Introduction to Game Programming

Session 3 – Collisions, motion improvement and other issues

Nacho Cabanes
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

Now it's time to talk about collisions. They help us control when our character touches an object, an enemy or a wall, and also when he collects something in the scene. Collisions are also used in shots, to check whether a shot has reached its target or not.

In console applications collision detection is quite easy, specially if our characters or elements are one cell width and height. That's why the collision detection that we are going to learn here is very basic. We will learn a more complex algorithm when talking about graphical environments.

Besides, we will improve the movement of the elements in our game. By now, the ball just moves to a random position in every iteration of the game loop, and it only moves each time we press a key. In this session, we will make it move step by step following a direction, and we will make this movement independent from our key input.

Finally, we will add some additional (minor) improvements to our video game, such as closing the window when we press a given key, or pausing the game to adjust its speed to our computer.

2. Colliding with screen limits

If we pay attention at how our scene has been built and drawn, is quite easy to detect if our character wants to go through a wall or screen limit. We only have to check if *characterX* is between the allowed X limits (0 and 40 – bar width). If this condition is true, then we can move our character to the new position. Otherwise, it will remain in its old position.

In order to centralise the bar width in a single place, we are going to define a constant to store it. Then, we can use this constant in step #2 of our game loop to check if we are goint to move to a valid position:

```
public class MyGame
{
    public static void Main()
    {
        // Constant to define bar width
        const int BAR_WIDTH = 5;
        …
        while(1 == 1)
        {
            …
            // 2 Read input and calculate player's new position
            …
            if (key.Key == ConsoleKey.LeftArrow && characterX > 0)
                characterX--;
            if (key.Key == ConsoleKey.RightArrow && characterX + BAR_WIDTH < 40)
                characterX++;
        …
```

Exercise 1

Update the project from previous sessions and add this check when we try to move the main character. It will only move if it goes to a valid X position.

3. Moving objects step by step

Let's change the movement of the ball. Instead of generating a random position in each iteration and place the ball right there, we are going to follow a direction. Initially, the ball will start from a given (static) position, and then it will start moving in a given direction. For instance, we can place it just on the bar. Besides, we are going to define an extra variable, called *ballYDirection*, to determine whether the ball is going up (negative) or down (positive).

```
public class MyGame
{
    public static void Main()
    {
        // Constant to define bar width
        const int BAR_WIDTH = 5;

        // Variables to store ball's coordinates
        int ballX = 20, ballY = 22, ballYDirection = -1;
        ...
```

3.1. Controlling ball bouncing

Then we can move the ball up until it reaches the upper border, then we start moving it down. At this point, the lines that generate a random position for the ball are pointless. You can remove them.

```
        while(1 == 1)
        {
            …
            ballX = DateTime.Now.Millisecond % 40;
            ballY = DateTime.Now.Millisecond % 24;
            …
            // 3 Move enemies and other objects

            if (ballY == 0)
                ballYDirection = -ballYDirection;
            ballY += ballYDirection;
        }
```

Note that, as soon as the ball reaches the upper border, we change the ball's Y direction to make it go down. However, if it reaches the lower border, it will keep going down for now. We will fix this later.

Exercise 2

Add this code to the videogame to make the ball start moving up from the bar's position.

3.2. Colliding with the bar

Now that we have our ball bouncing against the upper border, we can also make it bounce against the main character's bar. To do this, we check the collision in step #4 of our video game, and if there is a collision, then we change again the vertical direction of the ball. Try to think about what additional condition(s) must be checked in the “if” clause to make the ball bounce properly:

```
            // 4 Check collisions and update game state


            if (ballY == 22 && /* What else? */)
                ballYDirection = -ballYDirection;
```

Exercise 3

Add these changes to make the ball bounce against the bar and go up again.

4. Improving user input

One of the main problems of our video game by now is that the ball only moves when we press a key. In real video games, enemies or objects movements are independent from our input, so that they can reach us even if we don't move at all.

To do this, we must add this check before getting user input, so that we would only wait for this input if it actually exists:

```
if (Console.KeyAvailable)
{
   key = Console.ReadKey(false);
   // Calculate new position as we did before
}
```

Exercise 4

Add this check to user input (step #2 of the game loop) so that enemies can move independently from this input. You may notice that, with this update, enemies will move very fast now. We will correct this in next section.

5. Other improvements

5.1. Pausing the game loop

If we run the game loop with all the changes now, it may run too fast. To slow it down and adapt the video game to fast computers, we can add a pause at the end of every iteration (step #5 of our game loop). Just add this instruction:

```
while (true)
{
    ...
    System.Threading.Thread.Sleep(50);
}
```

This instruction tells the main thread (our application) to sleep a given number of milliseconds (300 in previous example, but you can change this value if you want to).

Exercise 5

Add a pause to the game loop so that it adjusts to your computer hardware and you can follow the ball movement with no effort. Initially, put the pause in step #5 of the game loop and check how it works. If it does not work properly, think where should it be placed to improve the game behavior.

5.2. Clearing the input buffer

Well, we have a reasonably valid version of the video game, but... Try to press and hold movement (for instance, the right arrow). What happens when you try to change your movement to the opposite direction?

If we press and hold a key, then a given number of "*ReadKey*" events accumulate, waiting to be consumed. This problem increases when we add a pause of X ms, because all the keys pressed during these milliseconds are waiting to be processed.

To solve this problem, whenever we detect that there is a key available (with *Console.KeyAvailable*, as seen before), we can use a *do..while* loop to read every pending key, until there is no keys left.

```
if (Console.KeyAvailable)
{
      do
      {
            key = Console.ReadKey(false);
      } while (Console.KeyAvailable);

      ... // if or switch to distinguish the movement type, as before
}
```

Note that we put our *Console.ReadKey* instruction inside the *do..while* loop, so that we read keys while there is any. Then, we check the last key processed and do this movement.

Exercise 6

Add this code to step #2 of our game loop to clear the input buffer after each iteration.

5.3. Closing the game window

Let's add a new key detection to let us close the game window when we don't want to play anymore. By now, we have to click on the corner of the window to close it, but this is not an appropriate way of closing a video game.

First of all, we are going to define a new boolean variable to check if we want to exit the video game. Declare it at the beginning of the *Main* method, along with the other existing variables:

```
public static void Main()
{
      ... // Other variables
      bool exitGame = false;
      ...
```

Next, we are going to change the condition of the main *while*. It will no longer be an infinite loop. We will keep on playing the game while we don't want to exit:

```
public static void Main()
{
      ...
      while(!exitGame)
      {
            ... Steps of the game loop
      }
}
```

Now, let's go to step #2 (user input). In the same "if" or "switch" structure that we have to decide the next movement for the main character, we are going to add a new branch to set the boolean variable to *true* when we press a given key (for instance, the Escape key):

```
public static void Main()
{
      ...
      while(!exitGame)
      {
            ...
            // 2. Read input and calculate player's new position
            ...
            if (key.Key == ConsoleKey.LeftArrow)
                  ...
```

```
            if (key.Key == ConsoleKey.Escape)
                  exitGame = true;
      }
}
```

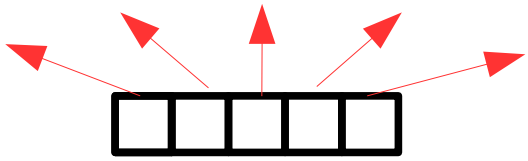
Exercise 7

Add this check to step #2 to let us quit the game whenever we want to. You may need to move to row number 24 after finishing the video game so that console message (*"Press a key to continue..."*) is not shown in the middle of the game screen.

5.4. To think a little...

To finish this session, let's think how to improve ball bouncing. For now, the ball only moves vertically, and it bounces against the upper limit and/or the bar. How could we make the ball move in other directions and bounce against the left/right walls as well?

To do this, we are going to make the ball change its direction every time it collides with a given section of the bar, according to this schema:



So, if ball collides with the left or right section of the bar, then it will start moving left or right, with an angle of 30º approximately. If it collides with the intermediate left and right sections, then it will start a 45º movement in this direction, and finally, if it collides with the center of the bar, it will move up.

Exercise 8

Think how we could implement this schema in our code, and try to implement it. Then, check collisions with every screen border (left and right), and change the X and Y direction of the ball.