Introduction to Game Programming

Session 3 – The game loop and sprites movement

Nacho Cabanes
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

In this session we are going to:

- Add the game loop to our video game.
- Start implementing some of the steps of the loop, such as moving the main character with key events
- Add some animation to the main character sprite

1.1. What is a “sprite”?

In previous session we learnt how to draw images. But in this session we are going to learn how to deal with a **sprite**. A sprite is a special type of image that represents an element of the game itself, such as the main character, an enemy, and obstacle...

2. The game loop

In order to move sprites in a video game, we need to add some kind of loop that lets us change the position of these sprites after each iteration. We are going to use the same game loop that we used in our console video game. Add it to the *Show* method of our *GameScreen* class. Besides, we are going to change the exit condition: instead of pressing the space bar key, we are going to exit this screen when pressing escape key:

```
public override void Show()
{
    int keyPressed;
    do
    {
        keyPressed = hardware.KeyPressed();

        // 1. Draw everything
        hardware.ClearScreen();
        hardware.DrawImage(imgCharacter);
        hardware.UpdateScreen();

        // 2. Move character from keyboard input

        // 3. Move enemies and objects

        // 4. Check collisions and update game state

        // 5. Pause game
    }while (keyPressed != Hardware.KEY_ESC);
}
```

If you try to run the project now, it should work as it did before these changes. In next sections we are going to cover some of the empty steps of the game loop.

3. Moving sprites with the keyboard

Now, we are going to make the main character move in all directions. Let's go to step #2 of the game loop and detect the key pressed (any of the four cursor keys) and move the character image to this direction. For instance, if we want to move it left, we can do something like this:

```
// 2. Move character from keyboard input
if (keyPressed == Hardware.KEY_LEFT)
    imgCharacter.MoveTo((short)(imgCharacter.X - 1), imgCharacter.Y);}
```

Note that we need to define a *cast* conversion when incrementing or decrementing the X property, since it is *short*, and every arithmetic operation with it turns it into *int*, but we need it to be short because the SDL drawing methods work with *short* parameters.

If you run the program now, you can see how the character moves left, but if we hold the key down, the character does not keep moving, so it only moves once per each key press.

If we want to improve this, and let the character move as we hold the key down, we need to add a new method to our *Hardware* class, called *IsKeyPressed*. This method will return *true* if a given key is being pressed, and *false* if not:

```
public bool IsKeyPressed (int key)
{
    bool pressed = false;
    Sdl.SDL_PumpEvents();
    Sdl.SDL_Event evt;
    Sdl.SDL_PollEvent(out evt);
    int numKeys;
    byte[] keys = Sdl.SDL_GetKeyState(out numKeys);
    if (keys[key] == 1)
        pressed = true;
    return pressed;
}
```

Then, we use this method instead of *KeyPressed* method in step #2 of the game loop:

```
// 2. Move character from keyboard input
if (hardware.IsKeyPressed(Hardware.KEY_LEFT))
    ...
```

Now, our character should move properly. But... why do we need this new method? We will use *KeyPressed* method whenever we want to register a single key press (for instance, to switch from welcome screen to game screen), and we will use *IsKeyPressed* method when we want to register any key hold event (for instance, to move a character).

Exercise 1

Add all the movements (right, up and down) to the main character, and check that it can move in any direction (even diagonally). Also add some *Sleep* to step #5 of the game loop (for instance, 10 ms), so that it does not move so fast.

4. Animating sprites

In this section we are going to see how to deal with animated sprites, this is, sprites that change their image as game goes on. It can be a character that changes its image as we move it with the keyboard, or a sprite that periodically changes its image according to some parameters (time, collisions…).

4.1. How to animate a sprite

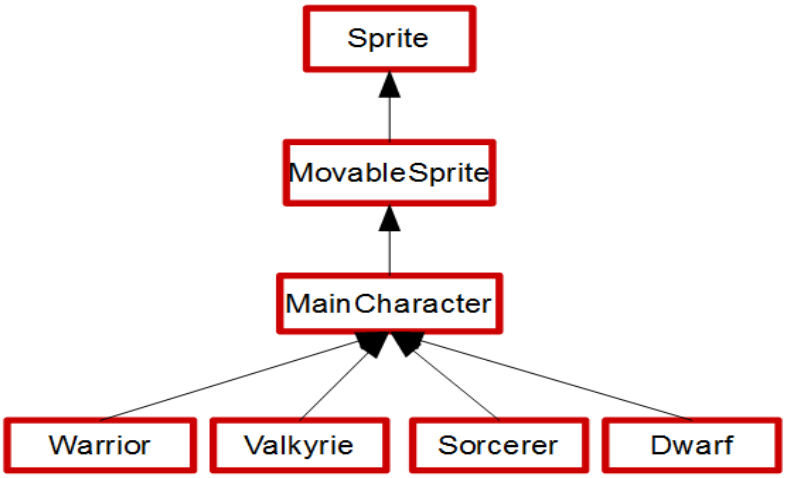
Animating a sprite implies having more than one image for this sprite, so that we can change the image shown every time. For instance, if we are implementing a Mario video game, and want to animate Mario as he moves in the scene, we need to have some different images of Mario, such as:

These images can be:

- In separate image files (for instance, *mario1.png, mario2.png, ...*). This is not a good idea if you have many images to deal with.
- In a single image file called **spritesheet** (for instance, *mariosprites.png*), so that we "crop" the part of the spritesheet that we need. This is the most appropriate option, normally, and this is what we are going to do with our video game.

4.2. Our sprite class structure

First of all, let's create a new set of classes to define our sprite class structure. The simplified class diagram to represent the relationships among these classes is like this:



4.2.1. The Sprite super class

First of all, we are going to define a *Sprite* superclass to store all the common information about every type of sprite:

- We define two constants to represent the width and height of every sprite in our video game (46px width and height)
- We also define two properties to store the screen coordinates in which the sprite needs to be drawn (X and Y)
- Finally, we define two more properties to determine the X and Y coordinates of the sprite in the sprite sheet that we want to draw (*SpriteX* and *SpriteY*)
- In addition, we define a method that lets us move the sprite to a new position in the screen whenever we want.

```
class Sprite
{
    public const short SPRITE_WIDTH = 46;
    public const short SPRITE_HEIGHT = 46;

    public short X { get; set; }
    public short Y { get; set; }
    public short SpriteX { get; set; }
    public short SpriteY { get; set; }

    public void MoveTo(short x, short y)
    {
        X = x;
        Y = y;
    }
}
```

4.2.2. The MovableSprite class

This class will inherit from *Sprite* superclass to represent every sprite that can move in the video game, such as the main character, the enemies, or a weapon that is thrown.

```
class MovableSprite: Sprite
{
    public enum SpriteMovement { LEFT, LEFT_UP, UP, RIGHT_UP, RIGHT,
                                 RIGHT_DOWN, DOWN, LEFT_DOWN };
    public SpriteMovement CurrentDirection { get; set; }

    public MovableSprite()
    {
        CurrentDirection = SpriteMovement.LEFT;
    }
```

```
}
```

We are going to define this basic structure for this class, but we are going to add some more new methods and properties soon. As you can see, we create a public *enum* to define all the possible movements of a movable sprite: up, down, left, right and the four diagonal movements, represented by a combination of left/right and up/down. Then, we define a property to store the current direction of the sprite (one of the eight directions defined in the *enum*), and we initialize this property in the constructor.

4.2.3. The MainCharacter class and its subclasses

We also define the *MainCharacter* subclass as a subtype of *MovableSprite* class, to represent the specific features of the main character, such as energy or points. In later sessions we will also define some other subtypes of *MovableSprite* class, such as enemies or weapons.

```
class MainCharacter: MovableSprite
{
    public ushort Energy { get; set; }
    public ushort Points { get; set; }
}
```

This class will have four specific subclasses, representing the four characters that can be chosen in the *PlayerSelectScreen*: the warrior, the valkyrie, the sorcerer and the dwarf. Initially, we are going to leave these classes empty, but we will fill them in a few minutes. For instance, this is the code for the *Warrior* class (all other three classes follow the same pattern):

```
class Warrior: MainCharacter
{
    public Warrior(): base()
    {
    }
}
```

4.3. Animation with spritesheets

Now that we have our sprite class structure defined, let's start with our sprite animation schema. If we are using a spritesheet, then we only need to load one (big) image, and crop the part that we need for each sprite. We are going to load the game sprite sheet (*gauntlet_spritesheet.png* image) as a static property of the *Sprite* class:

```
class Sprite
{
    public static Image SpriteSheet =
        new Image("imgs/gauntlet_spritesheet.png", 2385, 768);
    ...
```

Let's see how to animate our main character. If every sprite has the same size (as in our example), we can do this very easily by determining the X and Y coordinate of the region to be drawn... then, the width and height values will always be the same (we stored these

values in `Sprite.SPRITE_WIDTH` and `Sprite.SPRITE_HEIGHT` constants: 46px each).

Regarding our main character, we have 3 different sprites to animate each direction (left, right, up, down and each diagonal). Unfortunately, these sprites are separated in the spritesheet. Have a look at the warrior, for instance. If we want to move it left, the 3 sprites are placed in positions 7, 15 and 23 of the 10th row of sprites:

The X coordinates of these 3 sprites are 334, 766 and 1198, respectively, and the Y coordinate is 502 for all of them. It happens something similar with the rest of directions and with the rest of characters (valkyrie, sorcerer and dwarf), so we need an easy way of storing the coordinates of each group of sprites, and to switch from on group to another as we press the cursor keys.

4.3.1. Defining our sprite array for each character type

Every movable sprite of our video game will be able to move in 8 directions stored in *enum MovableSprite.SpriteMovement*. So we are going to define two bidimensional arrays of 8 rows, each one representing the X and Y coordinates of a given direction (all the directions of that *enum*, in the same order). So let's define these two arrays as properties of our *MovableSprite* class:

```
class MovableSprite: Sprite
{
    const byte TOTAL_MOVEMENTS = 8;
    public enum SpriteMovement { LEFT, LEFT_UP, UP, RIGHT_UP, RIGHT,
                                 RIGHT_DOWN, DOWN, LEFT_DOWN };
    public SpriteMovement CurrentDirection { get; set; }
    public int[][] SpriteXCoordinates = new int[TOTAL_MOVEMENTS][];
    public int[][] SpriteYCoordinates = new int[TOTAL_MOVEMENTS][];
...
```

We also define a constant to store the total number of possible movements (the length of the *enum*).

Now, it's time to fill this array in each subtype of *MainCharacter* class, as each subtype has its sprites in different coordinates. This is how it works for LEFT, UP and DOWN movements of the warrior:

```
class Warrior: MainCharacter
{
    public Warrior(): base()
    {
        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.LEFT] = new int[] { 334, 766, 1198 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.LEFT] = new int[] { 502, 502, 502 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.UP] = new int[] { 8, 442, 874 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.UP] = new int[] { 502, 502, 502 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.DOWN] = new int[] { 226, 658, 1090 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.DOWN] = new int[] { 502, 502, 502 };
    }
}
```

If you have a look at the spritesheet with some image processing software such as Gimp, you can see that left sprites for the warrior are located in coordinates (334, 502), (766, 502) and (1198, 502), which are represented by the first two lines of the constructor (X and Y coordinates, respectively).

4.3.2. Adding changes to MovableSprite class

In this step, we are going to add some more properties and methods to our *MovableSprite* class to let us animate the sprites. First of all, we are going to define a property to determine the current sprite of the sequence (left, right, up... or whatever) that we want to draw:

```
class MovableSprite: Sprite
{
    const byte TOTAL_MOVEMENTS = 8;
    public enum SpriteMovement { LEFT, LEFT_UP, UP, RIGHT_UP, RIGHT,
                                 RIGHT_DOWN, DOWN, LEFT_DOWN };
    public SpriteMovement CurrentDirection { get; set; }
    public byte CurrentSprite { get; set; }
```

Then, we initialize it in the constructor:

```
    public MovableSprite()
    {
        CurrentSprite = 0;
        CurrentDirection = SpriteMovement.LEFT;
    }
```

We also create a new method called *Animate* to animate the sprite, changing the direction and current sprite depending on the current movement:

```
    public void Animate(SpriteMovement movement)
    {
        if (movement != CurrentDirection)
        {
            CurrentDirection = movement;
            CurrentSprite = 0;
        } else {
                CurrentSprite = (byte)((CurrentSprite + 1) %
                    SpriteXCoordinates[(int)CurrentDirection].Length);
        }
        UpdateSpriteCoordinates();
    }
```

As you can see, if the movement has not changed from previous movement, then we simply increase the current sprite (taking the module with the total number of sprites of the sequence. If movement has changed, then we reset the current sprite to 0. Finally, we call a method called *UpdateSpriteCoordinates* that is not defined yet. We can add it next to this one:

```
    protected void UpdateSpriteCoordinates()
    {
        SpriteX = (short)(SpriteXCoordinates[(int)CurrentDirection][CurrentSprite]);
        SpriteY = (short)(SpriteYCoordinates[(int)CurrentDirection][CurrentSprite]);
    }
```

We just update *SpriteX* and *SpriteY* properties to store the coordinates of the currently selected sprite to be drawn in the screen. We can also call this method at the end of each constructor of the main character subtypes (*Warrior, Valkyrie, Sorcerer* and *Dwarf*). For instance, our *Warrior* constructor may look like this one:

```
    public Warrior(): base()
    {
        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.LEFT] = ...
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.LEFT] = ...
        ...
        UpdateSpriteCoordinates();
    }
```

4.3.3. Changes in Hardware class

To determine which sprite is going to be drawn each time, we are going to add a new method to our *Hardware* class:

```
public void DrawSprite(Image image, short xScreen, short yScreen,
                       short x, short y, short width, short height)
{
    Sdl.SDL_Rect src = new Sdl.SDL_Rect(x, y, width, height);
    Sdl.SDL_Rect dest = new Sdl.SDL_Rect(xScreen, yScreen, width, height);
    Sdl.SDL_BlitSurface(image.ImagePtr, ref src, screen, ref dest);
```

```
}
```

We had another version of this method with only an *Image* parameter. In this case, we are going to add six more parameters, to indicate the screen coordinates in which we want to draw the sprite (*xScreen* and *yScreen*), and the piece of the spritesheet that we want to draw (represented by its upper left coordinates x and y, and the width and height of the region to be drawn).

4.3.4. Changes in GameScreen class

Finally, we need to change some elements in our *GameScreen* class to adapt it to this sprite animation structure. Firs of all, we are no longer going to deal with an *Image* for the main character, so replace this line:

```
class GameScreen: Screen
{
    Image imgCharacter;
```

with this one:

```
    MainCharacter character;
```

As you can see, we are not going to deal with an *Image* object for the main character, but with a *Sprite* subtype. Then, when we select which character type we are going to play with, we just need to create an instance of the appropriate subtype:

```
    chosenPlayer = value;
    switch (value)
    {
        case 1:
            character = new Warrior();
            break;
        case 2:
            character = new Valkyrie();
            break;
        case 3:
            character = new Sorcerer();
            break;
        case 4:
            character = new Dwarf();
            break;
    }
    character.MoveTo(380, 280);
```

Now, let's move to the game loop. In step #1, we will use *DrawSprite* method defined before to draw the chosen sprite of the main character in every iteration:

```
    // 1. Draw everything
    hardware.ClearScreen();
    hardware.DrawSprite(Sprite.SpriteSheet, character.X, character.Y,
        character.SpriteX, character.SpriteY, Sprite.SPRITE_WIDTH, Sprite.SPRITE_HEIGHT);
    hardware.UpdateScreen();
```

In step #2 (moving character with user input), we move the sprite to the new position and then call to *Animate* method to animate the character in the given direction. This is how it works for left, up and down movements:

```
    // 2. Move character from keyboard input
    if (hardware.IsKeyPressed(Hardware.KEY_LEFT))
    {
        character.MoveTo((short)(character.X - 1), character.Y);
        character.Animate(MovableSprite.SpriteMovement.LEFT);
    }
    if (hardware.IsKeyPressed(Hardware.KEY_UP))
    {
        character.MoveTo(character.X, (short)(character.Y - 1));
        character.Animate(MovableSprite.SpriteMovement.UP);
    }
    if (hardware.IsKeyPressed(Hardware.KEY_DOWN))
    {
        character.MoveTo(character.X, (short)(character.Y + 1));
        character.Animate(MovableSprite.SpriteMovement.DOWN);
    }
```

4.3.5. Slowing down the animation

If you run your project right now, you can see how it works with the warrior, and move left, up or down (separately). But you can notice that the animation goes too fast. In order to slow it down, we can add some little changes to our *MovableSprite* class. What we are going to do is basically change the current sprite after some iterations or steps (not at every iteration), and to achieve this, we need to define a new constant to determine how many iterations must be completed before changing current sprite:

```
class MovableSprite: Sprite
{
    const byte TOTAL_MOVEMENTS = 8;
    const byte SPRITE_CHANGE = 10;
```

Then, we add a new attribute to count the iterations before changing the sprite:

```
    byte currentSpriteChange;
```

We initialize this attribute in the constructor:

```
    public MovableSprite()
    {
        CurrentSprite = 0;
        CurrentDirection = SpriteMovement.LEFT;
        currentSpriteChange = 0;
    }
```

and we use it in the *Animate* method to change the current sprite only when the counter reaches the constant value:

```
public void Animate(SpriteMovement movement)
{
    if (movement != CurrentDirection)
    {
        CurrentDirection = movement;
        CurrentSprite = 0;
        currentSpriteChange = 0;
    } else {
        currentSpriteChange++;
        if (currentSpriteChange >= SPRITE_CHANGE)
        {
            currentSpriteChange = 0;
            CurrentSprite = (byte)((CurrentSprite + 1) %
                SpriteXCoordinates[(int)CurrentDirection].Length);
        }
    }
    UpdateSpriteCoordinates();
}
```

Try to run the game again, and see how the animation slows down to an appropriate speed.

5. To think a little

You have been provided with a complete structure to animate the main character (as long as it is the warrior) to three basic directions: left, up and down. But there is a lot of work pending:

- (**1,5 points**) Complete the animation of all the 5 pending directions for the warrior (right, right-up, right-down, left-up and left-down). You may need to change the logic of step #2 of your game loop in order to allow diagonal animations.
- (**1,5 points... 0,5 points each**). Implement the animations of the 3 pending main character subtypes: the valkyrie, the sorcerer and the dwarf, in the same way that you did for the warrior (adding the X and Y coordinates array in the constructor).