Chapter 7

A Case Study: A Billiards Game

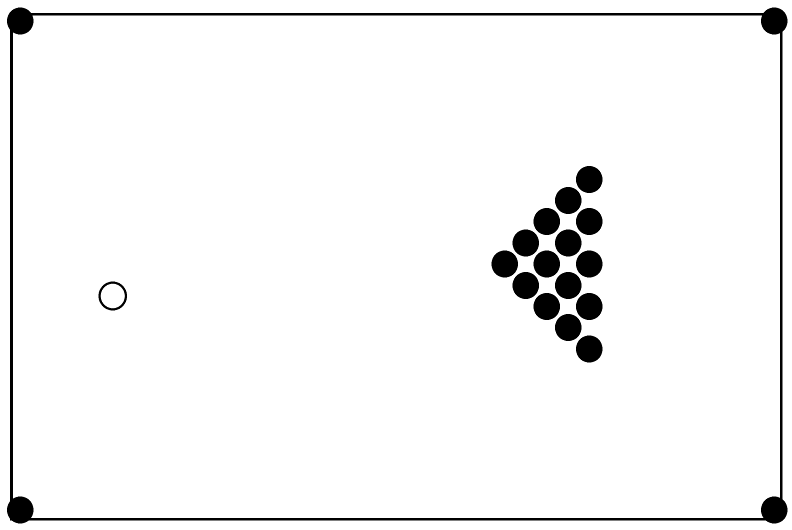
In our second case study, we will develop a simple simulation of a billiard table.[1] The program is written in Delphi Pascal.[2] As with the eight-queens program, the design of this program will stress the creation of autonomous interacting agents working together to produce the desired outcome.

7.1 The Elements of Billiards

The billiard table as the user sees it consists of a window containing a rectangle with holes (pockets) in the corners, 15 colored balls, and 1 white cue ball. By clicking the mouse the user simulates striking the cue ball, imparting a certain amount of energy to it. The direction of motion for the cue ball will be opposite to that of the mouse position in relation to the cue. Once a ball has energy it will start to move, reflecting off of walls, falling into holes, and potentially striking other balls. When a ball strikes another ball some of the energy of the first is given to the second, while the direction of movement of the two balls is changed by the collision.

[1]The game implemented by the program described in this chapter does not correspond to any actual game. It is not pool, it is not billiards, it is simply balls moving around a table consisting of walls and holes.

[2]Discussion of Delphi Pascal is complicated by the fact that graphical user interface elements of Delphi programs are constructed visually, using the integrated development environment. This style of design will be familiar to users of Visual Basic. However, the user interface aspects are not relevant to our purposes, which is the investigation of Delphi as an object-oriented programming language. The references at the end of the chapter provide pointers to further information regarding these other aspects of Delphi.



7.2 Graphical Objects

The heart of the simulation are three linked lists of *graphical objects*, which comprise the walls, holes, and balls. Each graphical object will include a link field and a field indicating the region of the screen occupied by the object.[3]

A simplifying assumption we have made is that all graphical objects occupy rectangular regions. This is, of course, quite untrue for a round object such as a ball. A more realistic alternative would have been to write a procedure that determined whether two balls have intersected based on the geometry of the ball rather than on the intersection of their regions. Once again, the complexity of the procedure would only have detracted from the issues we wish to address in our case study.

The primary objective in this case study is the way in which responsibility for behavior has been vested in the objects themselves. Every graphical object knows not only how to draw itself but how to move and how to interact with the other objects in the simulation.

[3]There are clear conflicts in ordering in the presentation of this case study. On the one hand, it is important for the reader to see examples of object-oriented principles as soon as possible; thus, placing this particular case study early in the book is desirable. On the other hand, this program, like almost all object-oriented programs, would benefit from more advanced techniques, which we will not discuss until later. In particular, the graphical objects might be better described by an inheritance hierarchy, such as we will describe in Chapter 8. Similarly, it is generally considered poor programming practice for the objects being maintained on a linked list to hold the link fields as part of their data area; a better design would separate the container from the elements in the list. Solving this problem is non-trivial and introduces complications not particularly relevant to the points addressed here. We will discuss container classes in Chapter 19.

7.2.1 The Wall Graphical Object

The first of our three graphical objects is a wall. It is defined by the following class description:

```
TWall = class(TObject)
public
    constructor create
        (ix, iy, iw, ih : Integer; cf : Real; ilink : TWall);
    procedure draw (canvas : TCanvas);
    function hasIntersected(aBall : Tball) : Boolean;
    procedure hitBy (aBall : TBall);
private
    x, y : Integer;
    height, width : Integer;
    convertFactor : Real;
    link : TWall;
end;
```

The x and y fields represent the upper left corner of the wall, while the height and width fields maintain the size. The link field maintains a linked list of wall objects. The constructor simply defines the region of the wall and sets the convert factor:

```
constructor TWall.create
    (ix, iy, iw, ih : Integer; cf : Real; ilink : Twall);
begin
    x := ix;
    y := iy;
    height := ih;
    width := iw;
    convertFactor := cf;
    link := ilink;
end;
```

A wall can be drawn simply by printing a solid rectangle. A graphics library routine performs this task:

```
procedure TWall.draw(canvas: TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clBlack;
                fillRect(Rect(x, y, x + width, y + height));
        end;
```

```
end;
```

The most interesting behavior of a wall occurs when it has been struck by a ball. The direction of the ball is modified by use of the convert factor for the wall. (Convert factors are either zero or pi, depending upon whether the wall is horizontal or vertical). The ball subsequently moves off in a new direction.

```
procedure TWall.hitBy (aBall : TBall);
begin
        { bounce the ball off the wall }
    aBall.direction := convertFactor - aBall.direction;
end;
```

7.2.2 The Hole Graphical Object

A hole is defined by the following class description:

```
THole = class(TObject)
public
    constructor create (ix, iy : Integer; ilink : THole);
    procedure draw (canvas : TCanvas);
    function hasIntersected(aBall : TBall) : Boolean;
    procedure hitBy (aBall : TBall);
private
    x, y : Integer;
    link : THole;
end;
```

As with walls, the initialization and drawing of holes is largely a matter of invoking the correct library routines:

```
constructor THole.create(ix, iy : Integer; ilink : THole);
begin
    x := ix;
    y := iy;
    link := ilink;
end;

procedure THole.draw(canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clBlack;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
```

```
end;
```

Of more interest is what happens when a hole is struck by a ball. There are two cases. If the ball happens to be the cue ball (which is identified with a global variable, CueBall), it is placed back into play at a fixed location. Otherwise, all the energy is drained from the ball and it is moved off the table to a special display area.

```
procedure THole.hitBy (aBall : TBall);
begin
        { drain enery from ball }
    aBall.energy := 0.0;

        { move ball }
    if aBall = CueBall then
        aBall.setCenter(50, 100)
    else begin
        saveRack := saveRack + 1;
        aBall.setCenter (10 + saveRack * 15, 250);
    end;
end;
```

7.2.3 The Ball Graphical Object

Our final graphical object is the ball, defined by the following class description:

```
TBall = class(TObject)
public
    constructor create (ix, iy : Integer; iLink : TBall);
    procedure draw (canvas : TCanvas);
    function hasIntersected(aBall : Tball) : Boolean;
    procedure hitBy (aBall : TBall);
    procedure update;
    procedure setCenter (nx, ny : Integer);
    procedure setDirection (nd : Real);
private
        x, y : Integer;
        direction : Real;
        energy : Real;
        link : TBall;
end;
```

In addition to the link and rectangle regions common to the other objects, a ball maintains two new data fields; a direction, measured in radians, and an energy, which is an arbitrary real value. Like a hole, a ball is initialized by arguments that specify the center of the ball. Initially a ball has no energy and a direction of zero.

```
constructor TBall.create(ix, iy : Integer; iLink : TBall);
begin
        setCenter(ix, iy);
        setDirection(0.0);
        energy := 0.0;
        link := iLink;
end;

procedure TBall.setCenter(nx, ny : Integer);
begin
        x := nx;
        y := ny;
end;

procedure TBall.setDirection(nd : Real);
begin
        direction := nd;
end;
```

A ball is drawn either as a frame or as a solid circle, depending upon whether or not it represents the cue ball.

```
procedure TBall.draw(canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                if (self = cueBall) then
                        Brush.Color := clWhite
                else
                        Brush.Color := clBlack;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
end;
```

The method update is used to update the position of the ball. If the ball has a nontrivial amount of energy, it moves slightly, then checks to see if it has hit another object. A global variable named ballMoved is set true if any ball on the table has moved. If the ball has hit another object, it notifies the second object that it has been struck. This notification process is divided into three steps, corresponding to hitting holes, walls, and other balls. Inheritance, which we will study in Chapter 8, will provide a means by which these three tests can be combined into a single loop.

```
procedure TBall.update;
var
    hptr : THole;
    wptr : TWall;
    bptr : TBall;
    dx, dy : integer;
begin
    if energy > 0.5 then begin
        ballMoved := true;
            { decrease energy }
        energy := energy - 0.05;
            { move ball }
        dx := trunc(5.0 * cos(direction));
        dy := trunc(5.0 * sin(direction));
                x := x + dx;
                y := y + dy;

            { see if we hit a hole }
        hptr := listOfHoles;
        while (hptr <> nil) do
            if hptr.hasIntersected(self) then begin
                hptr.hitBy(self);
                hptr := nil;
            end
            else
                hptr := hptr.link;

            { see if we hit a wall }
        wptr := listOfWalls;
        while (wptr <> nil) do
            if wptr.hasIntersected(self) then begin
                wptr.hitBy(self);
                wptr := nil;
            end
            else
                wptr := wptr.link;

            { see if we hit a ball }
        bptr := listOfBalls;
        while (bptr <> nil) do
            if (bptr <> self) and bptr.hasIntersected(self) then begin
                bptr.hitBy(self);
                bptr := nil;
            end
            else
                bptr := bptr.link;
    end;
end;
```

When one ball strikes another ball, the energy of the first one is split and half is given to the second one. The angles of both are also changed. (The physics is not exactly correct, but the results look reasonably realistic.)

```
procedure TBall.hitBy (aBall : TBall);
var
    da : real;
begin
        { cut the energy of the hitting ball in half }
    aBall.energy := aBall.energy / 2.0;

        { and add it to our own }
    energy := energy + aBall.energy;

        { set our new direction }
    direction := hitAngle(self.x - aBall.x, self.y - aBall.y);

        { and set the hitting balls direction }
    da := aBall.direction - direction;
    aBall.direction := aBall.direction + da;

        { continue our update }
    update;
end;

function hitAngle (dx, dy : real) : real;
    const
        PI = 3.14159;
    var
        na : real;
begin
    if (abs(dx) < 0.05) then
        na := PI / 2
    else
        na := arctan (abs(dy / dx));
    if (dx < 0) then
        na := PI - na;
    if (dy < 0) then
        na := - na;
    hitAngle := na;
```

```
end;
```

7.3 The Main Program

The previous section described the static characteristics of the program. The dynamic characteristics are set in motion when a mouse press occurs, at which time the following function is invoked:

```
procedure TfrmGraphics.DoClick (Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
    bptr : TBall;
begin
    cueBall.energy := 20.0;
    cueBall.setDirection(hitAngle(cueBall.x - x, cueBall.y - y));
        { then loop as long as called for }
    ballMoved := true;
    while ballMoved do begin
        ballMoved := false;
        bptr := listOfBalls;
        while bptr <> nil do begin
            bptr.update;
            bptr := bptr.link;
        end;
    end;
end;
```

The remainder of the program is relatively straight forward and will not be presented here. The complete source is given in Appendix B. The majority of the code is concerned with the initialization of the new objects and with the event loop that waits for the user to perform an action. The programmer uses the Delphi development environment to match events, such as mouse presses, with procesures, such as DoClick.

To stress the point we made at the beginning of this chapter, the most important feature of this case study is the fashion in which control has been decentralized and the objects themselves have been given the power to control and direct the flow of execution. When a mouse press occurs, all that happens is that the cue ball is provided with a certain amount of energy. Thereafter, the interaction of the balls drives the simulation.

7.4 Using Inheritance

In Chapter 1 we informally introduced inheritance, and in Chapter 8 we will discuss how inheritance works in each of the languages we are considering. In this section we will describe how inheritance can be used to simplify the billiards simulation, foreshadowing the discussion we will present in the next chapter. The reader may wish to return to this section after reading the general treatment of inheritance in the next chapter.

We have, in fact, been using inheritance throughout our development of the classes for our application. All of our classes inherit from the system class TObject. But as we did not use any behavior from this class, the issue was not very important. Now we will create parent classes that do embody useful behavior.

The first step in using inheritance in our billiards simulation is to define a general class for "graphical objects." This class includes all three items: balls, walls, and holes. The parent class is defined as follows:

```
type
  TBall = class; (* forward declaration *)

  TGraphicalObject = class(TObject)
  public
        constructor Create(ix, iy : Integer; il : TGraphicalObject);
        procedure draw (canvas : TCanvas); virtual; abstract;
        function hasIntersected (aBall : TBall): Boolean; virtual; abstract;
        procedure hitBy (aBall : TBall); virtual; abstract;
        procedure update; virtual;
  private
        x, y : Integer;
        link : TGraphicalObject;
  end;
```

Note the forward declaration for the class TBall. This allows the class TGraphicalObject to declare arguments of type TBall, even though the class definition has not yet been seen.

Every graphical object has a location and a link. The constructor sets these values. The methods draw, hasIntersected, and hitBy are declared as virtual and abstract. This means they are not defined in the parent class, but must be redefined in the child classes. The method update is declared as virtual, but not abstract. In the parent class it is defined to do nothing. This behavior will be overridden by class TBall, but not by the other two.

The classes Ball, Wall, and Hole are then declared as subclasses of the general class GraphicalObject and need not repeat the declarations for data areas or functions, unless they are being overridden.

```
THole = class(TGraphicalObject)
public
    constructor create
        (ix, iy : Integer; ilink : TGraphicalObject); overload;
    procedure draw (canvas : TCanvas);  override;
    function hasIntersected(aBall : TBall) : Boolean;  override;
    procedure hitBy (aBall : TBall); override;
end;
```

Compare this declaration to the one given earlier, and note how we have now eliminated the declaration for the data fields, since they have been moved to the parent class.

Constructors for the child classes must explicitly invoke the constructors for their parent classes, as in the following constructor for class TBall:

```
constructor TBall.Create (ix, iy : Integer; iLink : TGraphicalObject);
begin
    inherited Create(ix, iy, iLink);
    setDirection(0.0);
    energy := 0.0;
end;
```

By making CueBall a subclass of Ball, we can eliminate the conditional statement in the routine that draws the ball's image.

```
TCueBall = class(TBall)
public
        procedure draw (canvas : TCanvas); override;
end;

procedure TBall.draw(canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clBlack;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
end;

procedure TCueBall.draw (canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clWhite;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
end;
```

The greatest simplification comes from the fact that it is now possible to keep all graphical objects on a single linked list. Thus, the routine that draws the entire screen, for example, can be written as follows:

```
procedure TfrmGraphics.DrawExample(Sender: TObject);
var
    gptr : TGraphicalObject;
begin
    with imgGraph.Canvas do begin
        Brush.Color := clWhite;
        Brush.Style := bsSolid;
        FillRect(Rect(0, 0, 700, 700));
    end;
    gptr := listOfObjects;
    while (gptr <> nil) do begin
        gptr.draw(imgGraph.Canvas);
        gptr := gptr.link;
    end;
end;
```

The most important point in this code concerns the invocation of the function draw within the loop. Despite the fact that there is only one function call written here, sometimes the function invoked will be from class TBall; at other times it will be from class TWall, or class THole. The fact that one function call might result in many different function bodies being invoked is a form of *polymorphism*. We will discuss this important topic in more detail in Chapter 14.

The routine that tests to see if a moving ball has hit anything in the function Ball.update is similarly simplified. This can be seen in the complete source listing provided in Appendix B.

Chapter Summary

In our second case study we have examined a graphical program that simulates the behavior of a pool table. Once more our motivation for presenting the case study was not so much the problem being addressed, as it was the manner in which the problem was being solved. The balls, holes, and walls in the game are described as independently reacting agents. When the user interacts with the game by means of a mouse press, the effect is to impart some energy to the cue ball, thereby forcing it to move. Thereafter the objects interact among themselves, until all the balls run out of energy.

Further Information

In the first two editions of the book this case study was presented in Apple Object Pascal, instead of Delphi. Those versions can still be found in the web site, `ftp://ftp.cs.orst.edu/pub/budd/oopintro`.

As we noted at the beginning of this chapter, our concern here is with the programming language aspects of Delphi, which are only a small part of the entire Delphi system. Further information on Delphi can be found in [Lischner 2000, Kerman 2002]. Borland also provides a wealth of on-line material with the Delphi integrated program development system.

Self Study Questions

1. Give some examples of how the design makes holes, walls, and balls responsible for their own behavior.
2. By making each graphical object into a separate class, and making each responsible for a different aspect of behavior, the object-oriented design is able to support a great deal of information hiding. This, in turn, leads to programs which are considerably easier to modify than when conventional techniques are used. To illustrate this, explain what sections of code would need to be modified to produce each of the following changes:
   - Colored balls, rather than black and white.
   - Walls which absorb a bit of energy when they reflect a ball.
   - Holes which make a sound when they absorb a ball.
   - Balls which make a sound when they strike.

Exercises

1. Suppose you want to perform a certain action every time the billiards program executes the event loop task. Where is the best place to insert this code?
2. Suppose you want to make the balls colored. What portions of the program do you need to change?
3. Suppose you want to add pockets on the side walls, as on a conventional pool table. What portions of the program do you need to change?
4. The billiards program uses a “breadth-first” technique, cycling repeatedly over the list of balls, moving each a little as long as any ball has energy. An alternative, and in some ways more object-oriented, approach is to have each ball continue to update itself as long as it possesses any energy, and update any ball that it hits. With this technique, it is only necessary to

start the cue ball moving in order to put the simulation in motion. Revise the program to use this approach. Which do you think provides a more realistic simulation? Why?

5. A hole has the same graphical representation as a ball, namely a round black spot. Similarly the algorithms used to determine if a ball has intersected are the same for balls and holes. Given this, would it make sense to declare TBall as a child class of THole? What would be the advantages of doing so? What might be some problems introduced by this modification?