MonoGame Blocks Tutorial Part 5

Contents

Part 5 – Getting Keyboard and Mouse Inputs

Time required: 30 minutes

In the previous episode, we added the game border, completing the playing field. In this part, we'll start getting user input and applying it to the game.

The main game element that a user interacts with is the paddle. So, let's look at how we get inputs from the user and translate that into action in our game.

In our "Game1.cs" file, in the constructor, add the following two lines about the mouse and keyboard. We'll go over this new code in a bit.

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

    private Paddle paddle;
    private Wall wall;
    private GameBorder gameBorder;

    // Variables to store the screen size
    private int screenWidth = 0;
    private int screenHeight = 0;

    // Track mouse state
    private MouseState oldMouseState;
    private KeyboardState oldKeyboardState;
```

In "Game1.cs", add the indicated lines below to the `Update` method:

```
protected override void Update(GameTime gameTime)
{
    if (IsActive == false)
    {
        return;  // If the 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)
        {
            paddle.MoveTo(newMouseState.X);
        }
    }

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

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

    base.Update(gameTime);
}
```

Let's take a look at what we added. We added a test for `IsActive`. This tells us if our game is the active window. We don't want to process events if the program isn't the active window, so if we aren't active, we exit the update method.

We added two private variables called `oldMouseState` and `oldKeyboardState`. Why do we need those? This is one area where MonoGame programming is very different from what you are probably used to. In "normal" Windows programming, you get high-level events from the mouse like "Click". Those high-level events aren't available in MonoGame.

Instead, we have to compose them ourselves from the raw data that MonoGame provides to us. Basically, for the mouse, that is the current X and Y coordinate on the screen, button state (pressed or released), and thumb-wheel movement. We can't determine a "Click"

directly from this data. Instead, we need to have at least two data points from MonoGame, before we can detect a click event. Because of that, every time we get an Update event from MonoGame, we will need to save the previous Mouse and Keyboard state.

That's what `oldMouseState` and `oldKeyboardState` are for. At the end of each *Update* method call, we save the mouse and keyboard state, so that when the next *Update* event occurs, we can compare the new readings with the old ones. With that background, how do we create a mouse click event?

For a mouse click to have occurred, the user must press and release a mouse button without moving the mouse. The logic to detect a left mouse click is:

```
// Process left-click
if (newMouseState.LeftButton == ButtonState.Released &&
    oldMouseState.LeftButton == ButtonState.Pressed && oldMouseState.X ==
    newMouseState.X && oldMouseState.Y == newMouseState.Y && readyToServeBall)
```

We'll add that later to start the game and serve a ball.

For now, we will need to determine if the mouse has moved left or right. To do that, we have to see if the *X* coordinate has changed between the old and new readings. That's what this code does:

```
// 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);
    }
}
```

If the mouse *X* coordinate has changed, we'll change the paddle position to the new coordinate, by calling the `paddle.MoveTo` method. We'll only do this, if the mouse is within the boundary of our game field.

For the keyboard, we'll just move the paddle a fixed amount to the left or right when the user presses the left or right arrow, by calling the `paddle.MoveLeft` and `paddle.MoveRight` methods, respectively.

Press **F5** to run the game now. When you move the mouse left or right you should see the paddle on the screen move left and right. When you press the left or right keyboard arrow, you should see the paddle move left and right, as well.

Our paddle can move, but it can’t fulfill its life mission yet, since there is nothing to hit! It’s time to add a ball to our game.

Assignment Submission

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