Game Programming with the Allegro C Library

E

One picture is worth ten thousand words.
—Chinese proverb

Treat nature in terms of the cylinder, the sphere, the cone, all in perspective.
—Paul Cezanne

Nothing ever becomes real till it is experienced—even a proverb is no proverb to you till your life has illustrated it.
—John Keats

Objectives

In this appendix, you'll learn:

- How to install the Allegro game programming library to work with your C programs.
- To create games using Allegro.
- To import and display graphics.
- To use "double buffering" to create smooth animations.
- To import and play sounds.
- To recognize and process keyboard input.
- To create the simple game Pong.
- To use timers to regulate the speed of a game.
- To use datafiles to shorten the amount of code in a program.

Outline

E.1 Introduction

We now present game programming and graphics with the Allegro C library. Created in 1995 by Climax game programmer Shawn Hargreaves, **Allegro** is now an open source project maintained by the game developer community at `www.allegro.cc`.

In this appendix, we show how to use the Allegro C library to create the simple game of Pong. We demonstrate how to display graphics and smoothly animate moving objects. We explain additional features such as sound, keyboard input, text output and timers that are useful in creating games. The appendix includes web links to over 1,000 open source Allegro games and tutorials on advanced Allegro techniques.

E.2 Installing Allegro

Allegro was created to be a relatively simple programming library, but its installation can be tricky. Here we cover installation using Visual C++ 2008 Express Edition on Windows and GCC 4.3 on Linux. The full source code download of Allegro contains instructions for compiling from source on a wide variety of platforms in the `docs/build` directory.

Installing Allegro on Windows

To begin, you need the Allegro library itself, which is available at `www.allegro.cc/files`. Download the pre-compiled Allegro 4.2.2 for Microsoft Visual C++ 9.0 and the zip file containing the tools and examples. Extract these files to a convenient location, such as `C:\Allegro`.

In Visual Studio 2008, go to **Tools > Options > Projects and Solutions > VC++ Directories**, and select **Include files** from the **Show Directories for:** combobox. Select a blank line and browse for the `include` folder included with the Allegro download. Next, select **Library files** from the combobox and browse for the `lib` folder. Finally, copy all of the `.dll` files from the `bin` folder included with the Allegro download into the `C:\Windows\System32` folder.

Once the installation is complete, you must tell Visual Studio where to find the Allegro library when you create a new project. To do this, create a new **Win32 Project**. In the **Win32 Application Wizard**, select **Windows application** as the application type, and select the **Empty project** checkbox. Once the project has been created, go to **Project > Properties**, then **Configuration Properties > Linker > Input**. Select **Additional Dependencies** and add "`alleg.lib`". You must perform this step for every Allegro project you create. This

appendix's examples folder includes an empty project and solution with the Allegro library already added as a dependency that you can use for running the programs.

Installing Allegro on Linux

Most Linux distributions have precompiled packages for Allegro, though the exact name of the package differs. On Debian and Debian-based distributions such as Ubuntu, the package is `liballegro4.2-dev`; on Fedora it is called `allegro`; and on openSUSE it is split into the `allegro-devel` and `allegro-tools` packages. Install the proper Allegro package(s) and the `make` package using your distribution's package manager (e.g., Synaptic, PackageKit or YaST).

The `make` program makes compilation easier by letting you store instructions for compiling programs in a file called `Makefile` instead of typing them out on the command line each time you build the program. We have supplied a `Makefile` with the examples for this appendix—simply running "`make`" from inside the directory will use the `Makefile` to compile all of the examples in the appendix. If you are creating your own project, you can adapt our `Makefile` or use the `allegro-config` program, which displays the compiler options needed to compile and link Allegro programs.

E.3 A Simple Allegro Program

Consider a simple Allegro program that displays a message (Fig. E.1). In addition to the preprocessor directive to include the `allegro.h` header at line 3 (this directive must be present for an Allegro program to operate correctly), you will notice several functions in this program that we have not previously used. The function call `allegro_init` (line 7) initializes the Allegro library. It is required for an Allegro program to run, and must be called before any other Allegro functions.

```
1  /* Fig. E.1: figE_01.c
2     A simple Allegro program. */
3  #include <allegro.h>
4
5  int main( void )
6  {
7     allegro_init(); /* initialize Allegro */
8     allegro_message( "Welcome to Allegro!" ); /* display a message */
9     return 0;
10 } /* end function main */
11 END_OF_MAIN() /* Allegro-specific macro */
```

Fig. E.1 | A simple Allegro program.

The call to `allegro_message` (line 8) displays a message to the user in the form of an alert box. Since alert boxes interrupt running programs, you will probably not use this function much—it is preferable to use Allegro's text display functions to display messages on the screen without stopping the program. The `allegro_message` function is typically used to notify the user if something is preventing Allegro from working correctly.

The use of the `END_OF_MAIN` macro is the last of the new elements in this program. Windows, some Unix systems, and Mac OS X cannot run Allegro programs without this macro, as the executable that an Allegro program creates during compilation will not be able start correctly on those systems. The `END_OF_MAIN` macro checks the currently running operating system during compilation, and applies an appropriate fix to this problem if one is required. On systems other than the ones listed previously, `END_OF_MAIN` will do nothing, and your program will compile and run perfectly without it. However, you should still always add the macro after the right brace of your `main` to ensure that your program is compatible with systems that require it.

So, to get any Allegro program to run, you need to have three lines of code—the preprocessor directive to include `allegro.h`, the call to `allegro_init` and (if your system requires it) the call to the `END_OF_MAIN` macro following the right brace of your program's `main` function. Now we discuss Allegro's main capability—displaying graphics.

E.4 Simple Graphics: Importing Bitmaps and Blitting

Allegro can draw lines and simple shapes on its own, but the majority of the graphics it can display come from external sources. The `allegro.h` header defines several types that can be used to hold image data from external files—the **`BITMAP*` pointer type** is the most basic of these. A `BITMAP*` points to a `struct` in memory where image data is stored. Allegro has many functions that can be used to manipulate bitmaps. The most important of these are shown in Fig. E.2.

Function prototype	Description
`BITMAP *create_bitmap(int width, int height)`	Creates and returns a pointer to a blank bitmap with specified width and height (in pixels).
`BITMAP *load_bitmap(const char *filename, RGB *pal)`	Loads and returns a pointer to a bitmap from the location specified in `filename` with palette `pal`.
`void clear_bitmap(BITMAP *bmp)`	Clears a bitmap of its image data and makes it blank.
`void clear_to_color(BITMAP *bmp, int color)`	Clears a bitmap of its image data and makes the entire bitmap the color specified.
`void destroy_bitmap(BITMAP *bmp)`	Destroys a bitmap and frees up the memory previously allocated to it. Use this function when you are done with a bitmap to prevent memory leaks.

Fig. E.2 | Important BITMAP functions.

Now let's use Allegro's functions to load a bitmap and display it on the screen until a key is pressed. First we need a bitmap to load from, so copy the bitmap `picture.bmp` included with this appendix's examples and save it in the same folder as your Allegro project. The code in Fig. E.3 displays the bitmap until a key is pressed.

Common Programming Error E.1

Telling Allegro to load a file that is not in the same folder as the program being run will cause a runtime error, unless you specifically tell the program the folder in which the file is located by typing the full path name.

```
1  /* Fig. E.3: figE_03.c
2     Displaying a bitmap on the screen. */
3  #include <allegro.h>
4
5  int main( void )
6  {
7     BITMAP *bmp; /* pointer to the bitmap */
8
9     allegro_init(); /* initialize Allegro */
10    install_keyboard(); /* allow Allegro to receive keyboard input */
11    set_color_depth( 16 ); /* set the color depth to 16-bit*/
12    set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
13    bmp = load_bitmap( "picture.bmp", NULL ); /* load the bitmap file */
14    blit( bmp, screen, 0, 0, 0, 0, bmp->w, bmp->h ); /* draw the bitmap */
15    readkey(); /* wait for a keypress */
16    destroy_bitmap( bmp ); /* free the memory allocated to bmp */
17    return 0;
18 } /* end function main */
19 END_OF_MAIN() /* Allegro-specific macro */
```

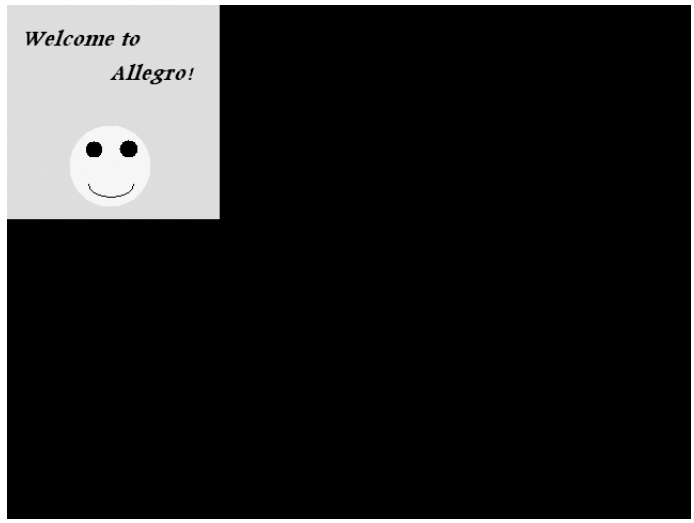

Fig. E.3 | Displaying a bitmap on the screen.

First, this program initializes the Allegro library and adds the ability to read keyboard input. Then, it sets the graphics mode of the program. Finally, it loads `picture.bmp`, displays it at the top-left corner of the screen, then waits for a key press. When a key is pressed, the program frees the memory that was allocated to the bitmap, then exits.

There are many new Allegro functions in this program. The call to `install_keyboard` at line 10 "installs" the keyboard so that Allegro can recognize and use it. It doesn't have anything to do with displaying our bitmap, but it needs to be called so that Allegro will know how to "wait" for a key press after the bitmap is displayed.

Line 11 calls our first graphics-related function, `set_color_depth`, which sets the color depth of the screen to *n*-bit, where *n* is the `int` passed to the function. A color depth of *n*-bits means that there are 2^n possible colors that can be displayed. A lower color depth will make a program require less memory, but its appearance may not be as good as a higher one. While we set the color depth to 16-bit in our program, you can use a different color depth if you wish, but the only values that `set_color_depth` accepts are 8, 15, 16, 24, and 32. It is advised, though, that you avoid using an 8-bit depth until you are more experienced with Allegro. If the color depth is set to 8-bit, loading a bitmap also requires that you load the bitmap's palette—in other words, the set of colors that the bitmap uses—which is a process that is beyond our scope. At the least, a 16-bit color depth is recommended. A 15-bit depth is also acceptable, but not all platforms support it.

In line 12 we call another graphics-related function, **`set_gfx_mode`**. This call sets the graphics mode of the program. This is an important function; let's examine its prototype.

```
int set_gfx_mode(int card, int width, int height, int v_w, int v_h);
```

If the function is successful, it returns 0; otherwise, it returns a non-zero value. In general, most Allegro functions that can fail follow the same paradigm. If you wish, you can insert `if` statements into your program to check if your functions work correctly and tell the program how to proceed if one fails, but to save space, our sample programs will assume that all our functions work as they should.

The first parameter of the `set_gfx_mode` function is `int card`. In earlier versions of Allegro, this parameter was used to tell the computer which video card driver to use, but determining which driver would work correctly with a given system and program was a difficult process. In newer versions of the library, a number of so-called "magic drivers" were added that detect the computer's drivers automatically—`GFX_AUTODETECT`, `GFX_AUTODETECT_FULLSCREEN`, `GFX_AUTODETECT_WINDOWED`, and `GFX_SAFE` (all of these are symbolic constants defined in `allegro.h`). In addition to specifying the driver to use, passing the parameter one of the preceding values also tells Allegro whether it should run the program in a window, or whether the program should use the entire screen. Passing `GFX_AUTODETECT_FULLSCREEN` or `GFX_AUTODETECT_WINDOWED` tells the program to run in fullscreen mode (the program takes up the entire screen) or windowed mode (the program runs in a standard window), respectively. Passing `GFX_AUTODETECT` makes the program try fullscreen mode first, then try windowed mode if fullscreen mode causes an error. `GFX_SAFE` mode is generally not used; though it acts the same way as `GFX_AUTODETECT`, there is one addition. If both fullscreen and windowed mode fail, `GFX_SAFE` mode forces the computer to use settings that it "knows" will work. However these "safe" modes usually have extremely low resolution and color depth, so they are generally avoided. There is also a fifth symbolic constant, `GFX_TEXT`, but this setting allows only text as the name

implies and passing it to `set_gfx_mode` will essentially "turn off" any window or fullscreen graphics that are already running.

Software Engineering Observation E.1

Avoid using the `GFX_SAFE` "magic driver" if possible. The "safe" graphics modes generally have a negative impact on your program's appearance.

The next two parameters, `width` and `height`, determine the number of pixels in the width and height of the screen (or the window, if you're using windowed mode). The last two parameters (`v_w` and `v_h`) define the minimum pixel width and height, respectively, of what is called the "**virtual screen**." The "virtual screen" was used in earlier versions of Allegro to help create games where action can occur out of the view of the visible screen, but it no longer has any real use in Allegro's current version as most systems today do not support it. Thus, the `v_w` and `v_h` parameters should simply be passed the value 0.

Now that we know how to set the graphics mode, we consider the function at line 13, `load_bitmap`. This function, introduced in Fig. E.2, loads the picture that we saved as `picture.bmp` and has the pointer `bmp` point to it. We did not pass the function a palette—this is not necessary. Recall that passing a bitmap's palette is required only when the color depth is set to 8-bit. However, it is important that you make sure you set the color depth and graphics mode in your program before you attempt to load a bitmap file. Otherwise, Allegro will have no information on how to store the bitmap in memory (it will not know how many bits in memory to use for each pixel, for example), and will instead try to "guess" how to do so. This can lead to various errors in your program.

Common Programming Error E.2

Loading a bitmap before setting the color depth and graphics mode of a program will likely result in Allegro storing the bitmap incorrectly.

If `load_bitmap` fails—if, for example, you tell it to load a bitmap that is not there—then it does not actually cause an error on that line. Instead, telling `load_bitmap` to load a nonexistent file simply makes the function return the value `NULL`. Obviously, this can cause problems later in the program, but compilers do not recognize a failed call to `load_bitmap` as an error. In general, any Allegro function that loads an external file behaves in the same way.

Line 14 contains a call to what is probably the most important function in this program, `blit`. The **`blit`** function (`blit` stands for BLock Transfer—the "i" is there only to make it pronounceable) is an Allegro function that takes a block of one bitmap (the block can be the entire picture, if you wish) and draws it onto another. Consider this function's prototype:

```
void blit(BITMAP *source, BITMAP *dest, int source_x, int source_y,
   int dest_x, int dest_y, int width, int height);
```

Because `blit` draws one bitmap onto another, we must specify those bitmaps for the function to work. The first two parameters (`source` and `dest`) define the source and destination bitmaps, respectively, so in our program we are taking a block of `bmp` and drawing it onto the `screen`. The symbolic constant `screen` (defined in `allegro.h`) refers to the screen of the computer. Blitting onto the screen is the only way to display graphics.

Since blitting takes a block from a bitmap, we also have to specify the position of this block in the source bitmap. The parameters `source_x` and `source_y` specify the coordinates of the top-left corner of the block we want to draw onto the destination bitmap. Allegro bitmap coordinates do not work the same way as they do on most graphs in your algebra and geometry classes: A larger *x*-value means further to the right, but a larger *y*-value means further down, not up. This means that the top-left point of any bitmap image is at the coordinates (0, 0). Figure E.4 illustrates this coordinate system using the output of the program we wrote in Fig. E.3.

Fig. E.4 | Allegro's coordinate system.

To draw onto a bitmap, we must also tell the function where we want the drawing to take place. The parameters `dest_x` and `dest_y` determine the coordinates of the top-left point of this location. Again, the coordinates (0, 0) represent the top left of a bitmap.

Finally, though we have specified the top-left corner of the block we want to copy from our source bitmap, we have not yet specified its size. The last two parameters specify the `width` and `height` of the block we are blitting. You may be wondering why in our program, we were able to pass these parameters the values of `bmp->w` and `bmp->h`. The reason for this is that the **`BITMAP*` type** defined by Allegro is a pointer to a `struct`. Along with the image data and a few other variables, the `BITMAP struct` contains two `int`s, `w` and `h`, that represent the width and height of the bitmap, respectively. Unless you do not want to display the entire image, you should pass these parameters the values `bmp->w` and `bmp->h`, replacing `bmp` with the name of your bitmap.

The `blit` function that we called at line 14 then copies a block from `bmp`, whose top-left corner and size are those of `bmp`, and draws it at the top-left corner of the virtual screen. In short, it displays the image in `picture.bmp` at the top-left of the screen.

After blitting the image onto the screen, the program makes a call to the function `readkey`. This function is the reason we called `install_keyboard` earlier in the program. Function `readkey` waits for a key to be pressed, then returns the value of the key pressed as an `int`. Even though we do nothing with the value it returns, this function is useful in this program because, like `scanf`, it causes the program to pause until the user provides some input. Without it, the program would end before we had a chance to see the bitmap.

Finally, after we hit a key, the program calls `destroy_bitmap`. As explained previously, this function destroys the bitmap passed to it and performs the equivalent of the `free` function on the memory that was allocated to it.

Error-Prevention Tip E.1

Use the `destroy_bitmap` function to free the memory of a bitmap that is no longer needed and prevent memory leaks.

Common Programming Error E.3

Trying to destroy a bitmap that has not been initialized causes a runtime error.

E.5 Animation with Double Buffering

Now that we know how to draw bitmaps on the screen, animating bitmaps and making them move around the screen is straightforward. We are now going to develop the simple game "Pong." We begin by using animation techniques to create a square "ball" that bounces around a white screen. For this purpose, we need a bitmap that serves as our "ball." Copy the file `ball.bmp` from this appendix's examples folder, and save it in a new project called `pong`. The image `ball.bmp` is 40 pixels wide by 40 pixels tall—this will be important as we code the program.

In most Pong games, the ball can travel at many different angles. However, since we are just starting with Allegro, we want to keep things as simple as possible. For this reason, in our Pong game, the ball only has four possible directions of travel: down-right, up-right, down-left, and up-left—all of these at 45-degree angles to the *x*- and *y*-axes in our program. We use an `int` to keep track of the ball's current direction, so we can use symbolic constants for each of the possible directions.

Moving a bitmap around the screen is simple—we just blit our bitmap onto the screen, then when we want to move it, we clear the screen and blit the bitmap at its new position. The program in Fig. E.5 creates a ball that moves around and bounces off the edges of the screen until we hit the *Esc* key.

```
1  /* Fig. E.5: figE_05.c
2     Creating the bouncing ball. */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
```

Fig. E.5 | Creating the bouncing ball. (Part 1 of 3.)

```
#define UP_LEFT 3

/* function prototypes */
void moveBall( void );
void reverseVerticalDirection( void );
void reverseHorizontalDirection( void );

int ball_x; /* the ball's x-coordinate */
int ball_y; /* the ball's y-coordinate */
int direction; /* the ball's direction */
BITMAP *ball; /* pointer to the ball's image bitmap */

int main( void )
{
   /* first, set up Allegro and the graphics mode */
   allegro_init(); /* initialize Allegro */
   install_keyboard(); /* install the keyboard for Allegro to use */
   set_color_depth( 16 ); /* set the color depth to 16-bit */
   set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
   ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
   ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
   ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
   srand( time( NULL ) ); /* seed the random function */
   direction = rand() % 4; /* and then make a random initial direction */

   while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
   {
      moveBall(); /* move the ball */
      clear_to_color( screen, makecol( 255, 255, 255 ) );
      /* now draw the bitmap onto the screen */
      blit( ball, screen, 0, 0, ball_x, ball_y, ball->w, ball->h );
   } /* end while */

   destroy_bitmap( ball ); /* destroy the ball bitmap */
   return 0;
} /* end function main */
END_OF_MAIN() /* don't forget this! */

void moveBall() /* moves the ball */
{
   switch ( direction ) {
      case DOWN_RIGHT:
         ++ball_x; /* move the ball to the right */
         ++ball_y; /* move the ball down */
         break;
      case UP_RIGHT:
         ++ball_x; /* move the ball to the right */
         --ball_y; /* move the ball up */
         break;
      case DOWN_LEFT:
         --ball_x; /* move the ball to the left */
         ++ball_y; /* move the ball down */
         break;
```

Fig. E.5 | Creating the bouncing ball. (Part 2 of 3.)

```
62          case UP_LEFT:
63             --ball_x; /* move the ball to the left */
64             --ball_y; /* move the ball up */
65             break;
66       } /* end switch */
67
68       /* make sure the ball doesn't go off the screen */
69
70       /* if the ball is going off the top or bottom... */
71       if ( ball_y <= 30 || ball_y >= 440 )
72          reverseVerticalDirection(); /* make it go the other way */
73
74       /* if the ball is going off the left or right... */
75       if ( ball_x <= 0 || ball_x >= 600 )
76          reverseHorizontalDirection(); /* make it go the other way */
77    } /* end function moveBall */
78
79    void reverseVerticalDirection() /* reverse the ball's up-down direction */
80    {
81       if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
82          ++direction; /* make the ball start moving up */
83       else /* "up" directions are odd numbers */
84          --direction; /* make the ball start moving down */
85    } /* end function reverseVerticalDirection */
86
87    void reverseHorizontalDirection() /* reverses the horizontal direction */
88    {
89       direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
90    } /* end function reverseHorizontalDirection */
```

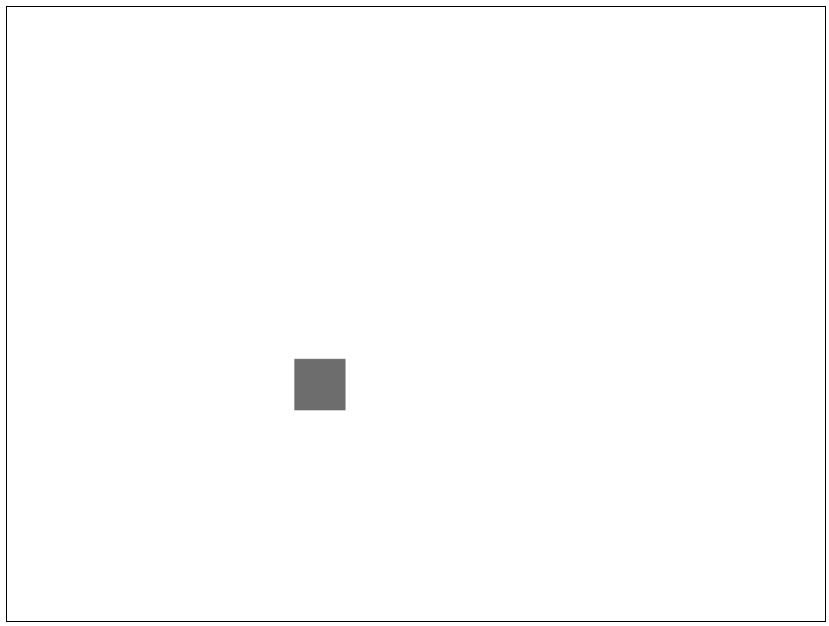

Fig. E.5 | Creating the bouncing ball. (Part 3 of 3.)

Not much is new in this program. The first lines of interest are at lines 29 and 30 where we set the initial values for `ball_x` and `ball_y`. The symbolic constants `SCREEN_W` and `SCREEN_H` are reserved by Allegro—these correspond to the width and height (in pixels), respectively, of the screen set by calling `set_gfx_mode`. These lines, therefore, place the ball in the center of the screen.

Line 34 contains a new keyboard-related item—Allegro has an array of `int`s called `key`. The array contains an index for each key on the keyboard—if a certain key is being pressed, the value at its corresponding index is set to 1; if it is not being pressed, the value is set to 0. The index that we check in the `while` loop is `KEY_ESC`, an Allegro symbolic constant that corresponds to the *Esc* key. Because the condition for our `while` loop is `!key[KEY_ESC]`, our program will continue while `key[KEY_ESC]` has a value of 0—in other words, while the *Esc* key is not being pressed. We will discuss the `key` array and Allegro's symbolic constants for the keyboard in more detail in Section E.7.

Line 37 contains the function that we use to draw the background—`clear_to_color`. Allegro does not have any explicit "background" defined in its library, but the `clear_to_color` function, as described in Fig. E.2, sets an initial color for the screen onto which we can later draw. Note, though, that in our program we had to use a function to pass the `color` parameter to `clear_to_color`. This function is the `makecol` function. Its prototype is:

```
int makecol( int red, int green, int blue );
```

This function returns an `int` that represents the color with the specified red, green, and blue intensities. The intensities allowed can range from 0 to 255, so passing the values `( 255, 0, 0 )` will create a bright red color, `( 0, 255, 0 )` will create a bright green and `( 0, 0, 255 )` will make a bright blue. In Fig. E.5, we passed `( 255, 255, 255 )`, or the maximum intensity of all three colors, which creates white. This means that our program will set the color of the screen to white when the `clear_to_color` function is called. Figure E.6 shows a table of common colors and their red, green, and blue values.

Color	Red value	Green value	Blue value
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Orange	255	200	0
Pink	255	175	175
Cyan	0	255	255
Magenta	255	0	255
Yellow	255	255	0
Black	0	0	0
White	255	255	255
Gray	128	128	128
Light gray	192	192	192
Dark gray	64	64	64

Fig. E.6 | The red, green and blue intensities of common colors in Allegro.

The rest of the program is self-explanatory—we have the ball move around the screen, and when it hits an edge, it changes its direction so that it "bounces." The reason that the lines in `moveBall` and the two `reverseDirection` functions are highlighted is because the math might appear to be a bit odd. First we check if the ball is going off the screen at lines 71 and 75. You may be wondering why the right and bottom boundaries of our screen appear to be 600 and 440 in these `if` statements, as opposed to 640 and 480 (the actual size of the screen). Recall that when we blit a bitmap onto the screen, we pass it the top-left corner of the block onto which we want to draw. This means that if `ball_y` has a value of 440, its top boundary will be at the *y*-coordinate 440. However, since the ball is 40 pixels tall, its bottom boundary will actually be at the *y*-coordinate 480, which is the bottom of the screen. The same applies to the ball's *x*-coordinate, which explains why the largest value it can be is 600. Also, if you are wondering why the lowest *y*-value allowed is 30, this is simply because we will use the top 30 pixels of the screen for the scoreboard as we add more into our Pong game.

Consider the lines that change the ball's direction. Line 82 causes the ball to start moving up if it's currently moving down, while line 84 does the opposite. Line 89 makes the ball start moving left if it's currently moving right, and right if it's currently moving left. Why does this work? Because of the specific values of the symbolic constants (lines 6–9), performing the operations in these three lines will always get you the direction you want.

Our "ball" is a square and not a circle. This does not have much impact on our program now, but once we add paddles into our game, having a square ball makes it much easier to detect if the ball and paddles are touching. We will go into more detail on this issue in Section E.9 when we add this feature to our game.

Run the program and you'll see your ball bounce around the screen. Notice, though, that the screen flickers substantially as the ball moves, to the point where it is difficult to see the ball. To fix this, we introduce a technique called double buffering.

Double Buffering for Smooth Animation

If you ran the program of the previous section, you probably noticed that the screen flickered as the ball moved. Why is this? Recall that for our animation to work, we had to clear the screen every time the ball moved. Unfortunately, this isn't the best practice. Though most computers can clear and redraw the screen quickly, there is still a small amount of time that the screen is blank between when it's cleared and when the ball is blitted onto it. Even though the screen is blank only briefly, it is still enough to cause the screen to appear to flicker as it animates the ball, since the ball keeps vanishing before it is redrawn.

We can fix this with a technique called double buffering, which uses a screen-sized, intermediary bitmap called a buffer to make the animation of moving bitmaps smoother. Instead of blitting bitmaps to the screen, we blit objects we want the user to see to the buffer. Once everything we want is there, we blit the buffer to the screen and clear the buffer.

Why does this work? You'll notice that we never clear the screen when we use this technique. Instead of deleting everything before redrawing the images on the screen, we simply draw over what's already there, which eliminates the flicker that the program in Fig. E.5 produced when it cleared the screen. In addition, since the buffer is not visible to the user (remember, the user can see only the `screen`), we can blit to and clear the buffer without worrying about it affecting anything that the user sees. Figure E.7 shows this technique in practice.

Double buffering takes only slightly more code than blitting directly to the screen—our modified program has only four more lines of code than the original! Lines 40–43 show how to code double buffering. We simply make the buffer white and blit the ball to the buffer at position `ball_x`, `ball_y`. Then, blit the whole buffer to the screen, and clear the buffer. The screen is never cleared, only drawn over, so there is no flicker.

```
1  /* Fig. E.7: figE_07.c
2     Using double buffering. */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
9  #define UP_LEFT 3
10
11 /* function prototypes */
12 void moveBall( void );
13 void reverseVerticalDirection( void );
14 void reverseHorizontalDirection( void );
15
16 int ball_x; /* the ball's x-coordinate */
17 int ball_y; /* the ball's y-coordinate */
18 int direction; /* the ball's direction */
19 BITMAP *ball; /* pointer to the ball's image bitmap */
20 BITMAP *buffer; /* pointer to the buffer */
21
22 int main( void )
23 {
24    /* first, set up Allegro and the graphics mode */
25    allegro_init(); /* initialize Allegro */
26    install_keyboard(); /* install the keyboard for Allegro to use */
27    set_color_depth( 16 ); /* set the color depth to 16-bit */
28    set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
29    ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
30    buffer = create_bitmap( SCREEN_W, SCREEN_H );/* create buffer */
31    ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
32    ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
33    srand( time( NULL ) ); /* seed the random function ... */
34    direction = rand() % 4; /* and then make a random initial direction */
35
36    while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
37    {
38       moveBall(); /* move the ball */
39       /* now, perform double buffering */
40       clear_to_color( buffer, makecol( 255, 255, 255 ) );
41       blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
42       blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
43       clear_bitmap( buffer );
44    } /* end while */
```

Fig. E.7 | Using double buffering. (Part 1 of 3.)

```
45
46     destroy_bitmap( ball ); /* destroy the ball bitmap */
47     destroy_bitmap( buffer ); /* destroy the buffer bitmap */
48     return 0;
49  } /* end function main */
50  END_OF_MAIN() /* don't forget this! */
51
52  void moveBall() /* moves the ball */
53  {
54     switch ( direction ) {
55        case DOWN_RIGHT:
56           ++ball_x; /* move the ball to the right */
57           ++ball_y; /* move the ball down */
58           break;
59        case UP_RIGHT:
60           ++ball_x; /* move the ball to the right */
61           --ball_y; /* move the ball up */
62           break;
63        case DOWN_LEFT:
64           --ball_x; /* move the ball to the left */
65           ++ball_y; /* move the ball down */
66           break;
67        case UP_LEFT:
68           --ball_x; /* move the ball to the left */
69           --ball_y; /* move the ball up */
70           break;
71     } /* end switch */
72
73     /* make sure the ball doesn't go off the screen */
74
75     /* if the ball is going off the top or bottom ... */
76     if ( ball_y <= 30 || ball_y >= 440 )
77        reverseVerticalDirection();
78
79     /* if the ball is going off the left or right ... */
80     if ( ball_x <= 0 || ball_x >= 600 )
81        reverseHorizontalDirection();
82  } /* end function moveBall */
83
84  void reverseVerticalDirection() /* reverse the ball's up-down direction */
85  {
86     if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
87        ++direction; /* make the ball start moving up */
88     else /* "up" directions are odd numbers */
89        --direction; /* make the ball start moving down */
90  } /* end function reverseVerticalDirection */
91
92  void reverseHorizontalDirection() /* reverses the horizontal direction */
93  {
94     direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
95  } /* end function reverseHorizontalDirection */
```

Fig. E.7 | Using double buffering. (Part 2 of 3.)

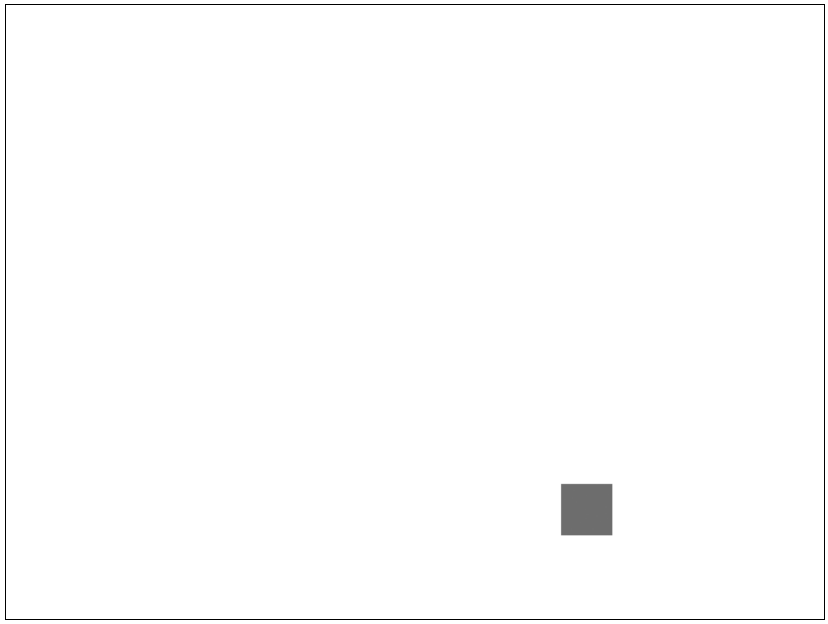

Fig. E.7 | Using double buffering. (Part 3 of 3.)

E.6 Importing and Playing Sounds

While a game with only graphics is fun to play, a game that uses sounds to enhance the player's experience is much more interesting. We now discuss importing sounds and playing sound files in Allegro programs, which we will use to "juice up" our game by having a "boing" sound play whenever the ball hits a wall.

Sound files are handled similarly to bitmaps—just as the `allegro.h` header defines several types that are used to hold image data, it defines several types that are used to hold sound data, as well. With images, the most basic of these types is `BITMAP*`. With sounds, the most basic of these types is the type `SAMPLE*`—Allegro refers to sound files as "digital samples." Like `BITMAP*`, the `SAMPLE*` type is a pointer.

The Allegro functions used to import and play sounds are analogous to those used for importing and displaying bitmaps. Figure E.8 shows the most important Allegro functions for manipulating sound files.

The `install_sound` function must be called before Allegro can play any sound files. Its prototype is:

```
int install_sound(int digi, int midi, const char *cfg_path);
```

The function returns an `int` for error-checking purposes—0 if the function is successful, and a non-zero value if it is not. The `digi` and `midi` parameters specify the sound card drivers used for playing digital samples and MIDI files, respectively. As with the graphic drivers, the newer versions of Allegro provide so-called "magic drivers" that automatically specify the audio drivers to use—`DIGI_AUTODETECT` and `MIDI_AUTODETECT`. These are the only values that you should pass to the first two parameters.

Function prototype	Description
`SAMPLE *load_sample(const char *filename)`	Loads and returns a pointer to a sound file with the specified filename. The file must be in `.wav` format. Returns `NULL` (with no error) if the specified file cannot be loaded.
`int play_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop)`	Plays the specified sample at the specified volume, pan position, and frequency. The sample will loop continuously if `loop` is non-zero.
`void adjust_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop)`	Adjusts a currently playing sample's parameters to the ones specified. Can be called on any sample without causing errors, but will affect only ones that are currently playing.
`void stop_sample(const SAMPLE *spl)`	Stops a sample that is currently playing.
`void destroy_sample(SAMPLE *spl)`	Destroys a sample and frees the memory allocated to it. If the sample is currently playing or looping, it will stop immediately.

Fig. E.8 | Important SAMPLE functions.

The `cfg_path` parameter has no effect on the program. Older versions of Allegro required that you specify a `.cfg` file that told the program how to play sound files, but this is no longer necessary in the current version.

Now let's add sounds to the bouncing ball program. As with bitmaps, we must provide the program with an external sound file to load, so copy the `boing.wav` sound file from the appendix's examples folder and save it in the same folder as your `pong` project. Then we can use the code in Fig. E.9 to make our ball emit a "boing" sound whenever it bounces off a side of the screen. The highlighted lines mark the changes from the previous section.

```
1  /* Fig. E.9: figE_09.c
2     Utilizing sound files */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
9  #define UP_LEFT 3
10
11 /* function prototypes */
12 void moveBall( void );
13 void reverseVerticalDirection( void );
14 void reverseHorizontalDirection( void );
15
16 int ball_x; /* the ball's x-coordinate */
17 int ball_y; /* the ball's y-coordinate */
18 int direction; /* the ball's direction */
```

Fig. E.9 | Utilizing sound files. (Part 1 of 3.)

```
19  BITMAP *ball; /* pointer to ball's image bitmap */
20  BITMAP *buffer; /* pointer to the buffer */
21  SAMPLE *boing; /* pointer to sound file */
22
23  int main( void )
24  {
25     /* first, set up Allegro and the graphics mode */
26     allegro_init(); /* initialize Allegro */
27     install_keyboard(); /* install the keyboard for Allegro to use */
28     install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
29     set_color_depth( 16 ); /* set the color depth to 16-bit */
30     set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
31     ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
32     buffer = create_bitmap(SCREEN_W, SCREEN_H);/* create buffer */
33     boing = load_sample( "boing.wav" ); /* load the sound file */
34     ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
35     ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
36     srand( time( NULL ) ); /* seed the random function ... */
37     direction = rand() % 4; /* and then make a random initial direction */
38     while ( !key[KEY_ESC] )/* until the escape key is pressed ... */
39     {
40        moveBall(); /* move the ball */
41        /* now, perform double buffering */
42        clear_to_color( buffer, makecol( 255, 255, 255 ) );
43        blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
44        blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
45        clear_bitmap( buffer );
46     } /* end while loop */
47     destroy_bitmap( ball ); /* destroy the ball bitmap */
48     destroy_bitmap( buffer ); /* destroy the buffer bitmap */
49     destroy_sample( boing ); /* destroy the boing sound file */
50     return 0;
51  } /* end function main */
52  END_OF_MAIN() /* don't forget this! */
53
54  void moveBall() /* moves the ball */
55  {
56     switch ( direction ) {
57        case DOWN_RIGHT:
58           ++ball_x; /* move the ball to the right */
59           ++ball_y; /* move the ball down */
60           break;
61        case UP_RIGHT:
62           ++ball_x; /* move the ball to the right */
63           --ball_y; /* move the ball up */
64           break;
65        case DOWN_LEFT:
66           --ball_x; /* move the ball to the left */
67           ++ball_y; /* move the ball down */
68           break;
69        case UP_LEFT:
70           --ball_x; /* move the ball to the left */
71           --ball_y; /* move the ball up */
```

Fig. E.9 | Utilizing sound files. (Part 2 of 3.)

```
            break;
      } /* end switch */

      /* make sure the ball doesn't go off screen */

      /* if the ball is going off the top or bottom ... */
      if ( ball_y <= 30 || ball_y >= 440 )
         reverseVerticalDirection();

      /* if the ball is going off the left or right ... */
      if ( ball_x <= 0 || ball_x >= 600 )
         reverseHorizontalDirection();
   } /* end function moveBall */

   void reverseVerticalDirection() /* reverse the ball's up-down direction */
   {
      if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
         ++direction; /* make the ball start moving up */
      else /* "up" directions are odd numbers */
         --direction; /* make the ball start moving down */
      play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
   } /* end function reverseVerticalDirection */

   void reverseHorizontalDirection() /* reverses the horizontal direction */
   {
      direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
      play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
   } /* end function reverseHorizontalDirection */
```

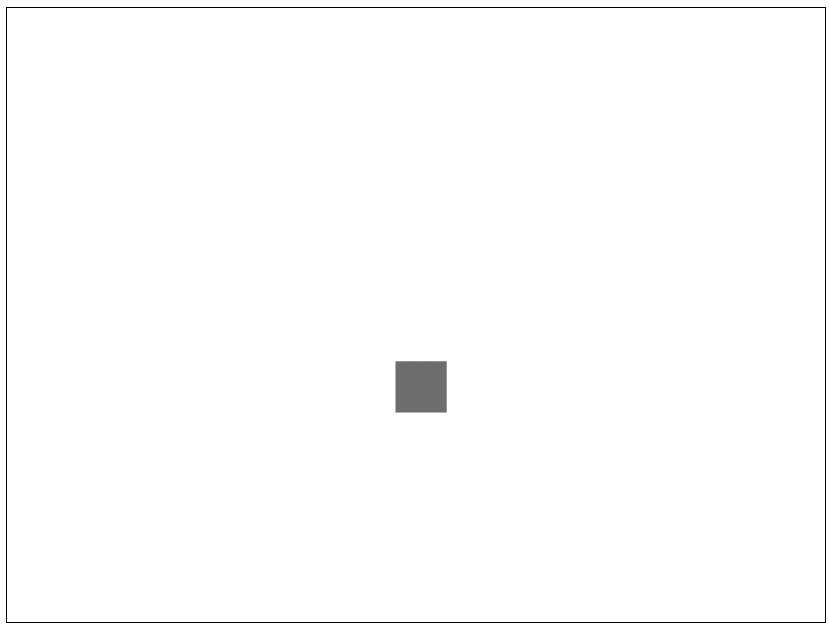

Fig. E.9 | Utilizing sound files. (Part 3 of 3.)

Because we placed the call to the `play_sample` function inside our functions that reverse the ball's direction, the "boing" sound will play whenever the ball's direction is reversed—in other words, whenever the ball hits the boundary of the screen. Consider the prototype of function `play_sample`:

```
int play_sample(const SAMPLE *sample, int volume, int pan,
   int frequency, int loop)
```

The `volume` and `pan` parameters determine the volume and pan position of the sample being played, and the values passed to them can range from 0 to 255. A volume of 0 means the sound will be muted, while a volume of 255 plays the sound at full volume. A pan position of 128 will play the sound out of both speakers equally, while a lower or higher value will play the sound more towards the left or right, respectively. The `frequency` parameter, which specifies the frequency at which the sample will be played, is a parameter whose value is relative rather than absolute. A `frequency` value of 1000 will play the sound at the frequency at which it was recorded, while a value of 2000 will play a sample at twice its normal frequency, which produces a much higher pitch, a value of 500 will play the sample at half its normal frequency, which produces a lower pitch and so on. Finally, as explained in Fig. E.8, the `loop` parameter will cause the sample to loop continuously if its value is not 0. Otherwise, the sample will play only once.

E.7 Keyboard Input

A game cannot be called a game unless the user can interact with it in some way. We have already used some keyboard input methods in this appendix; now we discuss keyboard input in Allegro in more detail.

The first thing we must do to allow Allegro to recognize and use the keyboard is call the `install_keyboard` function, which takes no parameters. Allegro does not need any driver information to install the keyboard.

Recall that Allegro defines an array of `int`s called `key`. This array enables us to determine when keys are pressed. In addition to the array, the `allegro.h` header defines symbolic constants that correspond to the keys on a standard keyboard. For example, the constant for the *A* key is `KEY_A`, and the constant for the spacebar is `KEY_SPACE`. The full list of these constants is available at `www.allegro.cc/manual/key`. These constants are used in tandem with the `key` array to determine if the key to which a given constant corresponds is being pressed at any given time.

Each symbolic constant corresponds to an index in the `key` array that keeps track of whether that key is being pressed or not. If the key is not being pressed, the array will hold 0 at that index, while if it is, the value at that index will be non-zero. Thus, if we want to see if the *A* key is being pressed, we look at the value returned by `key[KEY_A]`. If it is not zero, then we know the user is pressing the *A* key.

In our Pong game, we use this array to control the paddles on the sides of the screen. If the *A* or *Z* keys are being pressed, the paddle on the left side should move up and down, respectively. Likewise, if the user presses the up or down arrow keys, the paddle on the right side should move in the corresponding direction. For this purpose, we add a new function, `respondToKeyboard`, to our program that checks if any of these four keys are being pressed and moves the paddles accordingly.

Of course, we have not yet drawn the paddles in our program, so the first thing we need is a bitmap file that contains the image data for them. As with the ball bitmap, you can find the bitmap file for a paddle in this appendix's examples folder. Save the file as `bar.bmp` in the same folder as your Pong project (the bitmap is 20 by 100 pixels—we will use this information in the program). Once you have done that, you can use the code in Fig. E.10 to allow the user to move the paddles with the keyboard. As usual, new lines in the program are highlighted.

Lines 101–109 in Fig. E.10 show how we use the `key` array to determine whether certain keys are being pressed. In C, any statement that returns a non-zero value is considered "true" if used in the condition of an `if` statement, so Allegro's `key` array makes it easy to check for keypresses. Note, however, that the call to the `respondToKeyboard` function is inside the `while` loop in our `main` (line 49)—this is needed for the keyboard input to work correctly. Though the `if` statements in lines 101–109 are all that is necessary to check if certain keys have been pressed, each statement checks only once per call. Since we want to have our program check *constantly* for keyboard input, we must place the `respondToKeyboard` function inside some sort of loop that ensures the program will call it repeatedly. This holds true for most games besides Pong, as well.

```
1  /* Fig. E.10: figE_10.c
2     Adding paddles and keyboard input. */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
9  #define UP_LEFT 3
10
11 /* function prototypes */
12 void moveBall( void );
13 void respondToKeyboard( void );
14 void reverseVerticalDirection( void );
15 void reverseHorizontalDirection( void );
16
17 int ball_x; /* the ball's x-coordinate */
18 int ball_y; /* the ball's y-coordinate */
19 int barL_y; /* y-coordinate of the left paddle */
20 int barR_y; /* y-coordinate of the right paddle */
21 int direction; /* the ball's direction */
22 BITMAP *ball; /* pointer to ball's image bitmap */
23 BITMAP *bar; /* pointer to paddle's image bitmap */
24 BITMAP *buffer; /* pointer to the buffer */
25 SAMPLE *boing; /* pointer to sound file */
26
27 int main( void )
28 {
29    /* first, set up Allegro and the graphics mode */
30    allegro_init(); /* initialize Allegro */
31    install_keyboard(); /* install the keyboard for Allegro to use */
```

Fig. E.10 | Adding paddles and keyboard input. (Part 1 of 5.)

```
32      install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
33      set_color_depth( 16 ); /* set the color depth to 16-bit */
34      set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
35      ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
36      bar = load_bitmap( "bar.bmp", NULL); /* load the bar bitmap */
37      buffer = create_bitmap(SCREEN_W, SCREEN_H);/* create buffer */
38      boing = load_sample( "boing.wav" ); /* load the sound file */
39      ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
40      ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
41      barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
42      barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
43      srand( time( NULL ) ); /* seed the random function ... */
44      direction = rand() % 4; /* and then make a random initial direction */
45
46      while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
47      {
48         moveBall(); /* move the ball */
49         respondToKeyboard(); /* respond to keyboard input */
50         /* now, perform double buffering */
51         clear_to_color( buffer, makecol( 255, 255, 255 ) );
52         blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
53         blit( bar, buffer, 0, 0, 0, barL_y, bar->w, bar->h );
54         blit( bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h );
55         blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
56         clear_bitmap( buffer );
57      } /* end while */
58
59      destroy_bitmap( ball ); /* destroy the ball bitmap */
60      destroy_bitmap( bar ); /* destroy the bar bitmap */
61      destroy_bitmap( buffer ); /* destroy the buffer bitmap */
62      destroy_sample( boing ); /* destroy the boing sound file */
63      return 0;
64   } /* end function main */
65   END_OF_MAIN() /* don't forget this! */
66
67   void moveBall() /* moves the ball */
68   {
69      switch ( direction ) {
70         case DOWN_RIGHT:
71            ++ball_x; /* move the ball to the right */
72            ++ball_y; /* move the ball down */
73            break;
74         case UP_RIGHT:
75            ++ball_x; /* move the ball to the right */
76            --ball_y; /* move the ball up */
77            break;
78         case DOWN_LEFT:
79            --ball_x; /* move the ball to the left */
80            ++ball_y; /* move the ball down */
81            break;
82         case UP_LEFT:
83            --ball_x; /* move the ball to the left */
```

Fig. E.10 | Adding paddles and keyboard input. (Part 2 of 5.)

```
84             --ball_y; /* move the ball up */
85             break;
86       } /* end switch */
87
88       /* make sure the ball doesn't go off screen */
89
90       /* if the ball is going off the top or bottom ... */
91       if ( ball_y <= 30 || ball_y >= 440 )
92          reverseVerticalDirection();
93
94       /* if the ball is going off the left or right ... */
95       if ( ball_x <= 0 || ball_x >= 600 )
96          reverseHorizontalDirection();
97    } /* end function moveBall */
98
99    void respondToKeyboard() /* responds to keyboard input */
100   {
101      if ( key[KEY_A] ) /* if A is being pressed... */
102         barL_y -= 3; /* ... move the left paddle up */
103      if ( key[KEY_Z] ) /* if Z is being pressed... */
104         barL_y += 3; /* ... move the left paddle down */
105
106      if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
107         barR_y -= 3; /* ... move the right paddle up */
108      if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
109         barR_y += 3; /* ... move the right paddle down */
110
111      /* make sure the paddles don't go offscreen */
112      if ( barL_y < 30 ) /* if left paddle is going off the top */
113         barL_y = 30;
114      else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
115         barL_y = 380;
116      if ( barR_y < 30 ) /* if right paddle is going off the top */
117         barR_y = 30;
118      else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
119         barR_y = 380;
120   } /* end function respondToKeyboard */
121
122   void reverseVerticalDirection() /* reverse the ball's up-down direction */
123   {
124      if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
125         ++direction; /* make the ball start moving up */
126      else /* "up" directions are odd numbers */
127         --direction; /* make the ball start moving down */
128      play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
129   } /* end function reverseVerticalDirection */
130
131   void reverseHorizontalDirection() /* reverses the horizontal direction */
132   {
133      direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
134      play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
135   } /* end function reverseHorizontalDirection */
```

Fig. E.10 | Adding paddles and keyboard input. (Part 3 of 5.)

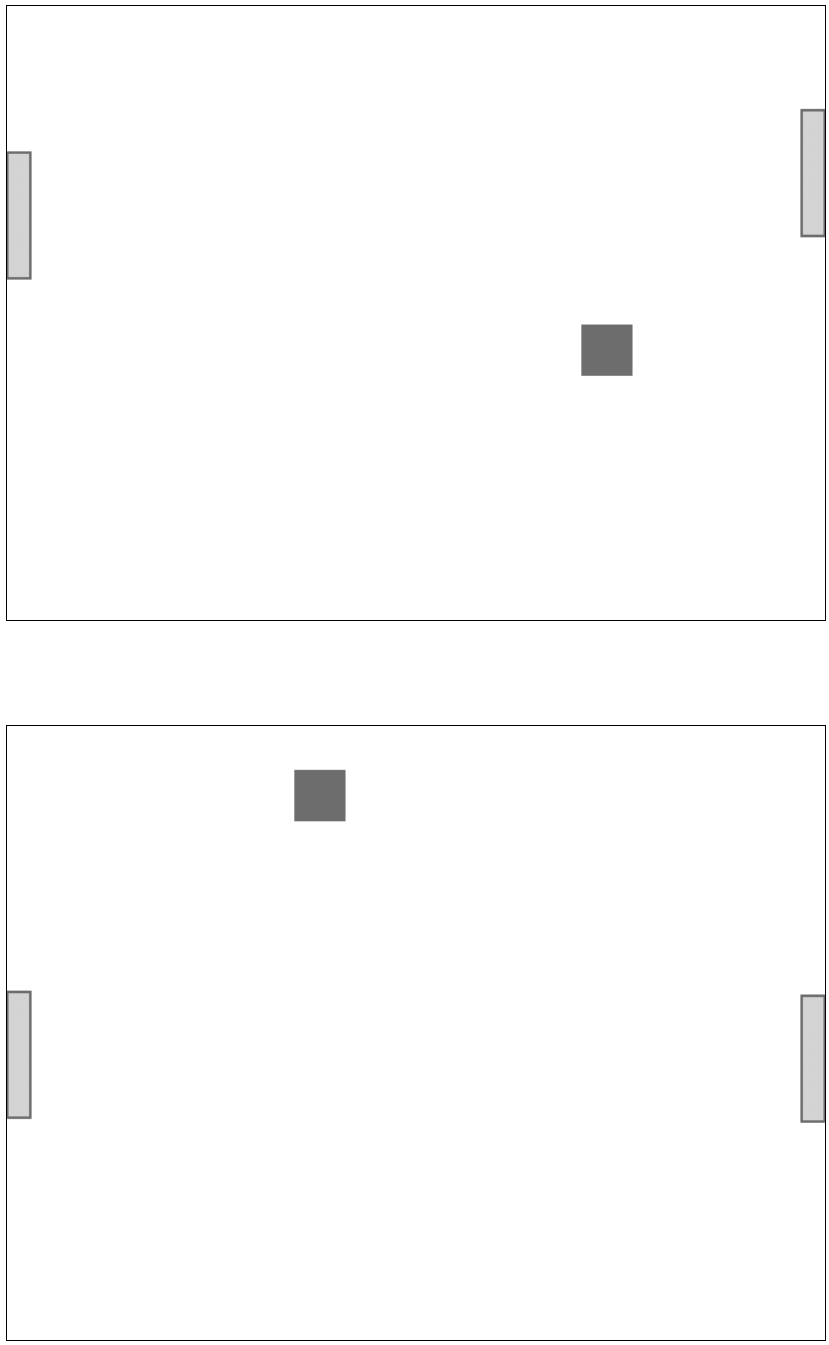

Fig. E.10 | Adding paddles and keyboard input. (Part 4 of 5.)

Fig. E.10 | Adding paddles and keyboard input. (Part 5 of 5.)

You may notice that we did not keep track of the paddles' *x*-coordinates in this program. Since the paddles cannot move horizontally, this is not necessary. Each paddle only has one *x*-coordinate for the duration of the game, so we do not need a variable that records those values. Also, as with the ball, the maximum *y*-coordinate allowed for the paddles is not 480 (the height of the screen); rather, it is 380. This is because the paddles are 100 pixels high, and a *y*-coordinate of 380 means that the bottom of the paddle is at the *y*-coordinate 480—the bottom of the screen.

One other thing to notice is that though we loaded the `bar` bitmap only once, we were able to draw it to the screen in two different places simultaneously. Allegro allows the same bitmap to be drawn in many different places, a feature that is useful if you need two identical bitmaps on the screen at once.

However, at this point, while the paddles can be moved with the keyboard, they do not have any effect on the ball—it will keep bouncing around the screen regardless of whether or not the paddles are in its way. We will deal with this problem in Section E.9, when we code this feature into our game. For now, we take a look at another Allegro capability—displaying text on the screen.

E.8 Fonts and Displaying Text

In almost all games, even the simplest ones, it is necessary for the game to communicate with the user in some way. This can range from giving the user instructions while the game is running, to simply telling the user how many points he or she has scored so far. To do this, the game developer needs some way of displaying text on the screen so that the player

can read it. In Allegro, displaying text is handled in a similar way to displaying bitmaps and playing sounds.

To display text in Allegro, the most important thing we must specify—aside from the actual text to be displayed, of course—is the font in which it should be displayed. As with bitmaps and sounds, Allegro can load fonts from external files, and defines a type which points to the place in memory where these files are stored—`FONT*`. Loading fonts is done with the `load_font` function, which works the same as `load_bitmap` and `load_sample`. Likewise, once you are done with a font, it must be destroyed with the `destroy_font` function to prevent memory leaks. If you wish to load a font from a file, it must be in `.fnt`, `.bmp`, or `.pcx` format. The parameters of `load_font` are slightly different from those of `load_bitmap` and `load_sample`, so we consider its prototype:

```
FONT *load_font( const char *filename, RGB *palette,
   void *parameter );
```

The first parameter is, obviously, the filename of the font file that is being loaded. The second parameter is something we have seen before—a palette. However, as with bitmaps, if the color depth is not 8-bit, we do not actually have to pass a palette to the function. It can safely be `NULL` without consequence. We will not use the third parameter—it is used to tell Allegro to load fonts in different ways, which we do not need to do. Like the second parameter, it can be set to `NULL` without causing any problems. Once a font has been loaded, you can use the functions in Fig. E.11 to draw text onto a bitmap or the screen.

Function prototype	Description
`void textprintf_ex(BITMAP *bmp, const FONT *f, int x, int y, int color, int bgColor, const char *fmt, ...)`	Draws the format control string specified by `fmt` and the parameters following it onto `bmp` at the specified coordinates. The text is drawn in the specified font and colors, and is left justified.
`void textprintf_centre_ex( BITMAP *bmp, const FONT *f, int x, int y, int color, int bgColor, const char *fmt, ...)`	Works the same way as `textprintf_ex`, but the text drawn is center justified at the specified coordinates.
`void textprintf_right_ex( BITMAP *bmp, const FONT *f, int x, int y, int color, int bgColor, const char *fmt, ...)`	Works the same way as `textprintf_ex`, but the text drawn is right justified at the specified coordinates.
`int text_length(const FONT *f, const char *string)`	Returns the width (in pixels) of the specified string when drawn in the specified font. Useful when aligning multiple text outputs.
`int text_height(const FONT *f, const char *string)`	Returns the height (in pixels) of the specified string when drawn in the specified font. Useful when aligning multiple text outputs.

Fig. E.11 | Functions that are useful for drawing text onto a bitmap.

(The `allegro.h` header defines a global variable, `font`, that contains the data for Allegro's "default" font. This font can be used if you do not have a font file from which to load. We provide a font file for our Pong game, but this is still a useful feature to know.)

As you can see, the text output functions have quite a number of parameters. It may be useful to take a look at an example of a function call to understand how the function works.

```
textprintf_ex( buffer, font, 0, 0, makecol( 0, 0, 0 ), -1,
   "Hello!" );
```

This function call displays the string `"Hello!"` at the top-left corner of a buffer (which can later be drawn onto the screen), using the default font. The text is displayed in black, with a transparent background color.

The string we pass to the `textprintf_ex` function is a format control string. This means that we can use any of the conversion specifiers discussed in Chapter 9 to print `int`s, `double`s and other variables. This is very useful if, for example, we want to print a player's score, as we will need to do in our Pong game.

We passed the `bgColor` parameter a value of `-1` in the example call. Allegro cannot create this color with a call to `makecol`, as it interprets the value of `-1` as "no color" or "transparent." This means that the text displayed will have no background color and that anything "behind" the text will be visible.

Normally the default Allegro font is fine when displaying text, but we have provided the `pongfont.pcx` font file with the appendix examples for use with our program. In our Pong game, we use the functions described above to display the scores of each of the players. For this reason, we add the `int`s, `scoreL` and `scoreR`, to this iteration of the program. Note, however, that since the paddles do not yet do anything, we cannot yet keep score in the game, and so the values of `scoreL` and `scoreR` will remain at 0 for the duration of the program. Nevertheless, Fig. E.12 shows how to use the functions explained previously to display text on the screen using the font provided on our website.

```
1  /* Fig. E.12: figE_12.c
2     Displaying text on the screen. */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
9  #define UP_LEFT 3
10
11 /* function prototypes */
12 void moveBall( void );
13 void respondToKeyboard( void );
14 void reverseVerticalDirection( void );
15 void reverseHorizontalDirection( void );
16
17 int ball_x; /* the ball's x-coordinate */
18 int ball_y; /* the ball's y-coordinate */
```

Fig. E.12 | Displaying text on the screen. (Part 1 of 4.)

```
19  int barL_y; /* y-coordinate of the left paddle */
20  int barR_y; /* y-coordinate of the right paddle */
21  int scoreL; /* score of the left player */
22  int scoreR; /* score of the right player */
23  int direction; /* the ball's direction */
24  BITMAP *ball; /* pointer to ball's image bitmap */
25  BITMAP *bar; /* pointer to paddle's image bitmap */
26  BITMAP *buffer; /* pointer to the buffer */
27  SAMPLE *boing; /* pointer to sound file */
28  FONT *pongFont; /* pointer to font file */
29
30  int main( void )
31  {
32     /* first, set up Allegro and the graphics mode */
33     allegro_init(); /* initialize Allegro */
34     install_keyboard(); /* install the keyboard for Allegro to use */
35     install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
36     set_color_depth( 16 ); /* set the color depth to 16-bit */
37     set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
38     ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
39     bar = load_bitmap( "bar.bmp", NULL); /* load the bar bitmap */
40     buffer = create_bitmap(SCREEN_W, SCREEN_H);/* create buffer */
41     boing = load_sample( "boing.wav" ); /* load the sound file */
42     pongFont = load_font( "pongfont.pcx", NULL, NULL ); /* load the font */
43     ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
44     ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
45     barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
46     barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
47     scoreL = 0; /* set left player's score to 0 */
48     scoreR = 0; /* set right player's score to 0 */
49     srand( time( NULL ) ); /* seed the random function ... */
50     direction = rand() % 4; /* and then make a random initial direction */
51
52     while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
53     {
54        moveBall(); /* move the ball */
55        respondToKeyboard(); /* respond to keyboard input */
56        /* now, perform double buffering */
57        clear_to_color( buffer, makecol( 255, 255, 255 ) );
58        blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
59        blit( bar, buffer, 0, 0, 0, barL_y, bar->w, bar->h );
60        blit( bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h );
61        /* draw text onto the buffer */
62        textprintf_ex( buffer, pongFont, 75, 0, makecol( 0, 0, 0 ),
63           -1, "Left Player Score: %d", scoreL );
64        textprintf_ex( buffer, pongFont, 400, 0, makecol( 0, 0, 0 ),
65           -1, "Right Player Score: %d", scoreR );
66        blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
67        clear_bitmap( buffer );
68     } /* end while */
69
70     destroy_bitmap( ball ); /* destroy the ball bitmap */
```

Fig. E.12 | Displaying text on the screen. (Part 2 of 4.)

```
71     destroy_bitmap( bar ); /* destroy the bar bitmap */
72     destroy_bitmap( buffer ); /* destroy the buffer bitmap */
73     destroy_sample( boing ); /* destroy the boing sound file */
74     destroy_font( pongFont ); /* destroy the font */
75     return 0;
76  } /* end function main */
77  END_OF_MAIN() /* don't forget this! */
78
79  void moveBall() /* moves the ball */
80  {
81     switch ( direction ) {
82        case DOWN_RIGHT:
83           ++ball_x; /* move the ball to the right */
84           ++ball_y; /* move the ball down */
85           break;
86        case UP_RIGHT:
87           ++ball_x; /* move the ball to the right */
88           --ball_y; /* move the ball up */
89           break;
90        case DOWN_LEFT:
91           --ball_x; /* move the ball to the left */
92           ++ball_y; /* move the ball down */
93           break;
94        case UP_LEFT:
95           --ball_x; /* move the ball to the left */
96           --ball_y; /* move the ball up */
97           break;
98     } /* end switch */
99
100    /* make sure the ball doesn't go off the screen */
101
102    /* if the ball is going off the top or bottom ... */
103    if ( ball_y <= 30 || ball_y >= 440 )
104       reverseVerticalDirection();
105
106    /* if the ball is going off the left or right ... */
107    if ( ball_x <= 0 || ball_x >= 600 )
108       reverseHorizontalDirection();
109 } /* end function moveBall */
110
111 void respondToKeyboard() /* responds to keyboard input */
112 {
113    if ( key[KEY_A] ) /* if A is being pressed... */
114       barL_y -= 3; /* ... move the left paddle up */
115    if ( key[KEY_Z] ) /* if Z is being pressed... */
116       barL_y += 3; /* ... move the left paddle down */
117
118    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
119       barR_y -= 3; /* ... move the right paddle up */
120    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
121       barR_y += 3; /* ... move the right paddle down */
122
```

Fig. E.12 | Displaying text on the screen. (Part 3 of 4.)

```
123    /* make sure the paddles don't go offscreen */
124    if ( barL_y < 30 ) /* if left paddle is going off the top */
125       barL_y = 30;
126    else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
127       barL_y = 380;
128    if ( barR_y < 30 ) /* if right paddle is going off the top */
129       barR_y = 30;
130    else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
131       barR_y = 380;
132 } /* end function respondToKeyboard */
133
134 void reverseVerticalDirection() /* reverse the ball's up-down direction */
135 {
136    if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
137       ++direction; /* make the ball start moving up */
138    else /* "up" directions are odd numbers */
139       --direction; /* make the ball start moving down */
140    play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
141 } /* end function reverseVerticalDirection */
142
143 void reverseHorizontalDirection() /* reverses the horizontal direction */
144 {
145    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
146    play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
147 } /* end function reverseHorizontalDirection */
```

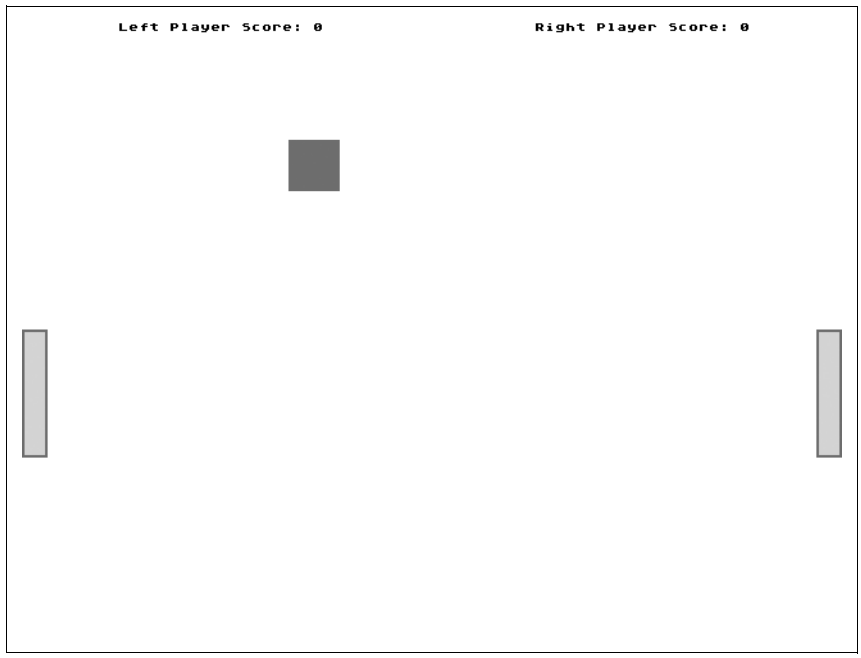

Fig. E.12 | Displaying text on the screen. (Part 4 of 4.)

E.9 Implementing the Game of Pong

We now have all of the elements of our Pong game in our program—a ball, moving paddles, sounds and a scoreboard. However, these elements are not yet capable of interacting with each other. In this section we tie up the loose ends in our program to make it run like an actual Pong game.

The weakness in the current version of our program is that the paddles don't yet stop the ball—it keeps moving regardless of whether or not the paddles are in its way. In addition, when the ball hits the left or right edge of the screen, it simply bounces instead of going off the screen, and the player is not awarded a point.

The method that we use to fix these problems is surprisingly simple. Allegro does not have any functions that determine whether or not two bitmaps are touching, but since we know the dimensions of the `ball` and `bar` bitmaps, we can easily test if the ball has hit a paddle. Since we need to check only whether the paddle is in the way of the ball if the ball is moving off the left or right of the screen, we can make this check inside the `if` statement that checks the ball's *x*-coordinate.

The only other issue we face is the fact that while we have created a boundary near the top of the screen that ensures the ball doesn't move into the scoreboard, there is no visual indication that the boundary is there—the ball just appears to bounce for no reason. While we know why this is happening, it may confuse the players.

We mentioned earlier in the appendix that Allegro can draw simple graphics. In addition to being able to draw rectangles, circles, and polygons, Allegro has a `line` function to draw a line from one point to another. We can use this function to draw a line where our boundary is. The prototype for this function follows:

```
void line(BITMAP *bitmap, int x1, int y1, int x2, int y2, int color)
```

This function draws a straight line onto the specified bitmap from the coordinates (`x1`, `y1`) to the coordinates (`x2`, `y2`). The line will be drawn in the given color, which can be specified by using the `makecol` function. We can now put the finishing touches on our Pong game with the code in Fig. E.13.

```
1  /* Fig. E.13: figE_13.c
2     Finishing up the Pong game. */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
9  #define UP_LEFT 3
10
11 /* function prototypes */
12 void moveBall( void );
13 void respondToKeyboard( void );
14 void reverseVerticalDirection( void );
15 void reverseHorizontalDirection( void );
16
```

Fig. E.13 | Finishing up the Pong game. (Part 1 of 7.)

```
17  int ball_x; /* the ball's x-coordinate */
18  int ball_y; /* the ball's y-coordinate */
19  int barL_y; /* y-coordinate of the left paddle */
20  int barR_y; /* y-coordinate of the right paddle */
21  int scoreL; /* score of the left player */
22  int scoreR; /* score of the right player */
23  int direction; /* the ball's direction */
24  BITMAP *ball; /* pointer to ball's image bitmap */
25  BITMAP *bar; /* pointer to paddle's image bitmap */
26  BITMAP *buffer; /* pointer to the buffer */
27  SAMPLE *boing; /* pointer to sound file */
28  FONT *pongFont; /* pointer to font file */
29
30  int main( void )
31  {
32     /* first, set up Allegro and the graphics mode */
33     allegro_init(); /* initialize Allegro */
34     install_keyboard(); /* install the keyboard for Allegro to use */
35     install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
36     set_color_depth( 16 ); /* set the color depth to 16-bit */
37     set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
38     ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
39     bar = load_bitmap( "bar.bmp", NULL); /* load the bar bitmap */
40     buffer = create_bitmap(SCREEN_W, SCREEN_H);/* create buffer */
41     boing = load_sample( "boing.wav" ); /* load the sound file */
42     pongFont = load_font( "pongfont.pcx", NULL, NULL ); /* load the font */
43     ball_x = SCREEN_W / 2; /* give ball its initial x-coordinate */
44     ball_y = SCREEN_H / 2; /* give ball its initial y-coordinate */
45     barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
46     barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
47     scoreL = 0; /* set left player's score to 0 */
48     scoreR = 0; /* set right player's score to 0 */
49     srand( time( NULL ) ); /* seed the random function ... */
50     direction = rand() % 4; /* and then make a random initial direction */
51
52     while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
53     {
54        moveBall(); /* move the ball */
55        respondToKeyboard(); /* respond to keyboard input */
56        /* now, perform double buffering */
57        clear_to_color( buffer, makecol( 255, 255, 255 ) );
58        blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
59        blit( bar, buffer, 0, 0, 0, barL_y, bar->w, bar->h );
60        blit( bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h );
61        line( buffer, 0, 30, 640, 30, makecol( 0, 0, 0 ) );
62        /* draw text onto the buffer */
63        textprintf_ex( buffer, pongFont, 75, 0, makecol( 0, 0, 0 ),
64           -1, "Left Player Score: %d", scoreL );
65        textprintf_ex( buffer, pongFont, 400, 0, makecol( 0, 0, 0 ),
66           -1, "Right Player Score: %d", scoreR );
67        blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
68        clear_bitmap( buffer );
69     } /* end while */
```

Fig. E.13 | Finishing up the Pong game. (Part 2 of 7.)

```
70
71     destroy_bitmap( ball ); /* destroy the ball bitmap */
72     destroy_bitmap( bar ); /* destroy the bar bitmap */
73     destroy_bitmap( buffer ); /* destroy the buffer bitmap */
74     destroy_sample( boing ); /* destroy the boing sound file */
75     destroy_font( pongFont ); /* destroy the font */
76     return 0;
77  } /* end function main */
78  END_OF_MAIN() /* don't forget this! */
79
80  void moveBall() /* moves the ball */
81  {
82     switch ( direction ) {
83        case DOWN_RIGHT:
84           ++ball_x; /* move the ball to the right */
85           ++ball_y; /* move the ball down */
86           break;
87        case UP_RIGHT:
88           ++ball_x; /* move the ball to the right */
89           --ball_y; /* move the ball up */
90           break;
91        case DOWN_LEFT:
92           --ball_x; /* move the ball to the left */
93           ++ball_y; /* move the ball down */
94           break;
95        case UP_LEFT:
96           --ball_x; /* move the ball to the left */
97           --ball_y; /* move the ball up */
98           break;
99     } /* end switch */
100
101    /* if the ball is going off the top or bottom ... */
102    if ( ball_y <= 30 || ball_y >= 440 )
103       reverseVerticalDirection(); /* make it go the other way */
104
105    /* if the ball is in range of the left paddle ... */
106    if (ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT))
107    {
108       /* is the left paddle in the way? */
109       if ( ball_y > ( barL_y - 39 ) && ball_y < ( barL_y + 99 ) )
110          reverseHorizontalDirection();
111       else if ( ball_x <= -20 ) { /* if the ball goes off the screen */
112          ++scoreR; /* give right player a point */
113          ball_x = SCREEN_W / 2; /* place the ball in the ... */
114          ball_y = SCREEN_H / 2; /* ... center of the screen  */
115          direction = rand() % 4; /* give the ball a random direction */
116       } /* end else */
117    } /* end if */
118
119    /* if the ball is in range of the right paddle ... */
120    if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
121    {
```

Fig. E.13 | Finishing up the Pong game. (Part 3 of 7.)

```
122         /* is the right paddle in the way? */
123         if ( ball_y > ( barR_y - 39 ) && ball_y < ( barR_y + 99 ) )
124            reverseHorizontalDirection();
125         else if ( ball_x >= 620 ) { /* if the ball goes off the screen */
126            ++scoreL; /* give left player a point */
127            ball_x = SCREEN_W / 2; /* place the ball in the ... */
128            ball_y = SCREEN_H / 2; /* ... center of the screen  */
129            direction = rand() % 4; /* give the ball a random direction */
130         } /* end else */
131      } /* end if */
132   } /* end function moveBall */
133
134   void respondToKeyboard() /* responds to keyboard input */
135   {
136      if ( key[KEY_A] ) /* if A is being pressed... */
137         barL_y -= 3; /* ... move the left paddle up */
138      if ( key[KEY_Z] ) /* if Z is being pressed... */
139         barL_y += 3; /* ... move the left paddle down */
140
141      if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
142         barR_y -= 3; /* ... move the right paddle up */
143      if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
144         barR_y += 3; /* ... move the right paddle down */
145
146      /* make sure the paddles don't go offscreen */
147      if ( barL_y < 30 ) /* if left paddle is going off the top */
148         barL_y = 30;
149      else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
150         barL_y = 380;
151      if ( barR_y < 30 ) /* if right paddle is going off the top */
152         barR_y = 30;
153      else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
154         barR_y = 380;
155   } /* end function respondToKeyboard */
156
157   void reverseVerticalDirection() /* reverse the ball's up-down direction */
158   {
159      if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
160         ++direction; /* make the ball start moving up */
161      else /* "up" directions are odd numbers */
162         --direction; /* make the ball start moving down */
163      play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
164   } /* end function reverseVerticalDirection */
165
166   void reverseHorizontalDirection() /* reverses the horizontal direction */
167   {
168      direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
169      play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
170   } /* end function reverseHorizontalDirection */
```

Fig. E.13 | Finishing up the Pong game. (Part 4 of 7.)

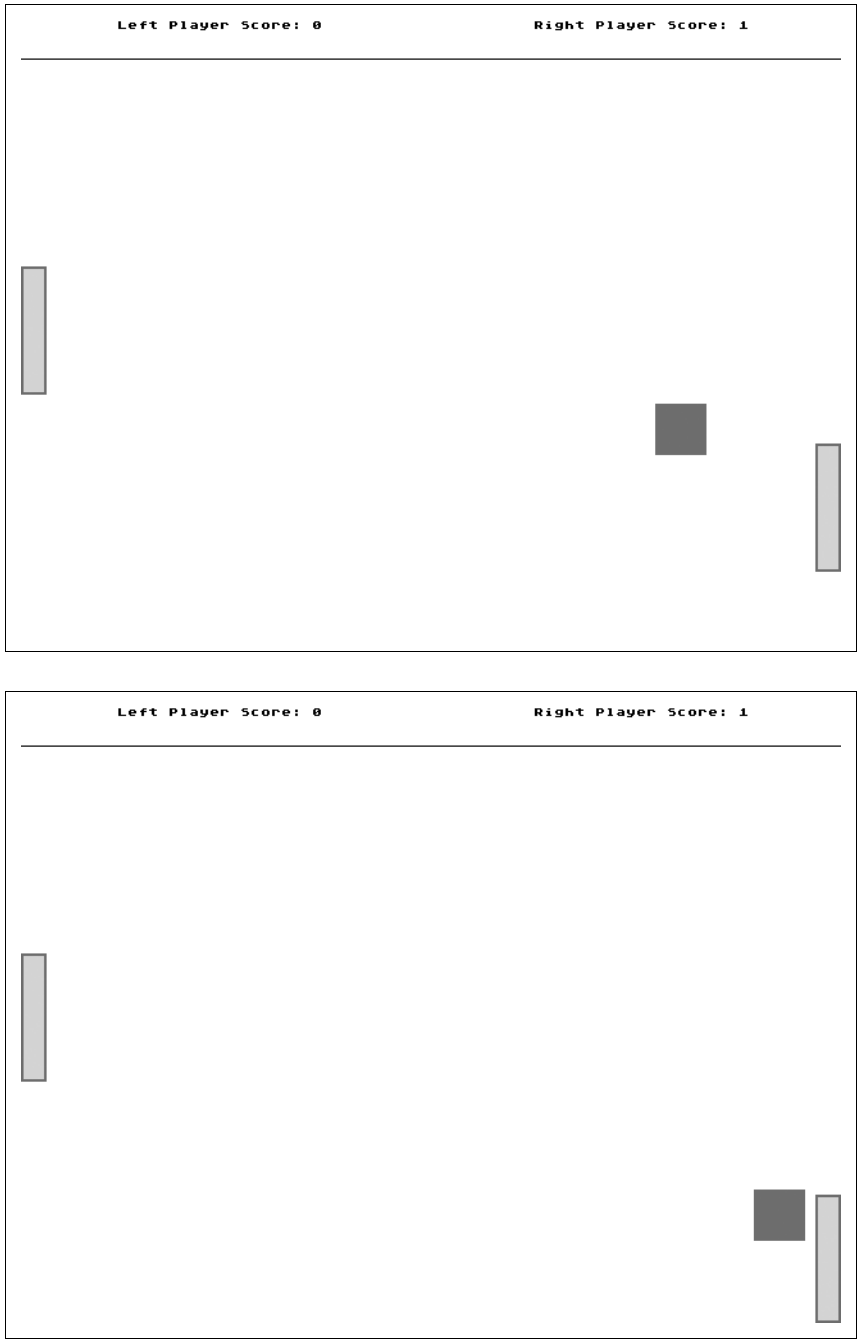

Fig. E.13 | Finishing up the Pong game. (Part 5 of 7.)

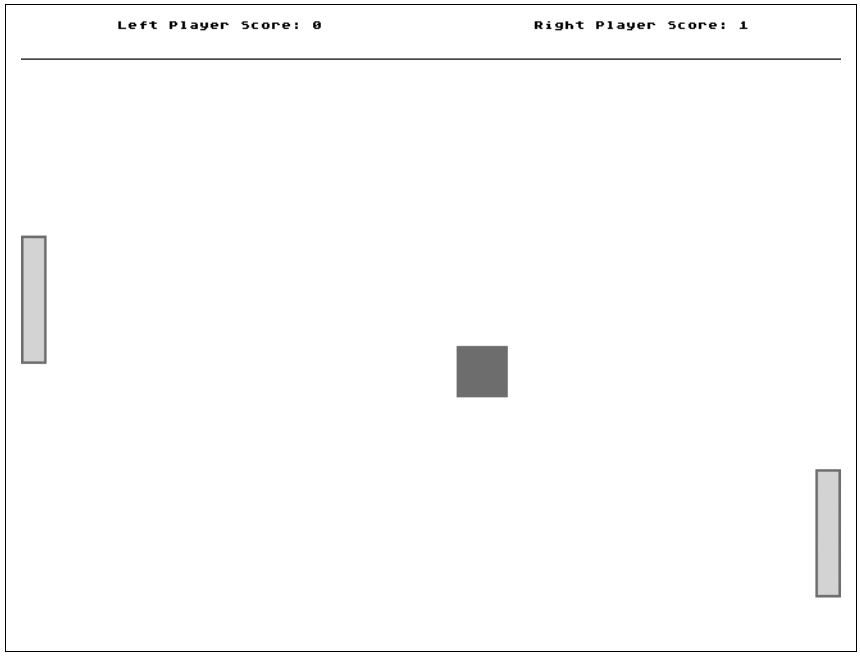

Fig. E.13 | Finishing up the Pong game. (Part 6 of 7.)

Fig. E.13 | Finishing up the Pong game. (Part 7 of 7.)

You may wonder why we chose the boundaries that we did for checking if the paddle is touching the ball. In order for the ball to "bounce" off a paddle, at least 1 pixel of the ball must be touching (i.e., immediately adjacent to) the paddle. Thus, If `ball_y` is 39 smaller than `barL_y`, the bottommost pixel of the ball is touching the bar, and if `ball_y` is 99 greater than `barL_y`, the topmost pixel is touching.

With this version of the program, our Pong game is now fully functional, but there are still a few improvements we can make. The first of these improvements is using timers to regulate the speed of the game, a topic we discuss in the next section.

E.10 Timers in Allegro

One missing feature of our Pong game is the ability to regulate the game's speed. At the moment, the main process of our game in Fig. E.13 is contained in a `while` loop. However, this is not a good practice, as the speed at which a program executes varies from system to system based on factors such as the processor speed. As such our `while` loop could operate at varying speeds depending on the system on which we are running our game, making our ball and paddles move faster or slower than we may want. In this section, we introduce Allegro **timers** to help regulate the speed of our game.

We install the timer handler for Allegro to use by calling the function `install_timer`, which takes no parameters. Once we have called the `install_timer` function, we can add timers to our program. This is done by calling the `install_int` function. The prototype for this function is:

```
int install_int( void ( *function )(), int interval );
```

You'll notice that the `function` parameter is a function pointer. Calling the `install_int` function adds a timer to your program that calls the function specified by `function` every `interval` milliseconds. Thus, if we wanted to call a function `timedFunction` once every second, we would add the following code:

```
install_int( timedFunction, 1000 );
```

There is no need to store a newly-installed timer in a variable—the timer will automatically start running in the background of the program and will remain running until it is removed or the program ends. Also, Allegro only identifies a timer by the function it is set to call. Calling `install_timer` and passing it a function that already has a timer tied to it does not create a new timer; instead, it simply changes the old timer's speed to the one specified by the `interval` parameter. Likewise, removing a timer is done by calling the `remove_timer` function and passing it the name of the function to which the timer is tied.

Allegro allows only 16 timers to be running at once regardless of the system being used. If we try to add more, the `install_int` function fails and returns a non-zero value. Generally, though, you should avoid using an excessive number of timers, as some Allegro processes also require timers to work, and they take up the same slots as regular timers. None of the Allegro functions we have discussed so far require timers, but some of the functions discussed in Section E.12—specifically, the functions that play `.fli` animations and MIDI music files—do need them.

The final step for everything to work correctly is to add the `volatile` qualifier discussed in Chapter 14 to any variable whose value might be changed by our timers. Because Allegro is an external library, compilers cannot recognize what the `install_int` function does. For this reason, the compiler may not "understand" that a variable can be modified by a timer, and it may attempt to optimize the main program code by loading the variable's value into one of the computer's registers, thus failing to notice when that value changes. Adding the `volatile` qualifier to a variable that can be modified by a timer warns the compiler that this value may change unexpectedly, so it must generate code that correctly reloads the latest value from memory.

We can now add timers to our Pong game to regulate how quickly our ball and paddles move on the screen (Fig. E.14).

```
1  /* Fig. E.14: figE_14.c
2     Adding timers to the Pong game. */
3  #include <allegro.h>
4
5  /* symbolic constants for the ball's possible directions */
6  #define DOWN_RIGHT 0
7  #define UP_RIGHT 1
8  #define DOWN_LEFT 2
9  #define UP_LEFT 3
10
11 /* function prototypes */
12 void moveBall( void );
13 void respondToKeyboard( void );
14 void reverseVerticalDirection( void );
15 void reverseHorizontalDirection( void );
```

Fig. E.14 | Adding timers to the Pong game. (Part 1 of 5.)

```
16
17  volatile int ball_x; /* the ball's x-coordinate */
18  volatile int ball_y; /* the ball's y-coordinate */
19  volatile int barL_y; /* y-coordinate of the left paddle */
20  volatile int barR_y; /* y-coordinate of the right paddle */
21  volatile int scoreL; /* score of the left player */
22  volatile int scoreR; /* score of the right player */
23  volatile int direction; /* the ball's direction */
24  BITMAP *ball; /* pointer to ball's image bitmap */
25  BITMAP *bar; /* pointer to paddle's image bitmap */
26  BITMAP *buffer; /* pointer to the buffer */
27  SAMPLE *boing; /* pointer to sound file */
28  FONT *pongFont; /* pointer to font file */
29
30  int main( void )
31  {
32     /* first, set up Allegro and the graphics mode */
33     allegro_init(); /* initialize Allegro */
34     install_keyboard(); /* install the keyboard for Allegro to use */
35     install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
36     install_timer(); /* install the timer handler */
37     set_color_depth( 16 ); /* set the color depth to 16-bit */
38     set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
39     ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
40     bar = load_bitmap( "bar.bmp", NULL); /* load the bar bitmap */
41     buffer = create_bitmap(SCREEN_W, SCREEN_H);/* create buffer */
42     boing = load_sample( "boing.wav" ); /* load the sound file */
43     pongFont = load_font( "pongfont.pcx", NULL, NULL ); /* load the font */
44     ball_x = SCREEN_W / 2; /* give ball its initial x-coordinate */
45     ball_y = SCREEN_H / 2; /* give ball its initial y-coordinate */
46     barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
47     barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
48     scoreL = 0; /* set left player's score to 0 */
49     scoreR = 0; /* set right player's score to 0 */
50     srand( time( NULL ) ); /* seed the random function ... */
51     direction = rand() % 4; /* and then make a random initial direction */
52     /* add timer that calls moveBall every 5 milliseconds */
53     install_int( moveBall, 5 );
54     /* add timer that calls respondToKeyboard every 10 milliseconds */
55     install_int( respondToKeyboard, 10 );
56
57     while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
58     {
59        /* now, perform double buffering */
60        clear_to_color( buffer, ( 255, 255, 255 ) );
61        blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
62        blit( bar, buffer, 0, 0, 0, barL_y, bar->w, bar->h );
63        blit( bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h );
64        line( buffer, 0, 30, 640, 30, makecol( 0, 0, 0 ) );
65        /* draw text onto the buffer */
66        textprintf_ex( buffer, pongFont, 75, 0, makecol( 0, 0, 0 ),
67           -1, "Left Player Score: %d", scoreL );
```

Fig. E.14 | Adding timers to the Pong game. (Part 2 of 5.)

```
68          textprintf_ex( buffer, pongFont, 400, 0, makecol( 0, 0, 0 ),
69             -1, "Right Player Score: %d", scoreR );
70          blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
71          clear_bitmap( buffer );
72       } /* end while */
73
74       remove_int( moveBall ); /* remove moveBall timer */
75       remove_int( respondToKeyboard ); /* remove respondToKeyboard timer */
76       destroy_bitmap( ball ); /* destroy the ball bitmap */
77       destroy_bitmap( bar ); /* destroy the bar bitmap */
78       destroy_bitmap( buffer ); /* destroy the buffer bitmap */
79       destroy_sample( boing ); /* destroy the boing sound file */
80       destroy_font( pongFont ); /* destroy the font */
81       return 0;
82    } /* end function main */
83    END_OF_MAIN() /* don't forget this! */
84
85    void moveBall() /* moves the ball */
86    {
87       switch ( direction ) {
88          case DOWN_RIGHT:
89             ++ball_x; /* move the ball to the right */
90             ++ball_y; /* move the ball down */
91             break;
92          case UP_RIGHT:
93             ++ball_x; /* move the ball to the right */
94             --ball_y; /* move the ball up */
95             break;
96          case DOWN_LEFT:
97             --ball_x; /* move the ball to the left */
98             ++ball_y; /* move the ball down */
99             break;
100         case UP_LEFT:
101            --ball_x; /* move the ball to the left */
102            --ball_y; /* move the ball up */
103            break;
104      } /* end switch */
105
106      /* if the ball is going off the top or bottom ... */
107      if ( ball_y <= 30 || ball_y >= 440 )
108         reverseVerticalDirection(); /* make it go the other way */
109
110      /* if the ball is in range of the left paddle ... */
111      if (ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT))
112      {
113         /* is the left paddle in the way? */
114         if ( ball_y > ( barL_y - 39 ) && ball_y < ( barL_y + 99 ) )
115            reverseHorizontalDirection();
116         else if ( ball_x <= -20 ) { /* if the ball goes off the screen */
117            ++scoreR; /* give right player a point */
118            ball_x = SCREEN_W / 2; /* place the ball in the ... */
119            ball_y = SCREEN_H / 2; /* ... center of the screen  */
```

Fig. E.14 | Adding timers to the Pong game. (Part 3 of 5.)

```
120          direction = rand() % 4; /* give the ball a random direction */
121       } /* end else */
122    } /* end if */
123
124    /* if the ball is in range of the right paddle ... */
125    if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
126    {
127       /* is the right paddle in the way? */
128       if ( ball_y > ( barR_y - 39 ) && ball_y < ( barR_y + 99 ) )
129          reverseHorizontalDirection();
130       else if ( ball_x >= 620 ) { /* if the ball goes off the screen */
131          ++scoreL; /* give left player a point */
132          ball_x = SCREEN_W / 2; /* place the ball in the ... */
133          ball_y = SCREEN_H / 2; /* ... center of the screen  */
134          direction = rand() % 4; /* give the ball a random direction */
135       } /* end else */
136    } /* end if */
137 } /* end function moveBall */
138
139 void respondToKeyboard() /* responds to keyboard input */
140 {
141    if ( key[KEY_A] ) /* if A is being pressed... */
142       barL_y -= 3; /* ... move the left paddle up */
143    if ( key[KEY_Z] ) /* if Z is being pressed... */
144       barL_y += 3; /* ... move the left paddle down */
145
146    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
147       barR_y -= 3; /* ... move the right paddle up */
148    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
149       barR_y += 3; /* ... move the right paddle down */
150
151    /* make sure the paddles don't go offscreen */
152    if ( barL_y < 30 ) /* if left paddle is going off the top */
153       barL_y = 30;
154    else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
155       barL_y = 380;
156    if ( barR_y < 30 ) /* if right paddle is going off the top */
157       barR_y = 30;
158    else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
159       barR_y = 380;
160 } /* end function respondToKeyboard */
161
162 void reverseVerticalDirection() /* reverse the ball's up-down direction */
163 {
164    if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
165       ++direction; /* make the ball start moving up */
166    else /* "up" directions are odd numbers */
167       --direction; /* make the ball start moving down */
168    play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
169 } /* end function reverseVerticalDirection */
170
171 void reverseHorizontalDirection() /* reverses the horizontal direction */
172 {
```

Fig. E.14 | Adding timers to the Pong game. (Part 4 of 5.)

```
173     direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
174     play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
175  } /* end function reverseHorizontalDirection */
```

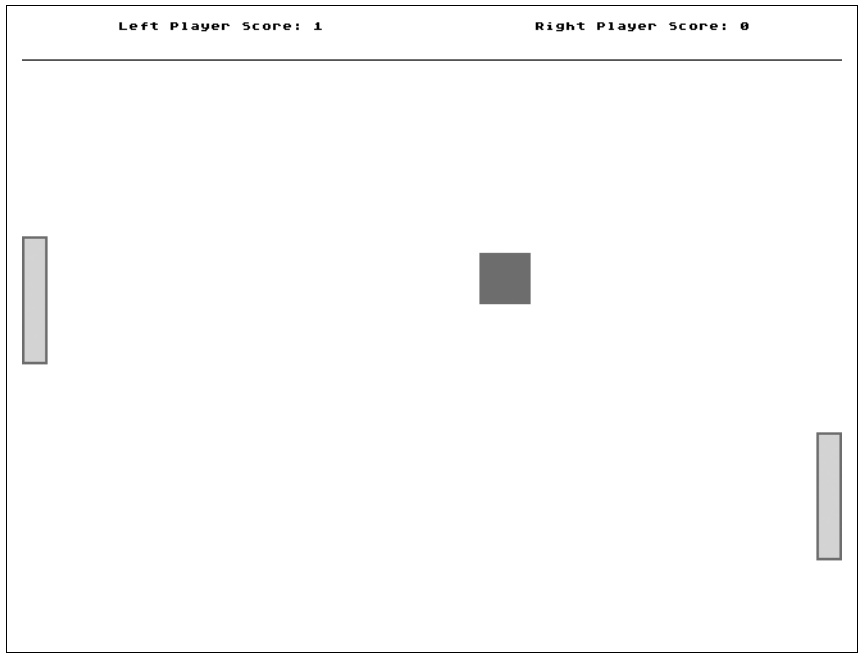

Fig. E.14 | Adding timers to the Pong game. (Part 5 of 5.)

The calls to the `moveBall` and `respondToKeyboard` have been removed from the `while` loop in the `main` of this version of the program.

As you can see, timers are simple to implement. With the code in Fig. E.14, the `moveBall` function is called once every 5 milliseconds, or 200 times per second, which means our ball will move as many pixels every second. Likewise, the program calls `respondToKeyboard` every 10 milliseconds, or 100 times per second, which means (since the `respondToKeyboard` function moves the paddles in intervals of 3 pixels) that the paddles can move 300 pixels in one second. However, while these are the speeds we have suggested for these two timers, feel free to change them so that the game runs at the speed you desire. Experimentation is the only real way to determine a speed that is best for the game.

E.11 The Grabber and Allegro Datafiles

Because most of Allegro's graphics and sounds come from external files, it is necessary for each program to load the files it needs, and to destroy them when the program is finished to prevent memory leaks. When we have a small number of external files, as we do in our Pong game, loading and destroying every file is not a difficult task. However, what if we had a large number of files that we needed to load? Not only would it be difficult to remember every single file that we had to load and destroy, but we would also have to distribute every one of these external files with our game if we chose to release it to the public.

Fortunately, the Allegro designers foresaw this problem, and to fix it, they created the **datafile**. A datafile is a single external file, created by Allegro, that holds the data of multiple external files in one place. A program can load a datafile and instantly have access to all of the files—be they bitmaps, sounds, fonts or anything else—that the datafile contains.

The Allegro directory contains a tool, known as the **grabber**, that we can use to import external files and create datafiles. In this section we use the grabber to create a datafile out of the bitmaps and sounds we used in our Pong game, then we use that datafile to reduce the number of lines in our program.

To create a datafile, we must first open the grabber. On Windows, it is the `grabber.exe` program in the `tools` directory of the tools and examples zip file. On other systems, you should just be able to run "`grabber`" from the command line. Run the grabber and the screen in Fig. E.15 should appear.

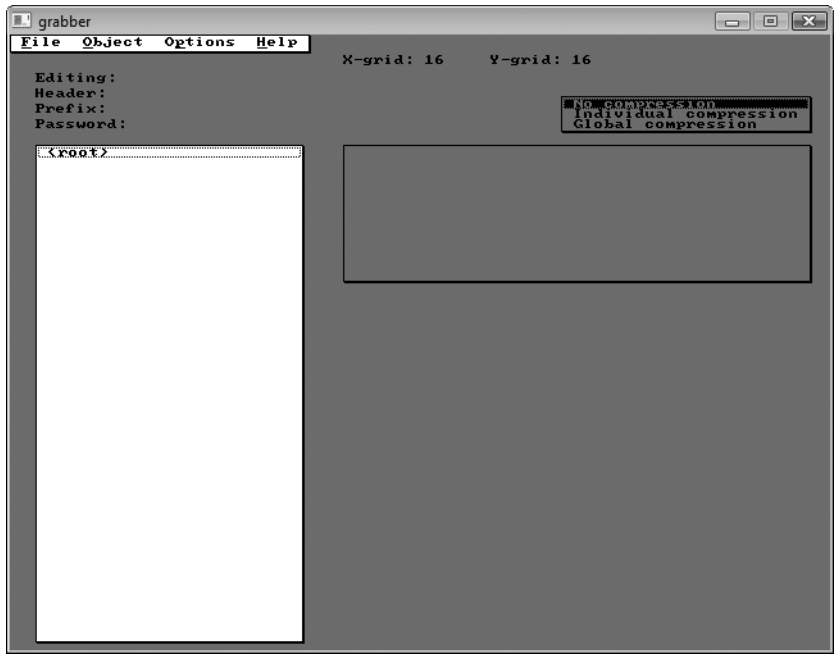

Fig. E.15 | Allegro's grabber utility.

There are four main areas of the grabber program's window. The top area, which contains several text fields, is the space that displays the properties of the datafile that is currently being modified. The white box on the left side of the window lists the objects that make up the datafile being edited; since we haven't yet created any objects, the box is currently empty. Once we import images and sounds into our datafile, the right side of the screen will be used to display the properties of a selected object.

While there are many items in the grabber's menus, we need only a few of them to create a datafile. To add any object to a datafile, we must first tell the grabber to create a new space in the datafile for the given object. We do this by selecting **New** in the **Object**

menu. A list of object types appears. The first thing we import is our ball bitmap, so select **Bitmap** and a dialog box appears asking for the object's name. Name it `BALL` (the reason why the name should be in all caps will become clear shortly). Your screen should now look like Fig. E.16.

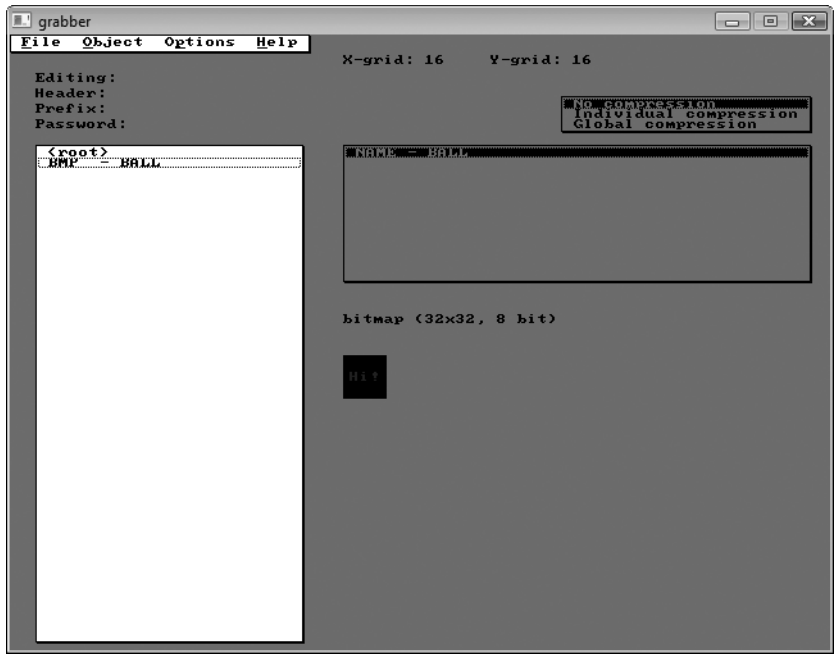

Fig. E.16 | Adding a bitmap to a datafile.

`BALL` has been added to the list of objects on the left side of the window, and that the right side of the window now contains information on the object. However, our object has no image data yet. To import data from a bitmap file, we must first select **Read Bitmap** from the **File** menu. The program asks where the bitmap file is located, so navigate to your Pong project folder and import our `ball.bmp` image. When this is done, the image appears in the window to confirm that it was loaded correctly.

Click anywhere in the window and the screen in Fig. E.16 reappears. The **Read Bitmap** menu item did not actually give our `BALL` object any image data—it merely loaded the image into the grabber's internal memory. To actually apply the bitmap data to our object, make sure that the `BALL` object is selected on the left side of the window, then select **Grab** from the **Object** menu. The screen in Fig. E.17 appears.

The program now asks us which part of the bitmap we want the object to contain. This feature is useful if we want our object to contain only part of an image, but right now we want the entirety of the ball bitmap. Move the cursor to the top-left corner of the image, then click and drag a box over the entire bitmap. The screen in Fig. E.18 should appear.

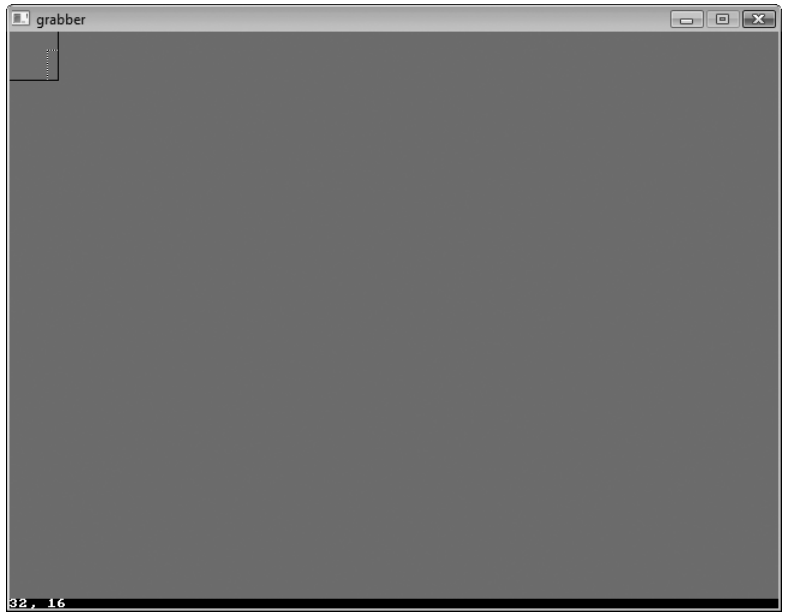

Fig. E.17 | Applying an imported bitmap to an object.

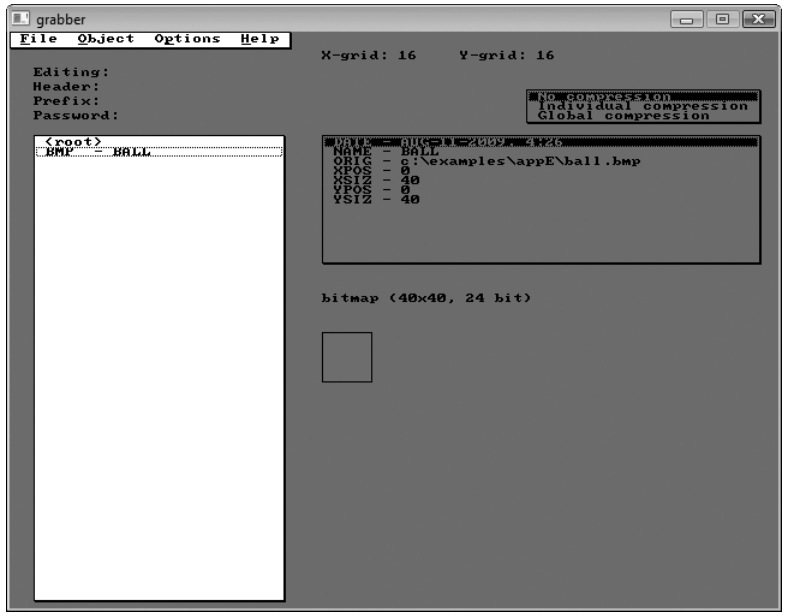

Fig. E.18 | A complete imported object.

With this done, we have finished importing our ball image into the datafile. Create a new bitmap in the grabber called `BAR` and repeat the process with the `bar.bmp` bitmap to import the paddle image as well. Both the `BALL` and `BAR` objects should now be in the list of items on the left side of the window.

Now that we have imported our bitmaps, we must also import our sound and font files. First, use the **Object** menu to create a new object of type **Sample** named `BOING`. Then, simply select **Grab** from the **Object** menu. The program asks for the location of the file, so locate our `boing.wav` file and double-click on it. The grabber then imports the sound file and applies its data to the `boing` object.

We use the same process to create our font object—create a new font called `PONGFONT`, and then use **Grab** to import the `pongfont.pcx` file. Once you have imported all the objects, your screen should look like Fig. E.19.

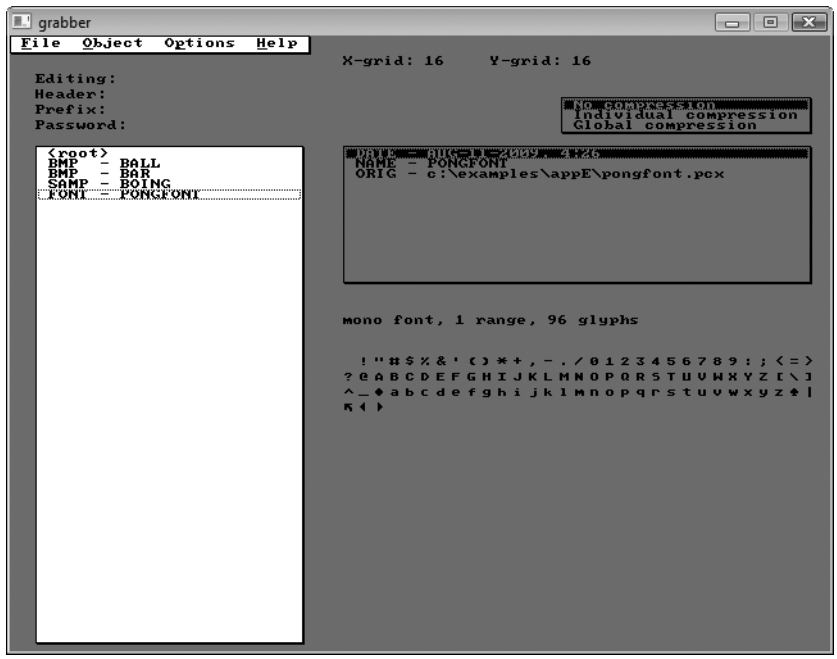

Fig. E.19 | The grabber window after importing all of our objects.

Now that we've imported all our objects into the grabber, there is still one final step we must perform before we can use the datafile effectively. You will notice that we have not yet done anything with the text fields at the top of the grabber window. However, the **Header** field is of interest to us. Filling in this field with a filename will make the grabber save a header file alongside the datafile that we are currently creating. This header file can be used to make it easier to access individual datafile objects in a program. We will discuss the header file in more detail shortly, but for now just enter `pong.h` in the text field. Then select **Save** from the **File** menu and save the datafile as `pongdatafile.dat` in the folder where your Pong project is located.

Now we face the task of loading the datafile into our program, which is not difficult. Just as Allegro defines `BITMAP*`, `SAMPLE*`, and `FONT*` pointer types for bitmaps, sounds and fonts, it also defines the **`DATAFILE*` type** that points to datafiles. Likewise, to load a datafile into a program, we call the function `load_datafile` and pass it the filename of the datafile we have created. Note, though, that Allegro does not define a function `destroy_datafile`. To free the memory allocated to a datafile, one must call the function `unload_datafile`.

Once a datafile is loaded into a program, Allegro considers it to be an array of objects, with each object having a specific index in the array. If we have a `DATAFILE*` variable in our program called `myDatafile`, accessing a specific object in the datafile is done with the code `myDatafile[i].dat`, where `i` is the object's index in the array. Normally, Allegro assigns indexes to the objects in a datafile by the order in which the objects were imported, so in our datafile, our `BALL` object has index 0, our `BAR` object has index 1, and so on.

It is easy for us to remember the indexes of the objects in our datafile, as it contains only four objects. However, with a large datafile, remembering the indexes of each and every object can be difficult. Fortunately, even with our small datafile, we do not have to memorize each object's index, as the `pong.h` header file we saved takes care of that for us. If you open the header file in your IDE, you will see the code in Fig. E.20.

```
1  /* Allegro datafile object indexes, produced by grabber v4.2.2, MinGW32 */
2  /* Datafile: c:\examples\appE\pongdatafile.dat */
3  /* Date: Mon Aug 10 16:59:35 2009 */
4  /* Do not hand edit! */
5
6  #define BALL                    0       /* BMP   */
7  #define BAR                     1       /* BMP   */
8  #define BOING                   2       /* SAMP  */
9  #define PONGFONT                3       /* FONT  */
```

Fig. E.20 | The `pong.h` header file.

By including the header file in our program, we eliminate the need to memorize the indexes of the objects in our datafile, as the header file assigns a symbolic constant to each index. The symbolic constant for any given object is the name we gave that object in the grabber when we imported it, which explains why we chose to name our objects in all caps—the convention for symbolic constants in C is to give them names with only capital letters. The Pong program in Fig. E.21 loads and accesses a datafile

```
1  /* Fig. E.21: figE_21.c
2     Using datafiles. */
3  #include <allegro.h>
4  #include "pong.h"
5
6  /* symbolic constants for the ball's possible directions */
7  #define DOWN_RIGHT 0
8  #define UP_RIGHT 1
9  #define DOWN_LEFT 2
10 #define UP_LEFT 3
```

Fig. E.21 | Using datafiles. (Part 1 of 5.)

```
11
12  /* function prototypes */
13  void moveBall( void );
14  void respondToKeyboard( void );
15  void reverseVerticalDirection( void );
16  void reverseHorizontalDirection( void );
17
18  volatile int ball_x; /* the ball's x-coordinate */
19  volatile int ball_y; /* the ball's y-coordinate */
20  volatile int barL_y; /* y-coordinate of the left paddle */
21  volatile int barR_y; /* y-coordinate of the right paddle */
22  volatile int scoreL; /* score of the left player */
23  volatile int scoreR; /* score of the right player */
24  volatile int direction; /* the ball's direction */
25  BITMAP *buffer; /* pointer to the buffer */
26  DATAFILE *pongData; /* pointer to the datafile */
27
28  int main( void )
29  {
30     /* first, set up Allegro and the graphics mode */
31     allegro_init(); /* initialize Allegro */
32     install_keyboard(); /* install the keyboard for Allegro to use */
33     install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
34     install_timer(); /* install the timer handler */
35     set_color_depth( 16 ); /* set the color depth to 16-bit */
36     set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
37     buffer = create_bitmap( SCREEN_W, SCREEN_H ); /* create buffer */
38     pongData = load_datafile( "pongdatafile.dat" ); /* load the datafile */
39     ball_x = SCREEN_W / 2; /* give ball its initial x-coordinate */
40     ball_y = SCREEN_H / 2; /* give ball its initial y-coordinate */
41     barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
42     barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
43     scoreL = 0; /* set left player's score to 0 */
44     scoreR = 0; /* set right player's score to 0 */
45     srand( time( NULL ) ); /* seed the random function ... */
46     direction = rand() % 4; /* and then make a random initial direction */
47     /* add timer that calls moveBall every 5 milliseconds */
48     install_int( moveBall, 5 );
49     /* add timer that calls respondToKeyboard every 10 milliseconds */
50     install_int( respondToKeyboard, 10 );
51
52     while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
53     {
54        /* now, perform double buffering */
55        clear_to_color( buffer, makecol( 255, 255, 255 ) );
56        blit( pongData[BALL].dat, buffer, 0, 0, ball_x, ball_y, 40, 40 );
57        blit( pongData[BAR].dat, buffer, 0, 0, 0, barL_y, 20, 100 );
58        blit( pongData[BAR].dat, buffer, 0, 0, 620, barR_y, 20, 100 );
59        line( buffer, 0, 30, 640, 30, makecol( 0, 0, 0 ) );
60        /* draw text onto the buffer */
61        textprintf_ex( buffer, pongData[PONGFONT].dat, 75, 0,
62           makecol( 0, 0, 0 ), -1, "Left Player Score: %d", scoreL );
```

Fig. E.21 | Using datafiles. (Part 2 of 5.)

```
63          textprintf_ex( buffer, pongData[PONGFONT].dat, 400, 0,
64             makecol( 0, 0, 0 ), -1, "Right Player Score: %d", scoreR );
65          blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
66          clear_bitmap( buffer );
67       } /* end while */
68
69       remove_int( moveBall ); /* remove moveBall timer */
70       remove_int( respondToKeyboard ); /* remove respondToKeyboard timer */
71       destroy_bitmap( buffer ); /* destroy the buffer bitmap */
72       unload_datafile( pongData ); /* unload the datafile */
73       return 0;
74    } /* end function main */
75    END_OF_MAIN() /* don't forget this! */
76
77    void moveBall() /* moves the ball */
78    {
79       switch ( direction ) {
80          case DOWN_RIGHT:
81             ++ball_x; /* move the ball to the right */
82             ++ball_y; /* move the ball down */
83             break;
84          case UP_RIGHT:
85             ++ball_x; /* move the ball to the right */
86             --ball_y; /* move the ball up */
87             break;
88          case DOWN_LEFT:
89             --ball_x; /* move the ball to the left */
90             ++ball_y; /* move the ball down */
91             break;
92          case UP_LEFT:
93             --ball_x; /* move the ball to the left */
94             --ball_y; /* move the ball up */
95             break;
96       } /* end switch */
97
98       /* if the ball is going off the top or bottom ... */
99       if ( ball_y <= 30 || ball_y >= 440 )
100         reverseVerticalDirection(); /* make it go the other way */
101
102      /* if the ball is in range of the left paddle ... */
103      if (ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT))
104      {
105         /* is the left paddle in the way? */
106         if ( ball_y > ( barL_y - 39 ) && ball_y < ( barL_y + 99 ) )
107            reverseHorizontalDirection();
108         else if ( ball_x <= -20 ) { /* if the ball goes off the screen */
109            ++scoreR; /* give right player a point */
110            ball_x = SCREEN_W / 2; /* place the ball in the ... */
111            ball_y = SCREEN_H / 2; /* ... center of the screen  */
112            direction = rand() % 4; /* give the ball a random direction */
113         } /* end else */
114      } /* end if */
115
```

Fig. E.21 | Using datafiles. (Part 3 of 5.)

```
116    /* if the ball is in range of the right paddle ... */
117    if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
118    {
119       /* is the right paddle in the way? */
120       if ( ball_y > ( barR_y - 39 ) && ball_y < ( barR_y + 99 ) )
121          reverseHorizontalDirection();
122       else if ( ball_x >= 620 ) { /* if the ball goes off the screen */
123          ++scoreL; /* give left player a point */
124          ball_x = SCREEN_W / 2; /* place the ball in the ... */
125          ball_y = SCREEN_H / 2; /* ... center of the screen  */
126          direction = rand() % 4; /* give the ball a random direction */
127       } /* end else */
128    } /* end if */
129 } /* end function moveBall */
130
131 void respondToKeyboard() /* responds to keyboard input */
132 {
133    if ( key[KEY_A] ) /* if A is being pressed... */
134       barL_y -= 3; /* ... move the left paddle up */
135    if ( key[KEY_Z] ) /* if Z is being pressed... */
136       barL_y += 3; /* ... move the left paddle down */
137
138    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
139       barR_y -= 3; /* ... move the right paddle up */
140    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
141       barR_y += 3; /* ... move the right paddle down */
142
143    /* make sure the paddles don't go offscreen */
144    if ( barL_y < 30 ) /* if left paddle is going off the top */
145       barL_y = 30;
146    else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
147       barL_y = 380;
148    if ( barR_y < 30 ) /* if right paddle is going off the top */
149       barR_y = 30;
150    else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
151       barR_y = 380;
152 } /* end function respondToKeyboard */
153
154 void reverseVerticalDirection() /* reverse the ball's up-down direction */
155 {
156    if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
157       ++direction; /* make the ball start moving up */
158    else /* "up" directions are odd numbers */
159       --direction; /* make the ball start moving down */
160    play_sample( pongData[BOING].dat, 255, 128, 1000, 0 ); /* play sound */
161 } /* end function reverseVerticalDirection */
162
163 void reverseHorizontalDirection() /* reverses the horizontal direction */
164 {
165    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
166    play_sample( pongData[BOING].dat, 255, 128, 1000, 0 ); /* play sound */
167 } /* end function reverseHorizontalDirection */
```

Fig. E.21 | Using datafiles. (Part 4 of 5.)

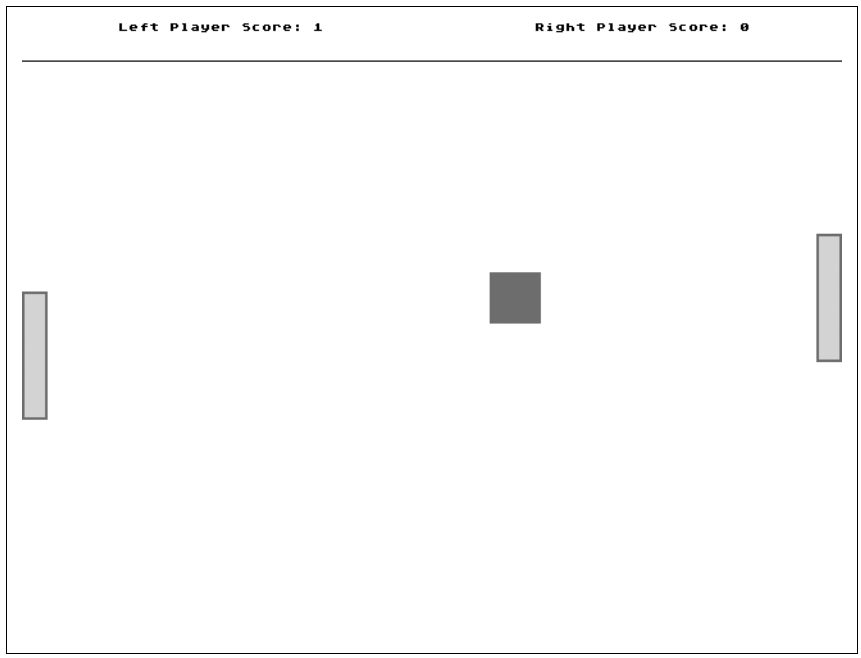

Fig. E.21 | Using datafiles. (Part 5 of 5.)

A downside of datafiles can be seen in lines 56–58. When we blitted our ball and paddle onto the screen, we had to pass the last two parameters (the `width` and `height` of the source bitmap) explicitly. Allegro considers objects loaded from a datafile to be of type `void *`, which cannot be dereferenced. As such, trying to pass these parameters `pongData[BALL].dat->w` and `pongData[BALL].dat->h` is a syntax error. Still, since we know the exact dimensions of our bitmaps, this is not a problem.

Using our datafile had no effect on the game itself—it runs exactly the same as the program of Fig. E.14. However, this program requires fewer lines of code for the same functionality. While it is only 8 lines in this case, a game with a larger number of objects in a datafile would most likely have a much larger difference. If you create a game with many bitmaps, sounds, and other files, make sure to use a datafile to reduce your program's length.

E.12 Other Allegro Capabilities

We mentioned several times in this appendix that Allegro is capable of drawing simple graphics; we used `line` functions to draw a line in our game. Allegro can also draw many other shapes, such as triangles, circles, arcs and polygons with an arbitrary number of sides. The full list of these functions is at `www.allegro.cc/manual/api/drawing-primitives/`.

Allegro also has a set of functions devoted solely to playing music using MIDI files. It defines a `MIDI*` type that points to MIDI files, and uses the `load_midi`, `play_midi`, `stop_midi` and `destroy_midi` functions to manipulate them. A full list of these functions is at `www.allegro.cc/manual/api/music-routines-(midi)/`.

If you have any `.fli` format animations that you would like to include in your game, Allegro can play them, as well. Allegro does not define a type that points to animations, but it can draw an animation directly onto a bitmap using the `play_fli` function. The documentation for this capability is at `www.allegro.cc/manual/api/flic-routines/`.

E.13 Allegro Resource Center

For more information on Allegro, visit the Allegro and C Game Programming section of our C resource center at `www.deitel.com/allegro/`.

Summary

Section E.3 A Simple Allegro Program

- Every Allegro program must include the `allegro.h` header, call the `allegro_init` function, and must have a call to the `END_OF_MAIN` macro immediately following the closing brace of the program's `main` function.
- The `allegro_init` function initializes the Allegro library. It must be called before any other Allegro functions, or the program will not work correctly.
- The `allegro_message` function is used to give the user a message when there is no graphical way of doing so.
- Windows, some Unix systems, and Mac OS X cannot run Allegro programs without the `END_OF_MAIN` macro. If it is missing, compilers on those systems will not be able to compile an Allegro program. Make sure to include the `END_OF_MAIN` macro to ensure compatibility with systems that require it.

Section E.4 Simple Graphics: Importing Bitmaps and Blitting

- Most of Allegro's graphics come from external files. Allegro defines several different variable types that point to image data in memory.
- The `BITMAP*` type is the most basic type defined by Allegro for pointing to stored image data in memory.
- The `set_color_depth` function is used to set the color depth of an Allegro program. The color depth can be set to 8-, 15-, 16-, 24-, or 32-bit. A lower color depth requires less memory, but a higher color depth results in a better-looking program.
- The `set_gfx_mode` function is used to set the graphics mode of an Allegro program. In addition to setting how the program is displayed (i.e., fullscreen mode or windowed mode), it also sets how many pixels are in the width and height of the screen or window.
- Allegro defines five "magic drivers" that can be passed to the `set_gfx_mode` function to specify whether the program should be run in fullscreen or windowed mode—`GFX_AUTODETECT`, `GFX_AUTODETECT_FULLSCREEN`, `GFX_AUTODETECT_WINDOWED`, `GFX_SAFE`, and `GFX_TEXT`.
- Most Allegro functions that can fail return `int`s for error-checking purposes. Generally these functions will return 0 if they succeed, and a non-zero value if they do not.
- An Allegro program must set the color depth and graphics mode of a program before attempting to do anything else with graphics.
- The `create_bitmap` function creates a new, blank bitmap.
- Use the `load_bitmap` function to load an image from an external bitmap file. If the color depth is set to 8-bit, you must pass this function the bitmap's palette in addition to the image itself.

- If the `load_bitmap` function fails, it returns `NULL`. It does not cause an error.
- The `blit` function is one of the most important functions in Allegro. It is used to draw a block from one bitmap onto another bitmap.
- A larger *x*-coordinate corresponds to further to the right in Allegro, but a larger *y*-coordinate corresponds to further down, not up.
- The `BITMAP*` type is a pointer to a `struct`. This structure contains two `int`s, `w` and `h`, that store the width and height of the bitmap in pixels, respectively.
- Use the `destroy_bitmap` function to destroy a bitmap and free the memory allocated to it to prevent memory leaks.

Section E.5 Animation with Double Buffering

- Animation in Allegro is done by blitting an object onto the screen in different places at regular intervals.
- The symbolic constants `SCREEN_W` and `SCREEN_H` are reserved by Allegro, and correspond to the width and height of the screen in pixels, respectively.
- Use the `clear_to_color` function to make the entirety of a bitmap a certain color. This is useful for setting a background color for a program.
- The `makecol` function is used to return `int`s that Allegro recognizes as various colors.
- The "double buffering" technique is a method that produces smooth animation. The technique consists of drawing everything onto an intermediary bitmap known as the buffer, and then drawing the entirety of the buffer onto the screen.

Section E.6 Importing and Playing Sounds

- Allegro defines the type `SAMPLE*` that points to sound file data stored in memory.
- Before any sounds can be played in Allegro, the `install_sound` function must be called.
- Allegro defines two "magic drivers" that should be passed to the `install_sound` function so that Allegro can determine which sound card drivers it should use for playing sounds. These "magic drivers" are `DIGI_AUTODETECT` and `MIDI_AUTODETECT`.
- The `load_sample` function is used to load an external sound file.
- Use the `play_sample` function to play a digital sample, and the `stop_sample` function to stop it.
- The `volume` parameter in the `play_sample` function determines the volume at which the sample should be played. A value of 0 mutes the sample, while a value of 255 plays it at maximum volume.
- The `pan` parameter in the `play_sample` function determines the pan position at which the sample should be played. A value of 128 plays the sample out of both speakers equally. A value lower than this will shift the sound towards the left speaker, while a greater value (up to a maximum of 255) will shift the sound towards the right speaker.
- The `frequency` parameter in the `play_sample` function determines the frequency (and therefore the pitch) at which the sample should be played. A value of 1000 will play the sample at normal frequency. A value of 2000 will play it at double the normal frequency, a value of 500 will play it at half, and so on.
- Passing the `loop` parameter of the `play_sample` function a value of 0 will make the sample play only once before stopping. Passing it any other value will cause the sample to loop indefinitely.
- Use the `destroy_sample` function to destroy a sample and free the memory allocated to it to prevent memory leaks.

Section E.7 Keyboard Input

- You must call the `install_keyboard` function to allow Allegro to recognize and use the keyboard.
- Allegro defines an array of `int`s called `key` which contains an index for each key on the keyboard. If a key is not being pressed, its respective index in the array will contain 0, while if the key is being pressed, the index will contain a non-zero number.
- Allegro defines several symbolic constants that correspond to keys on the keyboard. These constants are used in tandem with the `key` array to determine if specific keys are being pressed. The value stored at `key[KEY_A]` determines whether or not the *A* key is being pressed, the value stored at `key[KEY_SPACE]` determines whether the spacebar is being pressed, and so on.
- Any program that checks the keyboard for input using the `key` array should do so repeatedly. Otherwise, keypresses may be missed.

Section E.8 Fonts and Displaying Text

- Allegro defines the `FONT*` type that points to stored font data in memory.
- The symbolic constant `font` corresponds to Allegro's default font.
- Use the `load_font` function to load font data from external files. If the color depth is set to 8-bit, you must pass this function the font's palette in addition to the font itself.
- The `textprintf_ex` function prints text on the screen. Functions `textprintf_centre_ex` and `textprintf_right_ex` do the same thing, but justify the printed text at different positions.
- When Allegro is expecting an `int` that corresponds to a color as a parameter, passing the parameter a value of -1 will make Allegro interpret that color as "transparent."
- Use the `destroy_font` function to destroy a font and free the memory allocated to it to prevent memory leaks.

Section E.10 Timers in Allegro

- Any program that uses timers must call the `install_timer` function before attempting to add any timers.
- Allegro can have up to 16 timers running at once.
- Timers are added by calling the `install_int` function, and removed by calling the `remove_int` function.
- A timer calls a given function at regular intervals until it is removed. There is no need to store a timer in any type of variable.
- Allegro identifies a timer by the function it is programmed to call.
- Any variable whose value is modified by a function that a timer calls must be given the `volatile` qualifier to ensure the program works correctly.

Section E.11 The Grabber and Allegro Datafiles

- An Allegro datafile is a single external file that holds the data of many external files in one place.
- Allegro provides the grabber utility for the creation and editing of datafiles.
- The grabber can create header files that make it simple to access objects contained in a datafile.
- Loading a datafile into a program is done with the `load_datafile` function, and removing them is done with the `unload_datafile` function.
- Once a datafile is loaded in a program, Allegro considers it to be an array of objects. The index of each object in the array is dependent on the order in which the objects were imported into the datafile. The first object added to the datafile has index 0, the second one has index 1, and so on.
- Allegro considers objects loaded from a datafile to be of type `void *`.

Terminology

Allegro II
`BITMAP*` type IV, VIII
`blit` VII
datafile XLIII
`DATAFILE*` type XLVII
`FONT*` type XXVI
grabber XLII, XLIII, XLVI
`SAMPLE` type XVI
`set_gfx_mode` VI
timers in Allegro XXXVII
virtual screen VII

Self-Review Exercises

E.1 Fill in the blanks in each of the following statements:

a) Every Allegro program must include the _____ header.
b) The _____ function must be called before any other Allegro function.
c) Adding the _____ macro ensures compatibility with systems that require it.
d) Before Allegro can display any graphics, a program must call both the _____ and _____ functions.
e) The _____ function is used to draw a block of one bitmap onto another.
f) The main type defined by Allegro for pointing to sound file data is the _____ type.
g) Allegro defines the _____ symbolic constant that corresponds to Allegro's default font.
h) The _____ function is used to return an integer that Allegro interprets as a color.
i) Allegro can have up to _____ timers running at once.
j) The _____ function is used to add a timer to a program.
k) The _____ utility is used to create and edit Allegro datafiles.

E.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `screen` bitmap is the only bitmap visible to the user.
b) The coordinates (0, 0) refer to the bottom-left corner of a bitmap.
c) If Allegro attempts to load an external file that does not exist, a runtime error will occur.
d) The double buffering technique requires two intermediary bitmaps, or buffers, to work correctly.
e) Passing a value of 2 to the `loop` parameter in the `play_sample` function will cause the sound file to play twice before stopping.
f) The `install_keyboard` function must be passed a parameter that gives Allegro the driver information of the system's keyboard.
g) A program that draws text on the screen must specify a font in which that text should be drawn.
h) An Allegro program can have up to 32 timers running at once.
i) The function used for freeing the memory that is storing a datafile is the `destroy_datafile` function.

E.3 Write statements to accomplish each of the following:

a) Set the graphics mode of an Allegro program to a window that is 640 pixels wide by 480 pixels high.
b) Draw the bitmap `bmp` onto the top left corner of the bitmap `buffer`.
c) Play the digital sample `sample` at maximum volume, centered pan position, and normal frequency without looping.
d) If the spacebar is being pressed, set the value of the `int number` to 0.
e) Draw the string `"Hello!"` onto the top-left corner of the bitmap `buffer`, using Allegro's default font with a blue foreground color and transparent background color.
f) Add a timer that calls the function `timedFunction` four times every second.
g) Load the datafile with the filename `datafile.dat`.

E.4 Find the error in each of the following:

a) `BITMAP bmp;`

b) `set_gfx_mode( WINDOWED, 640, 480, 0, 0 );`

c) `makecol( 0, 0, 256 );`

Answers to Self-Review Exercises

E.1 a) `allegro.h`. b) `allegro_init`. c) `END_OF_MAIN`. d) `set_color_depth` and `set_gfx_mode`. e) `blit`. f) `SAMPLE*`. g) `font`. h) `makecol`. i) 16. j) `install_int`. k) grabber.

E.2 a) True.

b) False. The coordinates (0, 0) refer to the top-left corner of a bitmap.

c) False. The function that returns the pointer to the external file will return `NULL`, but no error will occur on that line.

d) False. Double buffering requires only one intermediary bitmap.

e) False. Passing any value other than 0 to the `loop` parameter will make the sound file loop indefinitely.

f) False. The `install_keyboard` function does not take any parameters.

g) True.

h) False. An Allegro program can have only 16 timers running at once.

i) False. This function is the `unload_datafile` function. The `destroy_datafile` function is not defined by Allegro.

E.3 a)
```
set_gfx_mode( GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0 );
```

b)
```
blit( bmp, buffer, 0, 0, 0, 0, bmp->w, bmp->h );
```

c)
```
play_sample( sample, 255, 128, 1000, 0 );
```

d)
```
if ( key[KEY_SPACE] )
   number = 0;
```

e)
```
textprintf_ex( buffer, font, 0, 0, makecol( 0, 0, 255 ), -1, "Hello!" );
```

f)
```
install_int( timedFunction, 250 );
```

g)
```
load_datafile( "datafile.dat" );
```

E.4 a) The variable `bmp` should instead be declared as a pointer to a `BITMAP`. All of Allegro's bitmap functions either take a pointer as a parameter or return a pointer. A `BITMAP` that is not a pointer is essentially useless.

b) Allegro does not define a `WINDOWED` "magic driver." Use the `GFX_AUTODETECT_WINDOWED` "magic driver" instead.

c) The `makecol` function can only accept parameters with values between 0 and 255.

Exercises

E.5 ***(Moving an Image Via the Arrow Keys)*** Write a program that draws the bitmap `ball.bmp` in the center of the screen. When the user presses one of the arrow keys, the bitmap should move ten pixels in that direction.

E.6 ***(Moving an Image Via the Arrow Keys)*** Modify the program from Exercise E.5 so that the ball will only move once for each time an arrow key is pressed. If the user holds down an arrow key, the ball should move once and then stop until the user releases and presses the key again.

E.7 ***(Moving an Image Via the Arrow Keys)*** Modify the program from Exercise E.5 so that if the user holds down an arrow key, the ball will only move once every second.

E.8 ***(Ending the Pong Game at 21)*** Modify the Pong game from Fig. E.21 so that when a player reaches 21 points, the game ends and displays a message that the left or right player has won.

E.9 *(Moving the Ball Faster in a Long Rally)* In most Pong games, when a rally between the two players lasts for a long time, the ball begins to speed up in order to prevent a stalemate. Modify the Pong game from Fig. E.21 so that the ball's speed increases for every ten times that it is hit in a rally. When either player scores, the ball should return to its original speed.

E.10 *(Altering the Paddle Speeds)* Some Pong games also modify the speed of one or both player's paddles in an effort to keep the game balanced. Modify the Pong game from Fig. E.21 so that when one player has a lead of at least 5 points, his or her paddle begins to slow down. The greater that player's lead, the slower his or her paddle should move. If the player's lead falls to under 5 points, his or her paddle should return to normal speed.

E.11 *(Adding a Pause Feature)* Video games often have a "pause" feature that allows a player to interrupt a game in progress and then resume it later. Modify the Pong game from Fig. E.21 so that pressing the *P* key will pause the game and halt the movement of the ball and paddles. Pausing the game should also make the message `"PAUSED"` appear in the center of the screen. If the game is paused, pressing the *R* key should clear the `"PAUSED"` message from the screen and resume the game.

E.12 *(Playing a Sound When a Ball Hits a Paddle)* Modify the Pong game from Fig. E.21 so that when the ball bounces off a wall or paddle, the word `"BOING!"` appears in blue at the point where the ball bounced and then fades away. The text should not simply vanish—it should gradually fade to white.

E.13 *(Providing Ball and Paddle Speed Options)* Modify the Pong game from Fig. E.21 so that before the game begins, a menu appears on the screen that allows the players to choose from several different ball and paddle speeds. This is tougher than it sounds! Allegro does not have any text input methods. You will have to find another method of solving this problem.