Game Programming and Graphics in C with Allegro
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 chapter you will learn:

- How to install the Allegro game programming library to work with your C programs.
- To create games using Allegro.
- To use Allegro to import and display graphics.
- To use the "double buffering" technique to create smooth animations.
- To use Allegro to import and play sounds.
- To have Allegro recognize and deal with keyboard input.
- To create the simple game "Pong" with Allegro.
- To use Allegro timers to regulate the speed of a game.
- To use Allegro datafiles to shorten the amount of code in a program.
- The many other features Allegro can add to a game.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>15.2</td>
<td>Installing Allegro</td>
</tr>
<tr>
<td>15.3</td>
<td>A Simple Allegro Program</td>
</tr>
<tr>
<td>15.4</td>
<td>Simple Graphics: Importing Bitmaps and Blitting</td>
</tr>
<tr>
<td>15.5</td>
<td>Animation with Double Buffering</td>
</tr>
<tr>
<td>15.6</td>
<td>Importing and Playing Sounds</td>
</tr>
<tr>
<td>15.7</td>
<td>Keyboard Input</td>
</tr>
<tr>
<td>15.8</td>
<td>Fonts and Displaying Text</td>
</tr>
<tr>
<td>15.9</td>
<td>Implementing the Game of Pong</td>
</tr>
<tr>
<td>15.10</td>
<td>Timers in Allegro</td>
</tr>
<tr>
<td>15.11</td>
<td>The Grabber and Allegro Datafiles</td>
</tr>
<tr>
<td>15.12</td>
<td>Other Allegro Capabilities</td>
</tr>
<tr>
<td>15.13</td>
<td>Allegro Internet and Web Resources</td>
</tr>
</tbody>
</table>
15.1 Introduction

- Allegro
  - C library created to aid in the development of games
  - Created in 1995 by Shawn Hargreaves
  - Adds many game-related features to C
    - Importing, displaying, and animating graphics
    - Playing sounds
    - Keyboard input
    - Outputting text graphically
    - Timers that call functions at regular intervals
  - In this chapter we will program the game “Pong” to demonstrate all of Allegro’s capabilities
15.2 Installing Allegro

- Allegro must be installed before it can be used with C
  - The installation process differs slightly for different systems
  - Detailed instructions in the book and the Allegro documentation
15.3 A Simple Allegro Program

- Every Allegro program must have three components:
  - `#include <allegro.h>`
  - `allegro_init`
  - `END_OF_MAIN`

- `#include <allegro.h>`
  - `allegro.h` header contains all Allegro function prototypes and variable type definitions
  - Header must be included or Allegro program will not compile
15.3 A Simple Allegro Program

- `allegro_init`
  - Initializes Allegro library
  - Must be present in all Allegro programs, and must be called before any other Allegro functions can be used

- `END_OF_MAIN`
  - Macro that must be placed immediately after the closing brace of `main` function
  - Ensures Allegro executable file will run on all systems
  - Required for an Allegro program to run on Windows, some UNIX systems, and Mac OS X
  - Not required on other systems, but recommended to ensure cross-platform compatibility
/* Fig. 15.1: fig15_01.c
A simple Allegro program */
#include <allegro.h>

int main( void )
{
    allegro_init(); /* initialize Allegro */
allegro_message( "Welcome to Allegro!" ); /* display a message */
    return 0;
} /* end function main */

END_OF_MAIN() /* Allegro-specific macro */
15.4 Importing Graphics and Blitting

**Importing Graphics**
- Allegro can draw simple lines and shapes on its own
- More complicated graphics usually come from external files
- Allegro can load data from image files into memory and defines several variable types to point to this data
  - **BITMAP** type is most basic variable type for pointing to image data

**BITMAP** type
- Almost always declared as a pointer
- Is a pointer to a struct
  - In addition to image data, the struct contains two integers, \( w \) and \( h \), that correspond to the bitmap’s width and height, respectively
- The screen is considered a **BITMAP** by Allegro
15.4 Importing Graphics and Blitting

- **Manipulating bitmaps**
  - Allegro defines several functions for manipulating `BITMAP*` objects
  - Most important: `load_bitmap` and `destroy_bitmap`

- **load_bitmap**
  - Loads image data and returns a `BITMAP*` that points to the data
  - Takes two arguments—filename of the image and a palette
    - Palette usually unnecessary; almost always passed as `NULL`
  - Returns a `BITMAP*` that points to the image data or `NULL` if the file cannot be loaded (no runtime error will occur)
15.4 Importing Graphics and Blitting

- **destroy_bitmap**
  - Removes image data from memory and frees that memory for future allocation
  - Takes a `BITMAP` as an argument
  - Once a bitmap has been destroyed, it cannot be used unless it is loaded again
  - Important to destroy all bitmaps once they are no longer needed to prevent memory leaks

- **Other bitmap manipulation functions**
  - Described on next slide
### Function prototype

<table>
<thead>
<tr>
<th>Description</th>
<th>Function Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates and returns a pointer to a blank bitmap with specified width and height (in pixels).</td>
<td><code>BITMAP *create_bitmap(int width, int height)</code></td>
</tr>
<tr>
<td>Loads and returns a pointer to a bitmap from the location specified in <code>filename</code> with palette <code>pal</code>.</td>
<td><code>BITMAP *load_bitmap(const char *filename, RGB *pal)</code></td>
</tr>
<tr>
<td>Clears a bitmap of its image data and makes it blank.</td>
<td><code>void clear_bitmap(BITMAP *bmp)</code></td>
</tr>
<tr>
<td>Clears a bitmap of its image data and makes the entire bitmap the color specified.</td>
<td><code>void clear_to_color(BITMAP *bmp, int color)</code></td>
</tr>
<tr>
<td>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.</td>
<td><code>void destroy_bitmap(BITMAP *bmp)</code></td>
</tr>
</tbody>
</table>

**Fig. 15.2** | Important BITMAP functions.
Common Programming Error 15.1

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.
/*Fig. 15.3: fig15_03.c
Displaying a bitmap on the screen. */
#include <allegro.h>

int main( void )
{
    BITMAP *bmp; /* pointer to the bitmap */

    allegro_init(); /* initialize Allegro */
    install_keyboard(); /* allow Allegro to receive keyboard input */
    set_color_depth( 16 ); /* set the color depth to 16-bit*/
    set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
    bmp = load_bitmap( "picture.bmp", NULL ); /* load the bitmap file */
    blit( bmp, screen, 0, 0, 0, 0, bmp->w, bmp->h ); /* draw the bitmap */
    readkey(); /* wait for a keypress */
    destroy_bitmap( bmp ); /* free the memory allocated to bmp */
    return 0;
} /* end function main */
END_OF_MAIN() /* Allegro-specific macro */
15.4 Importing Graphics and Blitting

- **Setting up graphics mode**
  - Before any graphics can be displayed, Allegro must set the graphics mode
  - Performed with two functions: `set_color_depth` and `set_gfx_mode`

- `set_color_depth`
  - Must be called before `set_gfx_mode`
  - Takes an integer as an argument
  - Sets color depth of the program
    - Color depth specifies how many bits of memory are used by the program to store the color of one pixel
  - Color depth can be set to 8-, 15-, 16-, 24-, or 32-bit
  - 8-bit not recommended as it complicates the image-loading process
15.4 Importing Graphics and Blitting

- **set_gfx_mode**
  - Sets graphics mode of the program
  - Takes five arguments, all integers
  - First argument specifies the graphics card driver Allegro should use for graphics
  - Should be passed a symbolic constant defined by Allegro
  - These constants are known as the graphics “magic drivers”
    - `GFX_AUTODETECT_FULLSCREEN` sets program to fullscreen mode
    - `GFX_AUTODETECT_WINDOWED` sets program to windowed mode
    - `GFX_AUTODETECT` tells program to try fullscreen mode and then windowed mode if fullscreen fails
    - `GFX_SAFE` is identical to `GFX_AUTODETECT`, but if both fullscreen and windowed mode fail to work, will set program to a low-quality “safe” graphics mode
15.4 Importing Graphics and Blitting

- **set_gfx_mode**
  - Second and third arguments determine width and height (in pixels) of graphics mode, respectively
  - Last two arguments determine minimum size of the “virtual screen”—usually set to 0
    - In current version of Allegro, virtual screen has no effect on the program, so these arguments can essentially be ignored
  - Returns 0 if graphics mode is set successfully, or a non-zero value otherwise
15.4 Importing Graphics and Blitting

- **blit**
  - Stands for “BLock Transfer”
  - Most important graphics function
  - Takes a rectangular block of one bitmap and draws it onto another
  - Takes eight arguments—two `BITMAP*`s and six integers
  - First argument specifies the bitmap from which the block will be taken
  - Second argument specifies the bitmap onto which the block will be drawn
    - To specify the screen, use the symbolic constant `screen`
15.4 Importing Graphics and Blitting

**blit**

- Third and fourth arguments specify the x- and y-coordinates of the top-left corner of the block to be taken from the source bitmap
- Fifth and sixth arguments specify the x- and y-coordinates on the destination bitmap onto which the top-left corner of the block will be drawn
- Seventh and eighth arguments specify the width and height, respectively, of the block to be taken from the source bitmap
- Note that in Allegro, the coordinates (0, 0) represent the top left corner of the screen or bitmap
  - A larger y-coordinate means further down on the screen, not up
Fig. 15.4 | Allegro’s coordinate system.
Software Engineering Observation 15.1

Avoid using the GFX_SAFE "magic driver" if possible. The "safe" graphics modes generally have a negative impact on your program’s appearance.
Common Programming Error 15.2

Loading a bitmap before setting the color depth and graphics mode of a program will likely result in Allegro storing the bitmap incorrectly.
Error-Prevention Tip 15.1

Use the `destroy_bitmap` function to free up the memory of a bitmap that is no longer needed and prevent memory leaks.
Common Programming Error 15.3

Trying to destroy a bitmap that has not been initialized causes a runtime error.
15.5 Animation with Double Buffering

- **Animation**
  - Very simple in Allegro
  - Draw one “frame” of animation, then clear the screen and draw next “frame”

- **Pong**
  - At this point we start programming Pong
  - Our “ball” will only travel in four directions—up-left, up-right, down-left, and down-right
/* Fig. 15.5: fig15_05.c  
Creating the bouncing ball. */
#include <allegro.h>

/* symbolic constants for the ball’s possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#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 */
    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! */

Outline

fig15_05.c
(2 of 4)

SCREEN_W and SCREEN_H correspond to the width and height of the screen
clear_to_color function clears the entire screen to white
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;
        case UP_LEFT:
            --ball_x; /* move the ball to the left */
            --ball_y; /* move the ball up */
            break;
    } /* end switch */

    /* make sure the ball doesn't go off the screen */
    /* if the ball is going off the top or bottom... */
    if (ball_y <= 30 || ball_y >= 440) {
        reverseVerticalDirection(); /* make it go the other way */
    }
} /* end moveBall */
/* if the ball is going off the left or right... */
if ( ball_x <= 0 || ball_x >= 600 )
    reverseHorizontalDirection(); /* make it go the other way */
} /* 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 */
} /* end function reverseVerticalDirection */

void reverseHorizontalDirection() /* reverses the horizontal direction */
{
    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
} /* end function reverseHorizontalDirection */
### 15.5 Animation with Double Buffering

- **while(!key[KEY_ESC])**
  - Allegro defines the `key` array that stores the state of each key on the keyboard.
  - `key[KEY_ESC]` corresponds to the state of the Esc key.
  - Program will continue while Esc is not being pressed.

- **makecol**
  - Returns an integer that Allegro interprets as a color.
  - Takes three integers—a red intensity, a green intensity, and a blue intensity.
  - Each intensity can vary from 0 (none) to 255 (maximum).
<table>
<thead>
<tr>
<th>Color</th>
<th>Red value</th>
<th>Green value</th>
<th>Blue value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Green</td>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>Orange</td>
<td>255</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td>Pink</td>
<td>255</td>
<td>175</td>
<td>175</td>
</tr>
<tr>
<td>Cyan</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Magenta</td>
<td>255</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>Yellow</td>
<td>255</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>Black</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>White</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Gray</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Light gray</td>
<td>192</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>Dark gray</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

**Fig. 15.6** | The red, green, and blue intensities of common colors in Allegro.
15.5 Animation with Double Buffering

- **Double Buffering**
  - In previous program, the ball often appears to flicker due to the screen constantly clearing itself to white
  - Double buffering removes this flicker
  - Uses an intermediary bitmap called the “buffer” that is the size of the screen
  - Anything meant to be drawn on the screen is drawn on the buffer instead
  - Once everything is on the buffer, the program then draws the buffer *over* the entirety of the screen
  - The buffer bitmap is cleared after it is drawn on the screen
/* Fig. 15.7: fig15_07.c  
   Using double buffering. */
#include <allegro.h>

/* symbolic constants for the ball’s possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#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 */
BITMAP *buffer; /* pointer to the buffer */

buffer bitmap is defined as a global variable
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 */
    buffer = create_bitmap( SCREEN_W, SCREEN_H ); /* create buffer */
    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 */
        /* now perform double buffering */
        clear_to_color( buffer, makecol( 255, 255, 255 ) );
        blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
        blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
        clear_bitmap( buffer );
    } /* end while */

    destroy_bitmap( ball ); /* destroy the ball bitmap */
    destroy_bitmap( buffer ); /* destroy the buffer bitmap */
    return 0;
} /* end function main */

END_OF_MAIN()
```c
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;
    case UP_LEFT:
        --ball_x; /* move the ball to the left */
        --ball_y; /* move the ball up */
        break;
    } /* end switch */

    /* make sure the ball doesn’t go off the 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 */
} /* end function reverseVerticalDirection */

void reverseHorizontalDirection() /* reverses the horizontal direction */
{
    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
} /* end function reverseHorizontalDirection */
15.6 Importing and Playing Sounds

- **Importing Sounds**
  - Importing sounds is done in a way similar to that of importing images
  - Main Allegro variable type for storing sound data is type `SAMPLE*`
    - Short for “digital sample”

- **load_sample**
  - As `load_bitmap` loads bitmaps, `load_sample` loads sounds
  - Takes one argument—the filename of the sound
  - Returns a `SAMPLE*` or `NULL` if the sound cannot be loaded
15.6 Importing and Playing Sounds

- `play_sample`
  - Plays a `SAMPLE` that has been loaded into the program
  - Takes five arguments—one `SAMPLE` and four integers
  - First argument is the sample to be played
  - Second argument specifies the volume of the sound
    - Can vary from 0 (mute) to 255 (max)
  - Third argument specifies sound’s pan position
    - Can vary from 0 (only left speaker) to 255 (only right speaker)—128 plays the sound out of both speakers equally
  - Fourth argument specifies sound’s frequency and pitch
    - A value of 1000 will play the sound at its normal frequency and pitch—greater and lesser values will raise and lower them
  - Last argument specifies if sound will loop
    - Sound will loop indefinitely if value is non-zero
15.6 Importing and Playing Sounds

- **destroy_sample**
  - Destroys a sample and frees its memory for later use
  - Will immediately stop the sample if it is playing
  - As with bitmaps, samples should be destroyed once they are no longer needed to prevent memory leaks
  - Takes a `SAMPLE` as an argument

- **Other sample manipulation functions**
  - Described on next slide
<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLE *load_sample(const char *filename)</td>
<td>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.</td>
</tr>
<tr>
<td>int play_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop)</td>
<td>Plays the specified sample at the specified volume, pan position, and frequency. The sample will loop continuously if loop is non-zero.</td>
</tr>
<tr>
<td>void adjust_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop)</td>
<td>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.</td>
</tr>
<tr>
<td>void stop_sample(const SAMPLE *spl)</td>
<td>Stops a sample that is currently playing.</td>
</tr>
<tr>
<td>void destroy_sample(SAMPLE *spl)</td>
<td>Destroys a sample and frees the memory allocated to it. If the sample is currently playing or looping, it will stop immediately.</td>
</tr>
</tbody>
</table>

**Fig. 15.8** | Important SAMPLE functions.
15.6 Importing and Playing Sounds

- **install_sound**
  - Must be called before any sounds can be played
  - Takes three arguments—two integers and one string
  - First two arguments specify what sound card drivers Allegro should use to play sounds
    - Should be passed the “magic drivers” `DIGI_AUTODETECT` and `MIDI_AUTODETECT`
  - Third argument is obsolete in current version of Allegro; should be passed `NULL`
    - Originally loaded a `.cfg` file that told Allegro how sounds should be played
/* Fig. 15.9: fig15_09.c */

#include <allegro.h>

/* symbolic constants for the ball’s possible directions */
define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#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 ball’s image bitmap */
BITMAP *buffer; /* pointer to the buffer */
SAMPLE *boing; /* pointer to sound file */

boing sample is defined as a global variable
# int main( void )

/* first, set up Allegro and the graphics mode */

allegro_init(); /* initialize Allegro */
install_keyboard(); /* install the keyboard for Allegro to use */
install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
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 */
buffer = create_bitmap( SCREEN_W, SCREEN_H );/* create buffer */
boing = load_sample( "boing.wav" ); /* load the sound file */
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 */
    /* now perform double buffering */
    clear_to_color( buffer, makecol( 255, 255, 255 ) );
    blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
    blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
    clear_bitmap( buffer );
} /* end while loop */
destroy_bitmap(ball); /* destroy the ball bitmap */
destroy_bitmap(buffer); /* destroy the buffer bitmap */
destroy_sample(boing); /* destroy the boing sound file */
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;
    case UP_LEFT:
      --ball_x; /* move the ball to the left */
      --ball_y; /* move the ball up */
      break;
  } /* end switch */
} /* end function moveBall */
/* 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();
}

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 */
}

Sample is played when the ball hits a wall
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 */

Sample is played when the ball hits a wall
15.7 Keyboard Input

- **Keyboard Input**
  - For a game to be called a game, the user must be able to interact with it somehow

- **install_keyboard**
  - Allows Allegro to receive and understand keyboard input
  - Takes no arguments
  - Must be called before keyboard input can be used in a program
15.7 Keyboard Input

- **key array**
  - Array of integers that stores the state of each key on the keyboard
  - Each key has a specific index in the array
  - If a key is not being pressed, its value in the array will be 0; otherwise, it will be non-zero

- **Keyboard symbolic constants**
  - Allegro defines a symbolic constant for each key that corresponds to its index in the key array
    - The constant for the A key is `KEY_A`
    - The constant for the spacebar is `KEY_SPACE`
  - For example, the value of `key[KEY_SPACE]` will be 0 if the spacebar is not being pressed, and non-zero if it is
  - if statements can be used to check if keys are being pressed
/* Fig. 15.10: fig15_10.c */

#include <allegro.h>

/* symbolic constants for the ball’s possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#define UP_LEFT 3

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

int ball_x; /* the ball’s x-coordinate */
int ball_y; /* the ball’s y-coordinate */
int barL_y; /* y-coordinate of the left paddle */
int barR_y; /* y-coordinate of the right paddle */
int direction; /* the ball’s direction */
BITMAP *ball; /* pointer to ball’s image bitmap */
BITMAP *bar; /* pointer to paddle’s image bitmap */
BITMAP *buffer; /* pointer to the buffer */
SAMPLE *boing; /* pointer to sound file */
int main( void )
{
    /* first, set up Allegro and the graphics mode */
    allegro_init(); /* initialize Allegro */
    install_keyboard(); /* install the keyboard for Allegro to use */
    install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
    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 */
    bar = load_bitmap( "bar.bmp", NULL ); /* load the bar bitmap */
    buffer = create_bitmap(SCREEN_W, SCREEN_H); /* create buffer */
    boing = load_sample( "boing.wav" ); /* load the sound file */
    ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
    ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
    barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
    barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
    srand( time( NULL ) ); /* seed the random function ... */
    direction = rand() % 4; /* and then make a random initial direction */
}

The paddle’s image is loaded

The two paddles are then given their initial coordinates
while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
{
    moveBall(); /* move the ball */
    respondToKeyboard(); /* respond to keyboard input */
    /* now perform double buffering */
    clear_to_color( buffer, makecol( 255, 255, 255 ) );
    blit( ball, buffer, 0, 0, ball_x, ball_y, ball_w, ball_h );
    blit( bar, buffer, 0, 0, barL_y, bar_w, bar_h );
    blit( bar, buffer, 0, 620, barR_y, bar_w, bar_h );
    blit( buffer, screen, 0, 0, 0, 0, buffer_w, buffer_h );
    clear_bitmap( buffer );
} /* end while */

destroy_bitmap( ball ); /* destroy the ball bitmap */
destroy_bitmap( bar ); /* destroy the bar bitmap */
destroy_bitmap( buffer ); /* destroy the buffer bitmap */
destroy_sample( boing ); /* destroy the boing sound file */
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;
    case UP_LEFT:
        --ball_x; /* move the ball to the left */
        --ball_y; /* move the ball up */
        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 */
```c
void respondToKeyboard() /* responds to keyboard input */
{
    if ( key[KEY_A] ) /* if A is being pressed... */
        barL_y -= 3; /* ... move the left paddle up */
    if ( key[KEY_Z] ) /* if Z is being pressed... */
        barL_y += 3; /* ... move the left paddle down */

    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
        barR_y -= 3; /* ... move the right paddle up */
    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
        barR_y += 3; /* ... move the right paddle down */

    /* make sure the paddles don't go offscreen */
    if ( barL_y < 30 ) /* if left paddle is going off the top */
        barL_y = 30;
    else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
        barL_y = 380;
    if ( barR_y < 30 ) /* if right paddle is going off the top */
        barR_y = 30;
    else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
        barR_y = 380;
}
/* end function respondToKeyboard */
```

**Outline**

**respondToKeyboard** function checks if various keys are being pressed and performs appropriate actions

(5 of 7)
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 */
Outline

fig15_10.c

(7 of 7)
15.8 Fonts and Displaying Text

- Displaying Text
  - In all games, it is necessary for the game to communicate with the user in some way

- Fonts
  - Allegro can display text on the screen, but it must be told in which font the text should be displayed
  - As with bitmaps and samples, Allegro defines a `FONT*` type that points to font data in memory

- `font` symbolic constant
  - The symbolic constant `font` corresponds to Allegro’s default font—can be used in place of any `FONT*` variable
15.8 Fonts and Displaying Text

- **load_font**
  - Loads a font file into memory
  - Takes two arguments—a file name and a palette
    - As with bitmaps, the palette is usually passed as NULL
  - Returns a `FONT*` or `NULL` if the font cannot be loaded

- **destroy_font**
  - Destroys a font and frees its memory for later use
  - Takes a `FONT*` as an argument
  - Remember to destroy fonts once they are no longer needed
15.8 Fonts and Displaying Text

- **Displaying Text**
  - Once a font has been loaded, one must use the `textprintf_ex` function to display the text on the screen

- **textprintf_ex**
  - Displays a string on the screen in the specified font
  - Takes at least seven arguments—a BITMAP*, a FONT*, four integers, and a format control string
  - First argument specifies the bitmap on which the text should be drawn
  - Second argument specifies the font in which the text should be drawn
15.8 Fonts and Displaying Text

- `textprintf_ex`
  - Third and fourth arguments specify the \( x \)- and \( y \)-coordinates at which the text should begin
  - Fifth and sixth arguments specify the foreground and background colors of the text being printed
    - Use `makecol` function to determine the correct values
    - Use a value of \(-1\) to specify a transparent color
  - Seventh argument specifies the string to be printed
    - This argument is a format control string, so conversion specifiers can be placed in it
    - If any conversion specifiers are present, arguments should be added following the string that specify their values
### Function prototype

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

**Fig. 15.11** | Functions that are useful for drawing text onto a bitmap.
/* Fig. 15.12: fig15_12.c
   Displaying text on the screen. */
#include <allegro.h>

/* symbolic constants for the ball’s possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#define UP_LEFT 3

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

int ball_x; /* the ball’s x-coordinate */
int ball_y; /* the ball’s y-coordinate */
int barL_y; /* y-coordinate of the left paddle */
int barR_y; /* y-coordinate of the right paddle */
int scoreL; /* score of the left player */
int scoreR; /* score of the right player */
int direction; /* the ball’s direction */
BITMAP *ball; /* pointer to ball’s image bitmap */
BITMAP *bar; /* pointer to paddle’s image bitmap */
BITMAP *buffer; /* pointer to the buffer */
SAMPLE *boing; /* pointer to sound file */
FONT *pongFont; /* pointer to font file */
int main( void )
{
    /* first, set up Allegro and the graphics mode */
allegro_init(); /* initialize Allegro */
install_keyboard(); /* install the keyboard for Allegro to use */
install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
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 */
bar = load_bitmap( "bar.bmp", NULL); /* load the bar bitmap */
buffer = create_bitmap(SCREEN_W, SCREEN_H); /* create buffer */
boing = load_sample( "boing.wav" ); /* load the sound file */
pongFont = load_font( "pongfont.pcx", NULL, NULL ); /* load the font */
ball_x = SCREEN_W/ 2; /* give the ball its initial x-coordinate */
bball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
scoreL = 0; /* set left player’s score to 0 */
scoreR = 0; /* set right player’s score to 0 */
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 */
    respondToKeyboard(); /* respond to keyboard input */
    /* now, perform double buffering */
    clear_to_color( buffer, makecol( 255, 255, 255 ) );
    blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
    blit( bar, buffer, 0, 0, barL_y, bar->w, bar->h );
    blit( bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h );
    /* draw text onto the buffer */
    textprintf_ex( buffer, pongFont, 75, 0, makecol( 0, 0, 0 ),
                   -1, "Left Player Score: %d", scoreL );
    textprintf_ex( buffer, pongFont, 400, 0, makecol( 0, 0, 0 ),
                   -1, "Right Player Score: %d", scoreR );
    blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
    clear_bitmap( buffer );
} /* end while */

destroy_bitmap( ball ); /* destroy the ball bitmap */
destroy_bitmap( bar ); /* destroy the bar bitmap */
destroy_bitmap( buffer ); /* destroy the buffer bitmap */
destroy_sample( boing ); /* destroy the boing sound file */
destroy_font( pongFont ); /* destroy the font */
return 0;
} /* end function main */
END_OF_MAIN() /* don’t forget this! */
```c
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;
    case UP_LEFT:
        --ball_x; /* move the ball to the left */
        --ball_y; /* move the ball up */
        break;
    } /* end switch */

    /* make sure the ball doesn't go off the screen */
    if (ball_y <= 30 || ball_y >= 440)
        reverseVerticalDirection();
} /* end moveBall */
```

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

void respondToKeyboard() /* responds to keyboard input */
{
    if ( key[KEY_A] ) /* if A is being pressed... */
        barL_y -= 3; /* ... move the left paddle up */
    if ( key[KEY_Z] ) /* if Z is being pressed... */
        barL_y += 3; /* ... move the left paddle down */

    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
        barR_y -= 3; /* ... move the right paddle up */
    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
        barR_y += 3; /* ... move the right paddle down */

    /* make sure the paddles don’t go offscreen */
    if ( barL_y < 30 ) /* if left paddle is going off the top */
        barL_y = 30;
    else if ( barL_y > 380 ) /* if left paddle is going off the bottom*/
        barL_y = 380;
    if ( barR_y < 30 ) /* if right paddle is going off the top */
        barR_y = 30;
    else if ( barR_y > 380 ) /* if right paddle is going off the bottom*/
        barR_y = 380;
} /* end function respondToKeyboard */
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 */
15.9 Implementing the Game of Pong

- **Unresolved Issues**
  - There are two issues we have to resolve before our Pong game can be considered complete
    - Making the ball bounce off the paddles
    - Creating a boundary between the scoreboard and the game

- **Making the ball bounce off the paddles**
  - In our current Pong game, the paddles do not stop the ball
  - We must make the ball reverse its direction if it hits a paddle
  - Since we know the dimensions of the ball and paddle, we can use `if` statements to determine if they are touching
15.9 Implementing the Game of Pong

- Creating a boundary
  - In our current Pong game, there is a boundary between the scoreboard and the game, but it is not visible
  - We use Allegro’s `line` function to create this boundary

- `line` function
  - Draws a line onto a bitmap
  - Takes six arguments—a `BITMAP*` and five integers
  - First argument specifies the bitmap onto which the line should be drawn
  - Second and third arguments specify the x- and y-coordinates of the point where the line starts
  - Fourth and fifth arguments specify the x- and y-coordinates of the point where the line ends
  - Sixth argument specifies the line’s color—use `makecol`
/* Fig. 15.13: fig15_13.c
Finishing up the Pong game. */

#include <allegro.h>

/* symbolic constants for the ball's possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#define UP_LEFT 3

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

int ball_x; /* the ball's x-coordinate */
int ball_y; /* the ball's y-coordinate */
int barL_y; /* y-coordinate of the left paddle */
int barR_y; /* y-coordinate of the right paddle */
int scoreL; /* score of the left player */
int scoreR; /* score of the right player */
int direction; /* the ball's direction */
BITMAP *ball; /* pointer to ball's image bitmap */
BITMAP *bar; /* pointer to paddle's image bitmap */
BITMAP *buffer; /* pointer to buffer */
SAMPLE *boing; /* pointer to sound file */
FONT *pongFont; /* pointer to font file */
int main( void )
{
    /* first, set up Allegro and the graphics mode */
    allegro_init(); /* initialize Allegro */
    install_keyboard(); /* install the keyboard for Allegro to use */
    install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
    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 */
    bar = load_bitmap( "bar.bmp", NULL); /* load the bar bitmap */
    buffer = create_bitmap(SCREEN_W, SCREEN_H); /* create buffer */
    boing = load_sample( "boing.wav" ); /* load the sound file */
    pongFont = load_font( "pongfont.pcx", NULL, NULL ); /* load the font */
    ball_x = SCREEN_W / 2; /* give ball its initial x-coordinate */
    ball_y = SCREEN_H / 2; /* give ball its initial y-coordinate */
    barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
    barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
    scoreL = 0; /* set left player’s score to 0 */
    scoreR = 0; /* set right player’s score to 0 */
    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 */
    respondToKeyboard(); /* respond to keyboard input */
    /* now, perform double buffering */
    clear_to_color(buffer, makecol(255, 255, 255));
    blit(ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h);
    blit(bar, buffer, 0, 0, barL_y, bar->w, bar->h);
    blit(bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h);
    line(buffer, 0, 30, 640, 30, makecol(0, 0, 0));
    /* draw text onto the buffer */
    textprintf_ex(buffer, pongFont, 75, 0, makecol(0, 0, 0),
                  -1, "Left Player Score: %d", scoreL);
    textprintf_ex(buffer, pongFont, 400, 0, makecol(0, 0, 0),
                  -1, "Right Player Score: %d", scoreR);
    blit(buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h);
    /* end while */
}

destroy_bitmap(ball); /* destroy the ball bitmap */
destroy_bitmap(bar); /* destroy the bar bitmap */
destroy_bitmap(buffer); /* destroy the buffer bitmap */
destroy_sample(boing); /* destroy the boing sound file */
destroy_font(pongFont); /* destroy the font */
return 0;
} /* end function main */
END_OF_MAIN() /* don’t forget this! */
```c
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;
        case UP_LEFT:
            --ball_x; /* move the ball to the left */
            --ball_y; /* move the ball up */
            break;
    } /* end switch */

    /* if the ball is going off the top or bottom... */
    if ( ball_y <= 30 || ball_y >= 440 )
        reverseVerticalDirection(); /* make it go the other way */
```
/* if the ball is in range of the left paddle ... */
if (ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT))
{
    /* is the left paddle in the way? */
    if (ball_y > (barL_y - 39) && ball_y < (barL_y + 99))
        reverseHorizontalDirection();
    else if (ball_x <= -20) { /* if the ball goes off the screen */
        ++scoreR; /* give right player a point */
        ball_x = SCREEN_W / 2; /* place the ball in the ... */
        ball_y = SCREEN_H / 2; /* ... center of the screen */
        direction = rand() % 4; /* give the ball a random direction */
    } /* end else */
} /* end if */

/* if the ball is in range of the right paddle ... */
if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
{
    /* is the right paddle in the way? */
    if (ball_y > (barR_y - 39) && ball_y < (barR_y + 99))
        reverseHorizontalDirection();
    else if (ball_x >= 620) { /* if the ball goes off the screen */
        ++scoreL; /* give left player a point */
        ball_x = SCREEN_W / 2; /* place the ball in the ... */
        ball_y = SCREEN_H / 2; /* ... center of the screen */
        direction = rand() % 4; /* give the ball a random direction */
    } /* end else */
} /* end if */

/* end function moveBall */

If the ball is moving off the side of the screen, the program checks if the paddle is in the way. If it is, the ball bounces; if not, the player on the other side gets a point and the ball is placed in the center of the screen.
void respondToKeyboard() /* responds to keyboard input */
{
    if ( key[KEY_A] ) /* if A is being pressed... */
        barL_y -= 3; /* ... move the left paddle up */
    if ( key[KEY_Z] ) /* if Z is being pressed... */
        barL_y += 3; /* ... move the left paddle down */

    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
        barR_y -= 3; /* ... move the right paddle up */
    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
        barR_y += 3; /* ... move the right paddle down */

    /* make sure the paddles don't go offscreen */
    if ( barL_y < 30 ) /* if left paddle is going off the top */
        barL_y = 30;
    else if ( barL_y > 380 ) /* if left paddle is going off the bottom*/
        barL_y = 380;
    if ( barR_y < 30 ) /* if right paddle is going off the top */
        barR_y = 30;
    else if ( barR_y > 380 ) /* if right paddle is going off the bottom*/
        barR_y = 380;
} /* end function respondToKeyboard */
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 */
Timers

- In our current Pong game, there is nothing regulating how quickly the game runs
  - On very fast systems the game may run too quickly to be playable
- Allegro’s timers allow us to control how often certain functions are called and how quickly our game runs

install_timer

- Must be called before any timers can be used
- Takes no arguments
- Can be cancelled by calling remove_timer function, which will remove all timers that are running
15.10 Timers in Allegro

- **install_int**
  - Installs a timer that calls a specified function at regular intervals
  - Takes two arguments—a function pointer and an integer
  - First argument specifies the function to be called
  - Second argument specifies the interval (in milliseconds) at which the function should be called
  - Allegro can have up to 16 timers running at once
  - Returns 0 if the timer is installed successfully, or a non-zero value if the function fails
Timers are not variables

- Unlike bitmaps, sounds, and fonts, timers are not stored in variables
- Once a timer is installed it will run in the background
- Allegro identifies each timer by the function it calls

remove_int

- Removes a timer previously installed by install_int
- Takes one argument—the function called by the timer to be removed
15.10 Timers in Allegro

- **Other notes**
  - Any variable that can be modified in a function called by a timer must be given the `volatile` qualifier
    - Because of the way timers are programmed, some compilers may not understand that a variable can be changed by a timer, and may optimize the code at compile time in a way that removes the variable’s modification
  - On systems running DOS or Mac OS 9 and below, the memory of variables and functions used by timers must be “locked” for the timers to work correctly
    - Not necessary on current systems
    - Detailed instructions on Allegro website
/* Fig. 15.14: fig15_14.c */
#include <allegro.h>

/* symbolic constants for the ball's possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#define UP_LEFT 3

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

volatile int ball_x; /* the ball's x-coordinate */
volatile int ball_y; /* the ball's y-coordinate */
volatile int barL_y; /* y-coordinate of the left paddle */
volatile int barR_y; /* y-coordinate of the right paddle */
volatile int scoreL; /* score of the left player */
volatile int scoreR; /* score of the right player */
volatile int direction; /* the ball's direction */
BITMAP *ball; /* pointer to ball's image bitmap */
BITMAP *bar; /* pointer to paddle's image bitmap */
BITMAP *buffer; /* pointer to buffer */
SAMPLE *boing; /* pointer to sound file */
FONT *pongFont; /* pointer to font file */
```c
int main( void )
{
    /* first, set up Allegro and the graphics mode */
    allegro_init(); /* initialize Allegro */
    install_keyboard(); /* install the keyboard for Allegro to use */
    install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
    install_timer(); /* install the timer handler */
    set_color_depth( 16 ); /* set the color depth to 16-bit */
    setgfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
    ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
    bar = load_bitmap( "bar.bmp", NULL ); /* load the bar bitmap */
    buffer = create_bitmap(SCREEN_W, SCREEN_H); /* create buffer */
    boing = load_sample( "boing.wav" ); /* load the sound file */
    pongFont = load_font( "pongfont.pcx", NULL, NULL ); /* load the font */
    ball_x = SCREEN_W / 2; /* give ball its initial x-coordinate */
    ball_y = SCREEN_H / 2; /* give ball its initial y-coordinate */
    barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
    barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
    scoreL = 0; /* set left player’s score to 0 */
    scoreR = 0; /* set right player’s score to 0 */
    srand( time( NULL ) ); /* seed the random function ... */
    direction = rand() % 4; /* and then make a random initial direction */
    /* add timer that calls moveBall every 5 milliseconds */
    install_int( moveBall, 5 );
    /* add timer that calls respondToKeyboard every 10 milliseconds */
    install_int( respondToKeyboard, 10 );
}
```

**Outline**

**fig15_14.c**

(2 of 7)

install_timer function must be called before timers can be used

moveBall function will be called every 5 milliseconds

**respondToKeyboard** function will be called every 10 milliseconds
while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
{
    /* now perform double buffering */
clear_to_color( buffer, ( 255, 255, 255 ) );
    blit( ball, buffer, 0, 0, ball_x, ball_y, ball->w, ball->h );
    blit( bar, buffer, 0, 0, barL_y, bar->w, bar->h );
    blit( bar, buffer, 0, 0, 620, barR_y, bar->w, bar->h );
    line( buffer, 0, 30, 640, 30, makecol( 0, 0, 0 ) );
    /* draw text onto the buffer */
textprintf_ex( buffer, pongFont, 75, 0, makecol( 0, 0, 0 ),
                    -1, "Left Player Score: %d", scoreL );
textprintf_ex( buffer, pongFont, 400, 0, makecol( 0, 0, 0 ),
                    -1, "Right Player Score: %d", scoreR );
    blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
    clear_bitmap( buffer );
} /* end while */

remove_int( moveBall ); /* remove moveBall timer */
remove_int( respondToKeyboard ); /* remove respondToKeyboard timer */
destroy_bitmap( ball ); /* destroy the ball bitmap */
destroy_bitmap( bar ); /* destroy the bar bitmap */
destroy_bitmap( buffer ); /* destroy the buffer bitmap */
destroy_sample( boing ); /* destroy the boing sound file */
destroy_font( pongFont ); /* destroy the font */
return 0;
} /* end function main */
END_OF_MAIN() /* don't forget this! */
```c
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;
    case UP_LEFT:
        --ball_x; /* move the ball to the left */
        --ball_y; /* move the ball up */
        break;
    }
    /* end switch */

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

    /* if the ball is in range of the left paddle... */
    if (ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT))
```
/* is the left paddle in the way? */
if (ball_y > (barL_y - 39) && ball_y < (barL_y + 99))
    reverseHorizontalDirection();
else if (ball_x <= -20) { /* if the ball goes off the screen */
    ++scoreR; /* give right player a point */
    ball_x = SCREEN_W/2; /* place the ball in the ... */
    ball_y = SCREEN_H/2; /* ... center of the screen */
    direction = rand() % 4; /* give the ball a random direction */
} /* end else */
else if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
    /* is the right paddle in the way? */
if (ball_y > (barR_y - 39) && ball_y < (barR_y + 99))
    reverseHorizontalDirection();
else if (ball_x >= 620) { /* if the ball goes off the screen */
    ++scoreL; /* give left player a point */
    ball_x = SCREEN_W/2; /* place the ball in the ... */
    ball_y = SCREEN_H/2; /* ... center of the screen */
    direction = rand() % 4; /* give the ball a random direction */
} /* end else */
} /* end function moveBall */

void respondToKeyboard() /* responds to keyboard input */
{  
  if ( key[KEY_A] ) /* if A is being pressed... */
    barL_y -= 3; /* ... move the left paddle up */
  if ( key[KEY_Z] ) /* if Z is being pressed... */
    barL_y += 3; /* ... move the left paddle down */
  if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
    barR_y -= 3; /* ... move the right paddle up */
  if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
    barR_y += 3; /* ... move the right paddle down */

  /* make sure the paddles don't go offscreen */
  if ( barL_y < 30 ) /* if left paddle is going off the top */
    barL_y = 30;
  else if ( barL_y > 380 ) /* if left paddle is going off the bottom*/
    barL_y = 380;
  if ( barR_y < 30 ) /* if right paddle is going off the top */
    barR_y = 30;
  else if ( barR_y > 380 ) /* if right paddle is going off the bottom*/
    barR_y = 380;
} /* end function respondToKeyboard */

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 */
15.11 The Grabber and Allegro Datafiles

Datafiles

- Every external file needed by an Allegro program must be loaded and later destroyed at program’s end
  - Simple when there is a small number of external files, but becomes very tedious when the number rises
- Allegro datafiles take data of multiple external files and store it in one place
- An Allegro program can load one datafile and gain access to the data of multiple external files
15.11 The Grabber and Allegro Datafiles

- **Grabber**
  - Datafiles are not very useful if we can’t make our own
  - The Allegro package contains a program called the grabber which is used to create datafiles
  - Executing the grabber program will cause the screen on the next slide to appear
Fig. 15.15 | Allegro’s grabber utility.
15.11 The Grabber and Allegro Datafiles

- **Four main areas of grabber window**
  - **Top area**
    - Contains information about the datafile being edited
  - **White area in bottom left**
    - Lists objects that the datafile contains
    - Currently empty as we have not loaded any external files
  - **Gray window in top right**
    - Contains information about the currently selected object
    - No object is currently selected, so it is empty
  - **Bottom right area**
    - Displays picture of currently selected object (if the object can be displayed as a picture—e.g. sounds cannot be displayed)
15.11 The Grabber and Allegro Datafiles

- Adding new objects to a datafile
  - First, we will add our ball bitmap to the datafile
  - Highlight “New” in the “Object” menu and a list of object types appears
  - Select “Bitmap”
  - Screen should now look like next slide
Fig. 15.16 | Adding a bitmap to a datafile.
15.11 The Grabber and Allegro Datafiles

- Applying data to an object
  - A new bitmap object has been created, but currently it is blank
  - To apply image data to the object, we must first read in a bitmap from an external file
  - Select “Read Bitmap” from “File” menu and import the ball.bmp file
  - Ball bitmap will appear on screen; click to return to main grabber screen
  - Next, select “Grab” from object menu
  - Screen on next slide will appear
Fig. 15.17 | Applying an imported bitmap to an object.
15.11 The Grabber and Allegro Datafiles

- **Applying image data to a bitmap object**
  - Grabber is asking what part of imported bitmap should be applied to BALL object
  - Drag a box over entire bitmap and release the mouse
  - Screen on next slide will appear
**Fig. 15.18** | A complete imported object.
15.11 The Grabber and Allegro Datafiles

- **Importing bitmaps**
  - BALL object has been successfully created
  - Repeat process with paddle bitmap to add it to the datafile as well

- **Importing other objects**
  - To add sound files or fonts to the datafile, choose the respective object from the “New” list in “Object” menu
  - Once object has been created, simply select “Grab” from object menu and select the external file to load
    - No need to “read” file as with bitmaps
  - Once all objects have been created, screen on next slide should appear
**Fig. 15.19** | The grabber window after importing all of our objects.
15.11 The Grabber and Allegro Datafiles

- **Saving the datafile**
  - Before choosing “Save” from the file menu, type `pong.h` into the “Header” field at the top of the grabber window.
  - This will make the grabber save a header file alongside the datafile.
    - The usage of this header will be explained shortly.
  - Then save the datafile in the same folder as the Pong program.
15.11 The Grabber and Allegro Datafiles

- **Loading datafiles into a program**
  - Just as Allegro defines the `BITMAP*`, `SAMPLE*`, and `FONT*` variable types to point to image, sound, and font data, it also defines the `DATAFILE*` type to point to datafile objects in memory.
  - Datafiles are loaded with `load_datafile` function, which takes a filename, and removed from memory with the `unload_datafile` function, which takes a `DATAFILE*` variable.
  - `destroy_datafile` function is not defined by Allegro.
15.11 The Grabber and Allegro Datafiles

- Accessing objects in a datafile
  - Allegro considers a `DATAF1 LE*` variable loaded into a program to be an array of objects
  - Each object’s index in the array corresponds to the order it was imported into the grabber
  - First object imported has index 0, second object has index 1, and so on
  - If there are many objects in the datafile, remembering each object’s index can be difficult
  - However, take a look at the header file we saved earlier
/* Allegro datafile object indexes, produced by grabber v4.2.0, MinGW32 */
/* Datafile: c:\Dev-Cpp\Projects\pongdatafile.dat */
/* Date: Wed Jun 21 12:57:10 2006 */
/* Do not hand edit */

#define BALL                             0        /* BMP */
#define BAR                              1        /* BMP */
#define BOING                            2        /* SAMP */
#define PONGFONT                         3        /* FONT */
15.11 The Grabber and Allegro Datafiles

- **The header file**
  - Defines a symbolic constant for each object that corresponds to that object’s index in the array
  - Each symbolic constant is the name that was given to the object when it was imported into the datafile

- **Accessing objects in the datafile**
  - If the datafile `myDatafile` has been loaded into the program, an object in it can be accessed with the code `myDatafile[i].dat`, where `i` is the object’s index
  - C considers any object accessed from a datafile to be of type `void *`, so they cannot be dereferenced
/* Fig. 15.21: fig15_21.c
   Using datafiles. */
#include <allegro.h>
#include "pong.h"

/* symbolic constants for the ball's possible directions */
#define DOWN_RIGHT 0
#define UP_RIGHT 1
#define DOWN_LEFT 2
#define UP_LEFT 3

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

volatile int ball_x; /* the ball's x-coordinate */
volatile int ball_y; /* the ball's y-coordinate */
volatile int barL_y; /* y-coordinate of the left paddle */
volatile int barR_y; /* y-coordinate of the right paddle */
volatile int scoreL; /* score of the left player */
volatile int scoreR; /* score of the right player */
volatile int direction; /* the ball's direction */
BITMAP *buffer; /* pointer to the buffer */
DATAFILE *pongData; /* pointer to the datafile */
int main( void )
{
    /* first, set up Allegro and the graphics mode */
    allegro_init(); /* initialize Allegro */
    install_keyboard(); /* install the keyboard for Allegro to use */
    install_sound( DIGI_AUTODETECT, MIDI_AUTODETECT, NULL );
    install_timer(); /* install the timer handler */
    set_color_depth( 16 ); /* set the color depth to 16-bit */
    set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
    buffer = create_bitmap( SCREEN_W, SCREEN_H ); /* create buffer */
    pongData = load_datafile( "pongdatafile.dat" ); /* load the datafile */
    ball_x = SCREEN_W / 2; /* give ball its initial x-coordinate */
    ball_y = SCREEN_H / 2; /* give ball its initial y-coordinate */
    barL_y = SCREEN_H / 2; /* give left paddle its initial y-coordinate */
    barR_y = SCREEN_H / 2; /* give right paddle its initial y-coordinate */
    scoreL = 0; /* set left player's score to 0 */
    scoreR = 0; /* set right player's score to 0 */
    srand( time( NULL ) ); /* seed the random function ... */
    direction = rand() % 4; /* and then make a random initial direction */
    /* add timer that calls moveBall every 5 milliseconds */
    install_int( moveBall, 5 ); /* add timer that calls respondToKeyboard every 10 milliseconds */
    install_int( respondToKeyboard, 10 );
}
while (!key[KEY_ESC]) /* until the escape key is pressed ... */ {
    /* now perform double buffering */
    clear_to_color(buffer, makecol(255, 255, 255));
    blit(pongData[BALL].dat, buffer, 0, 0, ball_x, ball_y, 40, 40);
    blit(pongData[BAR].dat, buffer, 0, 0, barL_y, 20, 100);
    blit(pongData[BAR].dat, buffer, 0, 0, 620, barR_y, 20, 100);
    line(buffer, 0, 30, 640, 30, makecol(0, 0, 0));
    /* draw text onto the buffer */
    textprintf_ex(buffer, pongData[PONGFONT].dat, 75, 0,
                  makecol(0, 0, 0), -1, "Left Player Score: %d", scoreL);
    textprintf_ex(buffer, pongData[PONGFONT].dat, 400, 0,
                  makecol(0, 0, 0), -1, "Right Player Score: %d", scoreR);
    blit(buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h);
    clear_bitmap(buffer);
} /* end while */

remove_int(moveBall); /* remove moveBall timer */
remove_int(respondToKeyboard); /* remove respondToKeyboard timer */
destroy_bitmap(buffer); /* destroy the buffer bitmap */
unload_datafile(pongData); /* unload the datafile */
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;
    case UP_LEFT:
        --ball_x; /* move the ball to the left */
        --ball_y; /* move the ball up */
        break;
    } /* end switch */

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

    /* if the ball is in range of the left paddle... */
    if ( ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT) )
/* is the left paddle in the way? */
if (ball_y > (barL_y - 39) && ball_y < (barL_y + 99))
    reverseHorizontalDirection();
else if (ball_x <= -20) { /* if the ball goes off the screen */
    ++scoreR; /* give right player a point */
    ball_x = SCREEN_W / 2; /* place the ball in the ... */
    ball_y = SCREEN_H / 2; /* ... center of the screen */
    direction = rand() % 4; /* give the ball a random direction */
} /* end else */
/* end if */
/* if the ball is in range of the right paddle ... */
if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
{
    /* is the right paddle in the way? */
    if (ball_y > (barR_y - 39) && ball_y < (barR_y + 99))
        reverseHorizontalDirection();
    else if (ball_x >= 620) { /* if the ball goes off the screen */
        ++scoreL; /* give left player a point */
        ball_x = SCREEN_W / 2; /* place the ball in the ... */
        ball_y = SCREEN_H / 2; /* ... center of the screen */
        direction = rand() % 4; /* give the ball a random direction */
    } /* end else */
} /* end if */
} /* end function moveBall */

void respondToKeyboard() /* responds to keyboard input */
if ( key[KEY_A] ) /* if A is being pressed... */
    barL_y -= 3; /* ... move the left paddle up */
if ( key[KEY_Z] ) /* if Z is being pressed... */
    barL_y += 3; /* ... move the left paddle down */
if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
    barR_y -= 3; /* ... move the right paddle up */
if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
    barR_y += 3; /* ... move the right paddle down */

/* make sure the paddles don't go offscreen */
if ( barL_y < 30 ) /* if left paddle is going off the top */
    barL_y = 30;
else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
    barL_y = 380;
if ( barR_y < 30 ) /* if right paddle is going off the top */
    barR_y = 30;
else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
    barR_y = 380;
} /* end function respondToKeyboard */

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( pongData[BOING].dat, 255, 128, 1000, 0 ); /* play sound */
} /* end function reverseVerticalDirection */
void reverseHorizontalDirection() /* reverses the horizontal direction */
{
    direction = (direction + 2) % 4; /* reverse horizontal direction */
    play_sample(pongData[BOING].dat, 255, 128, 1000, 0); /* play sound */
} /* end function reverseHorizontalDirection */
15.12 Other Allegro Capabilities

- Drawing primitives
  - Allegro can draw several types of simple polygons without external graphics
- Playing MIDI music files
- Playing .fli format animations in programs
- Displaying 3D graphics—very complex

Information on these capabilities in Allegro’s online manual at www.allegro.cc/manual