IN628 2020 Practical 08 – Bounds Action

In this practical you will modify your **Sprite** class to control its behaviour at the boundaries of its legal area.

Adding bounds actions

Up until now we have not been too concerned with what happened to our sprites when they reached the edges of the screen. We may have simply let them disappear, or we may have implemented a rough bounds check where we hard-coded in the width and height of the Form. For a useful game sprite, we need better control than this. Specifically, our sprite must:

1. Have a configurable bounds action. For example a sprite might bounce at the edges of the screen (like the ball in Breakout), or be removed from the game when it hits the edge of the screen (like the pellets in the Rainbow Spitting Chicken). Other common sprite bounds actions are Wrap (appear at the opposite edge) and Stop (quit moving, but don't actually be removed from the sprite collection). A sprite's desired bounds action may also change during the execution of the game.

2. Have a configurable bounding area. Not all sprites are allowed to move in the entire visible game area. We may wish for sprites to remain in some arbitrary rectangular part of the screen (imagine a bee that stays near its flowers, or a dragon who guards her treasure trove).

Therefore we will now add two new properties to our **Sprite** – a bounds action and a bounding rectangle.

Bounding rectangle

Your bounding rectangle is most sensibly implemented as a **Rectangle** instance. You will be able to access the properties **Left**, **Top**, **Right** and **Bottom** directly to determine when your sprite has hit an edge of its bounding rectangle. You will need to pass the values for the bounding rectangle into your **Sprite** constructor. If the sprite is restricted to a particular area, pass in a rectangle that represents the coordinates of that area. For now, this should be expressed in the coordinate space of the Form.

Determining that an edge has been hit

Each time you move your sprite, you will need to check if it has hit an edge of its bounding rectangle by inspecting the values of **xPos** and **yPos**. You can make this check after you modify **xPos** and **yPos** in the **Sprite::Move()** method. Checking for the **Left** and **Top** is quite straightforward: if **xPos** is less than the **Left** of the bounding rectangle, your sprite is off the left edge; if **yPos** is less than the **Top** of the bounding rectangle your sprite is off the top edge. However, checking for the **Right** and **Bottom** edges is more complex. If you wait for **xPos > boundingRectangle.Right** to be true, your sprite will appear to completely exit the screen (or its allowed area) before performing its bounds action, because this condition will not be true until the sprites' left edge has cleared the right hand border of its bounding area.

A sprite who is bouncing, for example, will seem to wander off the screen completely, and then slightly later, wander back on.

To detect the hitting of the right-hand edge of the bounding rectangle we need to inspect not **xPos** (the left-hand edge of the sprite), but **xPos + frameWidth** (the right-hand edge of the sprite's visible representation). Similarly, to detect the hitting of the bottom edge of the bounding rectangle we need to inspect **yPos + frameHeight**.
Therefore, in pseudocode, you have the logic:

```
if  (xPos < boundingRectangle.Left) ||
((xPos + frameWidth) > boundingRectangle.Right) ||
(yPos < boundingRectangle.Top) ||
((yPos + frameHeight) > boundingRectangle.Bottom)
```

The sprite is out of bounds and you need to implement the bounds action.

Representing bounds action state

It is efficient to represent your bounds action state with an integer. This allows for easy conditional logic (see below). But of course we don't want to use actual numbers in our code, as it will be very difficult to remember that "0 is bounce and 1 is wrap and ….." We therefore need to use #define statements or an enumeration to assign logical names to the integer variables we use. For example, in Sprite.h you might have these statements:

```
#define BOUNCE 0
#define WRAP 1
#define DIE 2
#define STOP 3
```

Or you might define

```
public enum EBoundsAction { BOUNCE, WRAP, DIE, STOP };
```

Implementing bounds action

The precise impact of hitting an edge depends on the current bounds action state of the sprite. For example, if the bounds action is **BOUNCE**, you need to change the **spriteDirection** property; if the bounds action is **STOP**, you need to set **xVel** and **yVel** to 0, and so on. Thus what action you take will depend on the value of the **boundsAction** state variable, and will require a switch statement. In pseudocode:

If the sprite has hit an edge, switch (boundsAction)

case (BOUNCE)
Change the sprite's direction as appropriate. Again you will need to distinguish between hitting a horizontal edge and hitting a vertical edge. (Although, if you have ordered your #define or enum directions: { EAST, SOUTH, WEST NORTH } => { 0, 1, 2, 3 }, the bounce can be implemented with a single mathematical expression that applies regardless of which edge was hit. This is left as a fun exercise for you.)

case (WRAP)
Move the sprite around to the opposite edge. You can get the location of the opposite edge from the **boundingRectangle**. Note that you will need to distinguish between hitting a horizontal edge and hitting a vertical edge.
case (STOP)
Set **xVel** and **yVel** to 0. You will also need to make the animation stop running, if that is appropriate.
case (DIE)
Get rid of the sprite. This may be implemented by setting an **IsAlive** property to false and dealing with it later.

Maintaining good modularity

The preceding discussion has added a lot of code to the **Sprite::Move()** method. If you just dump all this code directly into the method, it will become unwieldy and hard to read, modify and maintain. Make sure that each logically distinct computational element is encapsulated in its own method, and those methods are called as required in **Sprite::Move().**