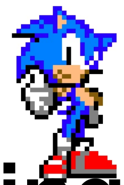
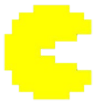
Introduction to Game Programming





Session 5 – Collisions and scroll

Nacho Cabanes
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

Collision detection can be used in video games for several purposes:

- Losing one life, or decrease energy levels, or destroy an enemy if a character collides with an enemy, or a fire shot collides with either the main character or an enemy
- Prevent the main character or an enemy from going beyond the scene limits
- Take objects that are present in the scene
- ... etc.

In this session we are going to add some more features to our video game:

- Firstly, we are going to control the screen boundaries, so that main character can't go beyond those limits.
- Then, we are going to implement a collision detection algorithm to check if the main character collides with static objects, such as walls.
- Next, we are going to add a big stage so that it does not fit in the screen size, and we will need to implement some scroll in order to reach all the corners of the stage
- Finally, you will be asked to implement some kind of stage edition, so that we can easily place walls all around the stage.

1.1. Previous steps

Before going on with the main contents of this session, lets add two minor changes to our current video game.

1.1.1. About the memory usage increase

If you play the game right now, you can notice that the total amount of memory needed is continuously growing as you load the *GameScreen* screen. The reason why this happens is a problem when rendering texts. So, first of all, let's add the two namespaces involved in this process:

```
using System;
using System.Threading;
using Tao.Sdl;
```

Next, let's add two attributes to our *GameScreen* class to pre-render the two text (energy and points) there:

```
class GameScreen: Screen
{
    MainCharacter character;
    Level level;
    int chosenPlayer;
    Font font;
    Audio audio;
    IntPtr textPoints, textEnergy;
```

Then, add a new, private method to this class to load the texts as images:

```
private void initTexts()
{
    Sdl.SDL_Color r = new Sdl.SDL_Color(255, 0, 0);
    Sdl.SDL_Color g = new Sdl.SDL_Color(0, 255, 0);
    textEnergy = SdlTtf.TTF_RenderText_Solid(font.GetFontType(), "ENERGY:", r);
    textPoints = SdlTtf.TTF_RenderText_Solid(font.GetFontType(), "POINTS:", g);
    if (textEnergy == IntPtr.Zero || textPoints == IntPtr.Zero)
        Environment.Exit(5);
}
```

And finally, call this method from the constructor:

```
public GameScreen(Hardware hardware): base(hardware)
{
    ...
    initTexts();
}
```

Besides, replace the old *Hardware.WriteText* method with this new one:

```
public void WriteText(IntPtr textAsImage, short x, short y)
{
    Sdl.SDL_Rect src = new Sdl.SDL_Rect(0, 0, screenWidth, screenHeight);
    Sdl.SDL_Rect dest = new Sdl.SDL_Rect(x, y, screenWidth, screenHeight);
    Sdl.SDL_BlitSurface(textAsImage, ref src, screen, ref dest);
}
```

1.1.2. Adding the session resources

Download and unzip the resources for this session. You will find a new DLL file to deal with JPEG images (*jpeg.dll*). Copy it in the main folder of your project. Besides, there is an image called *floor.jpg*. Copy it in the *imgs* folder. Remember to check the *Copy always* property for both files.

1.1.3. Adding and using new constants

In order to make our code more adaptive and understandable, let's create some constants referring to screen limits. Add them in *GameController* class, for instance:

```
public const short SCREEN_WIDTH = 800;
public const short SCREEN_HEIGHT = 500;
```

And use them whenever these dimensions are specified. For instance, here at *GameController*:

```
Hardware hardware = new Hardware(SCREEN_WIDTH, SCREEN_HEIGHT + 100, 24, false);
```

or here at *GameScreen*:

```
hardware.WriteText("ENERGY:", 5, GameController.SCREEN_HEIGHT + 50, 255, 0, 0, font);
hardware.WriteText("POINTS:", GameController.SCREEN_WIDTH - 200,
 GameController.SCREEN_HEIGHT + 50, 0, 255, 0, font);
```

1.1.4. Improving character motion code

In this session we are going to add some important features to main character motion detection, so step #2 of the game loop may become quite tricky. In order to avoid this, let's create a private method called *moveCharacter* in our *GameScreen* class, with this code:

```
private void moveCharacter()
{
    bool left = hardware.IsKeyPressed(Hardware.KEY_LEFT);
    bool right = hardware.IsKeyPressed(Hardware.KEY_RIGHT);
    bool up = hardware.IsKeyPressed(Hardware.KEY_UP);
    bool down = hardware.IsKeyPressed(Hardware.KEY_DOWN);

    if (up)
    {
        character.Y--;
    }
    if (down)
    {
        character.Y++;
    }
    if (left)
    {
        character.X--;
    }
    if (right)
    {
        character.X++;
    }

    if (left)
        if (up) character.Animate(MovableSprite.SpriteMovement.LEFT_UP);
        else if (down) character.Animate(MovableSprite.SpriteMovement.LEFT_DOWN);
        else character.Animate(MovableSprite.SpriteMovement.LEFT);
    else if (right)
        if (up) character.Animate(MovableSprite.SpriteMovement.RIGHT_UP);
        else if (down) character.Animate(MovableSprite.SpriteMovement.RIGHT_DOWN);
        else character.Animate(MovableSprite.SpriteMovement.RIGHT);
    else if (up)
        character.Animate(MovableSprite.SpriteMovement.UP);
    else if (down)
        character.Animate(MovableSprite.SpriteMovement.DOWN);
}
```

Then, we replace the whole step #2 of the game loop with this method call:

```
// 2. Move character from keyboard input
```

```
moveCharacter();
```

1.1.5. Changing player's initial position

In order to ease some aspects of the video game for future changes, let's change the main character's initial position to this one:

```
character.MoveTo(400, 250);
```

2. Checking screen boundaries

If you remember from past sessions, we have defined a playable scene of 800 pixels width and 500 pixels height (we drew a yellow line at Y = 500 last session, to write some information at the bottom of the game screen).

So, in every player move, we need to check if he's going to move out of these boundaries. In other words:

- New X coordinate must be >= 0 and <= 800 – player width
- New Y coordinate must be >= 0 and <= 500 – player height

We need to check these conditions in *moveCharacter* method:

```
private void moveCharacter()
{
    bool left = hardware.IsKeyPressed(Hardware.KEY_LEFT);
    bool right = hardware.IsKeyPressed(Hardware.KEY_RIGHT);
    bool up = hardware.IsKeyPressed(Hardware.KEY_UP);
    bool down = hardware.IsKeyPressed(Hardware.KEY_DOWN);

    if (up)
    {
        if (character.Y > 0)
            character.Y--;
    }
    if (down)
    {
        if (character.Y < GameController.SCREEN_HEIGHT - Sprite.SPRITE_HEIGHT)
            character.Y++;
    }
    ...
```

Add these checks with every possible movement of the main character.

3. Colliding with obstacles

Now, we are going to add some static obstacles to our video game. First of all, lets add a simple obstacle at some known coordinates, and check if the main character collides with it, and then we are going to define a basic level structure so that we can add obstacles and other objects to each level.

3.1. Defining the obstacles class hierarchy

An obstacle is some kind of sprite, so we need to define a class that inherits from *Sprite* class. Moreover, an obstacle is a static sprite, this is, a sprite that is not going to be moving around the scene. So let's create two classes.

3.1.1. StaticSprite class

This class is going to represent any static element of the scene, such as obstacles, exit points and so on. This class is going to be empty for now:

```
class StaticSprite : Sprite
{
}
```

3.1.2. Wall class

We are going to add some wall(s) to the scene, and these walls will be subtypes of *StaticSprite* class. So let's define a new class called *Wall*. This class will define the sprite coordinates in its constructor. Besides, we define an additional constructor to directly assign the screen coordinates to the wall.

```
class Wall: StaticSprite
{
    public Wall()
    {
        SpriteX = 1414;
        SpriteY = 664;
    }

    public Wall(short x, short y): this()
    {
        X = x;
        Y = y;
    }
}
```

3.2. Collision checking with single obstacles

Let's try adding a *Wall* object to the *GameScreen* class:

```
class GameScreen: Screen
```

```
{
    MainCharacter character;
    Wall wall;
    ...
```

Then, we initialize it at the constructor:

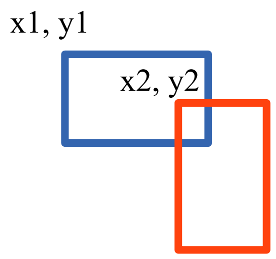
```
public GameScreen(Hardware hardware): base(hardware)
{
    ...
    wall = new Wall(200, 300);
}
```

And finally, we draw it at step #1 of the game loop:

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

3.2.1. Bidimensional simple collision detection algorithm

Regarding our video game, it is a 2D video game, so we only have to care about X and Y coordinates of both objects, along with their widths and heights. This is, we put every sprite inside a box that surrounds it, and we only have to check if both boxes collide.



Let's assume that one of the sprites has coordinates (x1, y1), and dimensions w1 and h1, and the other one is at (x2, y2), with dimensions w2 and h2 (see left image).

Both sprites will collide if there is any shared area. We can detect this with a simple method (we can add it to our *Sprite* class).

```
public bool CollidesWith(Sprite sp)
{
    return (X + Sprite.SPRITE_WIDTH > sp.X && X < sp.X + Sprite.SPRITE_WIDTH &&
            Y + Sprite.SPRITE_HEIGHT > sp.Y && Y < sp.Y + Sprite.SPRITE_HEIGHT);
}
```

3.2.2. Applying collision detection to the video game

Let's apply previous method to our video game. What we are going to do is move the character to the new position whenever he wants to move, then check if there is a collision with the obstacle(s) in this new position, and if so, we will put the character back to his previous coordinates.

First of all, we need two variables to store previous X and Y coordinates before moving the character:

```
public override void Show()
{
    short oldX, oldY;
```

Then, we store these coordinates at the beginning of step #2 of the game loop.

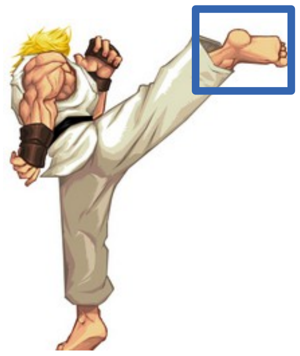
```
// 2. Move character from keyboard input
oldX = character.X;
oldY = character.Y;
moveCharacter();
...
```

Finally, we check collisions in step #4, and if there is any, we will just place the character in its previous coordinates:

```
// 4. Check collisions and update game state
if (character.CollidesWith(wall))
{
    character.X = oldX;
    character.Y = oldY;
}
```

3.2.3. More complex collisions

There are other video game types where collision detection is not so simple. If we are programming a fight video game such as Street Fighter, then we need to check if the arm or leg of one fighter collides with the other fighter. One way of achieving this is to determine a given area in each sprite, and check if this area collides with the other player. For instance, if one fighter is trying to kick his opponent, then we can make a box around his foot:



Then, we can use previous collision detection algorithm to check if this box collides with opponent's box.

3.3. Defining levels with obstacles

We are not going to draw single obstacles in the game screen, but whole levels with several obstacles, exit points, objects to be taken (keys, treasures...). All these elements are part of a given level, so we are going to define a *Level* class to store this information. To start with, we are only going to deal with a list of walls (sprites), and a floor background, along with the level width and height. Later we will add other elements to the level.

```
class Level
{
    public List<Sprite> Walls { get; }

    public Level()
    {
        Walls = new List<Sprite>();
    }

    public void AddWall(Wall w)
    {
        Walls.Add(w);
    }
}
```

Now, remove the lines referring to the single *Wall* object in the *GameScreen*, and add a *Level* attribute.

```
class GameScreen: Screen
{
    MainCharacter character;
    ~~Wall wall;~~
    Level level;
    ...
    public GameScreen(Hardware hardware): base(hardware)
    {
        ...
        ~~wall = new Wall(200, 300);~~
        level = new Level();
    }
```

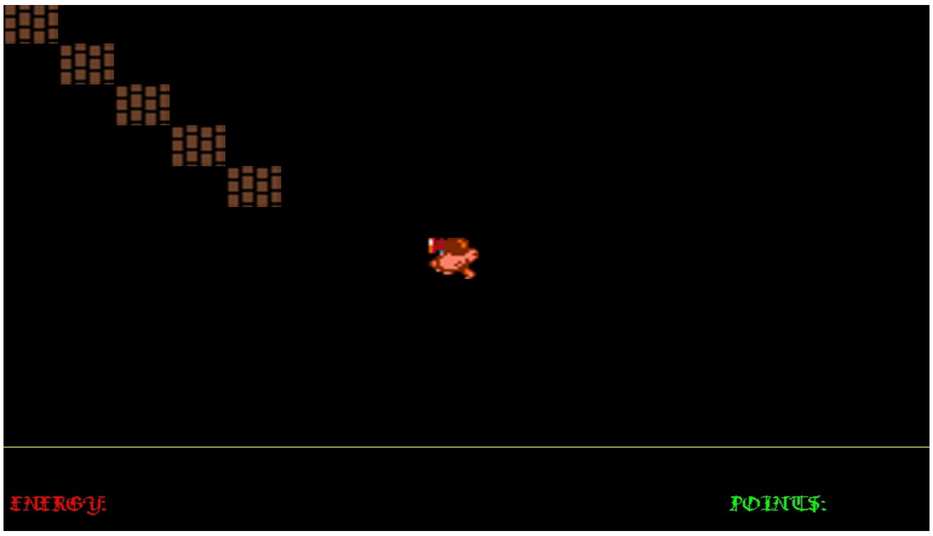
We may also define a private method called *loadLevel* in *GameScreen* to initialize the level with some walls:

```
private void loadLevel()
{
    for (int i = 0; i < 5; i++)
        level.AddWall(new Wall((short)(i * Sprite.SPRITE_WIDTH),
                               (short)(i * Sprite.SPRITE_HEIGHT)));
    level.AddWall(new Wall(1000, 200));
}
```

Then, in the *Show* method, we call *loadLevel* before the game loop, and then write the walls at step #1 of this loop:

```
public override void Show()
{
    short oldX, oldY;
    loadLevel();
    audio.PlayMusic(0, -1);
    do
    {
        // 1. Draw everything
        hardware.ClearScreen();
        foreach(Wall wall in level.Walls)
            hardware.DrawSprite(Sprite.SpriteSheet, wall.X, wall.Y,
                wall.SpriteX, wall.SpriteY,
                Sprite.SPRITE_WIDTH, Sprite.SPRITE_HEIGHT);
```

After these steps, we should see something like this, although our main character will not be able to collide with the obstacles yet.



Note that the last obstacles (at coordinates X = 1000, Y = 200) can't be seen yet. We will use it later, when we add scroll to the video game.

3.4. Adding collision with a list of obstacles

In order to allow our main character to collide with any obstacle of the list, we are going to add this new method to our *Sprite* class.

```
public bool CollidesWith(List<Sprite> sprites)
{
    foreach (Sprite sp in sprites)
        if (this.CollidesWith(sp))
            return true;
    return false;
}
```

Note that this method relies on previous *CollideWith(Sprite)* method to determine if the character collides with any obstacle of the list (then returns *true*) or not (then returns *false*).

Finally, replace the step #4 of the game loop with this code:

```
if (character.CollidesWith(level.Walls))
{
    character.X = oldX;
    character.Y = oldY;
}
```

Now the collision detection with the whole set of obstacles for current level is completed.

4. Adding scroll

When game map does not fit in screen size, we can either re-scale the whole game or (preferably) show only a part of the map at every game loop. This is what games like Super Mario Bros, Ghost'n Goblins or Ikari Warriors do, and it is called *scroll*. It can be horizontal, vertical or both (this last type of scroll is very typical in RPGs).

What we have to do when implementing a scroll is theoretically easy, although code might be quite difficult to understand and/or implement, depending on some issues such as the type of scroll, number of elements out of the screen that must be considered, and so on. Anyway, the basic rules of scroll are:

- If character is moving in a given direction and there is still a part of the map beyond the screen limits in that direction, then we move the map
- If there is no more map to show in character's current direction, then we move the main character until it reaches the border of the screen.

4.1. Updating level attributes

Regarding our video game, we are going to add a background floor of 1196px width x 920px height, called *floor.jpg.* You should have downloaded and copied it into your project by now.

Then, we need to add this floor as an attribute of our *Level* class, along with the level width and height:

```
class Level
{
    public List<Sprite> Walls { get; }
    public Image Floor { get; set; }
    public short Width { get; set; }
    public short Height { get; set; }
    ...
```

Then, we update these values in our *GameScreen.loadLevel* method:

```
private void loadLevel()
{
    level.Floor = new Image("imgs/floor.jpg", 1196, 920);
    level.Width = 1196;
    level.Height = 920;
    ...
}
```

4.2. Adding map coordinates

We need to store some kind of horizontal and vertical offset of the scene, so that we can update these values as the main character moves. To do this, we are going to add two more attributes to the level:

```
class Level
{
    public List<Sprite> Walls { get; }
    public Image Floor { get; set; }
    public short Width { get; set; }
    public short Height { get; set; }
    public short XMap { get; set; }
    public short YMap { get; set; }

    public Level()
    {
        Walls = new List<Sprite>();
        XMap = YMap = 0;
    }
    ...
```

Now, let's go to our *GameScreen* class. Add two more variables inside the *Show* method to store old map offset:

```
public override void Show()
{
    short oldX, oldY, oldXMap, oldYMap;
```

Then, save current map offset before moving the character:

```
// 2. Move character from keyboard input
oldX = character.X;
oldY = character.Y;
oldXMap = level.XMap;
oldYMap = level.YMap;
moveCharacter();
```

and restore it whenever a collision happens (step #4 of the game loop):

```
if (character.CollidesWith(level.Walls))
{
    character.X = oldX;
    character.Y = oldY;
    level.XMap = oldXMap;
    level.YMap = oldYMap;
}
```

Next, we add these lines to our *moveCharacter* method:

```
private void moveCharacter()
{
    ...
    if (up)
    {
        if (character.Y == GameController.SCREEN_HEIGHT / 2 && level.YMap > 0)
            level.YMap--;
        else if (character.Y > 0)
            character.Y--;
    }
    if (down)
    {
        if (character.Y == GameController.SCREEN_HEIGHT / 2 &&
                level.YMap < level.Height - GameController.SCREEN_HEIGHT)
            level.YMap++;
        else if (character.Y < GameController.SCREEN_HEIGHT -
                 Sprite.SPRITE_HEIGHT)
            character.Y++;
    }
    if (left)
    {
        if (character.X == GameController.SCREEN_WIDTH / 2 && level.XMap > 0)
            level.XMap--;
        else if (character.X > 0)
            character.X--;
    }
    if (right)
    {
        if (character.X == GameController.SCREEN_WIDTH / 2 &&
            level.XMap < level.Width - GameController.SCREEN_WIDTH)
            level.XMap++;
        else if (character.X < GameController.SCREEN_WIDTH -
                 Sprite.SPRITE_WIDTH)
            character.X++;
    }
    ...
```

So, if player is set in the middle of the X or Y axis, then we just move the map horizontally and/or vertically. Otherwise, or if there is no more map to see, we just move the character until the map limits.

We also need to draw the corresponding part of the floor, in step #1 of the game loop:

```
// 1. Draw everything
hardware.ClearScreen();
hardware.DrawSprite(level.Floor, 0, 0, level.XMap, level.YMap,
    GameController.SCREEN_WIDTH, GameController.SCREEN_HEIGHT);
...
```

If you run the game now, you can see how the map scrolls horizontally and vertically. However, obstacles move with the map. Let's fix this.

4.3. Updating obstacles coordinates and collisions

In order to make the obstacles disappear (or appear) as we move along the scene, we need to change the way we draw them in step #1 of the game loop:

```
// 1. Draw everything
hardware.ClearScreen();
hardware.DrawSprite(level.Floor, 0, 0, level.XMap, level.Ymap, ...);
foreach (Wall wall in level.Walls)
    hardware.DrawSprite(Sprite.SpriteSheet, (short)(wall.X - level.XMap),
        (short)(wall.Y - level.YMap), wall.SpriteX, wall.SpriteY,
        Sprite.SPRITE_WIDTH, Sprite.SPRITE_HEIGHT);
```

Note that we are using current map offset (*XMap* and *YMap* attributes) to substract this value from obstacle coordinates. If you run the game now, obstacles will disappear.

4.3.1. Adding offset to collision detection

There is one more bug pending: if we try to collide with an obstacle that is initially out of the scene, we will not be able to do it (try with the last obstacle of the list). The problem is that this obstacle's coordinates are (1000, 200), and main character coordinates are always within the screen limits (800 x 500), so he will never reach that point, because we are moving the map instead of the main character sometimes.

So we need to provide current map offset to collision detection methods, to add this offset to character position:

```
public bool CollidesWith(Sprite sp, short xOffset, short yOffset)
{
    return (X + xOffset + Sprite.SPRITE_WIDTH > sp.X &&
            X + xOffset < sp.X + Sprite.SPRITE_WIDTH &&
            Y + yOffset + Sprite.SPRITE_HEIGHT > sp.Y &&
            Y + yOffset < sp.Y + Sprite.SPRITE_HEIGHT);
}

public bool CollidesWith(List<Sprite> sprites, short xOffset, short yOffset)
{
    foreach (Sprite sp in sprites)
        if (this.CollidesWith(sp, xOffset, yOffset))
            return true;
    return false;
}
```

And update this method call from step #4 of the game loop:

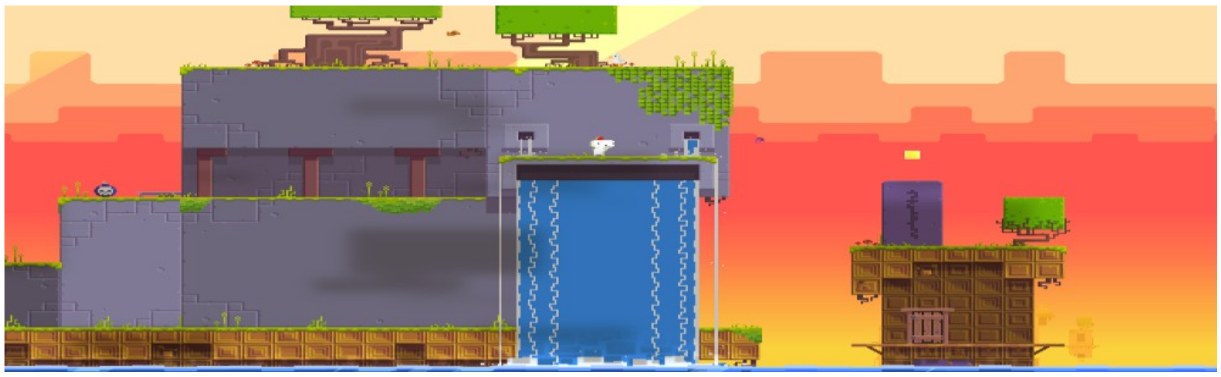
```
// 4. Check collisions and update game state
if (character.CollidesWith(level.Walls, level.XMap, level.YMap))
```

Now the collision detection with scroll mechanism should work perfectly.

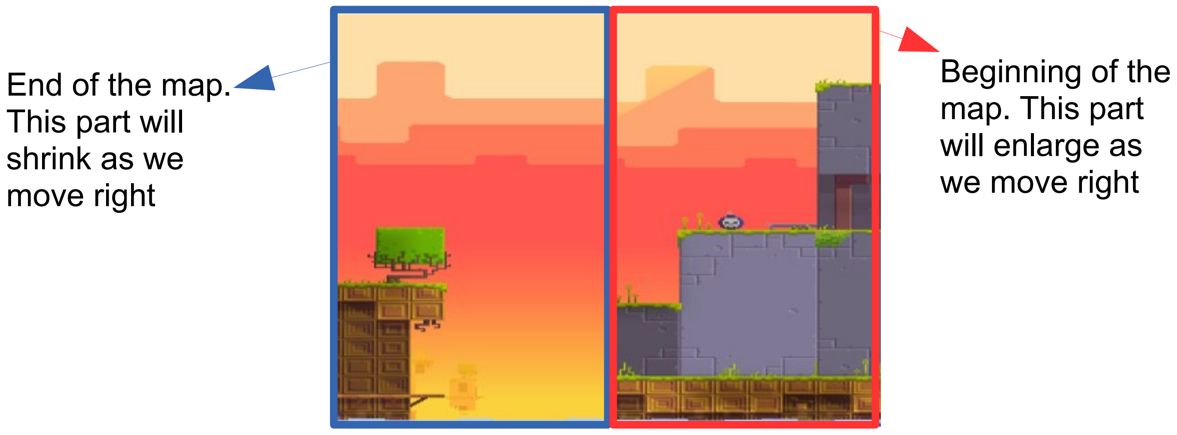
4.4. Other scroll types: continuous scroll

Some games have some kind of "infinite" scenes, that basically are continuous loops over the same map. To do this, when we reach the end of the map, we must draw the part of the map that is visible at this border, and then start drawing the beginning of the map from the opposite border.

For instance, consider this map:



When we reach the right border, we must get a piece of this border, and then link this piece with the beginning of the left border. We would have something like this:


End of the map. This part will shrink as we move right
Beginning of the map. This part will enlarge as we move right

5. To think a little...

Now it's your turn. Try to think an easy way of defining maps with obstacles, so that we can read, for instance, an array of strings or characters, and load the corresponding walls inside the level.

```
string[] map = { "WWWWWWWW",
                 "       W          WWWWWW",
                 "       W          W    W",
                 "       W          W    W",
                 "       W               W",
                 "           WWWWW       W",
                 "                  WWWWWW"};
```