MonoGame Blocks Tutorial Part 6

Contents

MonoGame Blocks Tutorial Part 6 1
Part 6 – The Ball 1
Assignment Submission 12

Part 6 – The Ball

Time required: 60 minutes

In the previous section, we added code to move the paddle, but we still don't have a ball to hit! It is time to add a ball to our game.

1. Right-click on the project **Blocks** in the solution explorer and select **Add → Class** from the drop-down menus.
2. In the **Add New Item** Dialog, enter "Ball.cs" for the file name, then click **Add**.
3. Enter the following code in "Ball.cs".

```
1   using System;
2   using Microsoft.Xna.Framework;
3   using Microsoft.Xna.Framework.Graphics;
4
5   namespace Blocks6
6   {
        2 references
7       class Ball
8       {
            11 references
9           public float BallX { get; set; }
            11 references
10          public float BallY { get; set; }
            19 references
11          public float BallXVelocity { get; set; }
            9 references
12          public float BallYVelocity { get; set; }
            5 references
13          public float BallHeight { get; set; }
            6 references
14          public float BallWidth { get; set; }
            5 references
15          public float Rotation { get; set; }
            2 references
16          public bool UseRotation { get; set; }
            3 references
17          public float ScreenWidth { get; set; } // Width of game screen
            2 references
18          public float ScreenHeight { get; set; } // Height of game screen
            7 references
19          public bool IsBallVisible { get; set; }  // Is ball visible on screen
            4 references
20          public int Score { get; set; }
            2 references
21          public int BlocksCleared { get; set; } // Number of blocks cleared this level
22
            4 references
23          private Texture2D imgBall { get; set; }
24
25          // Allows us to write on backbuffer when we need to draw self
26          private SpriteBatch spriteBatch;
27          private GameContent gameContent;
```

```
29          // Constructor
            1 reference
30          public Ball(float screenWidth, float screenHeight,
31                      SpriteBatch spriteBatch, GameContent gameContent)
32          {
33              BallX = 0;
34              BallY = 0;
35              BallXVelocity = 0;
36              BallYVelocity = 0;
37              Rotation = 0;
38              imgBall = gameContent.imgBall;
39              BallWidth = imgBall.Width;
40              BallHeight = imgBall.Height;
41              this.spriteBatch = spriteBatch;
42              this.gameContent = gameContent;
43              ScreenWidth = screenWidth;
44              ScreenHeight = screenHeight;
45              IsBallVisible = false;
46              Score = 0;
47              BlocksCleared = 0;
48              UseRotation = true;
49          }
```

```
51          public void Draw()
52          {
53              if (IsBallVisible == false)
54              {
55                  return;
56              }
57              if (UseRotation)
58              {
59                  Rotation += .1f;
60                  if (Rotation > 3 * Math.PI)
61                  {
62                      Rotation = 0;
63                  }
64              }
65              spriteBatch.Draw(imgBall, new Vector2(BallX, BallY), null,
66                               Color.White, Rotation, new Vector2(BallWidth / 2,
67                               BallHeight / 2), 1.0f, SpriteEffects.None, 0);
68          }
```

```
70          public void Launch(float x, float y, float xVelocity, float yVelocity)
71          {
72              if (IsBallVisible == true)
73              {
74                  return;  // Ball already exists, ignore
75              }
76              IsBallVisible = true;
77              BallX = x;
78              BallY = y;
79              BallXVelocity = xVelocity;
80              BallYVelocity = yVelocity;
81          }
```

```
83      public bool Move(Wall wall, Paddle paddle)
84      {
85          if (IsBallVisible == false)
86          {
87              return false;
88          }
89          BallX = BallX + BallXVelocity;
90          BallY = BallY + BallYVelocity;
91
92          // Check for wall hits
93          if (BallX < 1)
94          {
95              BallX = 1;
96              BallXVelocity = BallXVelocity * -1;
97          }
98          if (BallX > ScreenWidth - BallWidth + 5)
99          {
100             BallX = ScreenWidth - BallWidth + 5;
101             BallXVelocity = BallXVelocity * -1;
102         }
103         if (BallY < 1)
104         {
105             BallY = 1;
106             BallYVelocity = BallYVelocity * -1;
107         }
108         if (BallY > ScreenHeight)
109         {
110             IsBallVisible = false;
111             BallY = 0;
112             return false;
113         }
```

```
115             // Check for paddle hit
116             // Paddle is 70 pixels.
117             // Logically divide it into segments that will determine the angle of the bounce
118             Rectangle paddleRect = new Rectangle((int)paddle.PaddleX, (int)paddle.PaddleY,
119                                     (int)paddle.PaddleWidth, (int)paddle.PaddleHeight);
120             Rectangle ballRect = new Rectangle((int)BallX, (int)BallY,
121                                 (int)BallWidth, (int)BallHeight);
122             if (HitTest(paddleRect, ballRect))
123             {
124                 int offset = Convert.ToInt32((paddle.PaddleWidth - (paddle.PaddleX +
125                             paddle.PaddleWidth - BallX + BallWidth / 2)));
126                 offset = offset / 5;
127                 if (offset < 0)
128                 {
129                     offset = 0;
130                 }
131                 switch (offset)
132                 {
133                     case 0:
134                         BallXVelocity = -6;
135                         break;
136                     case 1:
137                         BallXVelocity = -5;
138                         break;
139                     case 2:
140                         BallXVelocity = -4;
141                         break;
142                     case 3:
143                         BallXVelocity = -3;
144                         break;
145                     case 4:
146                         BallXVelocity = -2;
147                         break;
148                     case 5:
149                         BallXVelocity = -1;
150                         break;
151                     case 6:
152                         BallXVelocity = 1;
153                         break;
154                     case 7:
155                         BallXVelocity = 2;
156                         break;
157                     case 8:
158                         BallXVelocity = 3;
159                         break;
160                     case 9:
161                         BallXVelocity = 4;
162                         break;
163                     case 10:
164                         BallXVelocity = 5;
165                         break;
166                     default:
167                         BallXVelocity = 6;
168                         break;
169                 }
170                 BallYVelocity = BallYVelocity * -1;
171                 BallY = paddle.PaddleY - BallHeight + 1;
172                 return true;
```

```
            }
            bool IsBlockHit = false;
            for (int i = 0; i < 7; i++)
            {
                if (IsBlockHit == false)
                {
                    for (int j = 0; j < 10; j++)
                    {
                        Block block = wall.BlockWall[i, j];
                        if (block.IsBlockVisible)
                        {
                            Rectangle BlockRect = new Rectangle((int)block.BlockX, (int)block.BlockY,
                                                                (int)block.BlockWidth, (int)block.BlockHeight);
                            if (HitTest(ballRect, BlockRect))
                            {
                                block.IsBlockVisible = false;
                                Score = Score + 7 - i;
                                BallYVelocity = BallYVelocity * -1;
                                BlocksCleared++;
                                IsBlockHit = true;
                                break;
                            }
                        }
                    }
                }
            }
            return true;
        }
        2 references
        public static bool HitTest(Rectangle r1, Rectangle r2)
        {
            if (Rectangle.Intersect(r1, r2) != Rectangle.Empty)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

This class has much of the logic for our game. Some of its properties are similar to our other classes. The *X* and *Y* properties are the ball's coordinates on the screen. `Height` and `Width` are the dimensions of the ball. Like the other classes, we are passing `spriteBatch` which will be used when we need to Draw the ball.

We are getting a reference to the ball image from the `gameContent.imgBall` field, and saving it in `imgBall`. The `Visible` property is used to determine if the ball needs to be drawn on the screen. The `XVelocity` and `YVelocity` fields are used to store how many `X` and `Y` pixels the ball will move every time the frame is updated. They are used to update the ball position on the screen.

Our ball will have the ability to rotate as it moves, so we have a `Rotation` property to specify how much our ball image should rotated when drawing it on the screen. We will update the rotation whenever we draw to the screen so the ball appears to spin as it moves.

We've added a property called `UseRotation` to determine if the ball should rotate. In the game, we will be drawing a ball next to the counter showing the number of balls remaining in the game, and we don't want that ball to spin like the regular game ball does. We can set this property to control the `Rotation`.

Since this class will be detecting block hits, it is a handy place to store a counter for Blocks that we have cleared on a game level, and also to keep a running game score. That's what the `BlocksCleared` and `Score` fields are used for.

All of our fields are initialized on the constructor call via passed arguments.

Our *Draw* method is similar to our other *Draw* methods, with a couple of exceptions. First, we are checking the `Visible` property to determine if the ball needs to be drawn. Also, we are checking the `UseRotation` property to determine whether to rotate the ball image before drawing, and if this property is true, we increase the rotation by a small amount. This is the first time that we are passing a non-zero value for the *Rotation* argument on the `spriteBatch.Draw` call. MonoGame actually does the rotation for us. We just have to provide the rotation angle, which we have stored in the `Rotation` property. The field after the `Rotation` argument on the `Draw` call specifies the origin for the rotation as an X, Y coordinate. We'll set this to the mid-point of the ball image.

The next method is called `Launch`. It will be called by the `Game1` class whenever we need to serve a new ball. It just initializes the ball's position and velocity and makes it visible.

The `Move` method is responsible for computing the ball's new position on the screen. It is called from the `Game1 Draw` method. In addition to updating the ball's coordinates, `Move` is responsible for detecting collisions with other objects in our game field. If the ball hits a side wall, we reverse the `XVelocity` by multiplying by "-1". If the ball hits the top wall we reverse the `YVelocity` by multiplying by "-1".

We use the `HitTest` method to determine if the ball hits the paddle or a block. It just checks if the rectangle formed by the ball intersects the rectangle formed by the paddle or one of the Blocks. If we hit the paddle, we reverse the `YVelocity` and the ball bounces back toward the top of the screen. The `X` position on the paddle will determine the direction and angle of deflection of the ball. If it hits the left half of the paddle, the ball will be deflected to the left, and the closer the ball hits to the left edge of the paddle, the sharper the angle of deflection will be. Similarly, if the ball hits the right half of the paddle, the ball will be deflected to the right, and the closer the ball hits to the right edge of the paddle, the sharper the angle of deflection will be.

For block hit detection, we just iterate through all of the Blocks in our `Wall` object. If the block `Visible` property is false, it has already been destroyed, so no collision with it can occur. If a hit is detected, we set that block's `Visible` property to false, increment our `BlocksCleared` field, and increment the users score. We give higher scores for the Blocks higher on the screen. So, Blocks in the first row are worth 1 point. Blocks in the second row are worth 2 points, and so on. If we do hit a block, we reverse the `YVelocity`.

If the ball's Y coordinate is greater than the `ScreenHeight` Property, the ball has fallen out of play, so we just set the `Visible` property to false.

Okay, that's it for the `Ball` class for now, but we need to connect it up to the game. So it's time to head back to the "Game1.cs" file. We need to add a field for our `Ball` object. In "Game1.cs", add the following line:

```
public class Game1 : Game
{
    // Create reference variables
    private GraphicsDeviceManager _graphics;
    private SpriteBatch _spriteBatch;
    GameContent gameContent;

    private Paddle paddle;
    private Wall wall;
    private GameBorder gameBorder;
    private Ball ball;
```

We'll initialize the `ball` object in our `LoadContent` method. Add the indicated line as shown below:

```
protected override void LoadContent()
{
    // New spriteBatch, used to draw bitmaps
    _spriteBatch = new SpriteBatch(GraphicsDevice);

    // Create a GameContent object that references the content assets
    gameContent = new GameContent(Content);

    // Get the current screensize
    screenWidth = GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Width;
    screenHeight = GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Height;

    // Set game to 502x700 or screen max if smaller
    if (screenWidth >= 502)
    {
        screenWidth = 502;
    }
    if (screenHeight >= 700)
    {
        screenHeight = 700;
    }
    _graphics.PreferredBackBufferWidth = screenWidth;
    _graphics.PreferredBackBufferHeight = screenHeight;
    _graphics.ApplyChanges();

    // Center the paddle horizontally on the screen
    int paddleX = (screenWidth - gameContent.imgPaddle.Width) / 2;

    // Set Paddle 100 pixels from the bottom of the screen
    int paddleY = screenHeight - 100;

    // Create game objects
    paddle = new Paddle(paddleX, paddleY, screenWidth, _spriteBatch, gameContent);
    wall = new Wall(1, 50, _spriteBatch, gameContent);
    gameBorder = new GameBorder(screenWidth, screenHeight, _spriteBatch, gameContent);
    ball = new Ball(screenWidth, screenHeight, _spriteBatch, gameContent);
}
```

We need to be able to serve the ball and keep track of the balls remaining. We need to add a couple of new fields. The `ballsRemaining` field will indicate how many balls the user still has available to play in the game. And the `readyToServeBall` will be used to determine if the game state permits the user to launch a new ball (i.e. there isn't a ball already in play). Add the indicated lines to the **Game1.cs** file as shown below:

```
23          // Track mouse state
24          private MouseState oldMouseState;
25          private KeyboardState oldKeyboardState;
26
27          private bool readyToServeBall = true;
28          private int ballsRemaining = 3;
29
```

We need to add logic to serve the ball. Add a new method called **ServeBall** to your **Game1.cs** file:

```
private void ServeBall()
{
    if (ballsRemaining < 1)
    {
        ballsRemaining = 3;
        ball.Score = 0;
        wall = new Wall(1, 50, _spriteBatch, gameContent);
    }
    readyToServeBall = false;
    float ballX = paddle.PaddleX + (paddle.PaddleWidth) / 2;
    float ballY = paddle.PaddleY - ball.BallHeight;
    ball.Launch(ballX, ballY, -3, -3);
}
```

The **ServeBall** method will check to see if we have any balls remaining. If we don't, it will reset the game screen for a new game. If we do, it will call the **ball.Launch** method to launch the ball. The ball is always launched from the coordinates of the middle of the paddle.

In our update method, we have to add code to Launch the ball when the user clicks the left mouse button, or when the user hits the space bar on the keyboard. Add the following lines to the **Update** method in **Game1.cs** as shown below:

```
protected override void Update(GameTime gameTime)
{
    if (IsActive == false)
    {
        return;  // Window is not active don't update
    }

    // Press the ESC key to exit the program
    if (Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // Get the current state of the keyboard and the mouse
    KeyboardState newKeyboardState = Keyboard.GetState();
    MouseState newMouseState = Mouse.GetState();

    // Process mouse move
    if (oldMouseState.X != newMouseState.X)
    {
        if (newMouseState.X >= 0 && newMouseState.X < screenWidth &&
            newMouseState.Y >= 0 && newMouseState.Y < screenHeight)
        {
            paddle.MoveTo(newMouseState.X);
        }
    }

    // Process left-click to serve ball
    if (newMouseState.LeftButton == ButtonState.Released &&
        oldMouseState.LeftButton == ButtonState.Pressed && oldMouseState.X ==
        newMouseState.X && oldMouseState.Y == newMouseState.Y && readyToServeBall)
    {
        ServeBall();
    }

    // Process keyboard events
    if (newKeyboardState.IsKeyDown(Keys.Left))
    {
        paddle.MoveLeft();
    }
    if (newKeyboardState.IsKeyDown(Keys.Right))
    {
        paddle.MoveRight();
    }

    // Process space bar to serve ball
    if (oldKeyboardState.IsKeyUp(Keys.Space) &&
        newKeyboardState.IsKeyDown(Keys.Space) && readyToServeBall)
    {
        ServeBall();
    }

    // Save the current state into the old state
    oldMouseState = newMouseState;
    oldKeyboardState = newKeyboardState;

    base.Update(gameTime);
}
```

Finally, in our "Game1.cs" file `Draw` method, we need to add logic to move and draw the ball. Add the indicated lines to "Game1.cs" as shown below:

```
151         protected override void Draw(GameTime gameTime)
152         {
153             GraphicsDevice.Clear(Color.Black);
154
155             // Begin drawing to buffer
156             _spriteBatch.Begin();
157
158             // Call the game objects Draw methods
159             paddle.Draw();
160             wall.Draw();
161             gameBorder.Draw();
162
163             if (ball.IsBallVisible)
164             {
165                 bool inPlay = ball.Move(wall, paddle);
166                 if (inPlay)
167                 {
168                     ball.Draw();
169                 }
170                 else
171                 {
172                     ballsRemaining--;
173                     readyToServeBall = true;
174                 }
175             }
176
177             // Write buffer to screen
178             _spriteBatch.End();
179
180             base.Draw(gameTime);
181         }
182     }
183 }
```

We've made quite a few updates, so now would be a good time to run the game by pressing **F5**. You should be able to press **Space** or click the mouse to launch the ball. You should see the ball moving and spinning. It should bounce off the walls, and destroy Blocks when it hits them. The paddle should deflect the ball, and the ball should fall out of play if you miss it with the paddle.

It's starting to look like a real breakout-type game, but some sounds would sure spice up game play!

Assignment Submission

Zip up the pseudocode and the project folder. Submit in Blackboard.