize(400, 400);
   winopen("bullseye");
   ortho2(-1.0, 1.0, -1.0, 1.0);
   color(BLACK);
   clear();
   color(GREEN);
   circf(0.0, 0.0, 0.9);
   color(YELLOW);
   circf(0.0, 0.0, 0.7);
   color(BLUE);
   circf(0.0, 0.0, 0.5);
   color(CYAN);
   circf(0.0, 0.0, 0.3);
   color(RED);
   circf(0.0, 0.0, 0.1);
   sleep(10);
   gexit();
   return 0;
}
```

Arcs

Arcs are also 2-D figures, and like circles and rectangles, they lie in the plane z=0. Arcs can be either unfilled (segments of circles) or filled (pie wedges). Arcs have a center *(x, y)*, a *radius*, a *starting angle*, and an *ending angle*. Angles are measured counterclockwise from the positive x axis; negative angles are clockwise. Angles are in tenths of degrees, so a 90-degree angle is written as 900.

An arc is drawn from the starting angle to the ending angle, so if *startang* is 0 and *endang* is 100, a 10 degree arc is drawn. Arcs are approximated by straight lines, and a full 360 degree arc consists of 80 segments. You can speed up arc drawing by making an arc primitive that uses fewer line segments.

Table 2-6 lists the `arc()` subroutines.

Argument Type	Filled	Unfilled
16-bit integer	`arcfs()`	`arcs()`
32–bit integer	`arcfi()`	`arci()`
32-bit float	`arcf()`	`arc()`

Table 2-6 Arc Subroutines

The following sample program draws a pie chart using filled arcs:

```
#include <gl/gl.h>

main()
{
   prefsize(400, 400);
   winopen("piechart");
   ortho2(-1.0, 1.0, -1.0, 1.0);
   color(BLACK);
   clear();
   color(RED);
   arcf(0.0, 0.0, 0.9, 0, 800);
   color(GREEN);
   arcf(0.0, 0.0, 0.9, 800, 1200);
   color(YELLOW);
   arcf(0.0, 0.0, 0.9, 1200, 2200);
   color(MAGENTA);
   arcf(0.0, 0.0, 0.9, 2200, 3400);
   color(BLUE);
   arcf(0.0, 0.0, 0.9, 3400, 0);
   sleep(10);
   gexit();
   return 0;
}
```

2.2.2 Nonrecommended Old-Style Subroutines

This section describes *nonrecommended* subroutines from previous releases of the GL. Skip this section if you are developing new GL applications.

The naming conventions for the rest of the subroutines in this chapter are similar to those used by the `arc()`, `circle()`, and `rectangle()` subroutines. However, because the remaining subroutines are usually three-dimensional, they come in 2-D and 3-D versions. As with `arc()`, `circle()`, and `rectangle()`, the two-dimensional versions are assumed to lie in the plane $z = 0$, but those figures can be transformed out of that plane by the various transformation and viewing subroutines discussed in Chapter 7.

The naming convention assumes that most subroutines are three-dimensional, so, for example, the point subroutine `pnt()` is the three-dimensional version, and `pnt2()` requires no *z* component to its arguments.

Current Graphics Position

In the new architecture, graphical figures are sent together—a set of points, a polyline, and a polygon are sent bracketed by a `bgn`*`geometry`* and an `end`*`geometry`* subroutine. The drawing of the figure might not start until the `end`*`geometry`* arrives.

In older systems, points were sent as individual subroutines, lines as a series of move and draw subroutines, and polygons as a polygon move, followed by a polygon draw subroutines, and finally a polygon close subroutine.

Between the old-style subroutines drawing polylines or polygons, the system maintains a current graphics position. Each draw subroutine draws from the current graphics position to the point specified by `draw()`. The current graphics position is then set to the new point. The current graphics position as used by the old-style polygon subroutines is discussed in the next sections.

getgpos

Because the system automatically maintains the current graphics position, few applications need to access it directly. Those that do use `getgpos()` to return the current graphics position. Its arguments include four pointers to floating point numbers in which the homogeneous coordinates of the current transformed point are returned. The returned values are in clip coordinates.

For compatibility, the current graphics position is maintained in exactly the same way for all the graphics subroutines listed in the rest of this chapter.

Old-Style Points

Table 2-7 shows the old-style point subroutines.

Argument Type	2-D	3-D
16-bit integer	`pnt2s()`	`pnts()`
32-bit integer	`pnt2i()`	`pnti()`
32-bit float	`pnt2()`	`pnt()`

Table 2-7 Old-Style Point Subroutines

The arguments are (*x*, *y*) for the 2-D subroutines, (*x*, *y*, *z*) for the 3-D subroutines. In addition to drawing a point, `pnt()` updates the current graphics position to its location.

This sample program, *pointsquare.c*, draws 100 points in a square area of the window:

```
#include <gl/gl.h>

main()
{
   int i, j;

   prefsize(400, 400);
   winopen("pointsquare");
   color(BLACK);
   clear();
   color(GREEN);
      for (i = 0; i < 100; i++) {
         for (j = 0; j < 100; j++)
            pnt2i(i*4 + 1, j*4 + 1);
   }
   sleep(10);
   gexit();
   return 0;
}
```

Old-Style Lines

Old-style lines are drawn using two subroutines: `move()` and `draw()`. The arguments and types of the `move()` and `draw()` subroutines are the same as for the `point()` subroutines. The `move()` subroutine sets the current graphics position to the specified vertex and `draw()` draws from the current graphics position to the specified point, then updates the current graphics position to that vertex.

Table 2-8 lists the `move()` and `draw()` subroutines.

Argument Type	2–D	3-D
16-bit integer	`move2s()`	`moves()`
32-bit integer	`move2i()`	`movei()`
32-bit float	`move2()`	`move()`
16-bit integer	`draw2s()`	`draws()`
32-bit integer	`draw2i()`	`drawi()`
32-bit float	`draw2()`	`draw()`

Table 2-8 Old-Style Move and Draw Subroutines

This sample program, *bluebox.c*, draws the outline of a blue box on the screen using the `move()` and `draw()` subroutines:

```
#include <gl/gl.h>
main()
{
    prefsize(400, 400);
    winopen("bluebox");
    color(BLACK);
    clear();
    color(BLUE);
    move2i(200, 200);
    draw2i(200, 300);
    draw2i(300, 300);
    draw2i(300, 200);
    draw2i(200, 200);
    sleep(10);
    gexit();
    return 0;
}
```

Old-Style Polygons

The old-style subroutines that draw filled polygons corresponding to the `move()` and `draw()` subroutines are `pmv()` and `pdr()`.

Table 2-9 lists the filled polygon subroutines.

Argument Type	2-D	3-D
16-bit integer	`pmv2s()`	`pmvs()`
32-bit integer	`pmv2i()`	`pmvi()`
32-bit float	`pmv2()`	`pmv()`
16-bit integer	`pdr2s()`	`pdrs()`
32-bit integer	`pdr2i()`	`pdri()`
32-bit float	`pdr2()`	`pdr()`

Table 2-9 Old-Style Filled Polygon Move and Draw Subroutines

A polygon is specified by a `pmv()` to locate the first point on the boundary, then a sequence of `pdr()` subroutines for each additional vertex, and finally a `pclos()` to close and fill the polygon. The `pclos()` subroutine has no arguments; all the other subroutines take either two or three arguments of the appropriate type.

Caution: Be sure not to spell the `pclos()` command *pclose*, because *pclose* is the IRIX command to close a pipe.

The following sample program, *bluebox3.c*, draws a filled blue polygon:

```
#include <gl/gl.h>

main()
{
    prefsize(400, 400);
    winopen("bluebox3");
    color(BLACK);
    clear();
    color(BLUE);
    pmv2i(200, 200);
    pdr2i(200, 300);
    pdr2i(300, 300);
    pdr2i(300, 200);
    pclos();
    sleep(10);
    gexit();
    return 0;
}
```

The GL has two sets of subroutines that take arrays of vertex coordinates and draw filled and unfilled polygons. Filled polygons are drawn by `polf()`, and polygon outlines are drawn by `poly()`.

Both the `polf()` and the `poly()` subroutines take two arguments. The first argument, *n*, is the number of vertices in the polygon, and the second is a two-dimensional array containing the coordinates.

Table 2-10 lists the polygon and filled polygon subroutines.

Argument Type	2-D	3-D
16-bit integer	`poly2s()`	`polys()`
32-bit integer	`poly2i()`	`polyi()`
32-bit float	`poly2()`	`poly()`
16-bit integer	`polf2s()`	`polfs()`
32-bit integer	`polf2i()`	`polfi()`
32-bit float	`polf2()`	`polf()`

Table 2-10 Old-Style Polygon and Filled Polygon Subroutines

This sample program, *hexagon.c*, draws a filled hexagon using `polf()`:

```
#include <gl/gl.h>

long parray[6][2] = {
   {100, 0},
   { 0, 200},
   {100, 400},
   {300, 400},
   {400, 200},
   {300, 0}
};

main()
{
   prefsize(400, 400);
   winopen("hexagon");
   color(BLACK);
   clear();
   color(GREEN);
   polf2i(6, parray);
   sleep(10);
   gexit();
   return 0;
}
```