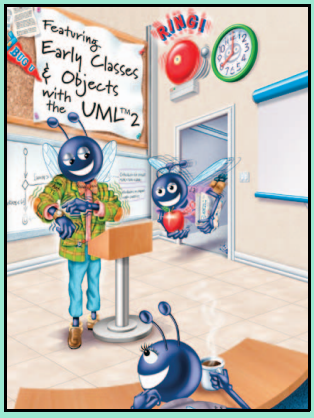
21



Come, Watson, come! The game is afoot.
—Sir Arthur Conan Doyle

For it's one, two, three strikes you're out at the old ball game.
—Jack Norworth

The game is up.
—William Shakespeare

If you wish to avoid foreign collision, you had better abandon the ocean.
—Henry Clay

There is strong shadow where there is much light.
—Johann Wolfgang von Goethe

Wagner's music is better than it sounds.
—Mark Twain

Game Programming with OGRE

OBJECTIVES

In this chapter you will learn:

- Some basics of game programming.
- Where to find instructions on how to install the Ogre 3D graphics engine to work with your C++ programs on the Windows, Linux and Mac platforms.
- To create games using Ogre.
- To perform collision detection.
- To use Ogre to import and display graphics.
- To use OgreAL to integrate the OpenAL audio library into your games.
- To have Ogre accept keyboard input.
- To create the simple game Pong® with Ogre and OgreAL.
- To use Ogre to regulate the speed of a game.

Outline

21.1 Introduction

We now introduce game programming and graphics with the Ogre 3D graphics engine. Created in 2000 by Steve Streeting, Ogre is an open source project maintained by the Ogre team at `www.ogre3d.org`. First, we discuss basic issues involved in game programming. Then we show how to use Ogre to create the simple game of Pong® originally developed and marketed by Atari, one of the pioneering companies of the computer games industry. We demonstrate how to create a scene with colored 3D graphics, smoothly animate moving objects, use timers to control animation speed, detect collisions between objects, add sound, accept keyboard input and display text output.

21.2 Installing Ogre, OgreAL and OpenAL

Ogre is powerful and easy to use, but the installation is a bit involved and varies for different platforms and compilers. There are two installation options—you can install the Ogre SDK or download the source code and compile it. You must also install OgreAL and the OpenAL audio library. OpenAL handles the sound functionality and OgreAL allows you to integrate the OpenAL capabilities with the Ogre code.

The Ogre Wiki site has instructions for installing Ogre on several platforms including Windows, Mac OS X and various Linux distributions. There are also instructions for configuring your compiler to use Ogre. You can find help with any problems you encounter in the Ogre forums. We provide links to all the instruction pages and help forums on this book's web page at `www.deitel.com/books/cpphtp6`, as well as in our C++ Game Programming Resource Center at `www.deitel.com/CplusplusGameProgramming`.

21.3 Basics of Game Programming

This section introduces the general fundamentals of game programming. Section 21.4 presents an implementation of the game of Pong, using the Ogre and OgreAL libraries.

Graphics

Graphics are perhaps the most crucial feature of any video game. Once a specialty, graphics programming is becoming more accessible even to novice programmers. There are many **3D graphics engines** available—these frameworks hide the often tedious and complex programming required with graphics APIs, and allow you to manage graphics more easily.

Ogre (Object-oriented Graphics Rendering Engine), one of the leading graphics engines, has been used in many commercial products including video games. It provides an object-oriented interface for 3D graphics programming. It supports the Direct3D and OpenGL graphics APIs and runs on the Windows, Linux and Mac platforms. **Direct3D** is Microsoft's Windows graphics API. **OpenGL** is a graphics specification implemented by many video card vendors across all major platforms, including Windows.

Ogre is strictly a graphics rendering engine—it does not directly support sound, physics, collision detection, networking or other game-related needs. The Ogre community has produced many add-ons that allow users to integrate other libraries with Ogre to support those features.

3D Models

A **3D model** is a computer representation of an object which can be drawn on the screen—a process called **rendering**. **Materials** determine an object's appearance by setting lighting properties, colors and textures. A **texture** is an image that is wrapped around the model.

Most objects displayed in 3D graphics are 3D models, everything from the terrain to the characters and the buildings. The models are created in **3D modeling tools**. Some popular 3D modeling tools are Maya (`http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018`), SoftImage XSI (`http://www.softimage.com/`) and Blender (`http://www.blender.org/`). They are all available for Windows, Linux and Mac platforms. Blender is free and Maya offers a free version. SoftImage has a free 30-day trial available. The Ogre community has also produced several tools to allow users to **export 3D models** from these and other popular modeling tools into Ogre.

Materials, Textures and Colors

Colors are determined by red, green and blue light intensities, which can range from 0 to 1.0—a color value of `(1.0, 0, 0)` will create a bright red color, `(0, 1.0, 0)` will create a bright green, and `(0, 0, 1.0)` will make a bright blue. The first value is the red intensity, the second is the green intensity and the third is the blue intensity. To create white, use the maximum intensities of all three color values, `(1.0, 1.0, 1.0)`. To create black—the absence of all color—use `(0, 0, 0)`. Color values sometimes include an **alpha channel** to represent transparency, also ranging from 0 to 1.0, 0 being completely transparent and 1 being completely opaque. Figure 21.1 shows common colors and their red, green, and blue intensity values. You can find color charts on the web as well. There is one at `www.tayloredmktg.com/rgb/`.

In 3D graphics, materials are used to determine the color of a 3D model. A material determines how the model should reflect different types of light and applies textures to the model. Materials can be set to use different **levels of detail (LoD)** depending on how far away from the viewer the model is. When close up, the model should be rendered with as much detail as possible. But, when the object in the scene is far away, there is no point in wasting computing power rendering the object with details that the viewer can't see—to increase performance the object can be rendered with much less detail.

Color	Red value	Green value	Blue value
Red	1.0	0.0	0.0
Green	0.0	1.0	0.0
Blue	0.0	0.0	1.0
Orange	1.0	0.784	0.0
Pink	1.0	0.686	0.686
Cyan	0.0	1.0	1.0
Magenta	1.0	0.0	1.0
Yellow	1.0	1.0	0.0
Black	0.0	0.0	0.0
White	1.0	1.0	1.0
Gray	0.5	0.5	0.5
Light gray	0.75	0.75	0.75
Dark gray	0.25	0.25	0.25

Fig. 21.1 | Red, green, and blue intensities of common colors.

Lighting

There are four different types of light in a 3D scene—ambient, diffuse, emissive and specular.[1] **Ambient light** is the general lighting in the scene that doesn't come from any apparent source. **Diffuse light** appears to come from a particular direction and is reflected evenly off any surfaces it hits. **Emissive light** appears to come from an object in the scene. Emissive light won't affect the objects around it, but will make the object emitting it seem brighter. **Specular light** comes from a particular direction and is reflected off an object based on the surface texture and angles. This is used to make an object appear shiny.

Collision Detection and Response

Collision detection is the process of determining whether two objects in a game are touching. You must know which objects to test and deal with some complex mathematics. Checking whether one square hit another is relatively simple if each is parallel to flat ground. Checking circles and spheres is more difficult—the mathematics of curved surfaces is more complex.

Objects need to react appropriately when they collide with other objects. Some objects, such as walls, are stationary, while others move throughout the scene. Modeling the physics of moving objects can be complex. There are collision detection and physics modeling libraries that handle these complexities for you. Such libraries help to create a realistic game playing experience.

1. Astel, D., and K. Hawkins, *Beginning OpenGL: Game Programming*, 2005, pp. 104–110.

Sound
Sound is crucial to the game playing experience. Gamers want to hear the lasers on their ships blasting away or the engines of their street racers revving up as they "peel out" at the starting line. Audio libraries help you enrich your games with sound. Many of those libraries support **3D sound**. In a 3D scene, objects emitting sound may be at various distances and directions from the user. The sound libraries take these factors into account when playing sounds. A sound from an object close to the listener will be louder than the sound from an object farther away. Also, sounds from one side of the listener will be played differently than sounds from the other.

Text
Games often communicate with the user by displaying text. This can range from giving the user instructions, to simply reporting how many points he or she has scored so far. In many games, text is a crucial form of communication between players. You can find free text fonts to use in your games at `www.1001freefonts.com`.

Timers
The speed at which a game runs can vary between systems due to differences in processor speeds. To solve this problem, game programmers use **timers** to control animation speed. If an object moves the same distance every **frame** (each time the screen is redrawn) then it may move at different speeds on different computers. Timers help make animations look more natural by regulating their speed to keep them smooth.

User Experience
Games should be fun to play and should appeal to the player in as many ways as possible. The basics we've discussed contribute to the overall user experience. You can get the player's attention through graphics and sound. Actions in games often have sounds associated with them. Many web sites offer free sounds you can use in your games. Some popular sound sites are Sound Hunter (`www.soundhunter.com`), Absolute Sound Effects Archive (`www.grsites.com/sounds`) and the search engine FindSounds (`www.findsounds.com`). You can also play a sound track in the background. Be sure to get permission to use any copyrighted songs if you plan on releasing your game as a product.

Players need to interact with games. User input devices include the keyboard, mouse, joystick and the game controller. Keep the controls simple—the game should be easy to use, but not easy to beat. You can communicate with the player using text.

21.4 The Game of Pong: Code Walkthrough

In the next several sections, we present a complete C++/Ogre implementation of the game of Pong. We walk through the code, explaining the Ogre capabilities as they are encountered. This is one of the largest example programs in the book. You should test drive the Pong program thoroughly before reading the code walkthrough. Download the exercises from `www.deitel.com/books/cpphtp6`. You'll find all the Pong files in the `ch21` folder. Copy the `PongResources` folder to the `OgreSDK\media` folder. Ogre will throw a runtime exception if any of the resources can't be found.

Once you have compiled the code and placed the `PongResources` folder in the correct location, run the program. If you followed the Ogre installation instructions correctly, the executable should be in your `OgreSDK\bin\debug` folder.

Pong has four major game objects, a ball, two paddles and a rectangular box (Fig. 21.2). The ball will bounce across the screen inside the box while the players control the paddles to keep the ball from hitting the left or right sides. If the ball hits the left or right side of the box, the player "attacking" that side is awarded a point. The score is displayed at the top of the screen.

Play the game for a while, using the *A* and *Z* keys to control the left paddle and the up and down arrow keys to control the right paddle. Notice the colors of the objects, the ball interacting with the other objects and the score displayed at the top of the screen. You won't be able to use your mouse while the Pong window is active—hit the *Esc* key to quit.

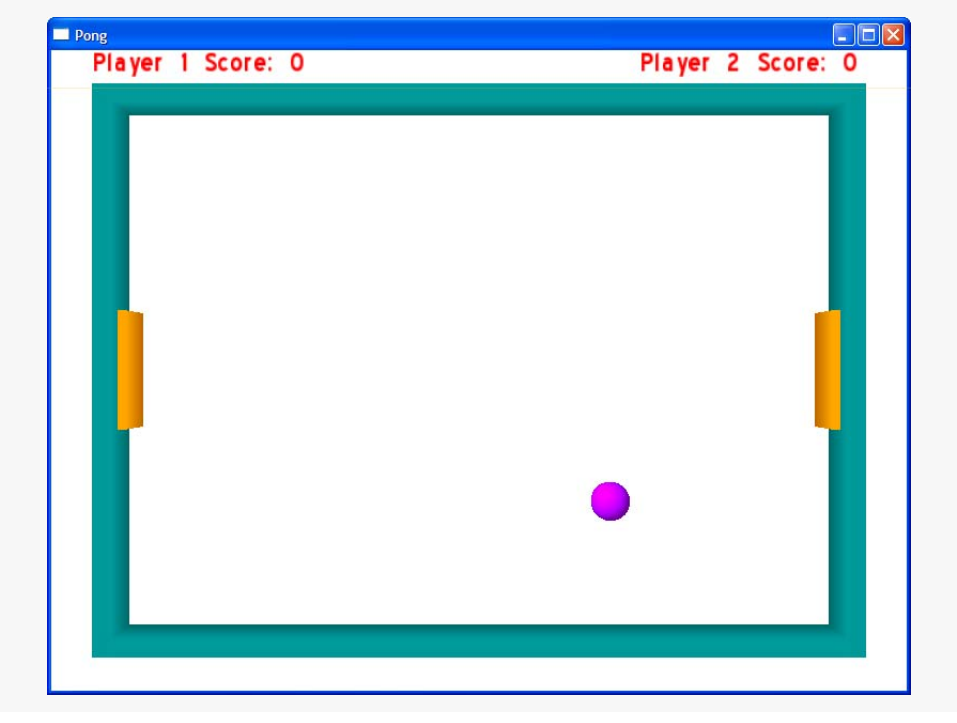


Fig. 21.2 | Pong game objects.

21.4.1 Ogre Initialization

Class `Pong` (Figs. 21.3–21.4) represents the game of Pong. Figure 21.3 contains the `Pong` class definition. Line 20 is the function prototype for the `run` function, which will start and run the game. Lines 23–24 contain the function prototypes for handling user input from the keyboard. Lines 27–28 define the function prototypes for performing game logic between frames. We also declare pointers to important Ogre objects (lines 37–41), input handling objects (lines 44–45) and the game objects (lines 48–50). Lines 53–55 define some variables used to control the behavior of the game—we'll discuss these later.

Before we can display any graphics we need to initialize the Ogre engine settings and create certain Ogre objects. The **OGRE Engine Rendering Setup** dialog box (Fig. 21.5) enables the user to choose the rendering settings, including which rendering subsystem to

```
// Pong.h
// Pong class definition (represents a game of Pong).
#ifndef PONG_H
#define PONG_H

#include <OISEvents.h> // OISEvents class definition
#include <OISInputManager.h> // OISInputManager class definition
#include <OISKeyboard.h> // OISKeyboard class definition
#include <Ogre.h> // Ogre class definition
using namespace Ogre; // use the Ogre namespace

class Ball; // forward declaration of class Ball
class Paddle; // forward declaration of class Paddle

class Pong : public FrameListener, public OIS::KeyListener
{
public:
   Pong(); // constructor
   ~Pong(); // destructor
   void run(); // run a game of Pong

   // handle keyPressed and keyReleased events
   bool keyPressed( const OIS::KeyEvent &keyEventRef );
   bool keyReleased( const OIS::KeyEvent &keyEventRef );

   // move the game objects and control interactions between frames
   virtual bool frameStarted( const FrameEvent &frameEvent );
   virtual bool frameEnded( const FrameEvent &frameEvent );

   static void updateScore(); // update the score text
   static int player1Score; // player 1's score
   static int player2Score; // player 2's score
private:
   void createScene(); // create the scene to be rendered

   // Ogre objects
   Root *rootPtr; // pointer to Ogre's Root object
   SceneManager *sceneManagerPtr; // pointer to the SceneManager
   RenderWindow *windowPtr; // pointer to RenderWindow to render scene in
   Viewport *viewportPtr; // pointer to Viewport, area that a camera sees
   Camera *cameraPtr; // pointer to a Camera in the scene

   // OIS input objects
   OIS::InputManager *inputManagerPtr; // pointer to the InputManager
   OIS::Keyboard *keyboardPtr; // pointer to the Keyboard

   // game objects
   Ball *ballPtr; // pointer to the Ball
   Paddle *leftPaddlePtr; // pointer to player 1's Paddle
   Paddle *rightPaddlePtr; // pointer to player 2's Paddle

   // variables to control game states
   bool quit, pause; // did user quit or pause the game
```

Fig. 21.3 | Pong class definition (represents a game of Pong). (Part 1 of 2.)

```
54      static bool wait; // should the Ball's movement be delayed
55      Real time; // used to delay the motion of a new Ball
56   }; // end class Pong
57
58   #endif // PONG_H
```

Fig. 21.3 | Pong class definition (represents a game of Pong). (Part 2 of 2.)

```
1    // Pong.cpp
2    // Pong class member function definitions.
3    #include "Pong.h" // Pong class definition
4    #include "Ball.h" // Ball class definition
5    #include "Paddle.h" // Paddle class definition
6    #include <OISEvents.h> // OISEvents class definition
7    #include <OISInputManager.h> // OISInputManager class definition
8    #include <OISKeyboard.h> // OISKeyboard class definition
9    #include <OgreAL.h> // OgreAL class definition
10   #include <OgreStringConverter.h> // OgreStringConverter class definition
11   #include <Ogre.h> // Ogre class definition
12   using namespace Ogre; // use the Ogre namespace
13
14   int Pong::player1Score = 0; // initialize player 1's score to 0
15   int Pong::player2Score = 0; // initialize player 2's score to 0
16   bool Pong::wait = false; // initialize wait to false
17
18   const int SPEED_INCREMENT = 10; // amount to increase the Ball's speed
19   const int SPEED_DECREMENT = -10; // amount to decrease the Ball's speed
20
21   // directions to move the Paddles
22   const Vector3 PADDLE_DOWN = Vector3( 0, -15, 0 );
23   const Vector3 PADDLE_UP = Vector3( 0, 15, 0 );
24
25   // constructor
26   Pong::Pong()
27   {
28      rootPtr = new Root(); // initialize Root object
29
30      // use the Ogre Config Dialog Box to choose the settings
31      rootPtr->showConfigDialog();
32
33      // get a pointer to the RenderWindow
34      windowPtr = rootPtr->initialise( true, "Pong" );
35
36      // create the SceneManager
37      sceneManagerPtr = rootPtr->createSceneManager( ST_GENERIC );
38
39      // create the Camera
40      cameraPtr = sceneManagerPtr->createCamera( "PongCamera" );
41      cameraPtr->setPosition( Vector3( 0, 0, 200 ) ); // set Camera position
42      cameraPtr->lookAt( Vector3( 0, 0, 0 ) ); // set where Camera looks
43      cameraPtr->setNearClipDistance( 5 ); // near distance Camera can see
44      cameraPtr->setFarClipDistance( 1000 ); // far distance Camera can see
```

Fig. 21.4 | Pong class member function definitions. (Part 1 of 6.)

```
45
46      // create the Viewport
47      viewportPtr = windowPtr->addViewport( cameraPtr );
48      viewportPtr->setBackgroundColour( ColourValue( 0, 0, 0 ) );
49
50      // set the Camera's aspect ratio
51      cameraPtr->setAspectRatio( Real( viewportPtr->getActualWidth() ) /
52         ( viewportPtr->getActualHeight() ) );
53
54      // set the scene's ambient light
55      sceneManagerPtr->setAmbientLight( ColourValue( 0.75, 0.75, 0.75 )
56
57      // create the Light
58      Light *lightPtr = sceneManagerPtr->createLight( "Light" ); // a Light
59      lightPtr->setPosition( 0, 0, 50 ); // set the Light's position
60
61      unsigned long hWnd; // variable to hold the window handle
62      windowPtr->getCustomAttribute( "WINDOW", &hWnd ); // get window handle
63      OIS::ParamList paramList; // create an OIS ParamList
64
65      // add the window to the ParamList
66      paramList.insert( OIS::ParamList::value_type( "WINDOW",
67         Ogre::StringConverter::toString(  hWnd ) ) );
68
69      // create the InputManager
70      inputManagerPtr = OIS::InputManager::createInputSystem( paramList );
71      keyboardPtr = static_cast< OIS::Keyboard* >( inputManagerPtr->
72         createInputObject( OIS::OISKeyboard, true ) ); // create a Keyboard
73      keyboardPtr->setEventCallback( this ); // add a KeyListener
74
75      rootPtr->addFrameListener( this ); // add this Pong as a FrameListener
76
77      // load resources for Pong
78      ResourceGroupManager::getSingleton().addResourceLocation(
79         "../../media/PongResources", "FileSystem", "Pong" );
80      ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
81
82      quit = pause = false; // player has not quit or paused the game
83      time = 0; // initialize the time since Ball was reset to 0
84   } // end Pong constructor
85
86   // Pong default destructor erases objects contained in a Pong object
87   Pong::~Pong()
88   {
89      // free dynamically allocated memory for Keyboard
90      if ( keyboardPtr )
91      {
92         delete keyboardPtr; // release memory pointer points to
93         keyboardPtr = 0; // point pointer at 0
94         OIS::InputManager::destroyInputSystem( inputManagerPtr );
95      } // end if
96
```

Fig. 21.4 | Pong class member function definitions. (Part 2 of 6.)

```
97      // free dynamically allocated memory for Root
98      if ( rootPtr )
99      {
100        delete rootPtr; // release memory pointer points to
101        rootPtr = 0; // point pointer at 0
102     } // end if
103
104     // free dynamically allocated memory for Ball
105     if ( ballPtr )
106     {
107        delete ballPtr; // release memory pointer points to
108        ballPtr = 0; // point pointer at 0
109     } // end if
110
111     // free dynamically allocated memory for Paddle
112     if ( leftPaddlePtr )
113     {
114        delete leftPaddlePtr; // release memory pointer points to
115        leftPaddlePtr = 0; // point pointer at 0
116     } // end if
117
118     // free dynamically allocated memory for Paddle
119     if ( rightPaddlePtr )
120     {
121        delete rightPaddlePtr; // release memory pointer points to
122        rightPaddlePtr = 0; // point pointer at 0
123     } // end if
124  } // end Pong default constructor
125
126  // create the scene to be displayed
127  void Pong::createScene()
128  {
129     // get a pointer to the Score Overlay
130     Overlay *scoreOverlayPtr =
131        OverlayManager::getSingleton().getByName( "Score" );
132     scoreOverlayPtr->show(); // show the Overlay
133
134     // make the game objects
135     ballPtr = new Ball( sceneManagerPtr ); // make the Ball
136     ballPtr->addToScene(); // add the Ball to the scene
137     rightPaddlePtr = new Paddle( sceneManagerPtr, "RightPaddle", 90 );
138     rightPaddlePtr->addToScene(); // add a Paddle to the scene
139     leftPaddlePtr = new Paddle( sceneManagerPtr, "LeftPaddle", -90 );
140     leftPaddlePtr->addToScene(); // add a Paddle to the scene
141
142     // create the walls
143     Entity *entityPtr = sceneManagerPtr->
144        createEntity( "WallLeft", "cube.mesh" ); // create the left wall
145     entityPtr->setMaterialName( "wall" ); // set material for left wall
146     entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
147
148     // create the SceneNode for the left wall
149     SceneNode *nodePtr = sceneManagerPtr->getRootSceneNode()->
```

Fig. 21.4 | Pong class member function definitions. (Part 3 of 6.)

```
150       createChildSceneNode( "WallLeft" );
151    nodePtr->attachObject( entityPtr ); // attach left wall to SceneNode
152    nodePtr->setPosition( -95, 0, 0 ); // set the left wall's position
153    nodePtr->setScale( .05, 1.45, .1 ); // set the left wall's size
154    entityPtr = sceneManagerPtr->createEntity( "WallRight", "cube.mesh" );
155    entityPtr->setMaterialName( "wall" ); // set material for right wall
156    entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
157
158    // create the SceneNode for the right wall
159    nodePtr = sceneManagerPtr->getRootSceneNode()->
160       createChildSceneNode( "WallRight" );
161    nodePtr->attachObject( entityPtr ); // attach right wall to SceneNode
162    nodePtr->setPosition( 95, 0, 0 ); // set the right wall's position
163    nodePtr->setScale( .05, 1.45, .1 ); // set the right wall's size
164    entityPtr = sceneManagerPtr->createEntity( "WallBottom", "cube.mesh" );
165    entityPtr->setMaterialName( "wall" ); // set material for bottom wall
166    entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
167
168    // create the SceneNode for the bottom wall
169    nodePtr = sceneManagerPtr->getRootSceneNode()->
170       createChildSceneNode( "WallBottom" );
171    nodePtr->attachObject( entityPtr ); // attach bottom wall to SceneNode
172    nodePtr->setPosition( 0, -70, 0 ); // set the bottom wall's position
173    nodePtr->setScale( 1.9, .05, .1 ); // set bottom wall's size
174    entityPtr = sceneManagerPtr->createEntity( "WallTop", "cube.mesh" );
175    entityPtr->setMaterialName( "wall" ); // set the material for top wall
176    entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
177
178    // create the SceneNode for the top wall
179    nodePtr = sceneManagerPtr->getRootSceneNode()->
180       createChildSceneNode( "WallTop" );
181    nodePtr->attachObject( entityPtr ); // attach top wall to the SceneNode
182    nodePtr->setPosition( 0, 70, 0 ); // set the top wall's position
183    nodePtr->setScale( 1.9, .05, .1 ); // set the top wall's size
184 } // end function createScene
185
186 // start a game of Pong
187 void Pong::run()
188 {
189    createScene(); // create the scene
190    rootPtr->startRendering(); // start rendering frames
191 } // end function run
192
193 // update the score text
194 void Pong::updateScore()
195 {
196    char score[ 25 ]; // text to be displayed
197
198    sprintf( score, "Player 2 Score: %i", player2Score ); // make the text
199    OverlayElement *textElementPtr =
200       static_cast< OverlayElement* >( OverlayManager::getSingletonPtr()->
201       getOverlayElement( "right" ) ); // get the right player's TextArea
202    textElementPtr->setCaption( score ); // set the TextElement's text
```

Fig. 21.4 | Pong class member function definitions. (Part 4 of 6.)

```
203
204    sprintf( score, "Player 1 Score: %i", player1Score ); // make the text
205    textElementPtr =
206       static_cast< OverlayElement* >( OverlayManager::getSingletonPtr()->
207       getOverlayElement( "left" ) ); // get the left player's TextArea
208    textElementPtr->setCaption( score ); // set the TextElement's text
209
210    wait = true; // the game should wait to restart Ball
211 } // end function updateScore
212
213 // respond to user keyboard input
214 bool Pong::keyPressed( const OIS::KeyEvent &keyEventRef )
215 {
216    // if the game is not paused
217    if ( !pause )
218    {
219       // process KeyEvents that apply when the game is not paused
220       switch ( keyEventRef.key )
221       {
222       case OIS::KC_ESCAPE: // escape key hit: quit
223          quit = true;
224          break;
225       case OIS::KC_UP: // up-arrow key hit: move the right Paddle up
226          rightPaddlePtr->movePaddle( PADDLE_UP );
227          break;
228       case OIS::KC_DOWN: //down-arrow key hit: move the right Paddle down
229          rightPaddlePtr->movePaddle( PADDLE_DOWN );
230          break;
231       case OIS::KC_A: // A key hit: move the left Paddle up
232          leftPaddlePtr->movePaddle( PADDLE_UP );
233          break;
234       case OIS::KC_Z: // Z key hit: move the left Paddle down
235          leftPaddlePtr->movePaddle( PADDLE_DOWN );
236          break;
237       case OIS::KC_F: // F key hit: increase the Ball's speed
238          ballPtr->changeSpeed( SPEED_INCREMENT );
239          break;
240       case OIS::KC_S: // S key hit: decrease the Ball's speed
241          ballPtr->changeSpeed( SPEED_DECREMENT );
242          break;
243       case OIS::KC_P: // P key hit: pause the game
244          pause = true; // set pause to true when the user pauses the game
245          Overlay *pauseOverlayPtr =
246             OverlayManager::getSingleton().getByName( "PauseOverlay" );
247          pauseOverlayPtr->show(); // show the pause Overlay
248          break;
249       } // end switch
250    } // end if
251    else // game is paused
252    {
253       // user hit 'R' on the keyboard
254       if ( keyEventRef.key == OIS::KC_R )
255       {
```

Fig. 21.4 | Pong class member function definitions. (Part 5 of 6.)

```
256          pause = false; // set pause to false when user resumes the game
257          Overlay *pauseOverlayPtr =
258             OverlayManager::getSingleton().getByName( "PauseOverlay" );
259          pauseOverlayPtr->hide(); // hide the pause Overlay
260       } // end if
261    } // end else
262    return true;
263 } // end function keyPressed
264
265 // process key released events
266 bool Pong::keyReleased( const OIS::KeyEvent &keyEventRef )
267 {
268    return true;
269 } // end function keyReleased
270
271 // return true if the program should render the next frame
272 bool Pong::frameEnded( const FrameEvent &frameEvent )
273 {
274    return !quit; // quit = false if the user hasn't quit yet
275 } // end function frameEnded
276
277 // process game logic, return true if the next frame should be rendered
278 bool Pong::frameStarted( const FrameEvent &frameEvent )
279 {
280    keyboardPtr->capture(); // get keyboard events
281    // the game is not paused and the Ball should move
282    if ( !wait && !pause )
283    {
284       // move the Ball
285       ballPtr->moveBall( frameEvent.timeSinceLastFrame );
286       ballPtr->hitPaddle(); // check if the Ball hit a Paddle
287    } // end if
288    // don't move the Ball if wait is true
289    else if ( wait )
290    {
291       // increase time if it is less than 4 seconds
292       if ( time < 4 )
293          // add the seconds since the last frame
294          time += frameEvent.timeSinceLastFrame;
295       else
296       {
297          wait = false; // shouldn't wait to move the Ball anymore
298          time = 0; // reset the control variable to 0
299       } // end else
300    } // end else
301
302    return !quit; // quit = false if the user hasn't quit yet
303 } // end function frameStarted
```

Fig. 21.4 | Pong class member function definitions. (Part 6 of 6.)

use (Direct3D or OpenGL), resolution, color depth, full-screen mode, and other options that are beyond the scope of this chapter. Direct3D is exclusively for Windows. OpenGL is supported on all major platforms. The resolution is defined by two values, width and

Fig. 21.5 | **OGRE Engine Rendering Setup** dialog box.

height, which determine the number of pixels used to draw the scene. The resolution options for both rendering subsystems range from 640 × 400 to 1680 × 1050. A higher resolution will produce more detailed graphics. If you choose to turn off full-screen mode, the resolution will also determine the size of the window in which the game is displayed. We run the game at a resolution of 800 × 600. A **color depth** of n-bits means that there are 2^n possible colors that can be displayed on the screen. A lower color depth will make the program require less memory, but the graphics may not be as good as with a higher color depth. Direct3D and OpenGL each support 16-bit and 32-bit color depths.

To display the dialog box, you must create a **`Root` object** (Fig. 21.4, line 28)—the base object in Ogre used to start the engine. No other Ogre calls can be made until the `Root` object has been created. Next, call the `showConfigDialog` function of the `Root` class (line 31) to display the dialog. Ogre will save the settings and use them as the default settings the next time the dialog box is displayed.

Once the rendering subsystem and window options have been set, we can create the **`RenderWindow`** (line 34), a window in which Ogre will render graphics. The first parameter, `true`, tells Ogre to create the window with the settings the user chose in the dialog box. Passing `false` to this parameter allows you to manually create the window at a later time. The second, `"Pong"`, is the name of the window within the engine and will also appear in the title bar of the window if it is not full screen. Notice the British spelling of `initialise`, reflecting Ogre's origin in the U.K.

21.4.2 Creating a Scene

Now that we've initialized Ogre and set up a window to render our graphics in, we need to add objects to create our **scene**—the collection of images that we display on the screen.

SceneManager

To control the scene we use Ogre's **SceneManager object** (line 37). The SceneManager manages the **scene graph**, a data structure that contains all the objects in the scene, both visible and non-visible. The SceneManager is used to create objects that will be added to the scene graph and determines which objects will be rendered. Excluding objects that are not within the visible scene from being rendered, known as **culling**, decreases rendering time and increases performance. This is done automatically. We'll keep a pointer to this SceneManager object as it will be used extensively throughout the game.

There are several types of SceneManagers which are designed to handle different types of scenes, such as indoor scenes or expansive landscape scenes. An Ogre application can use more than one SceneManager, separately or at the same time. For the purposes of this chapter, we use only one instance of the generic scene type (ST_GENERIC), an Ogre SceneManager that is not optimized for any particular type of scene.

Camera

Once we have a SceneManager we can start constructing our scene. First we add a Camera. A **Camera** in Ogre is the eye through which you view the scene. A 3D scene is usually too large to be displayed in one window. The Camera looks into the scene and tells Ogre what part you can actually see. Cameras can be placed at any location in the scene or attached to SceneNodes—we'll discuss these shortly. If attached to a SceneNode, the Camera will follow that node if it moves within the scene. Ogre supports multiple Cameras in a single scene, but we need only one.

We use the SceneManager to create the Camera (line 40),then we set the position, orientation, clip distances and Viewport (lines 41–48). Position is the location of the Camera within the scene. We position the Camera 200 units from the origin along the positive z-axis, towards the player. This places the Camera far enough away from the origin of the scene to be able to center the game around it. That way we can set all of the z-coordinates to 0 and not have to change them. The orientation is the direction in which the Camera is looking. We have the Camera look at the origin (line 42) because we center the game around that point. The clip distances define how near and how far the Camera can see. If something is closer to the Camera than the near clip distance, then the Camera won't be able to see it. If something is farther away than the far clip distance, then it is too far for the Camera to see. The **Viewport** is the area that the Camera can see. We set the Viewport's background color to white. Note that a Camera can have more than one Viewport, but we will use only one for our game. Cameras have many other features and functions, but this is all we need for our game.

Light

One of the most important aspects of a 3D scene is lighting. Ogre has three types of **Lights**—Point, Spot and Directional. **Point lights** are common—they have a position in space and radiate light in all directions. **Spot lights** are much like a spot light—they have a position in space like Point lights, but radiate light in only one direction; the strength of the light fades as the distance from the source increases. **Directional lights**, unlike the other types, do not have a position in space, they have only a direction in which they shine—the light is assumed to come from the same direction no matter where you are in the scene.

We use a `Point` light in our game. Lines 58–59 use the `SceneManager` to create the `Light` and set its position within the scene. The argument to the `createLight` function is the name by which we'll refer to the `Light`. We set the `Light`'s position by specifying its *x*-, *y*- and *z*-coordinates. Our scene is now ready for use.

21.4.3 Adding to the Scene

As we mentioned earlier, Pong has four major game objects—a `Ball`, two `Paddles` and a rectangular box. All of these elements must be added to the scene before they can be displayed.

Add the Ball

Class `Ball` (Figs. 21.6–21.7) represents the ball that bounces around the screen. We have to add the `Ball` to the scene before it can be displayed. The member function `addToScene` (Fig. 21.7, lines 26–51) creates an `Entity` that represents the `Ball`, adds it to the scene and creates some sounds associated with the `Ball`; we'll discuss the sounds later. An **`Entity`** is an instance of a mesh within the scene. A **mesh** is a file that contains the geometry information of a 3D model. Many `Entity` objects can be based on the same mesh as long as each `Entity` has a unique name. Lines 29–30 create the `Entity`, which is referenced through a pointer. The first argument is the name of the `Entity`. The second argument, `"sphere.mesh"`, is the mesh file used to determine the `Entity`'s shape. We use the sphere mesh provided with the Ogre SDK. You can find the mesh in the `OgreSDK\media\models` folder on your computer.

```
1  // Ball.h
2  // Ball class definition (represents a bouncing ball).
3  #ifndef BALL_H
4  #define BALL_H
5
6  #include <OgreAL.h> // OgreAL class definition
7  #include <Ogre.h> // Ogre class definition
8  using namespace Ogre; // use the Ogre namespace
9
10 class Paddle; // forward declaration of class Paddle
11
12 class Ball
13 {
14 public:
15    Ball( SceneManager *sceneManagerPtr ); // constructor
16    ~Ball(); // destructor
17    void addToScene(); // add the Ball to the scene
18    void moveBall( Real time ); // move the Ball across the screen
19    void hitPaddle(); // control the Ball hitting the Paddles
20    void changeSpeed( int change ); // change the speed of the Ball
21 private:
22    SceneManager *sceneManagerPtr; // pointer to the SceneManager
23    SceneNode *nodePtr; // pointer to the SceneNode
24    OgreAL::SoundManager *soundManagerPtr; // pointer to the SoundManager
25    OgreAL::Sound *wallSoundPtr; // sound played when Ball hits a wall
```

Fig. 21.6 | `Ball` class definition (represents a bouncing ball). (Part 1 of 2.)

```
26      OgreAL::Sound *paddleSoundPtr; // sound played when Ball hits a Paddle
27      OgreAL::Sound *scoreSoundPtr; // sound played when someone scores
28      int speed; // speed of the Ball
29      Vector3 direction; // direction of the Ball
30
31      // private utility functions
32      void reverseHorizontalDirection(); // change horizontal direction
33      void reverseVerticalDirection(); // change vertical direction
34   }; // end class Ball
35
36   #endif // BALL_H
```

Fig. 21.6 | `Ball` class definition (represents a bouncing ball). (Part 2 of 2.)

```
1   // Ball.cpp
2   // Ball class member function definitions.
3   #include "Ball.h" // Ball class definition
4   #include "Paddle.h" // Paddle class definition
5   #include "Pong.h" // Pong class definition
6   #include <OgreAL.h> // OgreAL class definition
7   #include <Ogre.h> // Ogre class definition
8   using namespace Ogre; // use the Ogre namespace
9
10  // Ball constructor
11  Ball::Ball( SceneManager *ptr )
12  {
13     sceneManagerPtr = ptr; // set pointer to the SceneManager
14     soundManagerPtr = new OgreAL::SoundManager(); // create SoundManager
15     speed = 100; // speed of the Ball
16     direction = Vector3( 1, -1, 0 ); // direction of the Ball
17  } // end Ball constructor
18
19  // Ball destructor
20  Ball::~Ball()
21  {
22     // empty body
23  } // end Ball destructor
24
25  // add the Ball to the scene
26  void Ball::addToScene()
27  {
28     // create Entity and attach it to a node in the scene
29     Entity *entityPtr =
30        sceneManagerPtr->createEntity( "Ball", "sphere.mesh" );
31     entityPtr->setMaterialName( "ball" ); // set material for the Ball
32     entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
33     nodePtr = sceneManagerPtr->getRootSceneNode()->
34        createChildSceneNode( "Ball" ); // create a SceneNode
35     nodePtr->attachObject( entityPtr ); // attach the Entity to SceneNode
36     nodePtr->setScale( .05, .05, .05 ); // scale SceneNode
37
```

Fig. 21.7 | `Ball` class member function definitions. (Part 1 of 4.)

```
   // attach sounds to Ball so they will play from where Ball is
   wallSoundPtr = soundManagerPtr->
      createSound( "boing", "boing.wav", false );
   nodePtr->attachObject( wallSoundPtr ); // attach a sound to SceneNode
   paddleSoundPtr = soundManagerPtr->
      createSound( "boing1", "boing1.wav", false ); // create a Sound
   nodePtr->attachObject( paddleSoundPtr ); // attach sound to SceneNode
   scoreSoundPtr = soundManagerPtr->
      createSound( "cheer", "cheer.wav", false ); // create a Sound

   // attach the score sound to its own node centered at ( 0, 0, 0 )
   sceneManagerPtr->getRootSceneNode()->createChildSceneNode( "score" )->
      attachObject( scoreSoundPtr );
} // end function addToScene

// move the Ball across the screen
void Ball::moveBall( Real time )
{
   nodePtr->translate( ( direction * ( speed * time ) ) ); // move Ball
   Vector3 position = nodePtr->getPosition(); // get Ball's new position

   // check if the Ball hit the left side
   if ( ( position.x  - 5 ) <= -92.5 )
   {
      nodePtr->setPosition( 0, 0, 0 ); // place Ball in center of screen
      ++Pong::player2Score; // give player 2 a point
      Pong::updateScore(); // update the score
      scoreSoundPtr->play(); // play a sound when player 2 scores
   } // end if
   else if ( ( position.x + 5 ) >= 92.5 ) // if Ball hit right side
   {
      nodePtr->setPosition( 0, 0, 0 ); // place Ball in center of screen
      ++Pong::player1Score; // give player 1 a point
      Pong::updateScore(); // update the score
      scoreSoundPtr->play(); // play a sound when player 1 scores
   } // end else

   // see if the Ball hit the bottom wall
   else if ( ( position.y  - 5 ) <= -67.5 && direction.y < 0 )
   {
      // place the Ball on the bottom wall
      nodePtr->setPosition( position.x, ( -67.5 + 5 ), position.z );
      reverseVerticalDirection(); // make the Ball start moving up
   } // end else
   // see if the Ball hit the top wall
   else if ( ( position.y + 5 ) >= 67.5 && direction.y > 0 )
   {
      // place the Ball on the top wall
      nodePtr->setPosition( position.x, ( 67.5 - 5 ), position.z );
      reverseVerticalDirection(); // make the Ball start moving down
   } // end else
} // end function moveBall

```

Fig. 21.7 | `Ball` class member function definitions. (Part 2 of 4.)

```
91  // reverse the Ball's horizontal direction
92  void Ball::reverseHorizontalDirection()
93  {
94     direction *= Vector3( -1, 1, 1 ); // reverse the horizontal direction
95     paddleSoundPtr->play(); // play the "boing1" sound effect
96  } // end function reverseHorizontalDirection
97
98  // reverse the Ball's vertical direction
99  void Ball::reverseVerticalDirection()
100 {
101    direction *= Vector3( 1, -1, 1 ); // reverse the vertical direction
102    wallSoundPtr->play(); // play the "boing" sound effect
103 } // end function reverseVerticalDirection
104
105 // control the Ball colliding with the Paddle
106 void Ball::hitPaddle()
107 {
108    // get the position of the Paddles and the Ball
109    Vector3 leftPaddlePosition = sceneManagerPtr->
110       getSceneNode( "LeftPaddle" )->getPosition();  // left Paddle
111    Vector3 rightPaddlePosition = sceneManagerPtr->
112       getSceneNode( "RightPaddle" )->getPosition(); // right Paddle
113    Vector3 ballPosition = getPosition(); // get the Ball's position
114
115    // is the Ball in range of the left Paddle?
116    if ( ( ballPosition.x - 5 ) < ( leftPaddlePosition.x + 1 ) )
117    {
118       // check for collision with left Paddle
119       if ( ( ballPosition.y - 5 ) < ( leftPaddlePosition.y + 15 ) &&
120          ( ballPosition.y + 5 ) > ( leftPaddlePosition.y - 15 ) )
121       {
122          reverseHorizontalDirection(); // reverse the Ball's direction
123
124          // place the Ball at the edge of the Paddle
125          nodePtr->setPosition( ( leftPaddlePosition.x + 6 ),
126             ballPosition.y, ballPosition.z );
127       } // end if
128    } // end if
129    // is the Ball in range of the right Paddle
130    else if ( ( ballPosition.x + 5 ) > ( rightPaddlePosition.x - 1 ) )
131    {
132       // check for collision with right Paddle
133       if ( ( ballPosition.y - 5 ) < ( rightPaddlePosition.y + 15 ) &&
134          ( ballPosition.y + 5 ) > ( rightPaddlePosition.y - 15 ) )
135       {
136          reverseHorizontalDirection(); // reverse the Ball's direction
137
138          // place the Ball at the edge of the Paddle
139          nodePtr->setPosition( ( rightPaddlePosition.x - 6 ),
140             ballPosition.y, ballPosition.z );
141       } // end if
142    } // end else
143 } // end function hitPaddle
```

Fig. 21.7 | `Ball` class member function definitions. (Part 3 of 4.)

```
144
145 // change the Ball's speed
146 void Ball::changeSpeed( int change )
147 {
148    if ( speed <= 200 && speed >= 10 ) // Ball's minimum and maximum speed
149       speed += change;
150    if ( speed < 10 ) // keep Ball from going below minimum speed
151       speed = 10;
152    else if ( speed > 200 ) // keep Ball from going above maximum speed
153       speed = 200;
154 } // end function changeSpeed
```

Fig. 21.7 | `Ball` class member function definitions. (Part 4 of 4.)

Line 31 sets the material used to color the `Entity`. The argument, `"ball"`, is the name of the material used to color the `Ball`. In Ogre, a material is usually created with a script, though it can also be created with code directly in the program. A material **script** (Fig. 21.8) is a text file that Ogre uses to create a material. Be sure you save the text file with a `.material` extension.

Let's take a look at the `ball` material. You'll notice that the structure looks similar to C++ code, with curly braces enclosing each section. Line 2 indicates that we are defining a `material` called `ball`. Within the material, we define a `technique`—how the object will be rendered (lines 5–15). You can define multiple `techniques` for a material, but that is beyond our scope. Within each `technique`, one or more `passes` is defined (lines 8–14). Each `pass` defines a single step in the rendering process for this material. Using multiple `passes` is beyond our scope. The color is determined in the `pass` by setting color values for the different types of lighting in the scene. We want the `Ball` to be violet, so set the color values to (0.58, 0, 0.827) (lines 11–13). The numbers after each type of light (ambient, diffuse and specular) represent color values.

Note that `"ball"` is not the name of the file that the `material` is defined in, but rather the name of the `material` within that file. A `material` file can define multiple `materials`.

```
1  // ball material definition
2  material ball
3  {
4     // define one technique for rendering the Ball
5     technique
6     {
7        // render the Ball in one pass
8        pass
9        {
10          // color the Ball violet
11          ambient 0.58 0 0.827
12          diffuse 0.58 0 0.827
13          specular 0.58 0 0.827 120
14       }
15    }
16 }
```

Fig. 21.8 | `Ball` material script.

The same `material` can be used for multiple `Entity` objects, but each `material` defined must have a unique name.

We've created the `Entity` for our `Ball`, but it is not part of the scene yet. Lines 33–35 (Fig. 21.7) add it to the scene so it will be rendered on the screen. We use the `SceneManager` to create a **SceneNode** (lines 33–34)—a `Node` within the scene graph that holds information about an object and its position in the scene, visible or non-visible. A `SceneNode` may have many `ChildNodes` attached to it, but can have only one parent `Node`. The argument, `"Ball"`, is the name by which the `SceneNode` will be referred to in the Ogre engine. Every `SceneNode` in the scene graph must have a unique name. The call to `getRootSceneNode` retrieves the top-most node within the scene graph. The **root node** is the parent of all other nodes. When you create a child of the root node, its initial position is (0, 0, 0). Line 35 attaches the `Entity` representing the `Ball` to the newly created `SceneNode`. The `Ball` is now part of the scene and will be rendered. Notice that all of the functions used to add the `Ball` to the scene are member functions of the `SceneManager`. That is why the `Ball` constructor takes a pointer to the `SceneManager` as a parameter—the class must be able to access the `SceneManager` to add the `Ball` object to the scene.

The sphere mesh provided with the Ogre SDK has a radius of 100. This is much larger than we need. Line 36 changes the size of the `Entity` attached to the `SceneNode`, but it does not affect the size of the actual mesh that the node's `Entity` is based on. We supply a scaling factor for each axis (*x*, *y* and *z*). We pass .05 as the scaling factor for all three axes. Using the same scaling factor for all three axes uniformly scales the mesh so it maintains its original shape. The function multiplies the radius of the sphere on each axis by the scaling factor to change the radius from 100 to 5. When you scale a mesh, the lighting effects become somewhat distorted. We fix that by having the `Entity` calculate the new normals for the mesh each time it is scaled (line 32). A **normal** in this case refers to the direction the surface of the object is facing. If the surface is facing towards the light, it is brighter. If it's facing away from the light, it is darker.

There is also a function `scale` that will change the size of the `SceneNode`. The difference is that `scale` changes the size based on the current size while `setScale` changes it based on the original size of the node. These functions also scale all children of the `SceneNode` by the same factor. This can be changed by telling each child of the parent node that you do not wish to have it scaled when the parent is scaled—to do that, call the `setInheritScale` function and pass it `false`.

Add the Paddles

Class `Paddle` (Figs. 21.9–21.10) represents the `Paddles`. We add a `Paddle` to the scene in much the same way that we added the `Ball`. The member function `addToScene` (Fig. 21.10, lines 23–34) uses the same first five Ogre functions, but with different arguments.

```
1  // Paddle.h
2  // Paddle class definition (represents a paddle in the game).
3  #ifndef PADDLE_H
4  #define PADDLE_H
5
```

Fig. 21.9 | `Paddle` class definition (represents a paddle in the game). (Part 1 of 2.)

```
#include <Ogre.h> // Ogre class definition
using namespace Ogre; // use the Ogre namespace

class Paddle
{
public:
   // constructor
   Paddle( SceneManager *sceneManagerPtr, String paddleName,
   int positionX );
   ~Paddle(); // destructor
   void addToScene(); // add a Paddle to the scene
   void movePaddle( Vector3 direction ); // move a Paddle
   Vector3 getPosition(); // get the position of a Paddle
private:
   SceneManager* sceneManagerPtr; // pointer to the SceneManager
   SceneNode *nodePtr; // pointer to a SceneNode
   String name; // name of the Paddle
   int x; // x-coordinate of the Paddle
}; // end of class Paddle

#endif // PADDLE_H
```

Fig. 21.9 | `Paddle` class definition (represents a paddle in the game). (Part 2 of 2.)

```
// Paddle.cpp
// Paddle class member function definitions.
#include "Paddle.h" // Paddle class definition
#include <Ogre.h> // Ogre class definition
using namespace Ogre; // use the Ogre namespace

// constructor
Paddle::Paddle( SceneManager *ptr, String paddleName,
   int positionX )
{
   sceneManagerPtr = ptr; // set the pointer to the SceneManager
   name = paddleName; // set the Paddle's name
   x = positionX; // set the Paddle's x-coordinate
} // end Paddle constructor

// destructor
Paddle::~Paddle()
{
   // empty body
} // end Paddle default destructor

// add the Paddle to the scene
void Paddle::addToScene()
{
   Entity *entityPtr = sceneManagerPtr->
      createEntity( name, "cube.mesh" ); // create an Entity
   entityPtr->setMaterialName( "paddle" ); // set the Paddle's material
   entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
```

Fig. 21.10 | `Paddle` class member function definitions. (Part 1 of 2.)

```
29      nodePtr = sceneManagerPtr->getRootSceneNode()->
30         createChildSceneNode( name ); // create a SceneNode for the Paddle
31      nodePtr->attachObject( entityPtr ); // attach Paddle to the SceneNode
32      nodePtr->setScale( .02, .3, .1 ); // set the Paddle's size
33      nodePtr->setPosition( x, 0, 0 ); // set the Paddle's position
34   } // end function addToScene
35
36   // move the Paddle up and down the screen
37   void Paddle::movePaddle( Vector3 direction )
38   {
39      nodePtr->translate( direction ); // move the Paddle
40      if ( nodePtr->getPosition().y > 52.5 ) // top of the box
41         nodePtr->setPosition( x, 52.5, 0 ); // place Paddle at top of box
42      else if ( nodePtr->getPosition().y < -52.5 ) // bottom of the box
43
44         // place the Paddle at the bottom of the box
45         nodePtr->setPosition( x, -52.5, 0 );
46   } // end function movePaddle
47
48   // return the position of the Paddle
49   Vector3 Paddle::getPosition()
50   {
51      return nodePtr->getPosition(); // get the position of the Paddle
52   } // end function getPosition
```

Fig. 21.10 | `Paddle` class member function definitions. (Part 2 of 2.)

First we create an `Entity` to represent the `Paddle` on the screen (lines 25–26). Here we use the name supplied to the constructor as the `Entity` name. We can't just use `"Paddle"` like we used `"Ball"` because each `Entity` must have a unique name, and there are two `Paddle`s in the game. We use the cube mesh provided with the Ogre SDK as the model for the `Paddle`. The cube mesh is located in the `OgreSDK\media\models` folder. We color both `Paddle`s dark orange with the same `material` (Fig. 21.11). This `material`

```
1  // paddle material definition
2  material paddle
3  {
4     // define one technique for rendering a Paddle
5     technique
6     {
7        // render a Paddle in one pass
8        pass
9        {
10          // render a Paddle in one pass
11          ambient 1 0.549 0
12          diffuse 1 0.549 0
13          specular 1 0.549 0 120
14       }
15    }
16 }
```

Fig. 21.11 | `Paddle` material script.

script is almost identical to the script used for the ball. The only differences are the name of the `material` (line 2) and the color values (lines 11–13).

Then we create a `ChildNode` of the root node to hold the data for the `Paddle` (Fig. 21.10, lines 29–30). We use the name provided to the constructor for the `Node`'s name as we did for the `Entity`. This is allowed because `Node`s and `Entity` objects are separate types, so there is not a name conflict.

Next we attach the `Entity` to the node (line 31) and scale the node to an appropriate size (line 32). The cube mesh is $100 \times 100 \times 100$, but we scale it to $2 \times 30 \times 10$ to make it an appropriate size for a `Paddle`. We also set the `Entity` to recalculate its normals (line 28) as we did with the `Ball`. The only new Ogre function we use is `setPosition` (line 33). This function places the node at the given coordinates in the scene. We didn't need to use this function in the `Ball` class because we wanted `Ball`'s `SceneNode` to start at (0, 0, 0), which is the default position of any node attached to the root node. We want the `Paddle` to be positioned at the edge of the screen, so we have to move it there. In line 33, `x` is a data member of class `Paddle` that defines the `Paddle`'s *x*-coordinate.

Add the Walls

Now we will add the box contains the bouncing `Ball` and the moving `Paddle`s. We create this box in the `createScene` function of class `Pong` (Fig. 21.4, lines 143–183). We use the same cube mesh, provided with the Ogre SDK, for all four walls—scaling the walls appropriately to make the box and recalculating the normals for lighting. The walls are added to the scene similarly to the `Ball` and `Paddle`s. We create an `Entity` using the cube mesh to represent each wall. We use a simple material to color all the walls cyan (Fig. 21.12). The material script looks just like the other two we have seen, only the name and color values differ.

Now position and scale the walls. The left and right walls are each placed 95 units from the origin in the *x*-direction. The top and bottom walls are each placed 70 units from the origin in the *y*-direction. Each wall is then scaled to the correct size. The top and bottom walls are positioned 140 units apart in the *y*-direction. We give both a width of 5 units. This width is an arbitrary value. You can change the width to make the game look

```
1  // wall material definition
2  material wall
3  {
4     // define one technique for rendering a wall
5     technique
6     {
7        // render a wall in one pass
8        pass
9        {
10          // color the wall cyan
11          ambient 0 0.545 0.545
12          diffuse 0 0.545 0.545
13          specular 0 0.545 0.545 120
14       }
15    }
16 }
```

Fig. 21.12 | Wall material script.

how you want. If you change this, you'll also have to change the collision detection code—you'll see why when we discuss the collision logic. For the left and right walls to stretch between the top and bottom walls, they must be 145 units long (140 plus the half width of each wall). So the *x*-scaling factor for the left and right walls is 1.45. The left and right walls are also given a width of 5 units and are positioned 185 units apart in the *x*-direction. For the top and bottom walls to stretch between the left and right walls, they must be 190 units long, so their *y*-scaling factor is 1.9.

Add the Text

We will use text to display the score of the game. Ogre generally uses `Overlays` to display text. An **`Overlay`** refers to anything you want to render on top of the 3D elements of the scene. We use `Overlays` only for text in this chapter. The `Overlay` is defined by a script saved in a `.overlay` file.

`Overlays` are composed of `OverlayElements`. The first element in an `Overlay` must be an `OverlayContainer`. An **`OverlayContainer`** can hold any type of `OverlayElement`. A **`TextAreaOverlayElement`** holds the text that will be displayed on the screen. Every object in an `Overlay` has three main attributes—metrics mode, position and size. The position is determined by the top-left corner of the object and is always relative to the parent `OverlayContainer` of the object. Size is determined by width and height. The **metrics mode** determines how the object is positioned and sized. **Pixel mode** will size the object based on the width and height declared in pixels. **Relative mode** will position and size the object relative to the size of the parent `OverlayContainer` (or the window if it is the outermost `OverlayContainer`). In relative mode, size and position values range from 0.0 to 1.0—think of it as a percentage of the parent `OverlayElement`'s size. If you position an element at (0.0, 0.0) it will be at the top-left corner of the parent element, (0.5, 0.0) would be 50% across the top.

To display the score, we create an `Overlay` (Fig. 21.13). Line 2 names the `Overlay` Score. A single overlay file can hold several Overlay definitions. Ogre will reference each `Overlay` by the name rather than the file. The ***z*-order** of the `Overlay` (line 5) is used to define what this `Overlay` should be rendered over. When using multiple `Overlays`, an `Overlay` with a higher *z*-order will be rendered on top of an `Overlay` with a lower *z*-order. Lines 8–58 create a **`PanelOverlayElement`** container that holds two `TextAreaOverlayElements`. The `OverlayContainer` is positioned in the top-left corner of the screen (lines 13–14) and runs along the entire width (line 17). The container is 10% of the height of the window (line 18). The first `TextAreaOverlayElement` (lines 21–38) is positioned at the top of the container 5% away from the left side, runs half the width and is the same height as the container (lines 28–31). The other `TextAreaOverlayElement` (lines 40–57) is positioned 69% of the way across the top of the container and runs to the end. The `TextAreaOverlayElements` also declare a font to use (which is defined by a script in a `.fontdef` file in the `PongResources` folder), the character height, font color (again note the British spelling, "colour") and the caption (lines 34–37 and 53–56).

Figure 21.14 is the `.fontdef` file that defines the `BlueBold` font. Line 2 gives the font a name that Ogre will refer to. Line 5 tells Ogre what type of font it is. True type is a common font file format (a `.ttf` file). The source (line 8) is the file that contains the font. We put the `.ttf` file in the same folder as the `.fontdef` file. If you place the two files in different locations you will have to specify the path to the `.ttf` file in line 8.

```
// An Overlay to display the score
Score
{
   // set a high z-order, displays on top of anything with lower z-order
   zorder 500

   // create a PanelOverlayElement container to hold the text areas
   container Panel(ScorePanel)
   {
      // use relative metrics mode to position this container at the
      // top-left corner of the screen
      metrics_mode relative
      left 0.0
      top 0.0

      // make it 1/10 the height and the full width of the screen
      width 1.0
      height .1

      // create a TextAreaOverlayElement to display player 1's score
      element TextArea(left)
      {
         // position and size the element relative to the container
         metrics_mode relative

         // position it at the top of the container 5% from the left and
         // make it the same height and half as long as the container
         left 0.05
         top 0.0
         width 0.5
         height 1.0

         // set font use for caption and set the size and color
         font_name BlueBold
         char_height .05
         colour 1.0 0 0
         caption Player 1 Score: 0
      }
      // create a TextAreaOverlayElement to display player 2's score
      element TextArea(right)
      {
         // position and size the element relative to the container
         metrics_mode relative

         // position it at the topof the container 69% from the left and
         // make it the same height as the container, stretch to the end
         left 0.69
         top 0.0
         width 0.5
         height 1.0

         // set font use for caption and set the size and color
         font_name BlueBold
```

Fig. 21.13 | `Overlay` script to display the score. (Part 1 of 2.)

```
54            char_height 0.05
55            colour 1.0 0 0
56            caption Player 2 Score: 0
57         }
58      }
59   }
```

Fig. 21.13 | `Overlay` script to display the score. (Part 2 of 2.)

```
1   // define the BlueBold font
2   BlueBold
3   {
4      // define the font type
5      type truetype
6
7      // set the source file for the font
8      source bluebold.ttf
9
10     // set the font size
11     size 16
12
13     // set the font resolution (96 is standard)
14     resolution 96
15  }
```

Fig. 21.14 | BlueBold font definition script.

Lines 130–132 of class Pong (Fig. 21.4) display the score on the screen. We use the static member function `getSingleton` of class `OverlayManager` to get a pointer to the `OverlayManager` object. We then use that object to get a pointer to the score `Overlay`, then we call the `show` function to display it on the screen. When a player scores, we need to update the text within the `Overlay` to reflect the change (lines 194–211). First we create the new text. Then we get a pointer to the appropriate `TextAreaOverlayElement` from the `OverlayManager` and use the `TextAreaOverlayElement` member function `setCaption` to replace the text.

21.4.4 Animation and Timers

Now that we know how to draw a `Ball` on the screen, animating it and making it move around the screen is straightforward. The function `moveBall` (Fig. 21.7, lines 54–89) moves the `Ball` around the screen. In most Pong games, the ball can travel at many different angles. However, since we are just starting out with Ogre, we want to keep things as simple as possible. For this reason, in our Pong game, the ball has only four possible directions of travel: down-right, up-right, down-left, and up-left—all at 45° angles to the *x*- and *y*-axes in our program.

Line 56 actually makes the `Ball` move; the rest of the function controls collisions with various objects within the scene, as we'll see shortly. The `translate` function takes a `Vector3` as an argument, which is a three-dimensional vector type defined by Ogre. The vector represents the direction and distance to move the `Ball`. We pass to the `translate` function the `Ball`'s direction multiplied by the distance to travel (`speed` × `time`) to deter-

mine the final vector. The `speed` parameter is the number of units the ball will move per second. The `time` parameter is the number of seconds since the last time the `Ball` was moved. We'll see where this comes from in just a moment. `SceneNode` translations are done in **parent space** by default. That means that the node is translated with respect to its parent node's position and orientation (i.e., the direction the node is facing). Translations can also be performed in world or local space by adding another parameter to the `translate` function (`TS_LOCAL` or `TS_WORLD`). Translations in **world space** are done with respect to the origin of the scene (0, 0, 0). Translations in **local space** are done with respect to the node's origin (where ever the node is positioned and which ever direction it is facing).[2]

To continuously move the `Ball` across the screen, we need to call the `moveBall` function when each new frame is rendered. Figure 21.3 defines the `Pong` class, our game's main driving class, which inherits from the Ogre class `FrameListener`. A **`FrameListener`** is a class that processes `Ogre::FrameEvents`. A **`FrameEvent`** occurs every time a frame begins or ends. Every `FrameListener` has two functions, `frameStarted` and `frameEnded` (lines 27–28), that each return a `bool`. Ogre keeps rendering frames until one of these functions returns `false`. We use the `frameStarted` function (Fig. 21.4, lines 278–303) to control the animation of our `Ball`, specifically line 285. This function is called by Ogre before each new frame is rendered. The `frameStarted` function calls the `Ball` class member function `moveBall` before every frame, which continuously moves the `Ball` across the screen. As discussed earlier, controlling the speed of the animation is vital to creating smooth motion. Frame rates (i.e., how quickly the scene gets redrawn) may vary greatly on different machines, so the `Ball` could move at a different speed on each one. For that reason we pass the `FrameEvent`'s data member `timeSinceLastFrame`, in seconds, to the `moveBall` function. We multiply this time by the `Ball`'s speed to determine the distance the `Ball` should move across the screen. This is an example of using a timer to control animation.

21.4.5 User Input

Now we discuss moving the `Paddle` up and down on the screen with the `movePaddle` function of class `Paddle` (Fig. 21.10, lines 37–46). To move the `Paddle`, we again use the `SceneNode` function `translate` (line 39). Rather than moving the `Paddle` based on time, we move it based on user input from the keyboard. The user specifies a direction, up or down, by pressing the corresponding key and the `Paddle` moves accordingly. The direction is passed to `movePaddle` as a `Vector3`.

Ogre does not directly support user input from devices such as the keyboard, mouse or joystick. However, the Ogre SDK comes with the **Object Oriented Input System (OIS)** for handling user input.

We need to create an `InputManager`, a `Keyboard` and a `KeyListener` to handle the user input and control the calls to `movePaddle`. The **`InputManager`** is used to create the various input devices. We create the `InputManager` in class `Pong`'s constructor (Fig. 21.4, line 70). To create the `InputManager` we must provide it with a window in which to collect (line 61–67) user input.

We create a **`Keyboard`** object which represents the actual keyboard. To collect `KeyEvents`, we must call the `capture` function of class `Keyboard`. We want to call this function repeatedly, so we place it in the `frameStarted` function which is called at the begin-

2. Junker, Gregory, *Pro OGRE 3D Programming*, 2006, pp. 82—89.

ning of every frame. Class `Pong` inherits from class `KeyListener`, an OIS class that handles input from the keyboard, we register it with the `Keyboard` (line 73) to receive `KeyEvents`, i.e., indications that the player has hit a key. A `KeyListener` defines two member functions (Fig. 21.3, lines 23–24)—we use only one of these (line 23). We must implement the other one, though, because they are both declared pure `virtual` in the class `KeyListener`.

The implementation of the key handling method is in lines 214–263. Every time we capture a key press, the `Keyboard` sends the `KeyEvent` to this member function. OIS defines an enumeration of all the keys on the keyboard which we use to determine which key was pressed. The `switch` statement (lines 220–249) responds only to certain keys. We extract the key enumeration from the `KeyEvent` and pass it to the `switch` statement (line 220). If the *A* or *Z* key is being pressed, the `Paddle` on the left side should move up or down, respectively. Likewise, if the user presses the up or down arrow keys, the `Paddle` on the right side should move in the corresponding direction. The directions passed to the `movePaddle` function are defined as constant `Vector3`s (lines 22–23).

We allow the user to pause the game by hitting the *P* key (lines 243–248), which sets the `Pong` data member `pause` to `true`. The `if` statement (line 217) will skip the switch statement that controls the `Paddle` movement when `pause` is `true`. The `pause` data member will also stop the `Ball` from moving when it is `true` (line 282). We also use an `Overlay` (Fig. 21.15) to display `"Game Paused"` on the screen. The game resumes when the player hits the *R* key.

```
1  // An Overlay to display "Game Paused" when the player pauses the game
2  PauseOverlay
3  {
4     // set a high z-order, displays on top of anything with lower z-order
5     zorder 500
6
7     // create a PanelOverlayElement container to hold the text area
8     container Panel(Pause)
9     {
10       // use relative metrics mode to position and size this container
11       metrics_mode relative
12
13       // place the container at the top-left corner of the window
14       left 0.0
15       top 0.0
16
17       // make the container the same size as the window
18       width 1.0
19       height 1.0
20
21       // create a TextAreaOverlayElement to display the text
22       element TextArea(pauseText)
23       {
24          // position and size the element relative to its container
25          metrics_mode relative
26
```

Fig. 21.15 | `Overlay` script to display the message `"Game Paused"` when player pauses the game. (Part 1 of 2.)

```
27            // center it vertically in the container
28            vert_align center
29
30            // put the left corner 4/10 from the left of the container and
31            // make it 2/10 the width of the container and 1/10 the height
32            left 0.4
33            width 0.2
34            height 0.1
35
36            // set the font used for caption and set the size and color
37            font_name BlueBold
38            char_height 0.05
39            colour 0 0 0
40            caption Game Paused
41         }
42      }
43   }
```

Fig. 21.15 | `Overlay` script to display the message `"Game Paused"` when player pauses the game. (Part 2 of 2.)

If the user hits the *Esc* key, the game exits by setting the `quit` data member to true (Fig. 21.4 lines 222–224). Recall that Ogre continues to render frames until the `frameStarted` or `frameEnded` function returns `false`. These both return `!quit`, so when we set `quit` to `true` the functions return `false` and tell Ogre to shut down. If you don't use the *Esc* key to quit, the program won't stop properly, it keeps running in the background. Be sure to use the *Esc* key.

21.4.6 Collision Detection

The `Ball` collides with a number of objects as it bounces around the screen. We need to detect these collisions to make the `Ball` interact correctly with its surroundings. Lines 60–88 of Fig. 21.7 control collisions between the `Ball` and the walls of the playing area. The call to `getPosition` (line 57) returns a `Vector3` representing the node's position relative to its parent node. Because all of our nodes are direct children of the root node whose position is (0, 0, 0), the position returned is always relative to the origin. There is also a `getWorldPosition` member function that will return the position relative to the origin of any node regardless of its parent's position.

The function first checks if the `Ball` has hit the left wall of the playing area (line 60). If the `Ball`'s *x*-coordinate (minus the radius) is less than or equal to –92.5 (the *x*-coordinate of the inner edge of the left wall plus half the wall's width), then the `Ball` has collided with the left wall. Once the collision is determined, the proper actions are taken. The `Ball` is placed in the middle of the screen for the next round (line 62). Player 2 is given a point (line 63) by incrementing the `static` variable in class `Pong` holding that player's score. The score is updated on the screen (line 64). Finally, a sound is played to indicate that a player has scored (line 65)—we'll explain that function call in Section 21.4.7. The process is the same to determine if the `Ball` hit the right side. The `Ball`'s *x*-coordinate is checked against the right wall's inner *x*-coordinate. If the `Ball` hits the right side the appropriate actions are taken. Figure 21.16 shows player 1 scoring a point. Note that the Ball is not actually going into the wall, it's an illusion caused by the 3D graphics.

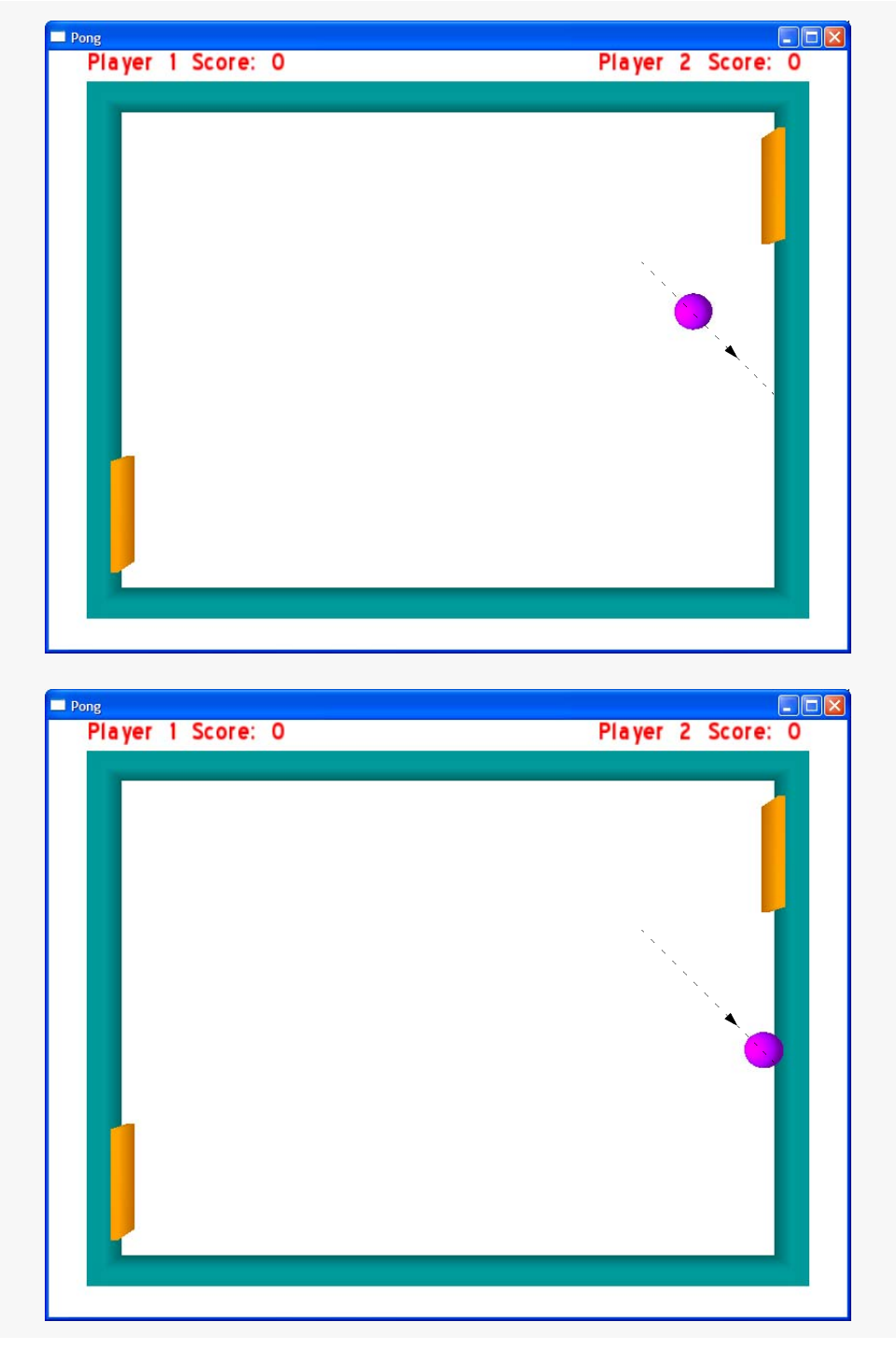


Fig. 21.16 | Player 1 scoring a point. (Part 1 of 2.)

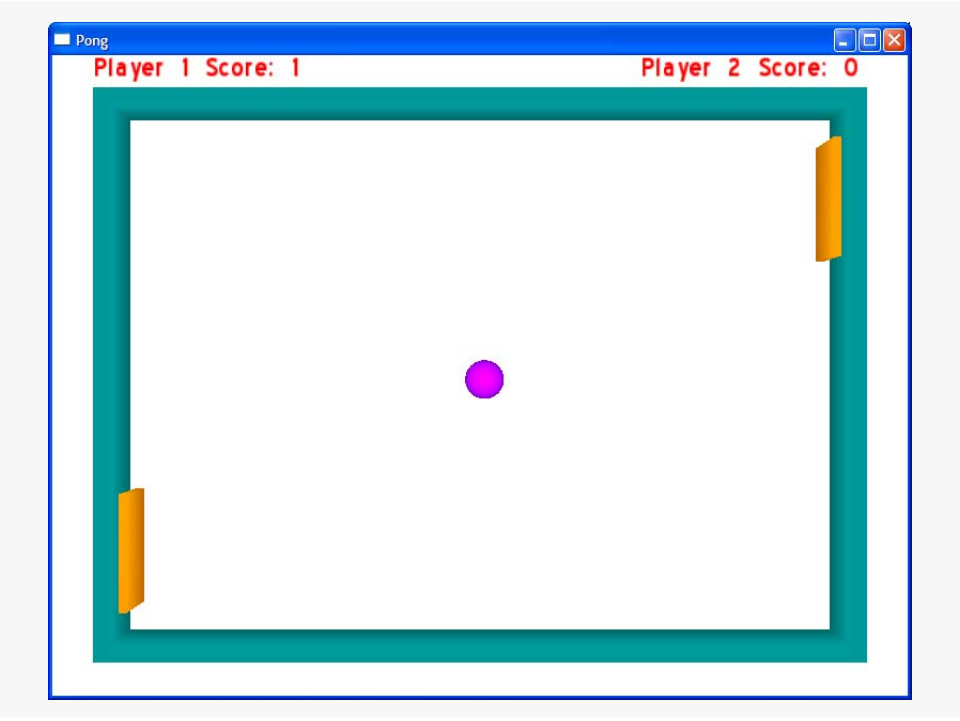


Fig. 21.16 | Player 1 scoring a point. (Part 2 of 2.)

The `Ball` is then checked against the top and bottom walls. The same collision logic is used. If the `Ball`'s *y*-coordinate (plus or minus the radius, depending on which wall it hits) after being moved would cross the wall's inner *y*-coordinate (which is physically impossible because they are both solid objects), the `Ball` has collided with either the top or bottom wall. To prevent the `Ball` from overlapping the wall, we place it at the edge of the wall after a collision (lines 79 and 86). Technically this violates the physics of the `Ball` by changing the distance it moved in the given time interval. To be accurate, we would have to determine the distance and direction it moved after hitting the wall and draw it at that point. We don't deal with this issue in the interest of keeping the code simple. The scene gets redrawn so quickly that the distance the Ball moves each frame is extremely small. The effect on the `Ball`'s movement is imperceptible.

Lines 106–143 check for collisions between the `Ball` and the `Paddle`s and take appropriate actions when one occurs. The `SceneManager` can retrieve any node within the scene graph by referencing the name given to the node when it was created. Lines 109–110 retrieve the node that the left `Paddle` is attached to, then return the position of the node. Lines 111–112 do the same thing for the right `Paddle`. We can use these positions to detect collisions between the `Paddle`s and the `Ball`. The logic is similar to the logic used for checking the walls.

Consider the lines that change the ball's direction. Line 94 causes the `Ball` to start moving left if it's currently moving right, and start moving right if it's currently moving left. Line 101 makes the `Ball` start moving up if it's currently moving down, and down if it's currently moving up. Why does this work? The direction of the `Ball` is determined by

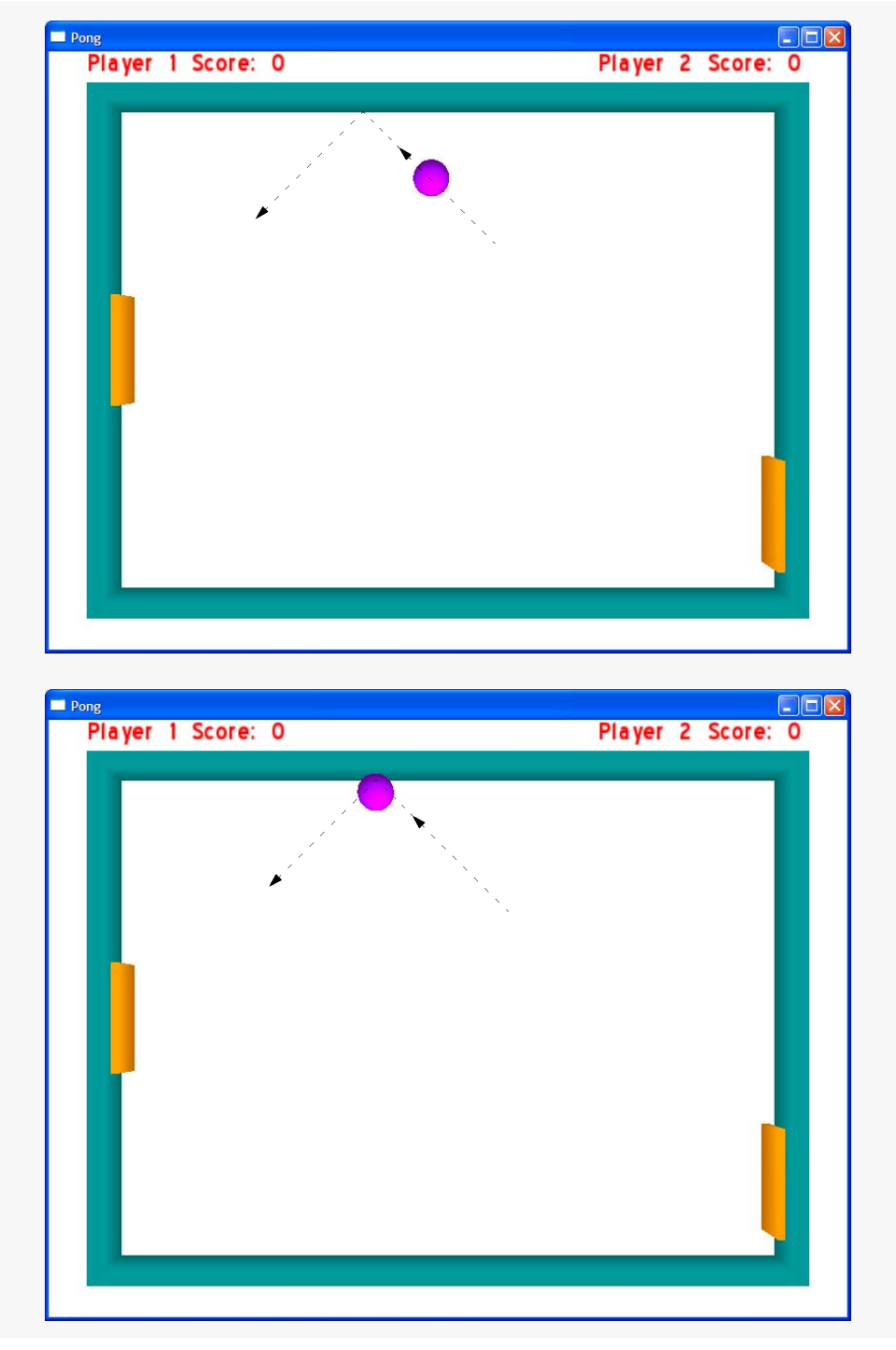


Fig. 21.17 | The `Ball` bouncing off the top wall. (Part 1 of 2.)

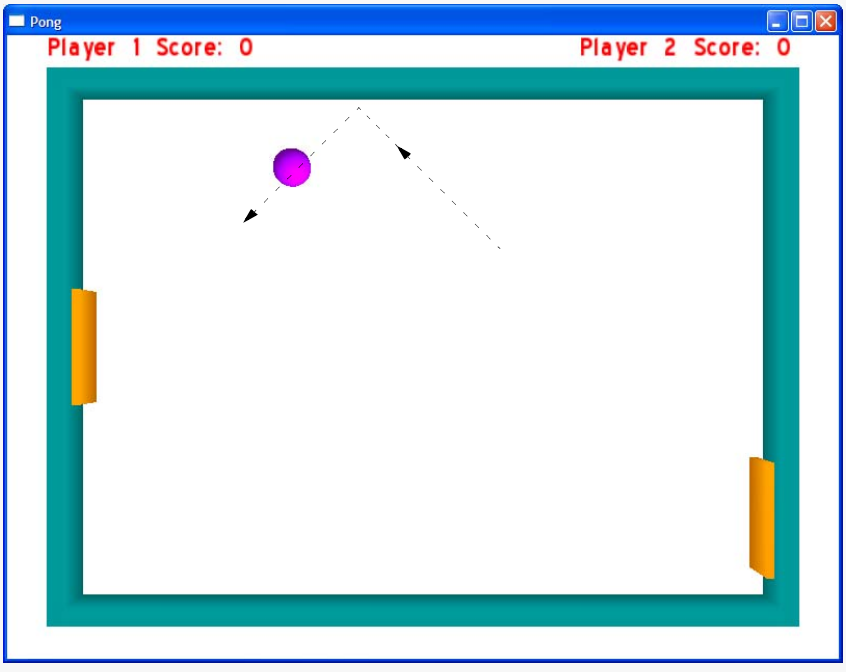


Fig. 21.17 | The `Ball` bouncing off the top wall. (Part 2 of 2.)

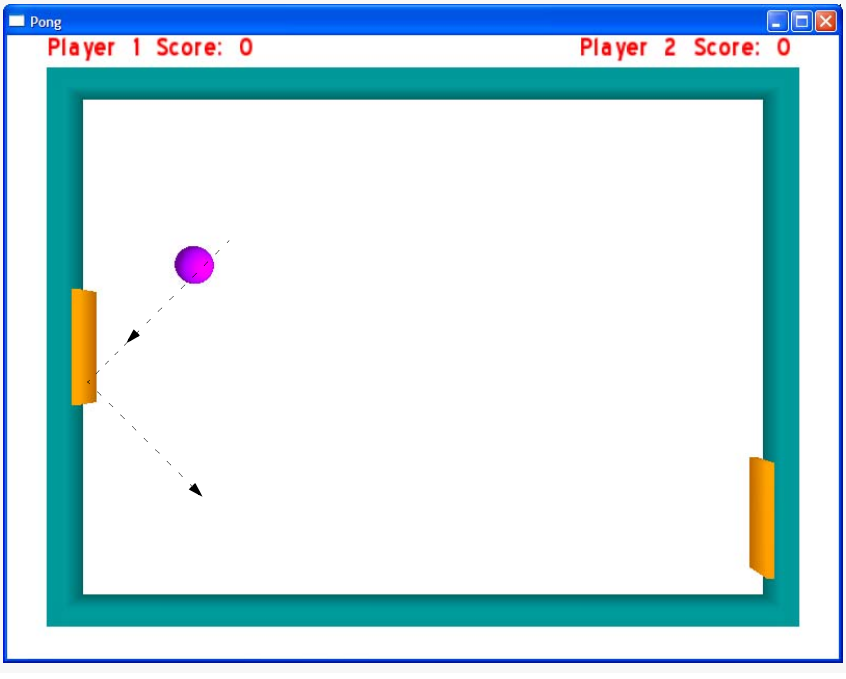


Fig. 21.18 | The `Ball` bouncing off the left `Paddle`. (Part 1 of 2.)

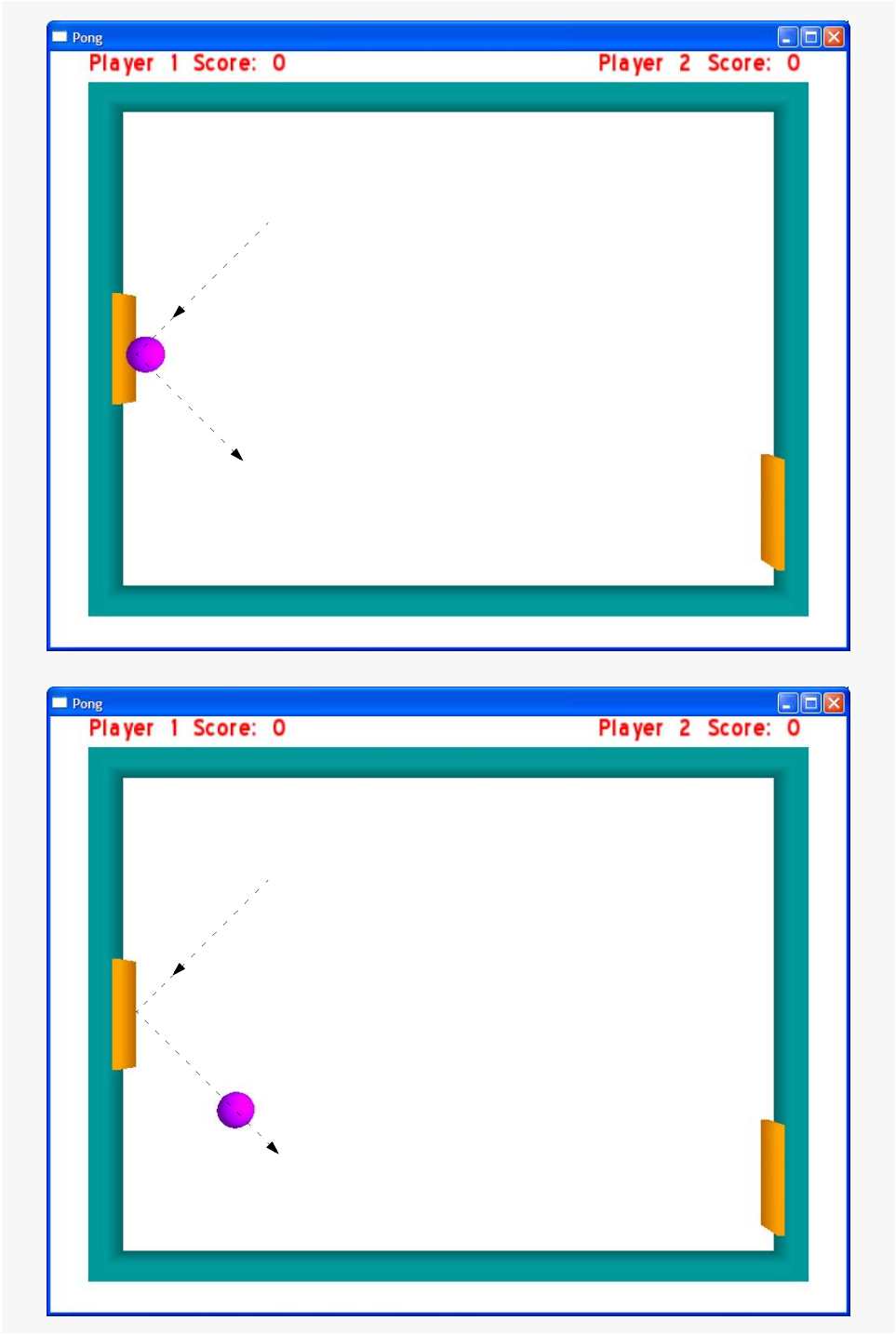


Fig. 21.18 | The `Ball` bouncing off the left `Paddle`. (Part 2 of 2.)

a `Vector3`. Each value represents a distance along the *x*-, *y*- or *z*-axes. A positive *x*-value means the `Ball` will move right along the *x*-axis, and a negative value will move the `Ball` left. If the `Ball` is moving right, multiplying its *x*-value by –1 will change the sign and reverse the direction. We do the same thing to change the vertical direction.

The collisions in our game are fairly simple cases, so we kept the logic simple. There are whole libraries dedicated to handling collisions and physics, such as Open Dynamics Engine (ODE, `http://ode.org/`), Bullet (`http://www.continuousphysics.com/Bullet/`) and Newton Game Dynamics (`http://www.newtondynamics.com/`). These libraries have Ogre bindings available on the Ogre Community Add-on page.

21.4.7 Sound

We now discuss importing sounds and playing sound files in Ogre programs, which we will use to "juice up" our Pong game. We play a "boing" sound whenever the `Ball` hits a wall, we play a different "boing" sound whenever the `Ball` hits a `Paddle` and we play a cheering sound whenever a player scores.

We will use OgreAL to add sound to our game. OgreAL is a wrapper around the OpenAL audio library. OgreAL was created and is maintained by Casey Borders (`www.mooproductions.org`), a member of the Ogre community. The OpenAL library is maintained by Creative Labs, `http://developer.creative.com`. The wrapper allows us to integrate sound functionality into the Ogre code by attaching the sounds to nodes within the scene graph. Because all of the sounds we play relate to the `Ball`, we place the OgreAL code in the `Ball` class. The OgreAL functions used to import and play sounds are analogous to those used for importing and displaying models in Ogre.

As with Ogre, we must include the `OgreAL.h` header. OgreAL manages the sounds using a `SoundManager` class, following the Ogre resource management scheme. We create a `SoundManager` in the `Ball` constructor (Fig. 21.7, line 14). There can be only one `SoundManager`. The `SoundManager` is used to create instances of `Sounds`, the OgreAL objects that contain the sound data. We create three `Sounds` and attach them to nodes (lines 39–50). The `createSound` function takes three parameters. The first is an `Ogre::String` that will be the name of the `Sound`. The second is the name of the sound file associated with the `Sound`. The third is a `bool` that determines if the `Sound` should be looped to continue playing. Passing `false` will play the `Sound` through once then stop. Passing `true` will continuously loop the sound until you stop it. We attach the `Sounds` to a node in same the way we would attach an `Entity` to a node with the `attachObject` function.

The first `Sound` we create (lines 39–40) will be played whenever the `Ball` bounces off the top or bottom wall. We attach it to the `Ball`'s node. OpenAL supports 3D sound, so by attaching the `Sound` to the `Ball`, it will play from wherever the `Ball` is. Our scene is relatively small so you may not notice the `Sound` being played in 3D, but if you listen closely it will sound slightly different. Because we placed the call to the `play` function (line 102) inside our function that reverses the `Ball`'s vertical direction, the "boing" sound will play whenever the `Ball`'s vertical direction is reversed—in other words, whenever the ball hits the top or bottom wall.

The second `Sound` is created the same way as the first, and again is attached to the `Ball`'s node. This sound will play whenever the `Ball` bounces off one of the `Paddles`. We play this `Sound` within the `reverseHorizontalDirection` function (line 95) for the same reason we play the other sound from `reverseVerticalDirection`.

The third `Sound` will play whenever a player scores. There is no particular location to play this `Sound` from—we attach it directly to the root node of the scene graph, which is positioned at the origin. We play the `Sound` from the `moveBall` function every time it is determined that a player has scored (lines 65 and 72).

There are a couple things to note about `Sounds` in OgreAL. Each `Sound` must have a unique name, just like `Entity` objects and `Nodes`. A `Sound` must finish playing before it can be played again. You should not attempt to delete the `SoundManager`. If you do, your program will throw an assertion error.

21.4.8 Resources

As mentioned earlier, Ogre uses scripts to create materials, `Overlays` and some other advanced features that are beyond the scope of this text. Ogre also uses `.mesh` files to represent 3D objects. OgreAL uses sound files. All of these resources must be loaded before we can use them. Ogre will throw a runtime exception if you try to use a resource that hasn't been loaded. To manage the game's resources we use a **`ResourceGroupManager`**. To load the resources for our game, we first tell the `ResourceGroupManager` where to find them. The `addResourceLocation` function (Figure 21.4, lines 78–79) takes three `Ogre::String` arguments. The first is the location of the resources. We placed all the resources in a folder called `PongResources` within the media folder in the Ogre SDK. Normally you would organize the resources in different folders by type, e.g. materials, models and overlays. But for simplicity we maintain one folder to hold all the resources we will need for the game. The second argument is the type of file the resources are in. The third is the resource group these files belong to. We will put these files in the `"Pong"` group. Now we load the resources in the location we just added (line 80).

21.4.9 Pong Driver

The last step is to write a `main` function (Fig. 21.19). Ogre supports various platforms so you should try not to write platform-specific code when you can avoid it. The preprocessor `if else` wrapper (lines 6–15) will determine if the program is running on a Windows platform or not. If it is, it will include the `windows.h` header and define the `WinMain` function. If not, it will define the normal `main` function. This allows to code to run on various platforms without having to be changed. You may not have seen the Windows-specific code before. The preprocessor directive to include the `windows.h` header gives the program the necessary access to the Windows API to run our program. The definition of `WIN32_LEAN_AND_MEAN` (line 7) will exclude rarely used headers in the `windows.h` header. This will speed up the compilation time for our program.

```
1  // PongMain.cpp
2  // Driver program for the game of Pong
3  #include "Pong.h" // Pong class definition
4
5  // If running on Windows, include windows.h and define WinMain function
6  #if OGRE_PLATFORM==PLATFORM_WIN32 || OGRE_PLATFORM==OGRE_PLATFORM_WIN32
7  #define WIN32_LEAN_AND_MEAN
8  #include "windows.h"
```

Fig. 21.19 | Driver program for the game of Pong. (Part 1 of 2.)

```
9
10  int WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
11
12  // If not, define normal main function
13  #else
14  int main( int argc, char **argv )
15  #endif
16  {
17      Pong game; // create a Pong object
18      game.run(); // start the Pong game
19      return 0;
20  } // end main
```

Fig. 21.19 | Driver program for the game of Pong. (Part 2 of 2.)

The `main` function creates the initial `Pong` object (line 17) and tells it to run the game (line 18). Class Pong member function `run` (Fig. 21.4, lines 187–191) first creates the game's scene (line 189), then calls the `startRendering` member function (line 190) of class `Root` to render the scene repeatedly until either the `frameStarted` or the `frameEnded` function returns false.

21.5 Wrap-Up

In this chapter you learned the basics of creating computer games. We discussed the basic concepts of graphics, briefly describing models, lighting and colors. You saw how to use the free Ogre 3D rendering engine to produce a 3D game. We showed you how to use the `SceneManager` to create and manage your scene. You learned how to use a `Camera` to view your scene. We discussed responding to user input from the keyboard with OIS. We demonstrated how to move an object at a constant speed. We covered the basics of collision detection, and showed how important it is to game programming. You learned how to display text on the screen using `Overlays`. We showed how Ogre uses scripts to manage materials and `Overlays` without having to recompile every time you change them.We also showed how to add sound to your games using the OgreAL wrapper for OpenAL.

This chapter should be viewed only a an introduction. We presented you with a basic example of Pong. Use it as a foundation for your own version. Go out and find your own sounds to use. Add new features to the game. Explore Ogre's other capabilities and create some cool visual effects. Really make this game your own. Game programming is all about being creative.

The next chapter presents bits, characters, C-strings and `structs`. You'll learn how to create and use `structs` and `typedefs`. We'll also discuss various bitwise operators and the string-handling library.

21.6 Ogre Internet and Web Resources

`ogre3d.org/`

The Ogre homepage. Here you can find the latest Ogre news, download Ogre or other Ogre related tools, browse the documentation or check out projects that use Ogre.

`www.ogre3d.org/index.php?option=com_content&task=view&id=411&Itemid=131`

Prebuilt SDK download page. There are SDKs available for Code::Blocks + MingGW C++ Toolbox, Visual C++ .Net 2003 and Visual C++ .Net 2005 (must install Service Pack 1)

`www.ogre3d.org/index.php?option=com_content&task=view&id=412&Itemid=132`
Ogre source code download page. Source code available for Windows, Linux and Mac OS X. Also download the 3rd-party dependencies package for your platform. Also has a link to a guide to building Ogre from source.

`www.ogre3d.org/index.php?option=com_content&task=view&id=415&Itemid=144`
Instructions on getting the Ogre source code from the CVS directory.

`www.ogre3d.org/wiki/index.php/Installing_An_SDK`
Installation instructions for the Ogre SDK on Windows with Visual C++, Code::Blocks + MinGW, Code::Blocks + MinGW + STLPort, Eclipse + MinGW + STLPort and GCC & Make/Any IDE. Linux, Debian, Gentoo, Fedora and Ubuntu. Mac OS X.

`www.ogre3d.org/wiki/index.php/Building_From_Source`
Instructions for building the Ogre source code on Windows with Visual C++, Visual C++ Toolkit 2003 & Code::Blocks, and GCC. Linux with GCC & Make, Debian, Fedora, Gentoo, Ubuntu/Kubuntu. Mac OS X with Xcode.

`www.ogre3d.org/wiki/index.php/BuildFAQ`
Solutions to common errors when building from the source code. Errors include not being able to find files, unresolved external symbols and other types of errors.

`www.ogre3d.org/wiki/index.php/SettingUpAnApplication`
Guide to setting up an Ogre Application project on Visual C++, Code::Blocks, GCC, Autotools, Scons, Eclipse, Anjuta IDE, KDevelop IDE

`www.ogre3d.org/phpBB2addons/viewtopic.php?t=3293&sid=17843ecbaa9b4cb5c19d4132437711ea`
OgreAL download and installation instructions.

`developer.creative.com/landing.asp?cat=1&sbcat=31&top=38`
OpenAL download and installation links.

`www.openal.org/downloads.html`
OpenAL download page.

`www.wreckedgames.com/wiki/index.php/WreckedLibs:OIS`
Object Oriented Input System (OIS) wiki page includes links to the OIS manual and API reference.

`www.tayloredmktg.com/rgb/`
Color code chart. Gives RGB values in hex and decimal. Colors are divided into general color range (e.g., grays, blues, greens, oranges).

`www.htmlcenter.com/tutorials/tutorials.cfm/89/General/`
Color chart gives RGB and hex color values.

Tutorials

`www.ogre3d.org/wiki/index.php/Ogre_Tutorials`
Ogre tutorials page. Tutorials range from basic to advanced on topics including introduction to Ogre, FrameListeners, animation multiple SceneManagers and content creation.

`www.blender.org/tutorials-help/`
Blender tutorials page.

`en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro`
The "Blender 3D: Noob to Pro" wikibook guides new Blender users through the process of 3D modeling. Teaches you how to work with models, lighting, rendering, animation, particles and soft bodies. Also has advanced tutorials on python scripting and advanced animation.

`http://www.cegui.org.uk/wiki/index.php/Tutorials`
Many tutorials on using Crazy Eddie's GUI System (CEGUI) that is supported by Ogre.

Tools

`www.ogre3d.org/index.php?option=com_content&task=view&id=413&Itemid=133`
Download model export tools for Blender, Maya, Softimage XSI and 3DS Max.

`usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7639525`
Autodesk Maya Personal Learning Edition page. Free version of Maya.

`www.softimage.com/downloads/default.aspx`
SoftImage XSI download page. Free 30-day trial available.

`www.blender.org/download/get-blender/`
Blender download page.

Code Examples

`www.ogre3d.org/wiki/index.php/CodeSnippits#HOWTO`
The Ogre Cookbook contains code samples explaining how to do various tasks relating to geometry, rendering, materials, textures, animation, input GUI and sound.

`www.ogre3d.org/phpBB2/viewtopic.php?t=27326`
Asteroid Wars. A game written using Ogre for graphics. Source code is available.

`www.ogre3d.org/phpBB2/viewtopic.php?t=27806`
Five games written with Ogre. The source code for all the games is available.

Books

`www.amazon.com/Pro-OGRE-3D-Programming/dp/1590597109/ref=pd_bbs_sr_1/102-2583408-2260151?ie=UTF8&s=books&qid=1173888297&sr=1-1`
Pro OGRE 3D Programming, by Gregory Junker.

Forums

`www.ogre3d.org/phpBB2/viewtopic.php?t=5706`
A forum post describing how to install Ogre on Debian GNU/Linux.

`www.ogre3d.org/phpBB2/viewforum.php?f=2`
Ogre Help forum. Get help from Ogre users on any problems you encounter while using Ogre.

`www.ogre3d.org/phpBB2/`
A number of forums on the Ogre site including help, using Ogre in practice, content creation, programming basics and more.

`www.ogre3d.org/phpBB2addons/`
Ogre add-ons forums. A number of forums dedicated to the more popular Ogre add-ons including OgreAL, OgreODE, NXOgre, PyOgre and more.

`www.ogre3d.org/phpBB2addons/viewforum.php?f=10`
The OgreAL forum provides information on installing and using OgreAL. Great place to find help.

`www.wreckedgames.com/forum/viewforum.php?f=6&sid=dc5f903554a80ac5194213329f5e46e4`
OIS forum. Get help on using OIS.

Summary

Section 21.3 Basics of Game Programming

- 3D graphics engines hide the tedious and complex programming required with graphics APIs.
- Ogre supports the Direct3D and OpenGL graphics APIs and runs on the Windows, Linux and Mac platforms.

- Ogre is strictly a graphics rendering engine. The Ogre community has produced many add-ons that allow users to integrate other libraries with Ogre to support those features.
- A 3D model is a computer representation of an object which can be drawn on the screen.
- Materials are used determine an object's appearance by setting lighting properties, colors and textures.
- A texture is an image that is wrapped around the model.
- Colors are determined by red, green and blue light intensities and an optional alpha value to represent transparency. Values can range from 0 to 1.0.
- There are four different types of light in a 3D scene—ambient, diffuse, emissive and specular.
- Collision detection is the process of determining whether two objects in a game are touching and reacting appropriately.
- There are collision detection and physics modeling libraries that handle the complexities for you.
- Audio libraries enrich your games with sound. Many of those libraries support 3D sound.
- Games often communicate with the user by displaying text.
- Timers control animation speed and make them look more natural.
- User input devices include the keyboard, mouse, joystick and the game controller.

Section 21.4.1 Ogre Initialization

- `Root` is the base object used in Ogre used to start the engine. No Ogre calls can be made until the `Root` object has been created.
- Call the `showConfigDialog` function of the `Root` class to display the dialog. The **OGRE Engine Rendering Setup** dialog box enables the user to choose the rendering settings.
- The resolution is defined by two values, width and height, which determine the number of pixels used to draw the scene. A higher resolution will produce more detailed graphics.
- A color depth of n-bits means that there are 2^n possible colors that can be displayed on the screen.
- The `RenderWindow` is a window in which Ogre will render graphics.

Section 21.4.2 Creating a Scene

- A scene is a collection of images that make up our graphics.
- The `SceneManager` manages the scene graph, a data structure that contains all the objects in the scene.
- The `SceneManager` is used to create objects and determine which objects will be rendered. An Ogre application can use more than one `SceneManager`.
- A `Camera` is the eye through which you view the scene. `Camera`s can be placed at any location in the scene or attached to `SceneNode`s. Ogre supports multiple `Camera`s in a single scene.
- The `Viewport` is the area that the `Camera` can see. A `Camera` can have more than one `Viewport`.
- Ogre has three types of `Light`s—`Point`, `Spot` and `Directional`. `Light`s are created with the `createLight` function of class `SceneManager`.

Section 21.4.3 Adding to the Scene

- An `Entity` is an instance of a mesh within the scene. A mesh is a file that contains the geometry information of a 3D model. Many `Entity` objects can be based on the same mesh as long as each `Entity` has a unique name.
- Use the `SceneManager` to create `SceneNode`s that hold information about an object and its position in the scene.

- The root node is the parent of all other nodes. When you create a child of the root node, its initial position is (0, 0, 0).
- Attach `Entity` objects to `SceneNodes` with `attachObject` function of class `SceneNode`.
- `scale` changes the size of the `Entity` attached to the `SceneNode` based on its current size, but it does not affect the size of the actual mesh that the node's `Entity` is based on. `setScale` changes the size based on the original size of the `Entity`. These functions also scale all children of the `SceneNode` by the same factor. To change that, call the `setInheritScale` function and pass it `false`.
- `setPosition` function places the node at the given coordinates in the scene.
- Ogre uses a `material` script to create a `material`. Save the file with a `.material` extension. A `material` file can define multiple materials, every `material` must have a unique name.
- An `Overlay` is defined by a script saved in a `.overlay` file. A single `.overlay` file can hold several `Overlay` definitions. Every object in an `Overlay` has three main attributes—metrics mode, position and size.
- `Overlays` are composed of `OverlayElements`. The first element in an `Overlay` must be an `OverlayContainer`. An `OverlayContainer` can hold any `OverlayElement`. A `TextAreaOverlayElement` holds text. Call the `show` function to display the `Overlay` on the screen
- Use `TextAreaOverlayElement` to display text. Call the `setCaption` function to change text on the screen.
- An `Overlay` with a higher *z*-order will be rendered on top of an `Overlay` with a lower *z*-order.
- Fonts are defined by a script in a `fontdef` file.
- Use the static member function `getSingleton` of class `OverlayManager` to get the `OverlayManager` object.

Section 21.4.4 Animation and Timers

- The `translate` function moves a `SceneNode`.
- `SceneNode` translations are done in parent space by default. Translations in parent space are done with respect to the parent's origin. Translations in world space are done with respect to the origin of the scene (0, 0, 0). Translations in local space are done with respect to the node's origin.
- A `FrameListener` processes `Ogre::FrameEvents`. A `FrameEvent` occurs every time a frame begins or ends.

Section 21.4.5 User Input

- Ogre does not directly support user input from devices such as the keyboard, mouse or joystick.
- Use the Object Oriented Input System (OIS) for handling user input.
- The `InputManager` is used to create the various input devices. To create the `InputManager` we must provide it with a window in which to collect.
- A `Keyboard` object collects `KeyEvents` and sends them to a `KeyListener`.
- OIS defines an enumeration of all the keys on the keyboard which we use to determine which key was pressed.

Section 21.4.6 Collision Detection

- `getPosition` returns a `Vector3` representing the node's position relative to its parent node; `getWorldPosition` returns the position relative to the origin.
- The `SceneManager` can retrieve any node within the scene graph by referencing the name given to the node when it was created.

- The direction of the `Ball` is determined by a `Vector3`. A positive *x*-value means the `Ball` will move right along the *x*-axis, and a negative value will move the `Ball` left. If the `Ball` is moving right, multiplying its *x*-value by –1 will change the sign and reverse the direction.
- There are whole libraries dedicated to handling collisions and physics.

Section 21.4.7 Sound

- OgreAL is a wrapper around the OpenAL audio library. The wrapper allows us to integrate sound functionality into the Ogre code by attaching the sounds to nodes within the scene graph.
- We must have the preprocessor directive to include the `OgreAL.h` header.
- `Sound` is the OgreAL objects that contains the sound data. Use the `createSound` function of class `SoundManager` to create sounds. There can be only one `SoundManager`.
- The `createSound` function takes three parameters. The first is an `Ogre::String` that will be the name of the `Sound` within the OgreAL system. The second is the name of the sound file associated with the `Sound`. The third is a `bool` that determines if the `Sound` should be looped to continue playing. Passing `false` will play the `Sound` through once then stop. Passing `true` will continuously loop the sound until you stop it.
- Attach the `Sound`s to a node with the `attachObject` function.
- Each `Sound` must have a unique name.
- A `Sound` must finish playing before it can be played again.

Section 21.4.8 Resources

- All of the resources must be loaded before we can use them.
- Use a `ResourceGroupManager` to manage the game's resources.
- The `addResourceLocation` function takes three `Ogre::String` arguments. The first is the location of the resources. The second argument is the type of file the resources are in. The third is the resource group these files belong to.

Section 21.4.9 Pong Driver

- Ogre supports various platforms so you should try not to write platform specific code when you can avoid it.

Terminology

3D graphics engines
3D model
3D modeling tool
3D sound
add-ons
alpha channel
ambient light
animation
`attachObject` function of class `SceneNode`
`Camera` class
`ChildNode` class
collision detection
color
color depth
`createSound` function of class `SoundManager`
culling
diffuse light
Direct3D
`Directional` lights
emissive light
`Entity` class
`EventProcessor` class
frame
`FrameEvent` class
`FrameListener` class
`getPosition` function of class `SceneNode`
`getRootSceneNode` function of class `SceneManager`
`getWorldPositionfunction` of `classSceneNode`
graphics
`initialise` function of class `Root`
`InputManager` class

`Keyboard` class
`KeyEvent` class
`KeyListener` class
`Light` class
lighting
local space
LoD (levels of detail)
material
mesh
metrics mode
`Node` class
normal
Ogre
Ogre Application Wizard
OgreAL
OIS (Object Oriented Input System)
OpenAL audio library
OpenGL
`Overlay` class
`OverlayContainer` class
`OverlayElement` class
`OverlayManager` class
`PanelOverlayElement` class
parent space
pixel mode
`Point` light
Pong
relative mode
render
rendering subsystem
`RenderWindow` class
resolution
`ResourceGroupManager` class
`Root` class
root node
`scale` function of class `SceneNode`
scaling factor
scene
scene graph
`SceneManager` class
`SceneNode` class
scripts
`setCaption` function of class `TextAreaOverlayElement`
`setInheritScale` function of class `SceneNode`
`setPosition` function of class `SceneNode`
`setScale` function of class `SceneNode`
`showConfigDialog` function of class `Root`
sound
`Sound` class
`SoundManager` class
specular light
`Spot` light
text
`TextAreaOverlayElement` class
texture
timer
`timeSinceLastFrame`
`translate` function of class `SceneNode`
user input
`Vector3`
`Viewport` class
`WIN32_LEAN_AND_MEAN`
`WinMain`
world space
z-order

Self-Review Exercises

21.1 Fill in the blanks in each of the following statements:
a) Every Ogre program must include the _____ header.
b) The _____ object must be created before any other Ogre function is called.
c) The main type defined by OgreAL for pointing to sound file data is _____.
d) A(n) _____ object is used to represent a color in Ogre.
e) Every OgreAL program must include the _____ header.
f) _____ are used to define materials and overlays for Ogre programs.
g) The _____ object is used to load resources for Ogre programs.
h) Ogre uses a(n) _____ object to manage the scene.
i) A 3D model is defined in an Ogre _____ file.

21.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) The coordinates (0, 0) refer to the bottom-left corner of an `Overlay Container`.
b) If Ogre attempts to load an external file that does not exist, a runtime error will occur.
c) Color values in Ogre range from 0 to 255.

d) Passing a value of `false` to the `createSound` function will cause the sound file to play continuously.
e) The `EventProcessor` constructor must be passed a parameter that gives Ogre the driver information of the system's keyboard.
f) An `Overlay` that draws text on the screen must specify a font in which that text should be drawn.
g) Every `Entity` must have a unique name.

21.3 Write statements to accomplish each of the following:
a) Attach an `Entity` pointer named `entityPtr` to a `SceneNode` pointer name `nodePtr`.
b) Scale the `Entity` from the previous question to half its original size.
c) Create the `Sound sample` that loops the `sound.wav` file.
d) If the spacebar is being pressed, set the value of the `int number` to 0.
e) Set an `Overlay Element` to position itself relative to the size of its parent `Container`.
f) Add a folder named `sounds` in the `media` folder of the OgreSDK as a `"General"` resource location.

21.4 Find the error in each of the following:
a)
```
SceneNode node;
```
b)
```
eventProcessorPtr->initialize();
```
c)
```
ColourValue( 0, 0, 255 );
```

Answers to Self-Review Exercises

21.1 a) `Ogre.h`. b) `Root`. c) `Sound`. d) `ColourValue`. e) `OgreAL.h`. f) scripts. g) `ResourceManager`. h) `SceneManager`. i) `.mesh`.

21.2 a) False. The coordinates (0, 0) refer to the top-left corner of an `Overlay Container`.
b) True.
c) False. Color values in Ogre range from 0.0 to 1.0.
d) False. The sound will play once then stop.
e) False. The `EventProcessor` constructor does not take any parameters.
f) True.
g) True.

21.3 a)
```
nodePtr->attachObject( entityPtr );
```
b)
```
nodePtr->setScale( .5, .5, .5 );
```
c)
```
soundManagerPtr->createSound( "sample", "sound.wav", true );
```
d)
```
if ( keyEvent.getKey() == KC_SPACE )
   number = 0;
```
e)
```
metrics_mode relative;
```
f)
```
ResourceGroupManager::getSigleton().addResourceLocation( "../../media/
sounds", "FileSystem", "General" );
```

21.4 a) The variable `node` should instead be declared as a pointer to a `SceneNode`. All of Ogre's `SceneNode` functions either take a pointer as a parameter or return a pointer.
b) The `initialise` function of an `EventProcessor` takes a `RenderWindow *` as an argument. It also uses the British spelling "initialise".
c) The `ColourValue` object can accept parameters only with values between 0 and 1.

Exercises

21.5 Look through the resources available in our Game Programming Resource Center at `www.deitel.com/computergames/gameprogramming/` and the C++ Game Programming Resource Center at `www.deitel.com/CplusplusGameProgramming/`.

21.6 Write a program that draws the mesh `sphere.mesh` in the center of the screen. When the user presses one of the arrow keys, the mesh should move ten units in that direction.

21.7 Modify the program from Exercise 21.6 so that if the user holds down an arrow key, the ball will move only once every second.

21.8 Modify the Pong game so that when a player reaches 21 points, the game ends and displays a message that the left or right player has won.

21.9 In most Pong games, when a rally between the two players lasts for a long time, the ball begins to speed up in order to prevent a stalemate. Modify the Pong game so that the ball's speed increases for every ten times that it is hit in a rally. When either player scores, the ball should return to its original speed.

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

21.11 Modify the Pong game so that when the ball bounces off a wall or paddle, the word "BOING!" appears in blue at the point where the ball bounced, then fades away. Note that the text should not simply vanish—it should gradually fade to white.

21.12 Modify the Pong game so that before the game begins, a menu appears on the screen that allows the players to choose from several different ball and paddle speeds. Ogre comes with the CEGUI (Crazy Eddie's GUI) library that could be helpful. The CEGUI library allows you to create a user interface in Ogre.

21.13 (*The Game of Snake*). The object of the game of snake is to maneuver the snake throughout the game area trying to eat bits of food. The snake is represented with a string of contiguous spheres in the game area, which is a two-dimensional grid. The snake can move up, down, left or right. If the snake eats a piece of food (shown by the "F"), it grows by adding another sphere to the end (Fig. 21.20). If the snake hits a wall of the game area (i.e., would be out of the array) the player loses (Fig. 21.21). If the snake runs into itself, the player loses (Fig. 21.22).

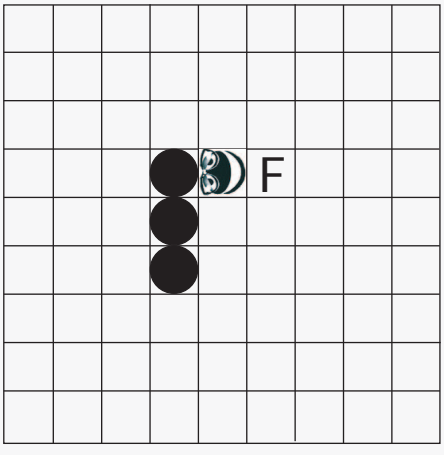


Fig. 21.20 | The snake grows when it eats. (Part 1 of 2.)

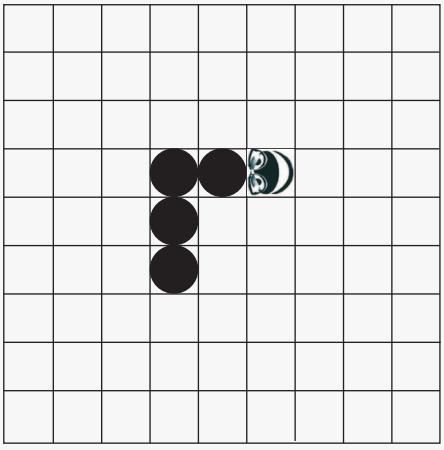


Fig. 21.20 | The snake grows when it eats. (Part 2 of 2.)

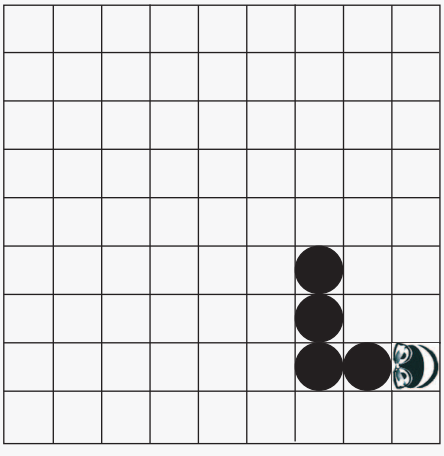


Fig. 21.21 | The snake dies if it hits a wall.

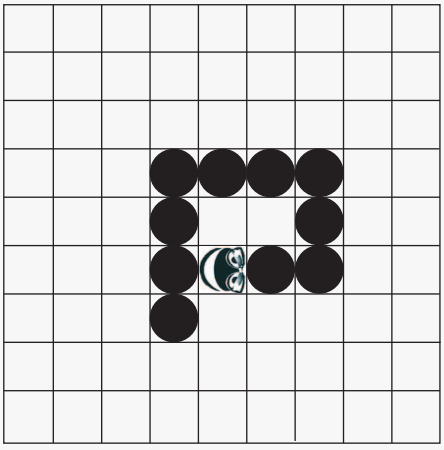


Fig. 21.22 | The snake dies if it hits itself.

21.14 Modify the program from Exercise 21.13 to add obstacles to the game area (Fig. 21.23). If the snake runs into an obstacle, the player loses.

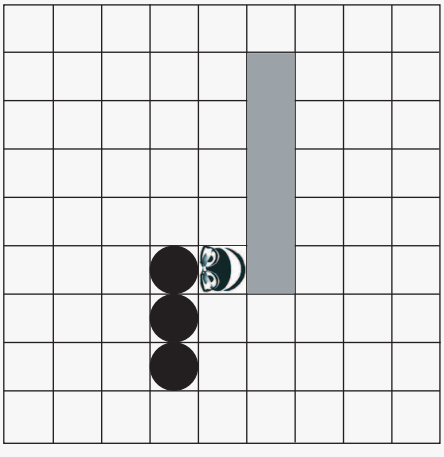


Fig. 21.23 | The snake dies if it hits an obstacle.

Index

Z