15

Programación de Juegos y Gráficos en C con Allegro

Referencia: C How to Program 5 ed

Deitel & Deitel, Pearson

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.

Outline

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 with Visual C 6

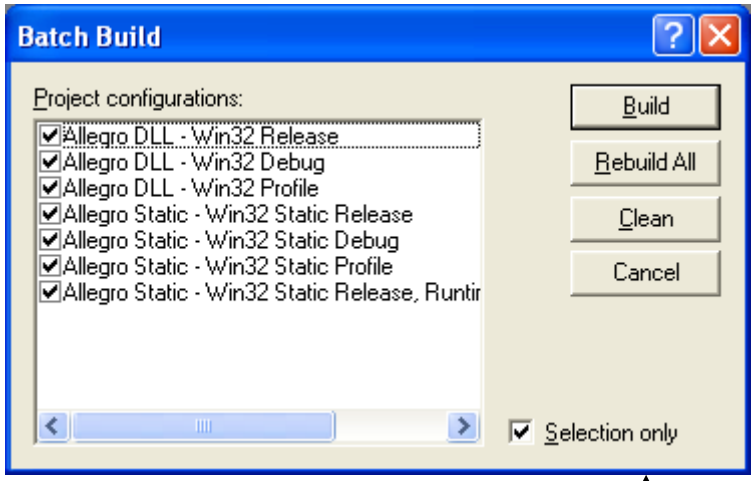
- **Allegro must be installed before it can be used with C**
  - **The installation process differs slightly for different systems**
  - **Detailed instructions in the Allegro documentation**
- **Links do download Allegro**
  - **Main page: http://alleg.sourceforge.net/**
  - **Download latest version:**

    http://alleg.sourceforge.net/wip.html
  - **Allegro Wiki with stuff information http://wiki.allegro.cc/index.php?title=Main_Page**

15.2 Building Allegro with Visual C 6

- **Requirements**
  - **Allegro 4.2.2**
  - **MSVC 6**
- **Instructions**

Allegro



- **Unzip content of all422 file to an $allegro_directory**
- **Run $allegro_directory \build\pre-build-msvc.bat**
- **Open $allegro_directory \build\msvc6\Allegro.dsw in MSVC 6**
- **Click Build / Batch Build.**
  - **Check all the versions you want to build.**
  - **Click Build. (Note: it is normal to get warnings of missing include files.)**
- **Note: Due to a bug in MSVC 6, it might be necessary to repeat Step 3 to perform the final links.**

15.2 Building Allegro with Visual C 6

- Copy the files
  - Copy the contents of $allegro_directory\include to $MVSC_directory\Microsoft Visual Studio\VC98\include
  - Copy the .lib files from $allegro_directory\lib\msvc to $MVSC_directory\Microsoft Visual Studio\VC98\lib
  - Copy the .dll files from $allegro_directory\lib\msvc to C:\Windows\System32

Demo

- Open $allegro_directory\build\msvc6\demo.dsw in MSVC 6
- Press F7 to Compile
- Press Ctrl-F5 to Run

15.2 Building Allegro with Visual C 6

Tools

- Open $allegro_directory\build\msvc6\tools.dsw in MSVC 6
- Click Build / Batch Build.
  - Check all the tools you want to build.
  - Click Build.

Note: some of the tools depend on libdat, so you should build that first.

Examples

- Open $allegro_directory\build\msvc6\examples.dsw in MSVC 6
- Click Build / Batch Build.
  - Check all the examples you want to build.
  - Click Build.

15.2 Building Allegro with Visual C 6

Tests

- **Open $allegro_directory\build\msvc6\tests.dsw in MSVC 6**
- **Click Build / Batch Build.**
  - **Check all the tools you want to build.**
  - **Click Build.**

- **Note: Due to a bug in MSVC 6, it might be necessary to repeat Build process to perform the final links for Demo, Tool, Examples and Tests.**

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**

Outline

```
1  /* Fig. 15.1: fig15_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 */
```

allegro.h header must be included

allegro_init function initializes Allegro library

fig15_01.c

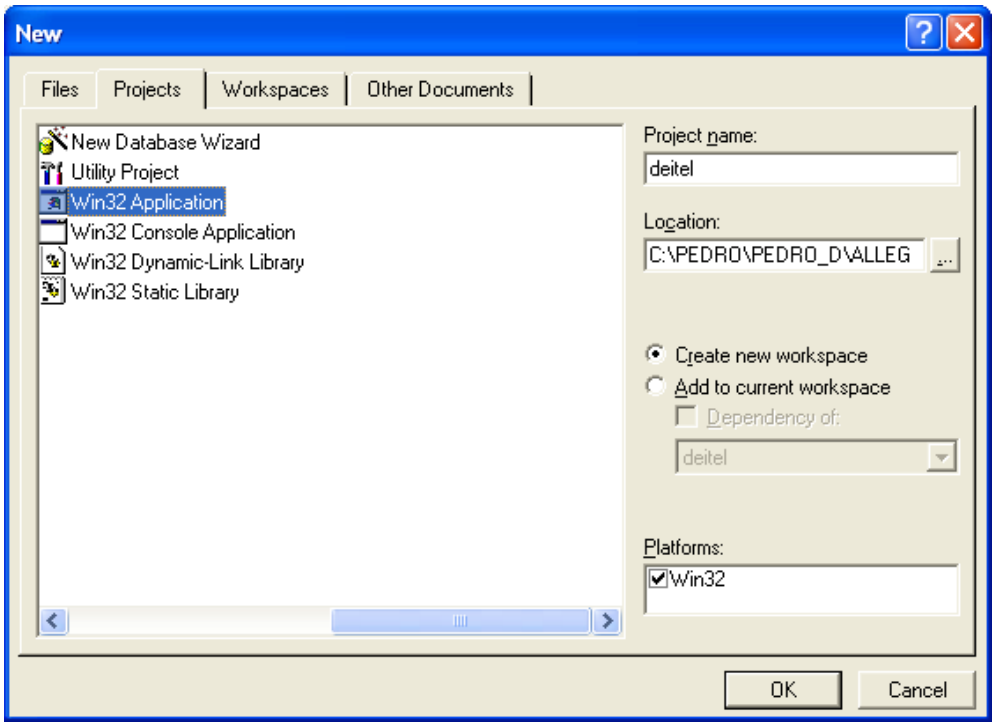
allegro_message function displays a text message in a dialog box like this

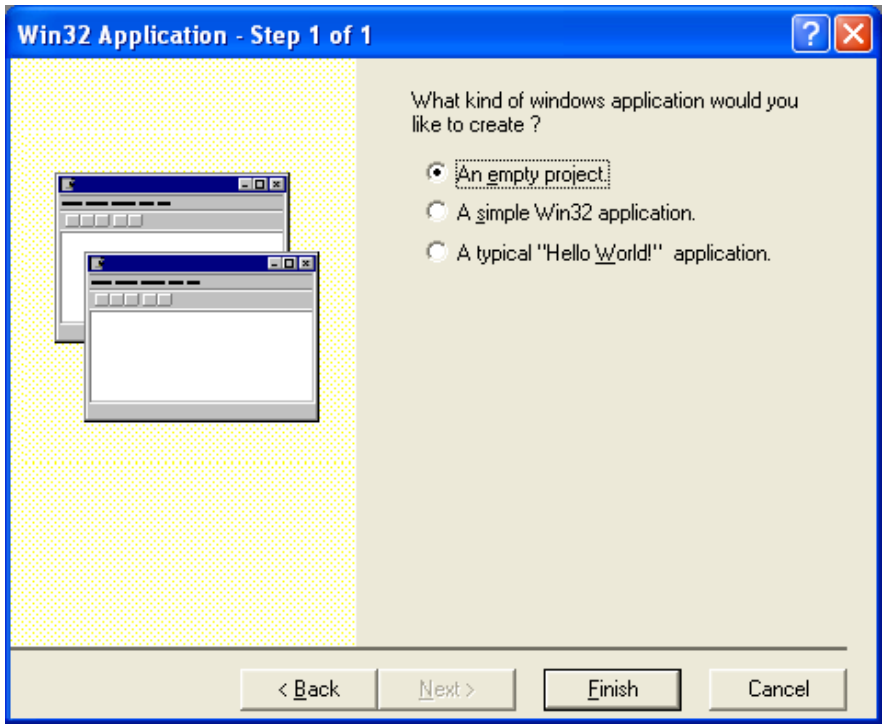
END_OF_MAIN macro ensures program will run on all systems

15.3 Compiling an Allegro Program

- Creating a Project
  - You need to have Allegro installed for MSVC 6.
  - Open Microsoft Visual Studio 6
  - Click on File / New.
  - In the tab Projects choose "Win32 Application"
  - Enter your project Name, choose a Location and hit "OK"
  - Choose "Empty project"
  - Hit "Finish"

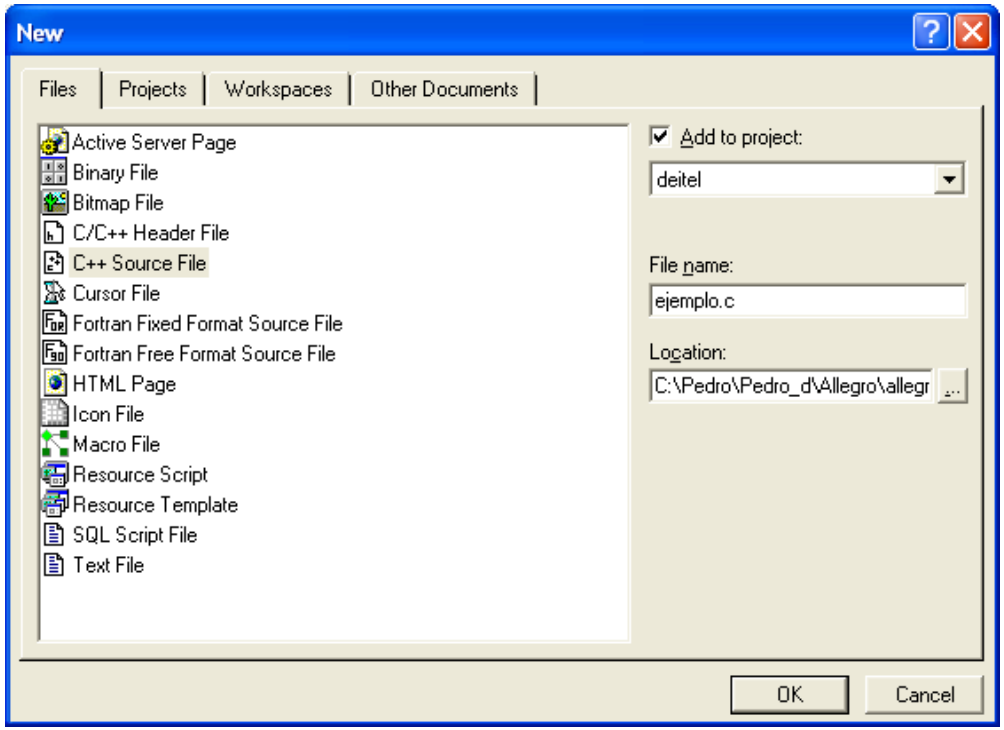
15.3 Compiling an Allegro Program





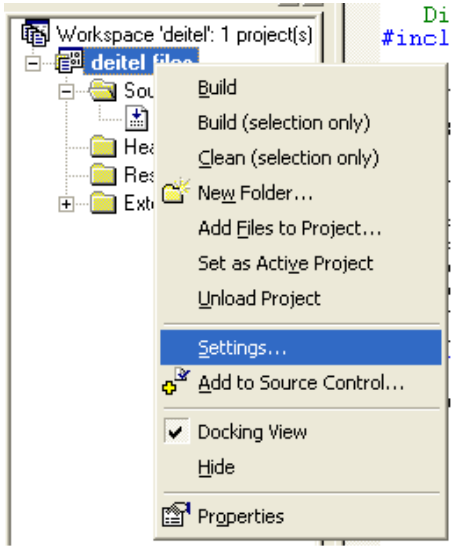
15.3 Compiling an Allegro Program

- Creating the main source file
  - Click on "File".
  - In tab Files choose "C++ Source File", enter the Name, and press "OK"
  - Enter the Allegro code.



15.3 Compiling an Allegro Program

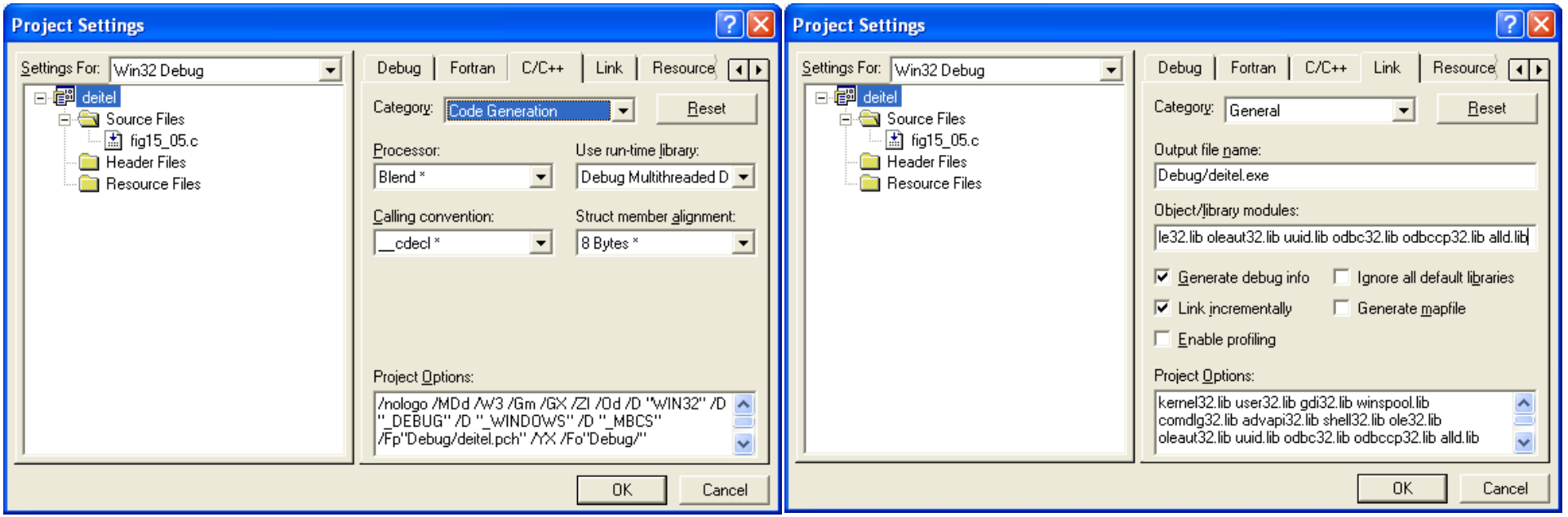
- Project Properties
  - The workspace has been set up, but now we need to link to the Allegro library. It is recommended configure the debugging version.
  - In order to get to the settings, you will need to Right-click on the Project Name in the “Workspace" and choose “Settings….".



15.3 Compiling an Allegro Program

- Dynamically Linking
- Debug Build
  - Under the Project Setting window, choose Setting For “Win32 Debug“.
  - Under the tab "C/C++“, choose Category "Code Generation" and select " Debug Multi-threaded DLL”, Click “OK"
  - Under the tab "Link“, add in the Object/libraries module "alld.lib“, Click “OK"

15.3 Compiling an Allegro Program



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	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. 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.

Outline

```
1  /*Fig. 15.3: fig15_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 */
```

install_keyboard allows Allegro to receive keyboard input

fig15_03.c

load_bitmap loads **picture.bmp** into memory and has **bmp** point to its data

readkey forces program to wait for a keypress

destroy_bitmap removes **bmp** from memory

blit draws **bmp** onto the top left corner of the screen

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**

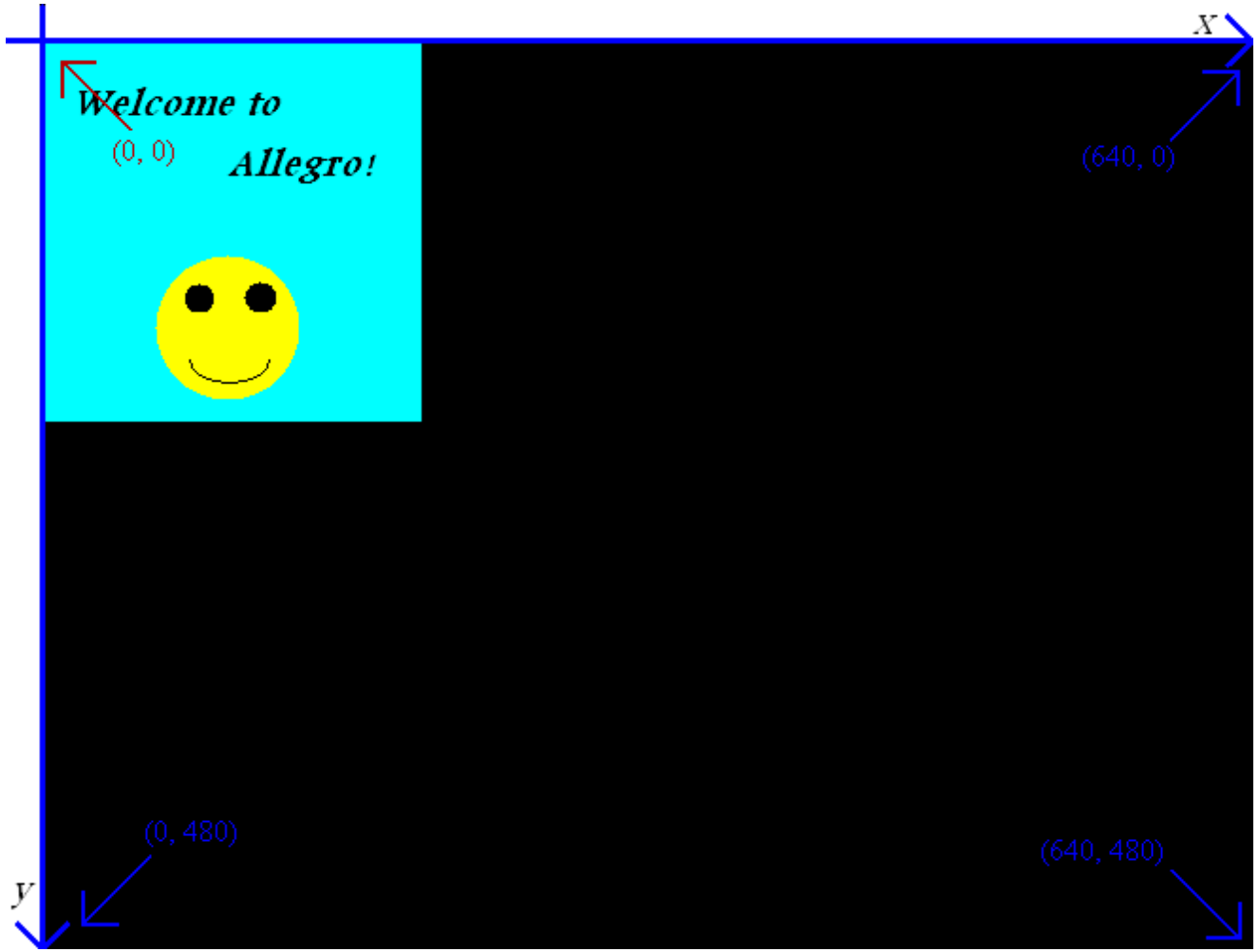


Fig. 15.4 | Allegro's coordinate system.

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**

Outline

```
1  /* Fig. 15.5: fig15_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
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
```

fig15_05.c

(1 of 4)

These symbolic constants correspond to the ball's four possible directions of travel

Outline

fig15_05.c

(2 of 4)

```
21 int main( void )
22 {
23    /* first, set up Allegro and the graphics mode */
24    allegro_init(); /* initialize Allegro */
25    install_keyboard(); /* install the keyboard for Allegro to use */
26    set_color_depth( 16 ); /* set the color depth to 16-bit */
27    set_gfx_mode( GFX_AUTODETECT, 640, 480, 0, 0 ); /* set graphics mode */
28    ball = load_bitmap( "ball.bmp", NULL ); /* load the ball bitmap */
29    ball_x = SCREEN_W / 2; /* give the ball its initial x-coordinate */
30    ball_y = SCREEN_H / 2; /* give the ball its initial y-coordinate */
31    srand( time( NULL ) ); /* seed the random function */
32    direction = rand() % 4; /* and then make a random initial direction */
33
34    while ( !key[KEY_ESC] ) /* until the escape key is pressed ... */
35    {
36       moveBall(); /* move the ball */
37       clear_to_color( screen, makecol( 255, 255, 255 ) );
38       /* now draw the bitmap onto the screen */
39       blit( ball, screen, 0, 0, ball_x, ball_y, ball->w, ball->h );
40    } /* end while */
41
42    destroy_bitmap( ball ); /* destroy the ball bitmap */
43    return 0;
44 } /* end function main */
45 END_OF_MAIN() /* don't forget this! */
46
```

SCREEN_W and **SCREEN_H** correspond to the width and height of the screen

clear_to_color function clears the entire screen to white

```
47 void moveBall() /* moves the ball */
48 {
49    switch ( direction ) {
50       case DOWN_RIGHT:
51          ++ball_x; /* move the ball to the right */
52          ++ball_y; /* move the ball down */
53          break;
54       case UP_RIGHT:
55          ++ball_x; /* move the ball to the right */
56          --ball_y; /* move the ball up */
57          break;
58       case DOWN_LEFT:
59          --ball_x; /* move the ball to the left */
60          ++ball_y; /* move the ball down */
61          break;
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
```

moveBall function moves the ball according to the value of **direction** variable

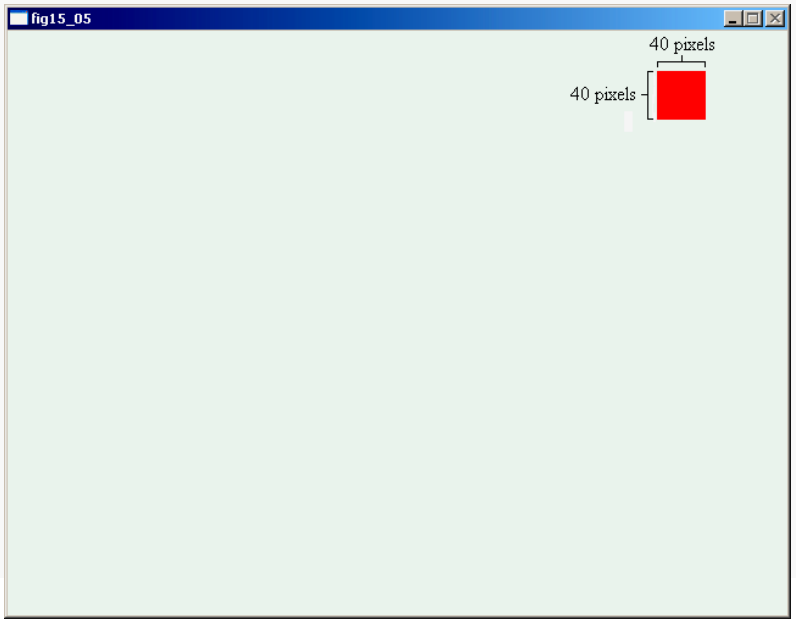
if statement tells program to reverse the ball's horizontal direction if it touches the top or bottom of the screen

Outline

fig15_05.c

(3 of 4)

```
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 */
```



Outline

`if` statement also reverses vertical direction

fig15_05.c

(4 of 4)

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)

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. 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**

Outline

fig15_07.c

(1 of 4)

```
1  /* Fig. 15.7: fig15_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
```

buffer bitmap is defined as a global variable

Outline

fig15_07.c

(2 of 4)

```
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 */
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
```

buffer is created to be the size of the screen

The ball is drawn onto the buffer, and then the buffer is drawn onto the screen

Buffer bitmap must also be destroyed at program's end

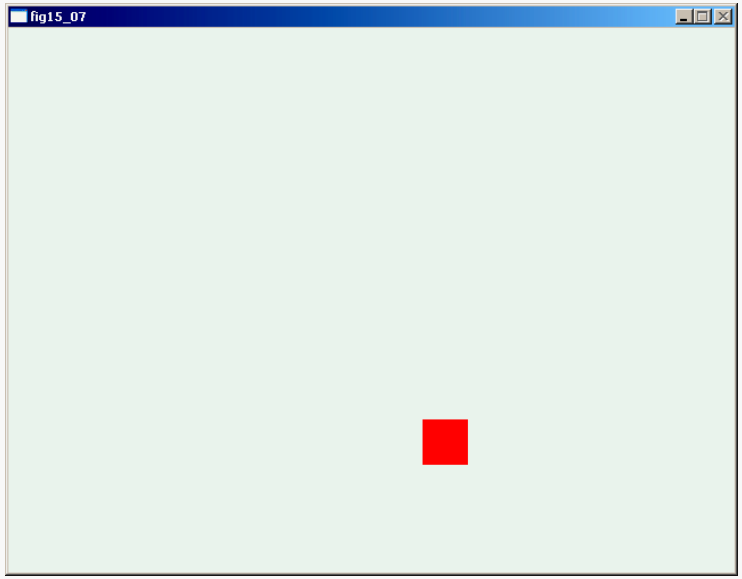
```
52 void moveBall() /* moves the ball */
53 {
54    switch ( direction ) {
55       case DOWN_RIGHT:
56          ++ball_x; /* move the ball to the right */
57          ++ball_y; /* move the ball down */
58          break;
59       case UP_RIGHT:
60          ++ball_x; /* move the ball to the right */
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
```

Outline

fig15_07.c

(3 of 4)

```
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 */
```



Outline

fig15_07.c

(4 of 4)

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

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. 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**

Outline

fig15_09.c

(1 of 5)

```
1  /* Fig. 15.9: fig15_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 */
19 BITMAP *ball; /* pointer to ball's image bitmap */
20 BITMAP *buffer; /* pointer to the buffer */
21 SAMPLE *boing; /* pointer to sound file */
22
```

boing sample is defined as a global variable

Outline

```
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 */
```

`install_sound` must be called to play sounds

`load_sample` loads sound data from a file

`fig15_09.c`

(2 of 5)

```
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 */
72          break;
73    } /* end switch */
74
```

Samples should be destroyed when they are no longer needed

Outline

fig15_09.c

(3 of 5)

Outline

fig15_09.c

(4 of 5)

```
75    /* make sure the ball doesn't go off screen */
76
77    /* if the ball is going off the top or bottom ... */
78    if ( ball_y <= 30 || ball_y >= 440 )
79       reverseVerticalDirection();
80
81    /* if the ball is going off the left or right ... */
82    if ( ball_x <= 0 || ball_x >= 600 )
83       reverseHorizontalDirection();
84 } /* end function moveBall */
85
86 void reverseVerticalDirection() /* reverse the ball's up-down direction */
87 {
88    if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
89       ++direction; /* make the ball start moving up */
90    else /* "up" directions are odd numbers */
91       --direction; /* make the ball start moving down */
92    play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
93 } /* end function reverseVerticalDirection */
```

Sample is played when the ball hits a wall

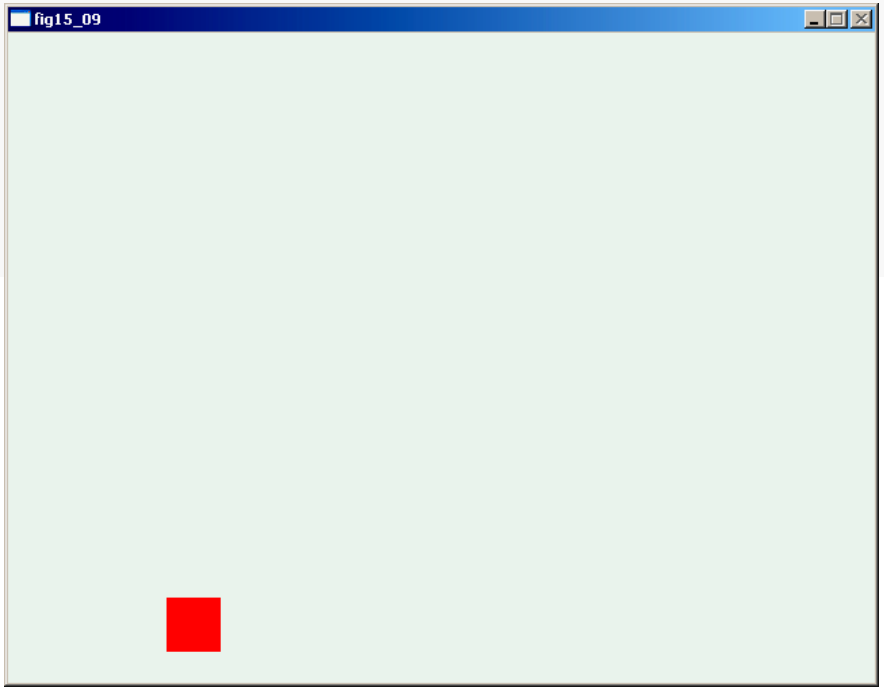
Outline

fig15_09.c

(5 of 5)

```
94
95 void reverseHorizontalDirection() /* reverses the horizontal direction */
96 {
97    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
98    play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
99 } /* 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

Outline

fig15_10.c

(1 of 7)

```
1  /* Fig. 15.10: fig15_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
```

New function has been added that checks if keys are being pressed

We are now adding paddles to the Pong game, so their bitmaps and coordinates must be stored

Outline

fig15_10.c

(2 of 7)

```
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 */
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
```

The paddle's image is loaded

The two paddles are then given their initial coordinates

```
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 */
```

Outline

respondToKeyboard function is called in main **while** loop

fig15_10.c

(3 of 7)

```
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 */
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 */
```

Outline

fig15_10.c

(4 of 7)

Outline

```
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
```

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

(5 of 7)

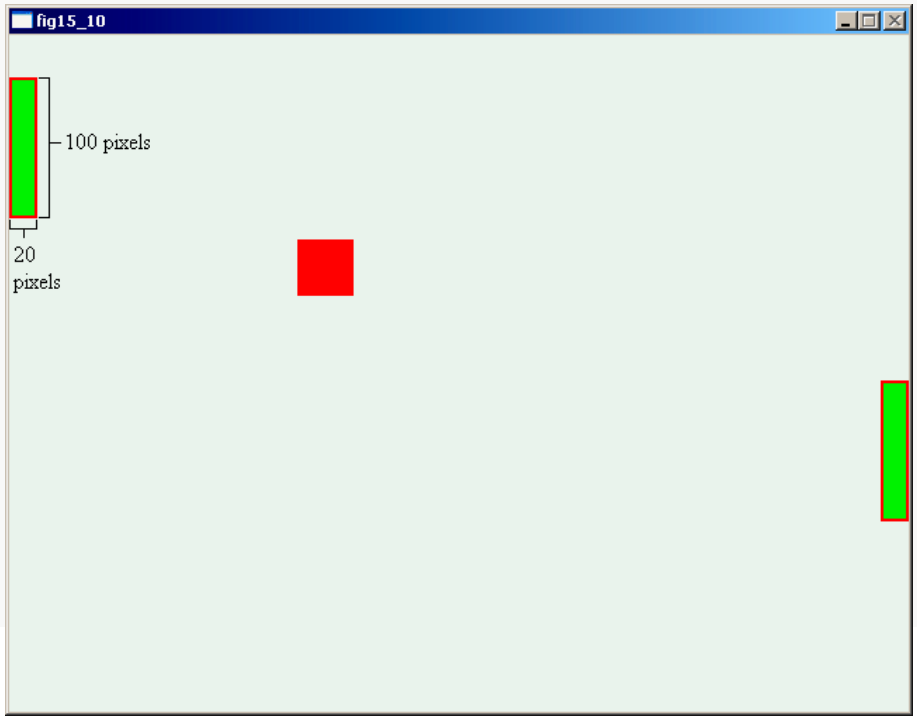
`if` statements use **`key`** array to check if keys are being pressed

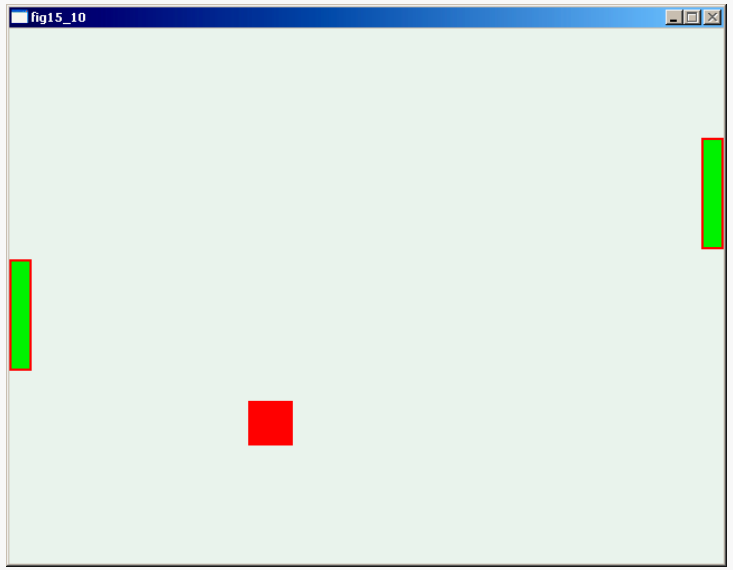
Outline

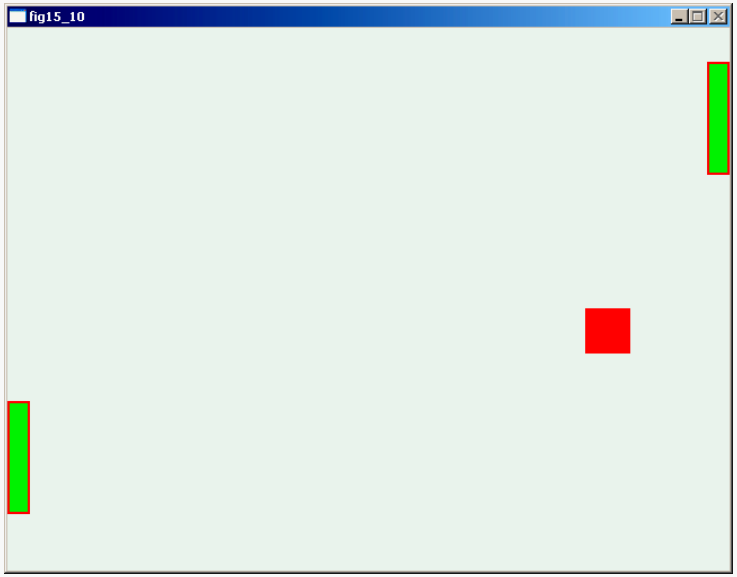
fig15_10.c

(6 of 7)

```
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 */
```







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	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. 15.11 | Functions that are useful for drawing text onto a bitmap.

Outline

fig15_12.c

(1 of 6)

```
1  /* Fig. 15.12: fig15_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 */
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
```

We now add a scoreboard to our Pong game, so there must be variables that store each player's score

FONT* variable will point to the data of our font in memory

Outline

fig15_12.c

(2 of 6)

```
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
```

load_font function loads font data into memory

Outline

fig15_12.c

(3 of 6)

```
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 */
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
```

textprintf_ex function displays text on the screen

```
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();
```

Outline

fig15_12.c

(4 of 6)

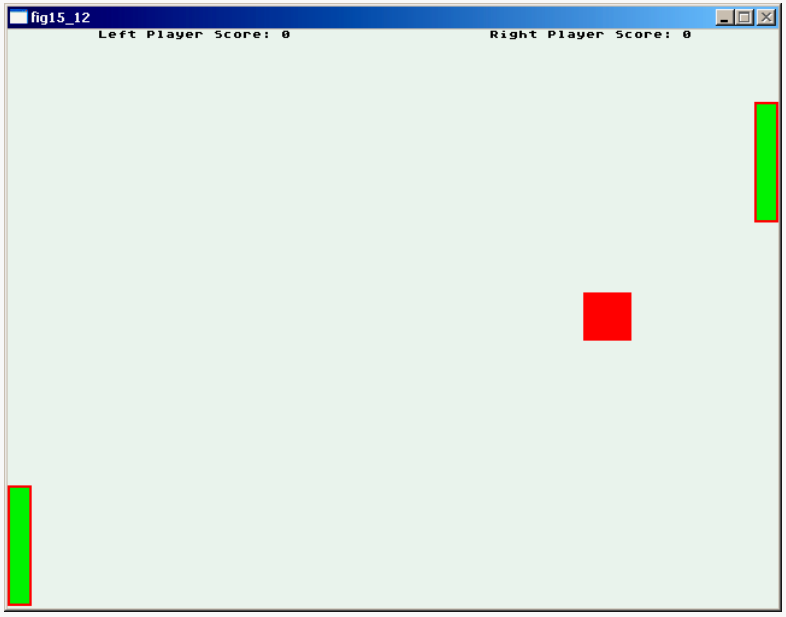
```
105
106    /* if the ball is going off the left or right ... */
107    if ( ball_x <= 0 || ball_x >= 600 )
108       reverseHorizontalDirection();
109 } /* end function moveBall */
110
111 void respondToKeyboard() /* responds to keyboard input */
112 {
113    if ( key[KEY_A] ) /* if A is being pressed... */
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
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
```

Outline

fig15_12.c

(5 of 6)

```
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 */
```



Outline

fig15_12.c

(6 of 6)

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`**

Outline

fig15_13.c

(1 of 9)

```
1  /* Fig. 15.13: fig15_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
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
```

Outline

fig15_13.c

(2 of 9)

```
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
```

Outline

fig15_13.c

(3 of 9)

```
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 */
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
```

line function draws a line onto the buffer

```
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
```

Outline

fig15_13.c

(4 of 9)

Outline

fig15_13.c

(5 of 9)

```
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   {
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 */
```

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.

Outline

fig15_13.c

(6 of 9)

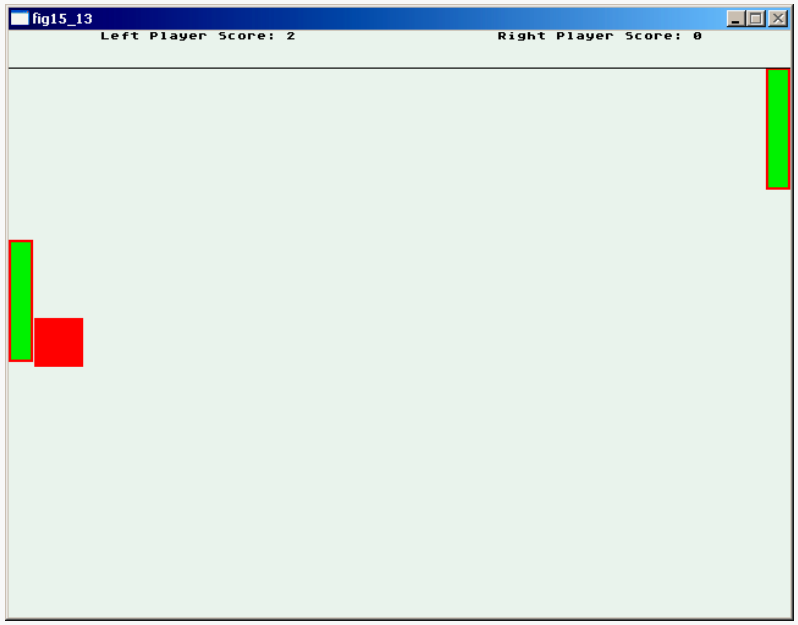
```
133
134 void respondToKeyboard() /* responds to keyboard input */
135 {
136    if ( key[KEY_A] ) /* if A is being pressed... */
137       barL_y -= 3; /* ... move the left paddle up */
138    if ( key[KEY_Z] ) /* if Z is being pressed... */
139       barL_y += 3; /* ... move the left paddle down */
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
```

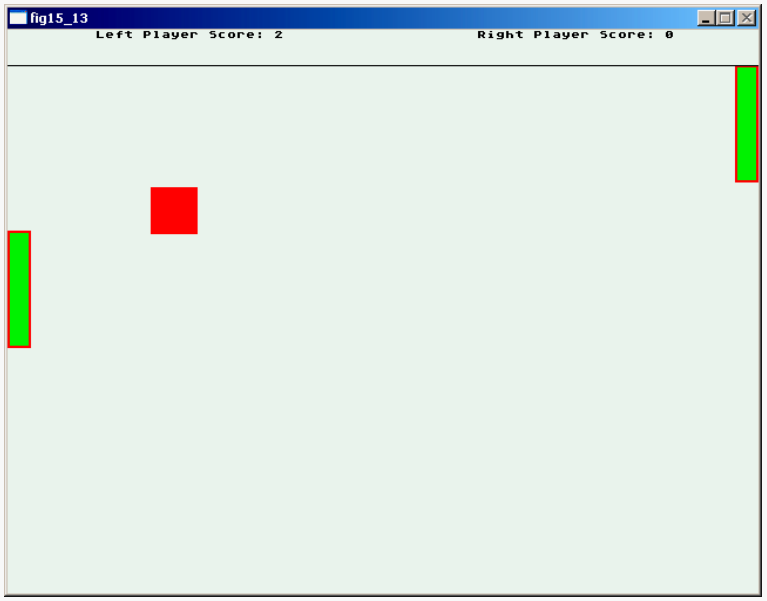
Outline

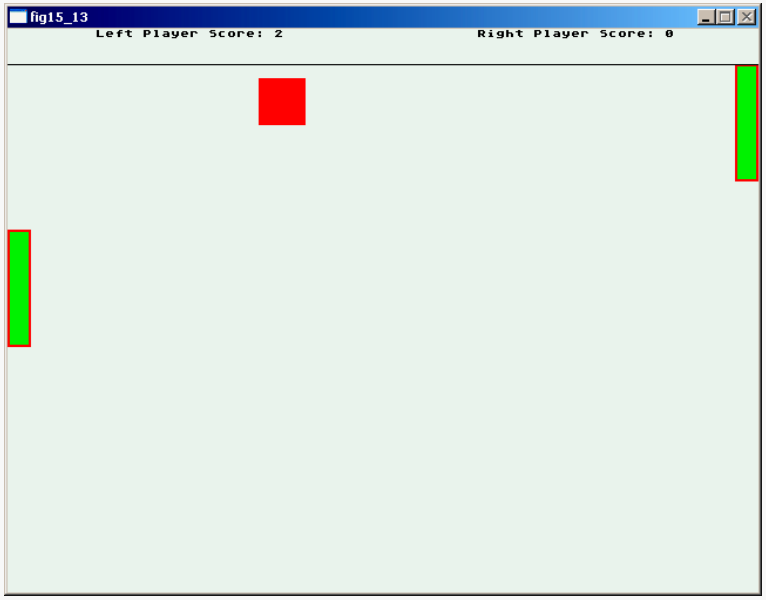
fig15_13.c

(7 of 9)

```
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 */
```



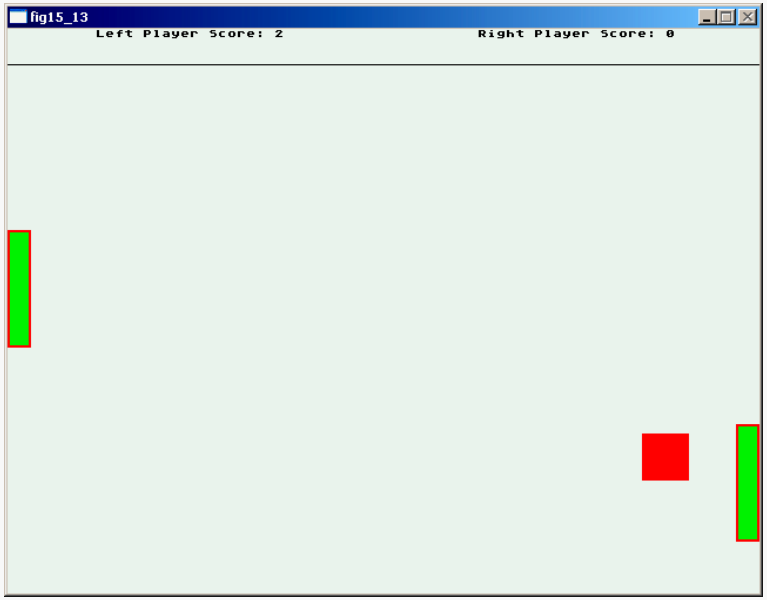


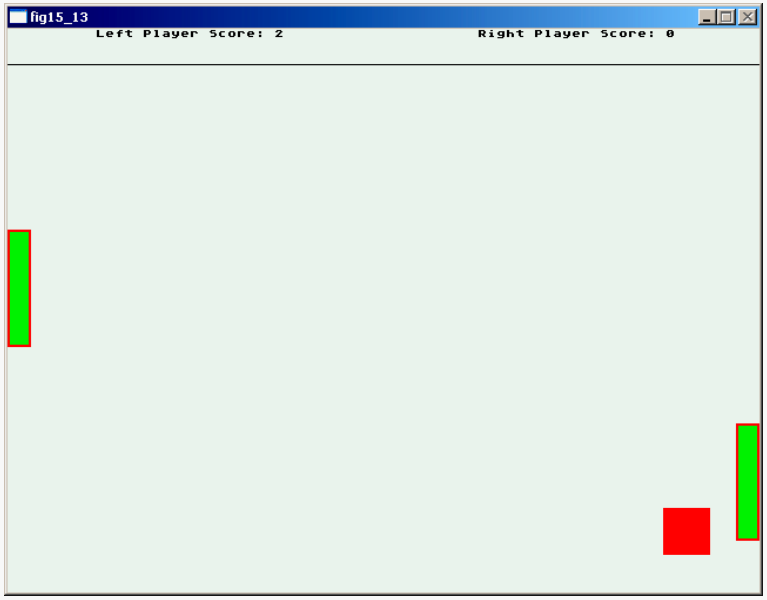


Outline

`fig15_13.c`

(8 of 9)





Outline

`fig15_13.c`

(9 of 9)

15.10 Timers in Allegro

- **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**

15.10 Timers in Allegro

- **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**

Outline

fig15_14.c

(1 of 7)

```
1  /* Fig. 15.14: fig15_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 );
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
```

Variables modified in any function called by a timer must be given the **volatile** qualifier

Outline

fig15_14.c

(2 of 7)

```
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
```

`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

Outline

fig15_14.c

(3 of 7)

```
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 );
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
```

remove_int function removes currently running timers

Note that the calls to **moveBall** and **respondToKeyboard** have been removed from the **while** loop—otherwise they would be called by both the timers *and* the **while** loop

```
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))
```

Outline

fig15_14.c

(4 of 7)

Outline

fig15_14.c

(5 of 7)

```
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  */
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 */
```

Outline

fig15_14.c

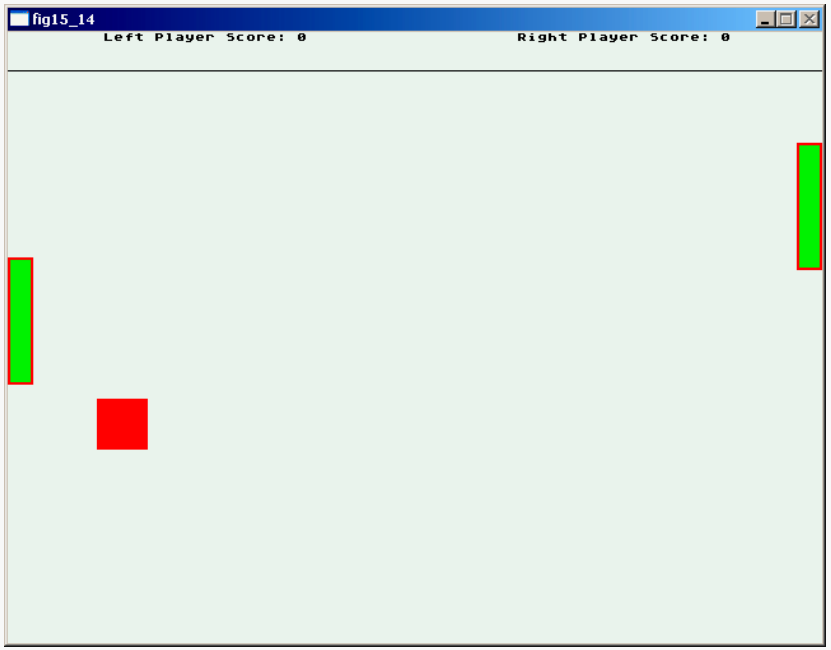
(6 of 7)

Outline

fig15_14.c

(7 of 7)

```
170
171 void reverseHorizontalDirection() /* reverses the horizontal direction */
172 {
173    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
174    play_sample( boing, 255, 128, 1000, 0 ); /* play "boing" sound once */
175 } /* 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**

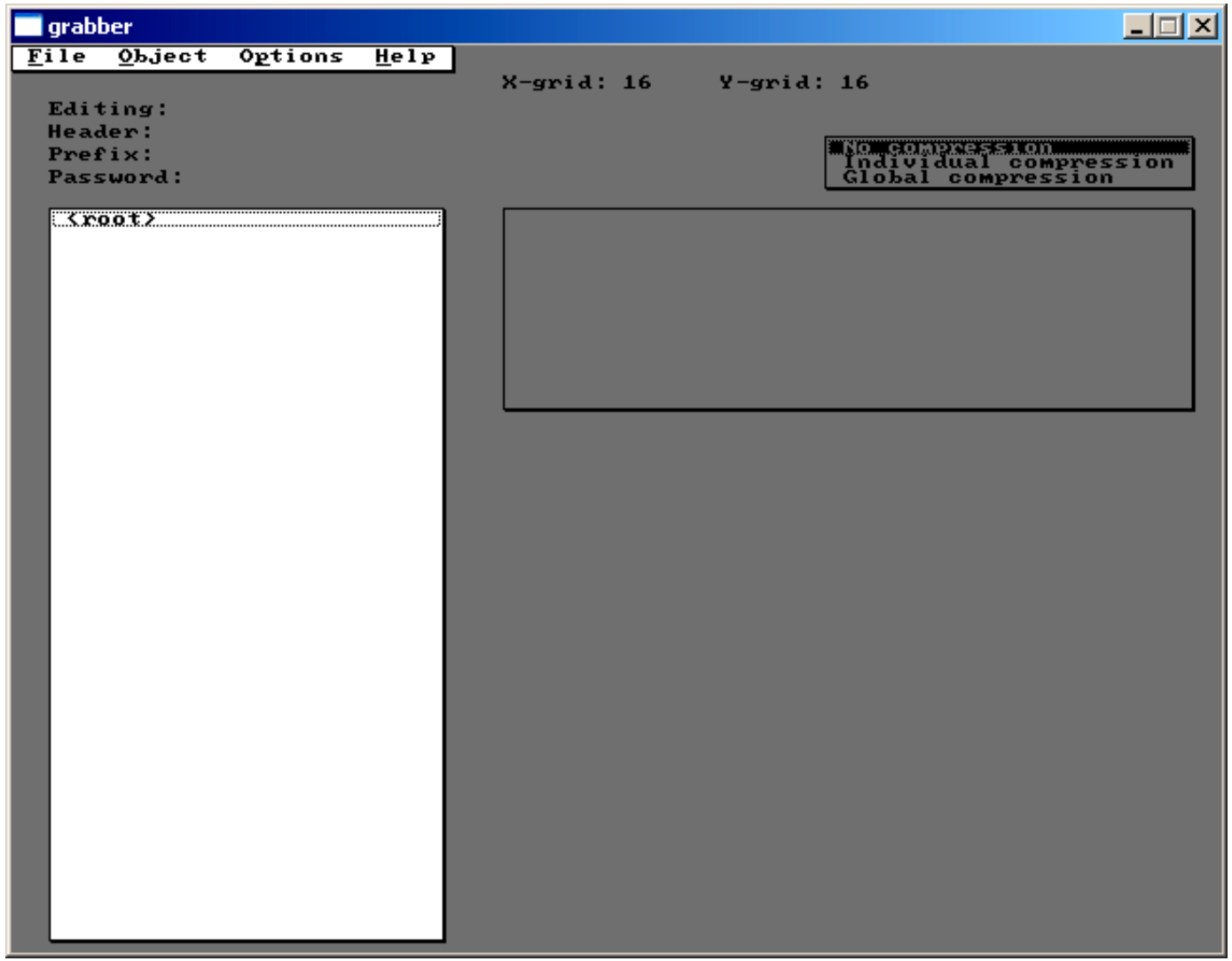


Fig. 15.15 | Allegro's grabber utility.

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**

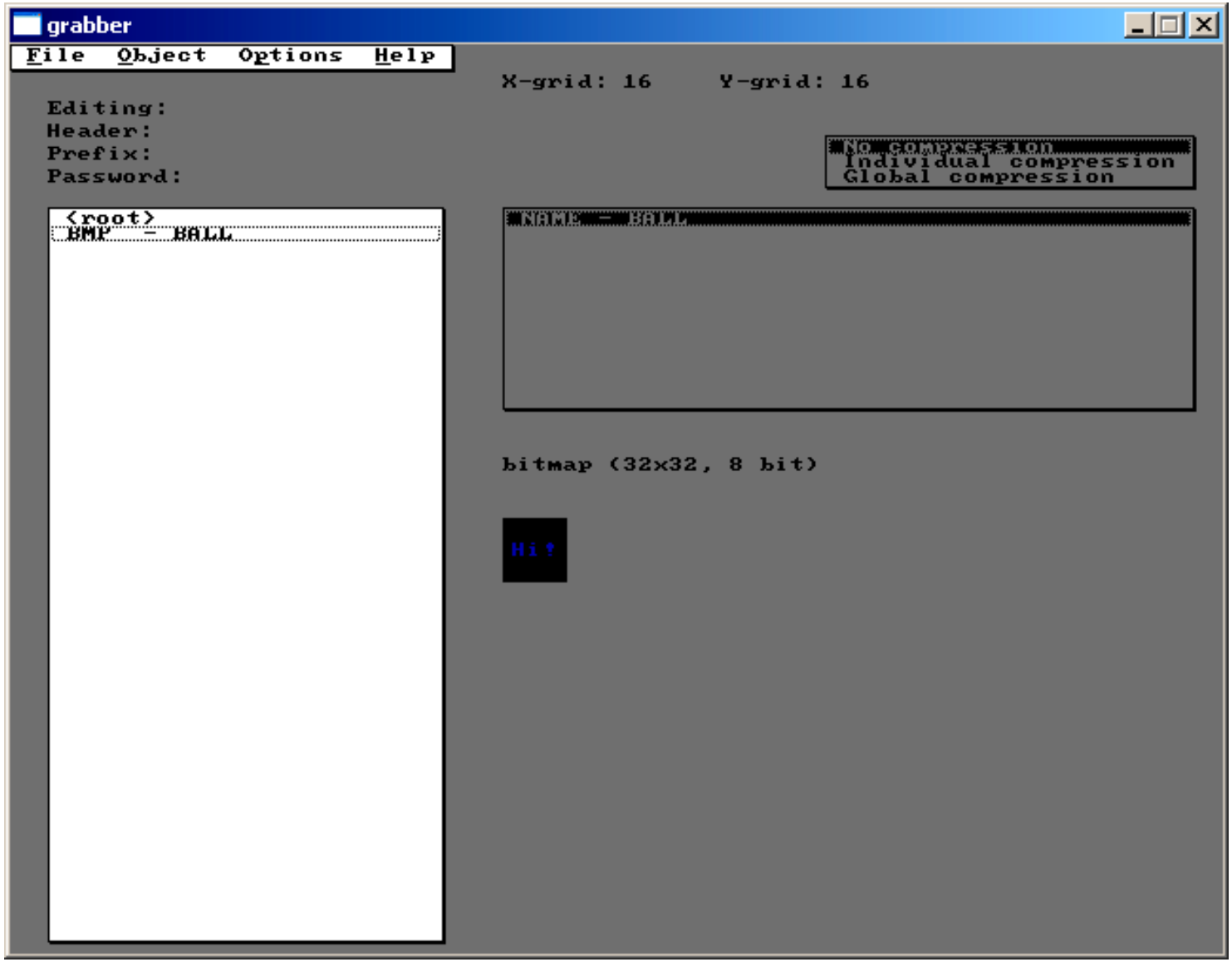


Fig. 15.16 | Adding a bitmap to a datafile.

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**

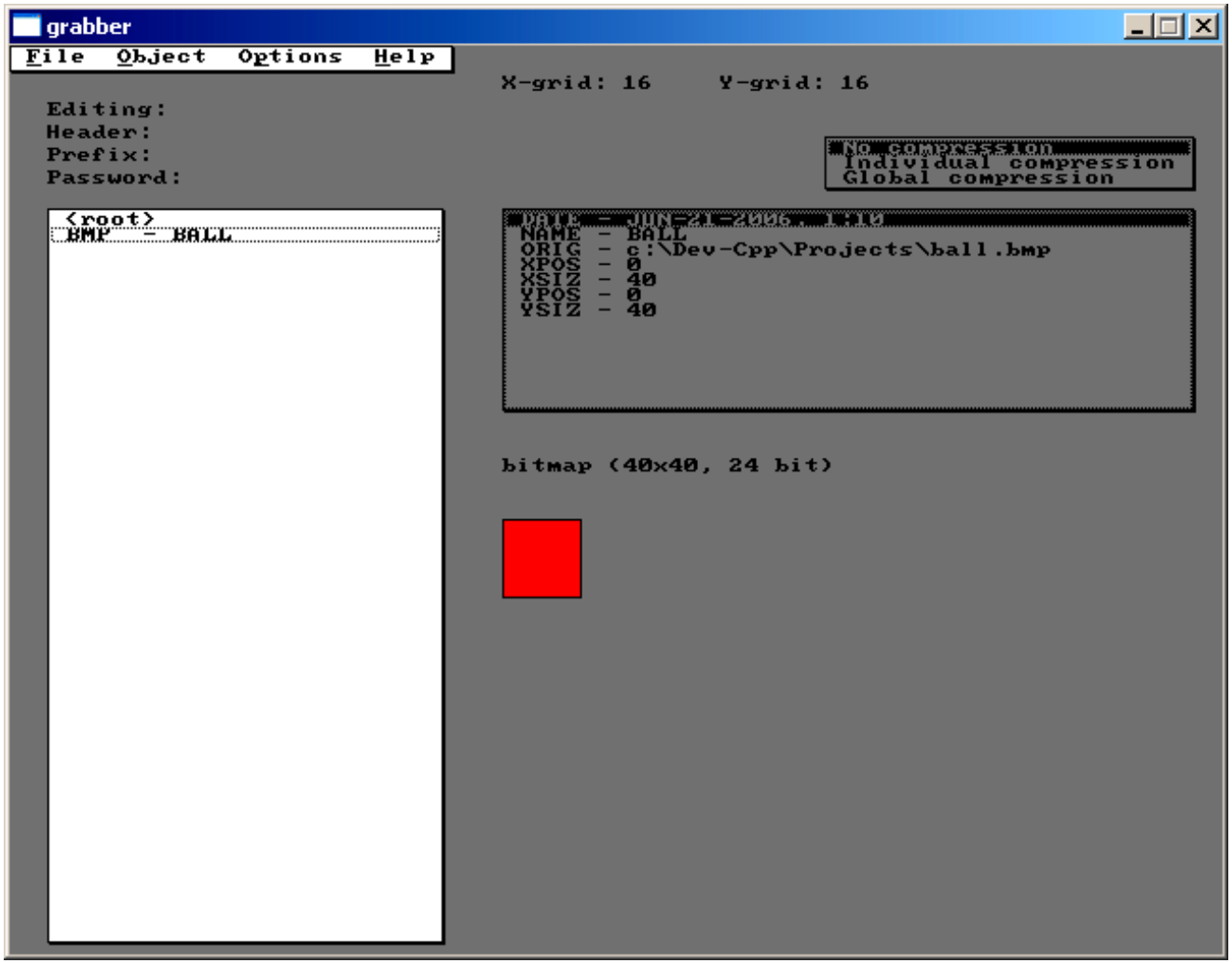


Fig. 15.18 | A complete imported object.

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**

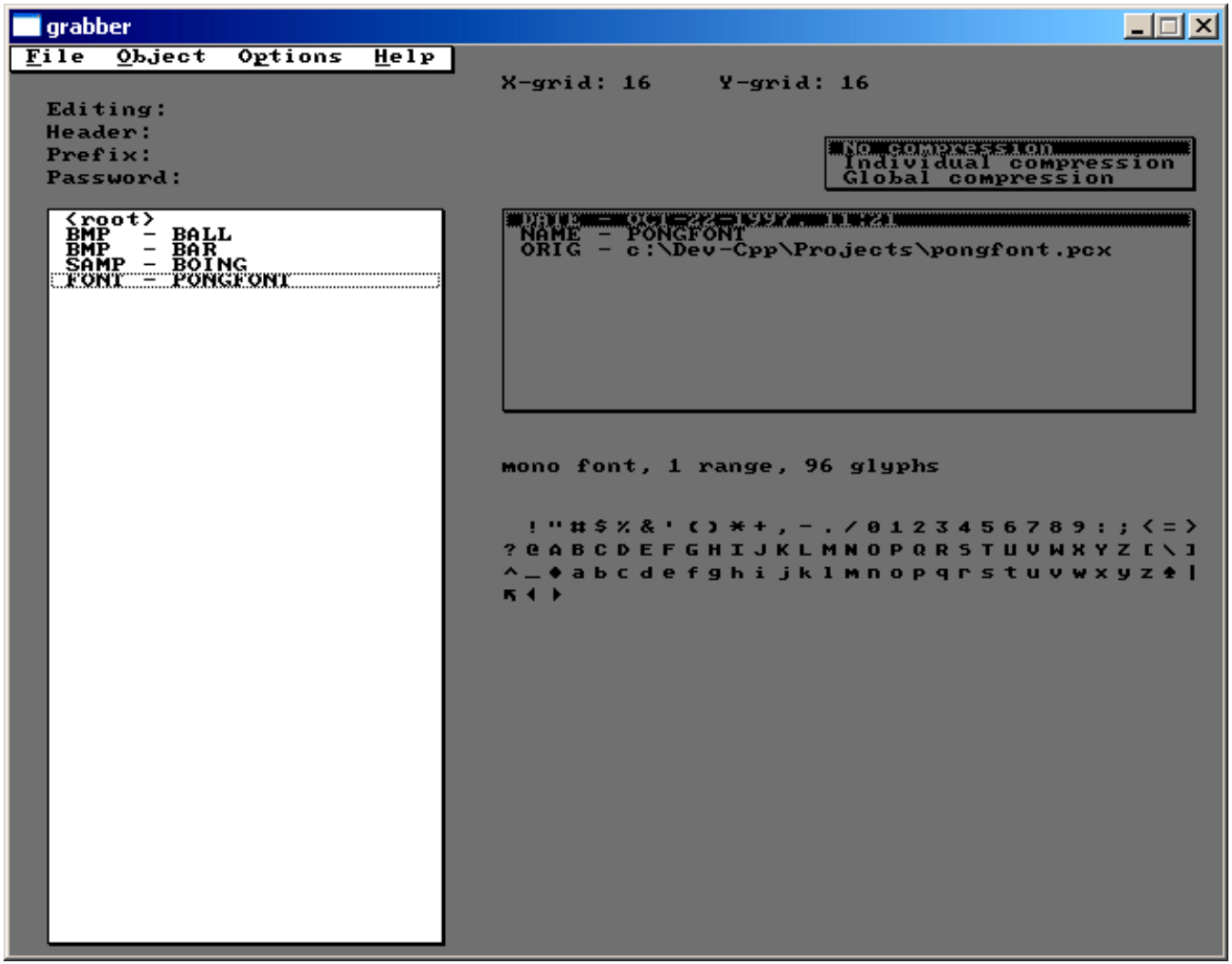


Fig. 15.19 | The grabber window after importing all of our objects.

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 `DATAFILE*` 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**

Outline

```
1  /* Allegro datafile object indexes, produced by grabber v4.2.0, MinGW32 */
2  /* Datafile: c:\Dev-Cpp\Projects\pongdatafile.dat */
3  /* Date: Wed Jun 21 12:57:10 2006 */
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 */
```

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**

Outline

fig15_21.c

(1 of 7)

```
1  /* Fig. 15.21: fig15_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
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
```

pong.h header included to help with accessing datafile objects

DATAFILE* object replaces all external files (the buffer is not external)

Outline

fig15_21.c

(2 of 7)

```
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
```

`load_datafile` function loads a datafile into the program

Outline

fig15_21.c

(3 of 7)

```
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",
63                      scoreL );
64       textprintf_ex( buffer, pongData[PONGFONT].dat, 400, 0,
65                      makecol( 0, 0, 0 ), -1, "Right Player Score: %d",
66                      scoreR );
67       blit( buffer, screen, 0, 0, 0, 0, buffer->w, buffer->h );
68       clear_bitmap( buffer );
69    } /* end while */
70
71    remove_int( moveBall ); /* remove moveBall timer */
72    remove_int( respondToKeyboard ); /* remove respondToKeyboard timer */
73    destroy_bitmap( buffer ); /* destroy the buffer bitmap */
74    unload_datafile( pongData ); /* unload the datafile */
75    return 0;
76 } /* end function main */
```

`unload_datafile` function removes datafile from memory

```
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   /* if the ball is going off the top or bottom ... */
101   if ( ball_y <= 30 || ball_y >= 440 )
102      reverseVerticalDirection(); /* make it go the other way */
103
104   /* if the ball is in range of the left paddle ... */
105   if (ball_x < 20 && (direction == DOWN_LEFT || direction == UP_LEFT))
```

Outline

fig15_21.c

(4 of 7)

Outline

fig15_21.c

(5 of 7)

```
106   {
107      /* is the left paddle in the way? */
108      if ( ball_y > ( barL_y - 39 ) && ball_y < ( barL_y + 99 ) )
109         reverseHorizontalDirection();
110      else if ( ball_x <= -20 ) { /* if the ball goes off the screen */
111         ++scoreR; /* give right player a point */
112         ball_x = SCREEN_W / 2; /* place the ball in the ... */
113         ball_y = SCREEN_H / 2; /* ... center of the screen  */
114         direction = rand() % 4; /* give the ball a random direction */
115      } /* end else */
116   } /* end if */
117
118   /* if the ball is in range of the right paddle ... */
119   if (ball_x > 580 && (direction == DOWN_RIGHT || direction == UP_RIGHT))
120   {
121      /* is the right paddle in the way? */
122      if ( ball_y > ( barR_y - 39 ) && ball_y < ( barR_y + 99 ) )
123         reverseHorizontalDirection();
124      else if ( ball_x >= 620 ) { /* if the ball goes off the screen */
125         ++scoreL; /* give left player a point */
126         ball_x = SCREEN_W / 2; /* place the ball in the ... */
127         ball_y = SCREEN_H / 2; /* ... center of the screen  */
128         direction = rand() % 4; /* give the ball a random direction */
129      } /* end else */
130   } /* end if */
131 } /* end function moveBall */
132
133 void respondToKeyboard() /* responds to keyboard input */
```

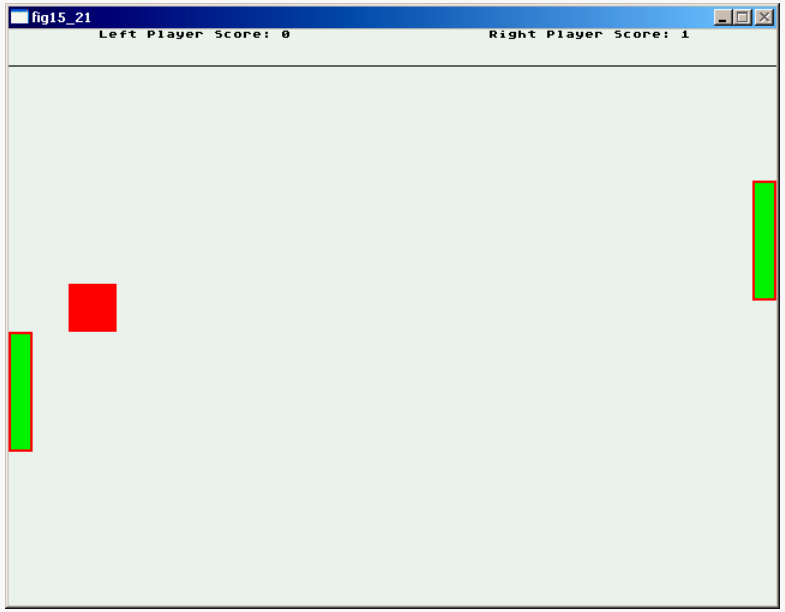
Outline

fig15_21.c

(6 of 7)

```
134 {
135    if ( key[KEY_A] ) /* if A is being pressed... */
136       barL_y -= 3; /* ... move the left paddle up */
137    if ( key[KEY_Z] ) /* if Z is being pressed... */
138       barL_y += 3; /* ... move the left paddle down */
139
140    if ( key[KEY_UP] ) /* if the up arrow key is being pressed... */
141       barR_y -= 3; /* ... move the right paddle up */
142    if ( key[KEY_DOWN] ) /* if the down arrow key is being pressed... */
143       barR_y += 3; /* ... move the right paddle down */
144
145    /* make sure the paddles don't go offscreen */
146    if ( barL_y < 30 ) /* if left paddle is going off the top */
147       barL_y = 30;
148    else if ( barL_y > 380 ) /* if left paddle is going off the bottom */
149       barL_y = 380;
150    if ( barR_y < 30 ) /* if right paddle is going off the top */
151       barR_y = 30;
152    else if ( barR_y > 380 ) /* if right paddle is going off the bottom */
153       barR_y = 380;
154 } /* end function respondToKeyboard */
155
156 void reverseVerticalDirection() /* reverse the ball's up-down direction */
157 {
158    if ( ( direction % 2 ) == 0 ) /* "down" directions are even numbers */
159       ++direction; /* make the ball start moving up */
160    else /* "up" directions are odd numbers */
161       --direction; /* make the ball start moving down */
162    play_sample( pongData[BOING].dat, 255, 128, 1000, 0 ); /* play sound */
163 } /* end function reverseVerticalDirection */
```

```
164
165 void reverseHorizontalDirection() /* reverses the horizontal direction */
166 {
167    direction = ( direction + 2 ) % 4; /* reverse horizontal direction */
168    play_sample( pongData[BOING].dat, 255, 128, 1000, 0 ); /* play sound */
169 } /* end function reverseHorizontalDirection */
```



Outline

fig15_21.c

(7 of 7)

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`**