Chapter 16

Graphical Objects

This chapter describes the subroutines that you use to build hierarchies of drawing modules so you can draw geometry that has multiple instances of the same figure. You often want to group together a sequence of drawing subroutines and give it an identifier. The entire sequence can then be repeated with a single reference to the identifier rather than by repeating all the drawing subroutines. In the Graphics Library, such sequences are called *graphical objects*; in other systems they are sometimes known as *display lists*.

- Section 16.1, "Creating an Object," tells you how to define the drawing modules that create an object.
- Section 16.2, "Working with Objects," describes the subroutines you use to edit objects and mark them for special operations.

16.1 Creating an Object

A graphical object is a list of graphics primitives (drawing subroutines) to display. For example, a drawing of an automobile can be viewed as a compilation of smaller drawings of each of its parts: windows, doors, wheels, and so on. Each part (for example, a wheel) might be a graphical object—a series of `point()`, `line()`, and `polygon()` subroutines.

To make the automobile a graphical object, you first create objects that draw its parts—a wheel object, a door object, a body object, and so on. The automobile object is a series of calls to the part objects, which together with appropriate rotation, translation, and scale subroutines, put all the parts in their correct places.

To create a graphical object, you call `makeobj()`, call the same drawing subroutines you would normally call to draw the object, and then call `closeobj()`. Between the `makeobj()` and `closeobj()` calls, drawing subroutines do not result in immediate drawing on the screen; rather, they are compiled into the object that is being created.

Thus, a graphical object is a list of primitive drawing subroutines to be executed. Drawing the graphical object consists of executing each routine in the listed order. There is no flow control, such as looping, iteration, or condition tests, except for tests that determine whether or not objects are in the viewport, as illustrated in Figure 16-3, in Section 16.2, "Working with Objects."

Note: Not all GL subroutines can be included within a graphical object. A general rule is to include drawing subroutines and not to include subroutines that return values. If you have a question about a particular routine, see the man page for that command.

makeobj

`makeobj()` creates a graphical object:

```
void makeobj (Object obj)
```

The argument *obj* is a 31-bit integer that is associated with the object. If *obj* is the number of an existing object, the contents of that object are deleted.

When `makeobj()` executes, the object number is entered into a symbol table and an empty graphical object is created. Subsequent graphics subroutines are compiled into the graphical object instead of being executed immediately. `makeobj()` creates a new object containing Graphics Library subroutines between `makeobj()` and `closeobj()`

closeobj

`closeobj()` terminates the object definition and closes the open object:

```
void closeobj(void)
```

All the subroutines in the graphical object between `makeobj()` and `closeobj()` are part of the object definition.

If you specify a numeric identifier that is already in use, the system replaces the existing object definition with the new one. To ensure that your object's numeric identifier is unique, use `isobj()` and `genobj()`.

Figure 16-1 shows the sphere defined as a graphical object that is created by the following code:

```
Object obj;
makeobj(sphere=genobj());
for (phi=0; phi<PI; phi+=PI/16) {
    bgnclosedline();
    for(theta=0; theta<2*PI; theta+=PI/18) {
       vert[0] = sin(theta) * cos(phi);
       vert[1] = sin(theta) * sin(phi);
       vert[2] = cos(theta);
       v3f(vert);
    }
    endclosedline();
}
closeobj();
```

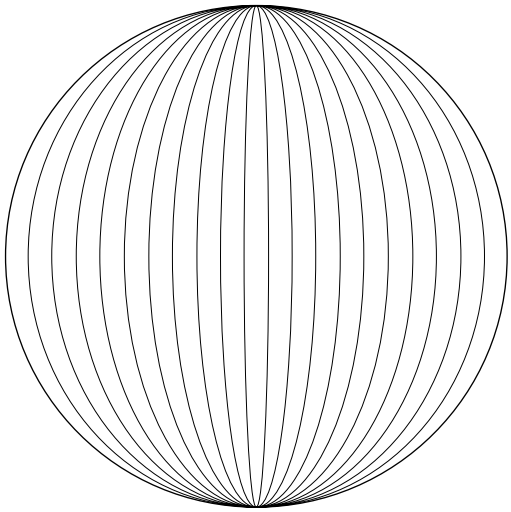

Figure 16-1 Sphere Defined as an Object

isobj

`isobj()` tests whether there is an existing object with a given numeric identifier. Its argument *obj* specifies the desired numeric identifier. `isobj()` returns TRUE if an object exists with the specified numeric identifier and FALSE if none exists.

genobj

`genobj()` generates a unique numeric identifier:

```
Object genobj(void)
```

`genobj()` is useful in naming objects when it is impossible to anticipate what the current numeric identifier will be when the routine is called.

delobj

`delobj()` deletes an object:

```
delobj(Object obj)
```

The system frees all memory storage associated with the deleted object. The numeric identifier is undefined until it is reused to create a new object. The system ignores calls to deleted or undefined objects.

16.2 Working with Objects

You can draw, modify, and delete objects. The following sections describe those operations.

16.2.1 Drawing an Object

Once you create an object, you can draw it with a single `callobj()` command.

`callobj()` draws a created object on the screen:

```
void callobj(Object obj)
```

The argument *obj* takes the numeric identifier of the object you want to draw.

Use `callobj()` to call one object from inside another. You can draw more complex pictures when you use a hierarchy of simple objects. For example, the program below uses a single `callobj(pearl)` to draw the object, a string of pearls, by calling the previously defined object *pearl* seven times.

```
Object pearl = 1, pearls = 2
makeobj(pearl);
    color(BLUE);
    for(angle=0; angle<3600; angle=angle+300) {
        rotate(300, 'y');
        circ(0.0, 0.0, 1.0);
    }
closeobj();
makeobj(pearls);
    for(i=0; i<7; i=i+1) {
        translate(2.0, 0.0, 0.0);
        color(i);
        callobj(pearl);
    }
closeobj();
```

The system does not save global attributes before `callobj()` takes effect. Thus, if an attribute, such as color, changes within an object, the change can affect the caller as well. You can use `pushattributes()` and `popattributes()` to preserve global attributes across `callobj()`.

As another example of using simple objects to build more complex objects, a solar system can be defined as a hierarchical object. Calling the object *solarsystem* draws all the other objects named in its definition (the sun, the planets and their orbits.

When you call a complex object, the system draws the whole hierarchy of objects in its definition. Because the system draws the whole object *solarsystem* it can draw objects that are not visible in the viewport.

Operations known as *pruning* and *culling* guarantee that only the objects that fit within the viewport are drawn. Culling determines which parts of the picture are less than the minimum feature size, and thus too small to draw on the screen. Pruning calculates whether an object is completely outside the viewport.

Figure 16-2 shows the solar system. The diagram below the solar system is a hierarchy diagram, also called a *tree*. Branches in the tree represent calling subroutines.

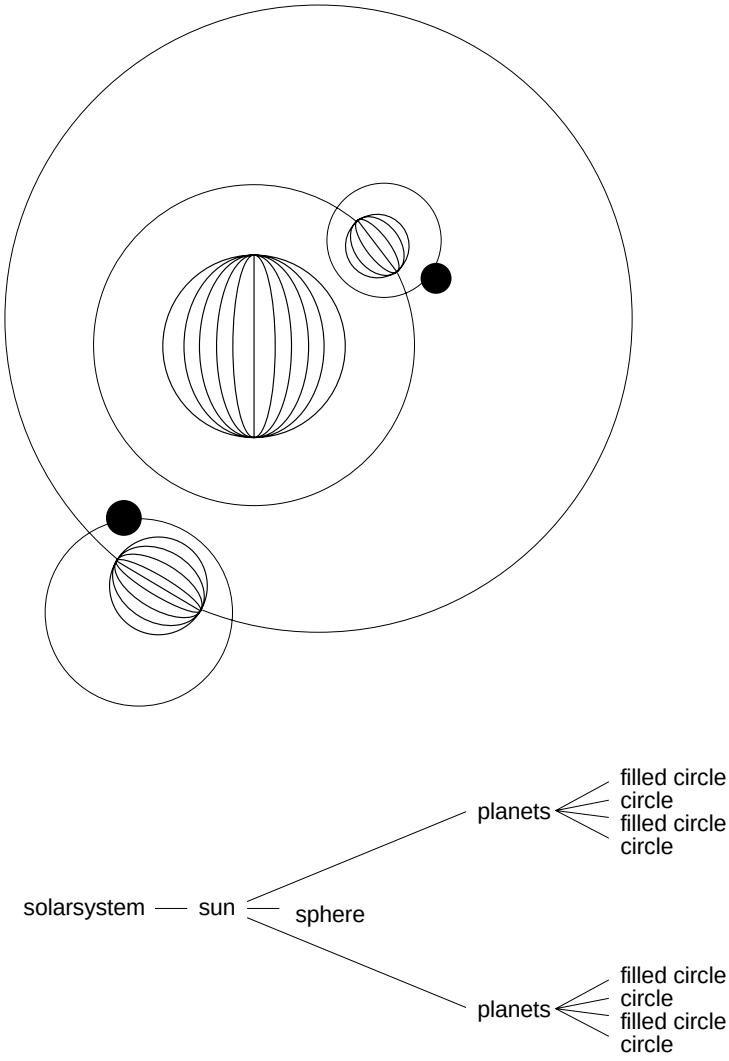

Figure 16-2 Drawing a Hierarchical Object

16.2.2 Bounding Boxes

Bounding boxes can be used to surround objects with irregular surfaces to make it easier to test them for pruning and culling.

Figure 16-3 shows some of the *solarsystem* objects surrounded by their bounding boxes. The bounding boxes can perform pruning to determine which objects are partially within the viewport.

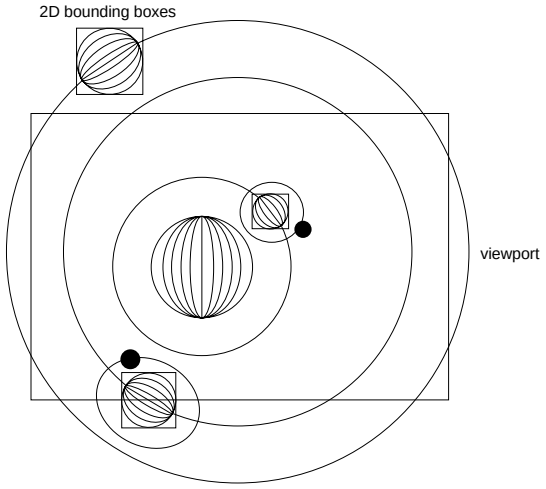

Figure 16-3 Bounding Boxes

bbox2

`bbox2()` determines whether or not an object is within the viewport, and whether it is large enough to be seen, by performing pruning and culling:

```
void bbox2(Screencoord xmin, Screencoord ymin,
          Coord x1, Coord y1, Coord x2, Coord y2)
```

`bbox2()` takes as its arguments an object space bounding box (*x1*, *y1*, *x2*, *y2*) in object coordinates, and minimum horizontal and vertical feature sizes (*xmin, ymin)* in pixels. The system calculates the bounding box, transforms it to screen coordinates, and compares it with the viewport. If the bounding box is completely outside the viewport, the subroutines between bbox2 and the end of the object are ignored.

If the bounding box is within the viewport, the system compares it with the minimum feature size. If it is too small in both the *x* and *y* dimensions, the rest of the subroutines in the object are ignored. Otherwise, the system continues to interpret the object.

16.2.3 Editing Objects

You can change an object by editing it. Editing requires that you identify and locate the drawing subroutines you want to change. You use two types of subroutines when you edit an object:

edit	add, remove, or replace drawing subroutines
tag	identify locations of drawing subroutines within an object

If you have to edit graphical objects frequently, you should build your own custom data structures and traversal subroutines, rather than use graphical objects. The editing subroutines that follow are best suited for infrequent and simple editing operations.

editobj

To open an object for editing, use `editobj()`:

```
void editobj(Object obj)
```

A pointer acts as a cursor that appends new subroutines. The pointer is initially set to the end of the object. The system appends graphics subroutines to the object until either a `closeobj()` or a pointer positioning routine `objdelete()`, (`objinsert()`, or `objreplace()`) executes.

The system interprets the editing subroutines following `editobj()`. Use `closeobj()` to terminate your editing session. If you specify an undefined object, an error message appears.

getopenobj

To determine if any object is open for editing, use `getopenobj()`:

```
Object getopenobj(void)
```

If an object is open, it returns the object's id. It returns -1 if no object is open.

16.2.4 Using Tags

Tags locate items within a graphical object that you want to edit. Editing subroutines require tag names as arguments. STARTTAG is a predefined tag that goes before the first item in the list; it marks the beginning of the list. STARTTAG does not have any effect on drawing or modifying the object. Use it only to return to (find) the beginning of the list.

ENDTAG is a predefined tag that is positioned after the last item on the list; it marks the end of the list. Like STARTTAG, ENDTAG does not have any effect on drawing or modifying the object. Use it to find the end of the graphical object. When you call `makeobj()` to create a list, STARTTAG and ENDTAG automatically appear. You cannot delete these tags. When an object is opened for editing, there is a pointer at ENDTAG, just after the last routine in the object. To perform edits on other items, refer to them by their tags.

maketag

You can use tags to mark items you may want to change. You explicitly tag subroutines with `maketag()`:

```
void maketag(Tag t)
```

Specify a 31-bit numeric identifier for *t*. The system places a marker (tag) between two items. You can use the same tag name in different objects.

newtag

`newtag()` also adds tags to an object, but uses an existing tag to determine its relative position within the object. `newtag()` creates a new tag that is offset beyond the other tag by the number of lines given in its argument *offst*.:

```
void newtag(Tag newtg, Tag oldtg, Offset offst)
```

istag

`istag()` tells whether a given tag is in use within the current open object:

```
Boolean istag(Tag t)
```

`istag()` returns TRUE if the tag is in use, and FALSE if it is not. The result is undefined if there is no currently open object.

gentag

`gentag()` generates a unique integer to use as a tag within the current open object:

```
Tag gentag(void)
```

deltag

`deltag()` deletes tags from the object currently open for editing:

```
void deltag(Tag t)
```

Note: You cannot delete the special tags `STARTTAG` and `ENDTAG`.

16.2.5 Inserting, Deleting, and Replacing within Objects

The subroutines `objinsert()`, `objdelete()`, and `objreplace()` allow you to add, delete, or replace subroutines in a graphical object.

objinsert

Use `objinsert()` to add subroutines to an object at the location specified in *t*.:

```
void objinsert(Tag t)
```

`objinsert()` positions an editing pointer on the tag you specify in *t*. The system inserts graphics subroutines immediately after the tag. To terminate the insertion, use `closeobj()` or another editing routine (`objdelete()`, `objinsert()`, `objreplace()`).

objdelete

`objdelete()` removes subroutines from the current open object:

```
void objdelete(Tag tag1, Tag tag2)
```

`objdelete()` removes everything between *tag1* and *tag2*, deletes subroutines and other tag names, and leaves the pointer at the end of the object after it executes. For example, `objdelete(STARTTAG, ENDTAG)` deletes every drawing routine. `objdelete()` is ignored if no object is open for editing.

objreplace

`objreplace()` combines the functions of `objdelete()` and `objinsert()`:

```
void objreplace(Tag t)
```

This provides a quick way to replace one drawing routine with another that occupies the same amount of space in the graphical object. Its argument is a single tag, *t*. Graphics subroutines that follow `objreplace()` overwrite existing subroutines until a `closeobj()` or editing routine (`objinsert()`, `objreplace()`, `objdelete()`) terminates the replacement.

Note: `objreplace()` requires that the new routine to be exactly the same length in characters as the previous one. Use `objdelete()` and `objinsert()` for more general replacement.

Example—Editing an Object

The following is an example of object editing. First, the object *star* is defined:

```
makeobj(star);
   color(GREEN);
   maketag(BOX);
   recti(1, 1, 9, 9);
   maketag(INNER);
   color(BLUE);
   poly2i(8, Inner);
   maketag(OUTER);
   color(RED);
   poly2i(8, Outer);
   maketag(CENTER);
   color(YELLOW);
   pnt2i(5, 5);
closeobj();
editobj(star);
   circi(1, 5, 5);
   objinsert(BOX);
   recti(0, 0, 10, 10);
   objreplace(INNER);
   color(GREEN);
closeobj();
```

The object resulting from the editing session is equivalent to an object created by the following code:

```
makeobj(star);
   color(GREEN);
   maketag(BOX);
   recti(0, 0, 10, 10);
   recti(1, 1, 9, 9);
   maketag(INNER);
   color(GREEN);
   poly2i(8, Inner);
   maketag(OUTER);
   color(RED);
   poly2i(8, Outer);
   maketag(CENTER);
   color(YELLOW);
   pnt2i(5, 5);
   circi(1, 5, 5);
closeobj();
```

16.2.6 Managing Object Memory

Editing can require large amounts of memory. The subroutines `compactify()` and `chunksize()` perform memory management tasks.

compactify

As memory is modified by the various editing subroutines, an open object can become fragmented and be stored inefficiently. When the amount of wasted space becomes large, the system automatically calls `compactify()` during the `closeobj()` operation.

`compactify()` allows you to perform the compaction explicitly:

```
void compactify(Object obj)
```

Unless you insert new subroutines in the middle of an object, compaction is not necessary.

Note: `compactify()` uses a significant amount of computing time. Do not call it unless the amount of available storage space is critical; use it sparingly when performance is a consideration.

chunksize

`chunksize()` lets you specify the minimum chunk of memory necessary to accommodate the largest number of vertices you want to call:

```
void chunksize(long chunk)
```

If there is a memory shortage, you can use `chunksize()` to allocate memory for an object. `chunksize()` specifies the minimum amount of memory that the system allocates to an object. The default *chunk* is 1020 bytes. When you specify *chunk*, its size should vary according to the needs of the application. As the object grows, more memory is allocated in units of size *chunk*. Call `chunksize()` only once after `winopen()`, and before the first `makeobj()`.

16.2.7 Mapping Screen Coordinates to World Coordinates

This section describes how to map screen coordinates to world coordinates.

mapw

`mapw()` takes a 2-D screen point and maps it onto a line in 3-D world space. Its argument *vobj* contains the viewing, projection, and viewport transformations that map the current displayed objects to the screen.

`mapw()` reverses these transformations and maps the screen coordinates back to world coordinates. It returns two points (*wx1*, *wy1*,*wz1)* and (*x2*, *wy2*, *wz2)*, which specify two different points on the line. The length of the line is arbitrary. s*x* and *sy* specify the screen point to be mapped.

```
void mapw(Object vobj, Screencoord sx, Screencoord sy, Coord *wx1,Coord *wy1,
Coord *wz1, Coord *wx2, Coord *wy2, Coord *wz2)
```

mapw2

`mapw2()` is the 2-D version of `mapw()`. In two dimensions, the system maps screen coordinates to world coordinates rather than to a line. Again, *vobj* contains the projection and viewing transformations that map the displayed objects to world coordinates; *sx* and *sy* define screen coordinates. w*x* and *wy* return the corresponding world coordinates. If the transformations in *vobj* are not 2D (i.e., not orthogonal projections), the result is undefined.

```
void mapw2(Object vobj, Screencoord sx, Screencoord sy, Coord *wx,Coord *wy)
```