Chapter 6

Animation

This chapter describes the subroutines that you use to create continuous motion in graphics scenes. This chapter introduces a technique called *double buffering*, which allows you to create graphics that are animated in the kind of smooth motion that looks like a movie.

- Section 6.1, "Understanding How Animation Works," presents an overview on how animation is done on the IRIS.
- Section 6.2, "Creating an Animation," tells you how to set up your system to do animation, and maximize animation performance.

6.1 Understanding How Animation Works

You can address frame buffer memory in either of two modes:

- Single buffer mode - a program addresses frame buffer memory as a single buffer whose pixels are always visible.
- Double buffer mode - a program addresses frame buffer memory as if it were two buffers, only one of which is displayed at a time.

The currently visible buffer is called the *front buffer* and the invisible, drawing buffer is the *back buffer*. The display hardware in the system constantly reads the contents of the visible buffer (the front buffer in double buffer mode), and displays those results on the screen. On a standard monitor, the electron guns sweep from the top of the screen to the bottom, refreshing all pixels, 60 to 76 times each second. If the graphics hardware changes the contents of the visible frame buffer, the next time the refresh hardware reads a changed pixel, the system draws the new value instead of the old one.

6.1.1 Single Buffer Mode

By default, the system is in single buffer mode. Whatever you draw into the bitplanes is immediately visible on the screen. For static drawings, this is acceptable, but it does not provide smooth animated motion. If you try to animate a drawing in single buffer mode, you can see a visible flicker in all but the simplest drawing operations.

In single buffer mode, the system simultaneously updates and displays the image data in the active bitplanes; consequently, incomplete or changing pictures can appear on the screen. `singlebuffer()` does not take effect until `gconfig()` is called.

6.1.2 Double Buffer Mode

For smooth motion, it is preferable to display a completely drawn image for a certain time (for instance, a few 60ths of a second), then present the next frame, also completely drawn, during the next time period. Scene frames are interleaved in this manner so that the changes from one frame to the next cannot be detected by your eye. If the frames are not swapped quickly enough, your eye can detect this motion and perceive it as flicker.

In double buffer mode, the bitplanes are partitioned into two groups, the front bitplanes and the back bitplanes. Double buffering works in either RGB mode or color map mode. Use `doublebuffer()` to set the display mode to double buffer mode. The mode change does not take effect until you call `gconfig()`. In double buffer mode, only the front bitplanes are displayed, and drawing routines normally update only the back bitplanes; `frontbuffer()` and `backbuffer()` can override this default. `gconfig()` sets `frontbuffer()` to `FALSE` and `backbuffer()` to `TRUE` in double buffer mode.

6.2 Creating an Animation

Before drawing anything, you must set the frame buffers into the correct configuration, such as color map mode or RGB mode.

6.2.1 Setting Modes for Animation

Configuring the frame buffer is a two-step process:

1. Indicate how to configure the frame buffer.
2. Call `gconfig()` to set the system into that particular configuration.

After a `gconfig()` call, the current writemask and color are no longer defined.

6.2.2 Swapping Buffers

After the entire frame is rendered into the back buffer, `swapbuffers()` is called to make it the visible buffer. The `swapbuffers()` call is ignored in single buffer mode. The `swapbuffers()` subroutine waits for the next screen refresh before exchanging the front and back buffers. If it did not wait, a frame would be drawn partly in one buffer, and partly in the other, causing a serious visual disturbance.

Because screen refresh occurs approximately every 60th of a second on the standard monitor, `swapbuffers()` can block the running process for up to that long. The default monitor is refreshed 60 to 76 times per second. Other monitor options can have other retrace periods. See `setmonitor()` and `getmonitor()` in Chapter 5.

Because `swapbuffers()` blocks the user program until the next screen refresh (1/60 second), every frame takes *n* screen refreshes to render and display, where *n* is the ceiling function of the actual rendering time. For example, if a scene takes 1.9 refreshes to render, then every frame takes 2 refreshes to render and display. Therefore, the application performs at 60/2 or 30 frames per second. If you add another polygon to the scene, and it now takes 2.1 refreshes to render, or 3 refreshes to render and display, the frame rate drops from 30 to 60/3 or 20 frames per second. There is no smooth degradation. While the geometry is moving about, the time it takes to render each frame varies.

6.2.3 Swapping Multiple Buffers

On IRIS-4D/VGX Series, SkyWriter, and RealityEngine systems, both overlay and underlay planes can also be double buffered. The `mswapbuffers()` subroutine lets you swap any combination of the available frame buffers at the

same time. Just like `swapbuffers()`, `mswapbuffers()` blocks until the next vertical retrace period. It is ignored by frame buffers that are not in double buffer mode.

The constants you use to indicate the buffers to be swapped are

`NORMALDRAW`	Swap the front and back buffers of the normal color bitplanes.
`OVERDRAW`	Swap the front and back buffers of the overlay bitplanes.
`UNDERDRAW`	Swap the front and back buffers of the underlay bitplanes.

These constants can be bitwise-ORed together to swap multiple buffers simultaneously. For example, to swap front and back for the normal frame buffer and for the underlay planes, include this line in your program:

```
mswapbuffers(NORMALDRAW | UNDERDRAW);
```

This sample program, *bounce.c*, demonstrates animation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>
#define RGB_BLACK 0x000000
#define RGB_WHITE 0xffffff
#define X           0
#define Y           1
#define XY          2
#define SIZE        0.05
#define BOUNDS      (1.0 + SIZE)
#define EDGE        (1.1 * BOUNDS)

float boundary[4][XY] = {
    {-BOUNDS, -BOUNDS},
    {-BOUNDS, BOUNDS},
    { BOUNDS, BOUNDS},
    { BOUNDS, -BOUNDS}
};


struct ball_s {
    float pos[XY];
    float delta[XY];
    unsigned long col;
};
```

```
void main(int argc, char *argv[])
{
    int i, j;
    int nballs;
    struct ball_s *balls;
    short val;
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB not available on this machine\n");
        return 1;
    }
    if (argc != 2) {
        fprintf(stderr, "Usage: bounce <ball count>\n");
        return 1;
    }
    nballs = atoi(argv[1]);
    if (!(balls = (struct ball_s *)malloc(nballs * sizeof(struct ball_s)))) {
        fprintf(stderr, "bounce: malloc failed\n");
        return 1;
    }
    for (i = 0; i < nballs; i++) {
        for (j = 0; j < XY; j++) {
            balls[i].pos[j] = 2.0 * (drand48() - 0.5);
            balls[i].delta[j] = 2.0 * (drand48() - 0.5) / 50.0;
        }
        balls[i].col = drand48() * 0xffffff;
    }
    prefsize(400, 400);
    winopen("bounce");
    doublebuffer();
    RGBmode();
    gconfig();
    shademodel(FLAT);
    qdevice(ESCKEY);
    ortho2(-EDGE, EDGE, -EDGE, EDGE);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        for (i = 0; i < nballs; i++) {
             for (j = 0; j < XY; j++) {
                    balls[i].pos[j] += balls[i].delta[j];
                    if ((balls[i].pos[j] >= 1.0) || (balls[i].pos[j] <= -1.0))
                        balls[i].delta[j] = -balls[i].delta[j];
                }
            }
            cpack(RGB_BLACK);
            clear();
            cpack(RGB_WHITE);
            bgnclosedline();
```

```
            for (i = 0; i < 4; i++)
                v2f(boundary[i]);
            endclosedline();
            for (i = 0; i < nballs; i++) {
                cpack(balls[i].col);
                sboxf(balls[i].pos[X]-SIZE, balls[i].pos[Y]-SIZE,
                        balls[i].pos[X]+SIZE, balls[i].pos[Y]+SIZE);
        }
        swapbuffers();
    }
    gexit();
    return 0;
}
```

6.2.4 Maximizing Animation Performance

The subroutine calls contained in the remainder of this chapter might be considered advanced topics. You do not need to use these calls unless you are tuning your program for maximum performance.

Sometimes in double buffer mode, it is useful to be able to write the same thing into both buffers at once. For example, suppose an animated image has both a fixed part and a changing part. The fixed part needs to be drawn only once, but into both buffers. It is most easily done by enabling the front buffer (as well as the back buffer) for writing, drawing the image, and then disabling the front buffer. The animation then proceeds by drawing the changing part of the image using the usual double buffering techniques.

When the value of the argument *b* is `FALSE` (the default value), the front buffer is not enabled for writing. When the value of *b* is `TRUE`, the front buffer is enabled for writing. This routine is useful only in double buffer mode. `frontbuffer(TRUE)` in double buffer mode enables simultaneous updating of (or writing into) both the front and the rear buffers. `gconfig()` sets `frontbuffer()` to `FALSE`.

backbuffer

It is sometimes convenient to update both the front and the back buffers, or to update the front buffer instead of the back. `backbuffer()` enables updating in the back buffer. Its argument, *b*, is a Boolean value. When the value of *b* is `TRUE`,

the default, the back buffer is enabled for writing. When the value of *b* is FALSE, the back buffer is not enabled for writing.

getbuffer

`getbuffer()` indicates which buffer(s) are enabled for writing in double buffer mode. The returned values operate as a bitmask (that is, the number returned represents the numeric value of all the bits currently set). The default, 1, means the back buffer is enabled (as does any odd value); 2 means that the front buffer is enabled (or any value in which the 2 bit is set); and 3 (or value in which the 3 bit is set) means that both are enabled. `getbuffer()` returns zero if neither buffer is enabled or if the system is not in double buffer mode. If the z-buffer (see Chapter 8) is enabled for drawing, `getbuffer()` can return 4, 5, 6, or 7.

Each of the possible values also has an associated symbolic value that you can use in the call to `getbuffer()`. See the `getbuffer()` man page for a list of these symbols.

swapinterval

`swapinterval()` defines a minimum time between buffer swaps. For example, a swap interval of 5 refreshes the screen at least five times between execution of successive calls to `swapbuffers()`. `swapinterval()` is typically used when you want to show frames at a constant rate, but the images vary in complexity. To achieve a constant rate, set the swap interval long enough that even the most complex frame can be drawn in that time. For drawing a simple frame, the user's process simply blocks and waits until the swap interval is used up. The default interval is 1.

Note: `swapinterval()` is valid only in double buffer mode.

getdisplaymode

`getdisplaymode()` returns the current display mode: 0 indicates RGB single buffer mode; 1 indicates single buffer mode; 2 indicates double buffer mode; 5 indicates RGB double buffer mode. Modes 3 and 4 are unused.

gsync

`gsync()` pauses execution until a vertical retrace occurs. It was intended as a method to synchronize drawing with the vertical retrace to achieve animation in single buffer mode, but it is not precise since the retrace rate varies among systems. You might think of calling `gsync()` a number of times to get short time delays (because each call waits until the next 60th of a second), but this is not the recommended procedure for setting up a delay. Use the system call `sginap()` instead.

`gsync()` is also used for rubber-banding in single buffer mode.

`gsync()` is included primarily for compatibility with systems that might not have enough bitplanes to use double buffer mode. `gsync()` waits for the next vertical retrace period. Due to pipeline and operating system delays, smooth motion in single buffer mode is often impossible. `gsync()` should be used only as a last resort, and it might not work.