A TopDown Epic Adventure (with a Z)

Gaëtan Blaise-Cazalet - Head of programming section

The Sprite Class

The Sprite class

- In the previous games, we used always the same pattern for our game object classes :

We had some position and transformation variable : x, y, rotation, ox, oy...

We had some functions : Load, Update, Draw

- This is called boilerplate code. Code we always have to recreate.

- We would like to encapsulate this code into a class we will reuse through our next games. This class will be called the Sprite class.

The Sprite class

```
class Sprite
{
    public float X { get; set; }
    public float Y { get; set; }
    public float Ox { get; set; }
    public float Oy { get; set; }
    public float Rotation { get; set; }
    public bool Visible { get; set; }
    public Color Color { get; set; }

    Texture2D image;
    string path;

    public Vector2 Position
    {
        get
        {
            return new Vector2(X, Y);
        }
        set
        {
            X = value.X;
            Y = value.Y;
        }
    }

    public Rectangle Rect
    {
        get
        {
            return new Rectangle((int)(X + Ox), (int)(Y + Oy), image.Width, image.Height);
        }
    }

    public byte Opacity
    {
        get
        {
            return Color.A;
        }
        set
        {
            Color = new Color(Color.R, Color.G, Color.B, value);
        }
    }
...
```

The Sprite class

```
...
    public Sprite(int x, int y, string path)
    {
        X = x;
        Y = y;
        this.path = path;
        Color = Color.White;
    }

    public virtual void Load(ContentManager content)
    {
        image = content.Load<Texture2D>(path);
    }

    public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
    {
        if (Visible)
        {
            Rectangle rect = new Rectangle(0, 0, image.Width, image.Height);
            spriteBatch.Draw(image, rect, null, Color, Rotation, new Vector2(Ox, Oy), SpriteEffects.None, 0);
        }
    }
}
```

The game objects classes

- From our sprite class, we can create different game object classes. They will inherit from Sprite.
- In our current game, we will want a Hero, a Projectile and an Enemy. They will all inherit from the Sprite class
- So create three Hero, Projectile and Enemy classes.

```
class Hero : Sprite
{
    public Hero(int x, int y, string path) : base(x, y, path)
    {

    }
}
```

```
class Projectile : Sprite
{
    public Projectile(int x, int y, string path) : base(x, y, path)
    {

    }
}
```

```
class Enemy : Sprite
{
    public Enemy(int x, int y, string path) : base(x, y, path)
    {

    }
}
```

Hero

Hero move

Make the Hero move up/down/left/right when pressing ZQSD.

We want a smooth move, like in the lander lesson.

Hero move : code

```
Vector2 speed;
const float ACCELERATION = 500;
const float DECELLERATION = 0.95f;
const float MAX_SPEED = 400;

public Hero(int x, int y, string path) : base(x, y, path)
{
    Visible = true;
}

public void Update(GameTime gameTime)
{
    float dt = (float)gameTime.ElapsedGameTime.TotalSeconds;
    KeyboardState ks = Keyboard.GetState();
    if (ks.IsKeyDown(Keys.Down))
    {
        speed.Y += ACCELERATION * dt;
    }
    if (ks.IsKeyDown(Keys.Up))
    {
        speed.Y -= ACCELERATION * dt;
    }
    if (ks.IsKeyDown(Keys.Right))
    {
        speed.X += ACCELERATION * dt;
    }
    if (ks.IsKeyDown(Keys.Left))
    {
        speed.X -= ACCELERATION * dt;
    }

    speed *= DECELLERATION;


    if (speed.X > 0) speed.X = Math.Min(MAX_SPEED, speed.X);
    if (speed.X < 0) speed.X = Math.Max(-MAX_SPEED, speed.X);
    if (speed.Y > 0) speed.Y = Math.Min(MAX_SPEED, speed.Y);
    if (speed.Y < 0) speed.Y = Math.Max(-MAX_SPEED, speed.Y);

    Position += speed * dt;
}
```

Hero rotation

Rotate the hero the right direction. You have two choices :

- Use the Rotation member of the Sprite class
- Create 4 textures in the Hero and display one in function of the direction

Projectiles

Projectile pop

Create a projectile class

Let's make a projectile appear on the right of the player

```
public Projectile(Hero hero) : base("projectile_horizontal")
{
        X = hero.X + 64;
        Y = hero.Y;
}
```

We need to change the Sprite class and add a constructor in it :

```
public Sprite(string path)
{
    X = 0;
    Y = 0;
    this.path = path;
    Color = Color.White;
}
```

And then add the Projectile in the Game1 class. The projectile will be created when we press Space (in the Update function) :

```
    Projectile projectile;
...
    protected override void Update(GameTime gameTime)
    {
        ...
        KeyboardState ks = Keyboard.GetState();
        if(ks.IsKeyDown(Keys.Space))
        {
            projectile = new Projectile(link);
            projectile.Load(Content);
            projectile.Visible = true;
            cooldownCounter = 0;
        }
        ...
    }
...
     protected override void Draw(GameTime gameTime)
    {
        ...
            projectile.Draw(gameTime, spriteBatch);
        ...
    }
```

Projectile List

The problem is we have only one projectile. We can have multiple projectile using a List<Projectile> instead of a single projectile variable.

```
List<Projectile> projectiles = new List<Projectile>();
```

```
protected override void Update(GameTime gameTime)
{
    ...
    KeyboardState ks = Keyboard.GetState();
    if(ks.IsKeyDown(Keys.Space))
    {
        Projectile projectile = new Projectile(link);
        projectile.Load(Content);
        projectile.Visible = true;
        projectiles.Add(projectile);
    }
    ...
}
```

```
protected override void Draw(GameTime gameTime)
{
    ...
    foreach (Projectile p in projectiles)
    {
        p.Draw(gameTime, spriteBatch);
    }
    ...
}
```

Projectile Cooldown

Our projectile generation is now continue. We have to add a cooldown to avoid that.

```
float cooldownCounter = 0;
const float COOLDOWN = 0.1f;
```

```
protected override void Update(GameTime gameTime)
{
    ...
    float dt = (float)gameTime.ElapsedGameTime.TotalSeconds;
    cooldownCounter += dt;

    ...
    if(ks.IsKeyDown(Keys.Space) && cooldownCounter > COOLDOWN)
    {
        ...
        cooldownCounter = 0;
    }
    ...
}
```

Projectile Direction : Hero

Now we want our projectile to go to the right when the player is orientated to the right, and to the left when the player is orientated to the left.

First we have to add a Direction property in the Hero class and to set the direction when we move.

```
public int Direction { get; set; }
```

```
public void Update(GameTime gameTime)
{
    ...
    if (ks.IsKeyDown(Keys.Down))
    {
        Direction = 2;
        speed.Y += ACCELERATION * dt;
    }
    if (ks.IsKeyDown(Keys.Up))
    {
        Direction = 8;
        speed.Y -= ACCELERATION * dt;
    }
    if (ks.IsKeyDown(Keys.Right))
    {
        Direction = 6;
        speed.X += ACCELERATION * dt;
    }
    if (ks.IsKeyDown(Keys.Left))
    {
        Direction = 4;
        speed.X -= ACCELERATION * dt;
    }
    ...
}
```

Projectile Direction

Then, in function of the hero's direction, we set the projectile when it is constructed.

We also set the move in function of the direction in the Update method.

```
int direction;
```

```
public Projectile(Hero hero) : base("projectile_horizontal")
{
    direction = hero.Direction;
    if (direction == 6)
    {
        X = hero.X + 64;
        Y = hero.Y;
    }
    else if (direction == 4)
    {
        X = hero.X;
        Y = hero.Y;
    }

}
```

```
public void Update(GameTime gameTime)
{
    if(direction == 6)
    {
        X = X + 5;
    }
    if(direction == 4)
    {
        X = X - 5;
    }
}
```

Do the same for the up and the down direction.

Tileset

What is a tileset ?

A tileset is a set of tiles (!). A tile is a small image part that is used to draw 2d environment in an optimised way. Each tile is drawn at multiple places to optimize asset creation and computer memory. Here is a 3-tile tileset :

We can count tiles from the first (0) to the third (2). If there were multiple lines, we would count colums, then lines. Let's call this number tileId.

Each tile will be drawn on the game screen, at specific coordinates, inside a table. The numbers inside the table are tileIds.

	0	1	2
0	1	1	1
1	1	2	2
2	1	2	2

In this table, we will draw walls on the top and left edge, and floor inside.

We will create a Tileset class, and a Tilemap class, that will hold the Tileset and the table data, plus the drawing logic.

The tileset class

The Tileset class will have the following members :

- Number of colums,
- Number of rows,
- The size of a tile,
- The image texture and path

```
class Tileset
{
    public int Rows { get; set; }
    public int Cols { get; set; }
    public int Tilesize { get; set; }
    public Texture2D Image
    {
        get
        {
            return image;
        }
    }

    string path;
    Texture2D image;

    public Tileset(int rows, int cols, int tilesize, string path)
    {
        Rows = rows;
        Cols = cols;
        Tilesize = tilesize;
        this.path = path;
    }

    public void Load(ContentManager content)
    {
        image = content.Load<Texture2D>(path);
    }
}
```

The tilemap class

The tilemap will hold the tileset and the bidimensional table that will contain the tileids. It will be constructed with those two pieces of data.

The Tilemap drawing is explained on the next page.

```
class Tilemap
{
    public Tileset Tileset
    {
        get
        {
            return tileset;
        }
    }

    public int[][] Data
    {
        get
        {
            return data;
        }
    }

    Tileset tileset;
    int[][] data;               // <- towdimensional table

    public Tilemap(Tileset tileset, int[][] data)
    {
        this.tileset = tileset;
        this.data = data;
    }

    public void Load(ContentManager content)
    {
        tileset.Load(content);
    }
    ...
```

The tilemap class (draw)

```
...
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    for (int r = 0; r < data.Length; r++)
    {
        for (int c = 0; c < data[r].Length ; c++)
        {
            Rectangle drawRect = new Rectangle(
                c * tileset.Tilesize, r * tileset.Tilesize,
                tileset.Tilesize, tileset.Tilesize
            );

            int tileId = data[r][c];
            int tilesetCoordX = tileId % tileset.Cols;
            int tilesetCoordY = tileId / tileset.Cols;
            Rectangle tileRect = new Rectangle(
                tilesetCoordX * tileset.Tilesize, tilesetCoordY * tileset.Tilesize,
                tileset.Tilesize, tileset.Tilesize
            );

            spriteBatch.Draw(tileset.Image, drawRect, tileRect,
                Color.White, 0, new Vector2(0, 0), SpriteEffects.None, 0);
        }
    }
}


}
```

To draw the tilemap, we will go through the bidimensional array. For each row and for each colum, we create two rectangles :

- The drawRect Rectangle will be used to draw on the game screen. This rects coordinates are the colums / row number multiplied by the tilesize, and its size is the tile size. This is the classic rectangle we have used until now to drow something on a screen.

- The tileRect Rectangle will be the part of the tileset texture we will draw on the tilemap. To get it, we must get its coordinates from the tileID. To get the Y coord, we divide (euclidian division) tileId by the number of tileset's colums. To get the X coordinate, we use the modulo (%) operator to get the rest of this euclidian division.

One we get those two rectangles, we can use them in the Draw function to draw the tile at the right place.

Tilemap in Game class

Then, we just have to integrate our classes in the Game class.

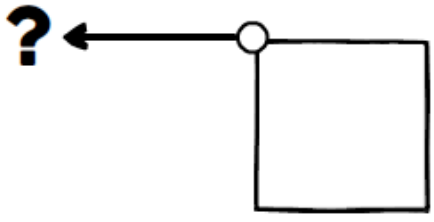
```
Tileset tileset;
Tilemap tilemap;
int[][] tilemapData = new int[][]
{
    new int[] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1 },
    new int[] { 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
};

protected override void Initialize()
{
    ...
    tileset = new Tileset(1, 3, 40, "tileset");
    tilemap = new Tilemap(tileset, tilemapData);
    ...
    base.Initialize();
}

protected override void LoadContent()
{
    ...
    tilemap.Load(Content);
    ...
}

protected override void Draw(GameTime gameTime)
{
    ...
    tilemap.Draw(gameTime, spriteBatch);
    ...
}
```

Collisions with the tilemap

The raycast concept

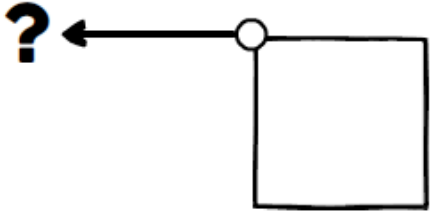
Let's say we have a wall tile on our left and we don't want the hero to go through it. We will use the following algorithm :

- Select a point on the left side of our hero, called collisionOrigin
- Check what will be at it's next position, by adding the X move to this point
- If there is a wall tile (tileId == 1) at this position, we will forbid the move.



This method, casting a ray from a point of our colliding element, is the simplest form of an algorithm called "raycasting".

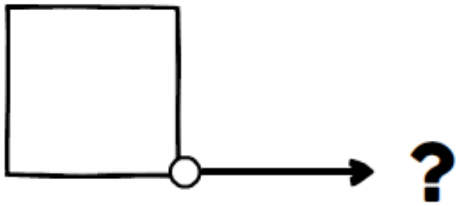
Simple code




```
public void Update(GameTime gameTime, Tilemap tilemap)
{
    ...
    float collisionOx = X;
    float collisionOy = Y;
    collisionOx += image.Width;
    collisionOy += image.Height;

    int nextTileCol = (int)Math.Floor((collisionOx + Speed.X * dt) / (float)tilemap.Tileset.Tilesize);
    int nextTileRow = (int)Math.Floor((collisionOy + Speed.Y * dt) / (float)tilemap.Tileset.Tilesize);

    int tile = tilemap.Data[nextTileRow][nextTileCol];
    if (tile == 1)
    {
        speed.X = 0;
        speed.Y = 0;
    }

    // Move
    Position += Speed * dt;
}
```

Fix right and down move

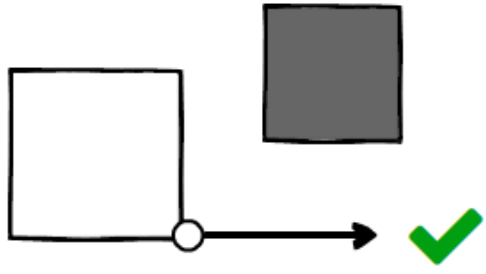
The previous code works for left and up move, but not for right and down move, because the collisionOrigin is not the right one. We must use the bottom-right point of our hero to set the collisionOrigin.




```
    public void Update(GameTime gameTime, Tilemap tilemap)
    {
        ...
        float collisionOx = X;
        float collisionOy = Y;

// new code
        if (Direction == 6 || Direction == 2)
        {
            collisionOx += image.Width;
            collisionOy += image.Height;
        }
// end new code

        collisionOx += image.Width;
        collisionOy += image.Height;

        int nextTileCol = (int)Math.Floor((collisionOx + Speed.X * dt) / (float)tilemap.Tileset.Tilesize);
        int nextTileRow = (int)Math.Floor((collisionOy + Speed.Y * dt) / (float)tilemap.Tileset.Tilesize);

        int tile = tilemap.Data[nextTileRow][nextTileCol];
        if (tile == 1)
        {
            speed.X = 0;
            speed.Y = 0;
        }

        // Move
        Position += Speed * dt;
    }
```

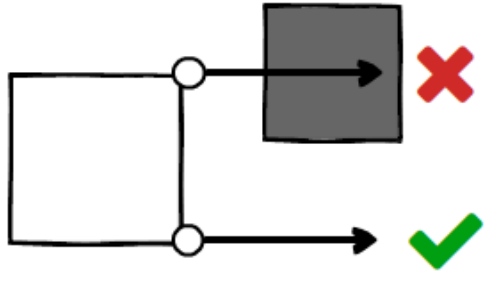
A second raycast

The previous code is in trouble if there is an blocking tile that should block the hero path, but is not hit by the raycast. For instance :



The collision algorithm won't detect the blocking tile, so the hero will go through.

To avoid that, we will cast two rays, one from the collisionOrigin, and one on the opposing edge of the collision origin. The collision will be detected if one of the two rays meet an obstacle.



The number of rays depends the size of the collidable object and the size of the tiles. If tilesize is 40 and hero is 64*64, we only need two rays.

A second raycast - Code

```
public void Update(GameTime gameTime, Tilemap tilemap)
{
    ...
    bool firstCollision = false;
    bool secondCollision = false;

    float collisionOx = X;
    float collisionOy = Y;
    // Premiere collision
    if (Direction == 6 || Direction == 2)
    {
        collisionOx += image.Width;
        collisionOy += image.Height;
    }
    int nextTileCol = (int)Math.Floor((collisionOx + Speed.X * dt) / (float)tilemap.Tileset.Tilesize);
    int nextTileRow = (int)Math.Floor((collisionOy + Speed.Y * dt) / (float)tilemap.Tileset.Tilesize);

    int tile = tilemap.Data[nextTileRow][nextTileCol];
    if (tile == 1)
    {
        firstCollision = true;
    }

    // Deuxieme collision
    if (Direction == 2)
    {
        collisionOx -= image.Width;
    }
    else if (Direction == 4)
    {
        collisionOy += image.Height;
    }
    else if (Direction == 6)
    {
        collisionOy -= image.Height;
    }
    else if (Direction == 8)
    {
        collisionOx += image.Width;
    }
    nextTileCol = (int)Math.Floor((collisionOx + Speed.X * dt) / (float)tilemap.Tileset.Tilesize);
    nextTileRow = (int)Math.Floor((collisionOy + Speed.Y * dt) / (float)tilemap.Tileset.Tilesize);

    tile = tilemap.Data[nextTileRow][nextTileCol];
    if (tile == 1)
    {
        secondCollision = true;
    }

    // Resolution
    if (firstCollision || secondCollision)      // <-- "||" means "or" in a condition
    {
        speed.X = 0;
        speed.Y = 0;
    }
    ...
}
```

Various scenes

A SceneMap class

Until now, we put all our game logic inside th Game class. Now imagine we want multiple levels in our game. The idea would be to replace the tilemap data, and the initialize function to set different tilemap data, tileset file and enemies. We would need one tilemap data, one tileset path and one set of enemies by level.

A simpler way to do this is to create a class that will hold all the level logic, in which we will pass the tilemap data, the enemies, and any other data needed.

All the game logic we have put in the Game class will now go in this SceneMap class. Let us create this class.

```
class SceneMap
{
    Hero link;
    List<Projectile> projectiles = new List<Projectile>();
    float cooldownCounter = 0;
    const float COOLDOWN = 0.1f;

    Tileset tileset;
    Tilemap tilemap;

    List<Enemy> enemies;

    ContentManager content;

    public SceneMap(int[][] data, List<Enemy> enemies, string tilesetPath)
    {

    }

    public void Load(ContentManager content)
    {

    }

    public void Update(GameTime gameTime)
    {

    }

    public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
    {

    }
}
```

We have put all Game1 fields into the scene map. We add the content manager to ease the loading for future steps. Now, we will create the content of our functions.

A SceneMap class

The SceneMap constructor content will be analog to the Game1 Initialise function.

```
public SceneMap(int[][] tilemapData, List<Enemy> enemies, string tilesetPath)
{
    link = new Hero(100, 100, "hero");
    tileset = new Tileset(1, 3, 40, tilesetPath);
    tilemap = new Tilemap(tileset, tilemapData);

    this.enemies = enemies;
}
```

We create the Tileset and the Tilemap with the constructors parameters. We hold a reference to the enemies list in the class.

The Load function will be similar to former Game1's Load function :

```
public void Load(ContentManager content)
{
    link.Load(content);
    tilemap.Load(content);
    foreach (Enemy e in enemies)
    {
        e.Load(content);
    }
}
```

Update is exactly the same function, except for the content reference when we load projectiles. Also, we don't need the first Game1's update line and the base.Update at the end of the function.

```
protected override void Update(GameTime gameTime)
{
    ...
    if(ks.IsKeyDown(Keys.Space) && cooldownCounter > COOLDOWN)
    {
        Projectile energyWave = new Projectile(link);
        energyWave.Load(content);
        ...
    }
    ...
}
```

Draw is the same function, except we don't need the first and last two lines of Game1's Draw.

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    tilemap.Draw(gameTime, spriteBatch);
    foreach (Enemy e in enemies)
    {
        e.Draw(gameTime, spriteBatch);
    }
    link.Draw(gameTime, spriteBatch);
    foreach (Projectile p in projectiles)
    {
        p.Draw(gameTime, spriteBatch);
    }
}
```

SceneMap in Game1

Now that our SceneMap is ready, we have to use it in the Game1 class.

```
class Game1 : Game
{
    ...
    SceneMap currentScene;

    int[][] tilemapData01 = new int[][]
    {
        ...
    };

    protected override void Initialize()
    {
        Enemy enemy0 = new Enemy(400, 200, "enemy");
        Enemy enemy1 = new Enemy(300, 300, "enemy");
        List<Enemy> enemies = new List<Enemy>();
        enemies.Add(enemy0);
        enemies.Add(enemy1);

        currentScene = new SceneMap(tilemapData01, enemies, "tileset");

        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);

        currentScene.Load(Content);
    }

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
            Exit();

        currentScene.Update(gameTime);

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.Black);
        spriteBatch.Begin();

        currentScene.Draw(gameTime, spriteBatch);

        spriteBatch.End();
        base.Draw(gameTime);
    }
}
```

The game will be strictly the same, except it runs in a SceneMap class now.

A second level

Imagine we want a second level. We will use the same tileset, but create an other tilemap datatable and an other list of enemies. We will then create two SceneMap, one with the first data set, one with the second. The currentScene variable will be set to be the first level at the beginning of the game.

```
class Game1 : Game
{
    ...
    SceneMap level01;
    SceneMap level02;
    SceneMap currentScene;

    int[][] tilemapData01 = new int[][]
    {
        ...
    };
    int[][] tilemapData02 = new int[][]
    {
        ...
    };

    protected override void Initialize()
    {
        Enemy enemy0 = new Enemy(400, 200, "enemy");
        Enemy enemy1 = new Enemy(300, 300, "enemy");
        List<Enemy> enemies01 = new List<Enemy>();
        enemies01.Add(enemy0);
        enemies01.Add(enemy1);

        Enemy enemy2 = new Enemy(200, 300, "enemy");
        Enemy enemy3 = new Enemy(50, 400, "enemy");
        Enemy enemy4 = new Enemy(550, 100, "enemy");
        List<Enemy> enemies02 = new List<Enemy>();
        enemies02.Add(enemy2);
        enemies02.Add(enemy3);
        enemies02.Add(enemy4);

        level01 = new SceneMap(tilemapData01, enemies01, "tileset");
        level02 = new SceneMap(tilemapData02, enemies02, "tileset");
        currentScene = level01;

        base.Initialize();
    }
    ...
}
```

The two levels are set in the Initialize function. Only the level in the currentScene variable (namely level01) will be updated and drawn.

Change scene - Setup

Now is a big abstraction leap.

Until now, we have passed values as arguments. We passed simple values, as int, string, float. We also passed objects, like Sprite or GameTime.

Now, we will pass a function as an argument. To change the scene, we will pass to our SceneMap a ChangeScene function, that will become a member of SceneMap, and will be used when every enemy of the SceneMap is dead, to change level.

Here is how. First, we need to declare a delegate ChangeSceneFunc(), that will be the type of our function argument. We also declare the variable that will hold this function as a member of our SceneMap class.

```
class SceneMap
{
    ...
    public delegate void ChangeSceneFunc();
    ChangeSceneFunc changeScene;
    ...
}
```

The keyword delegate is used to tell we create a function type ChangeSceneFunc.

Now, we change the SceneMap constructor to pass a ChangeSceneFunc argument, and to store it in the changeScene member variable.

```
    public SceneMap(int[][] tilemapData, List<Enemy> enemies, string tilesetPath, ChangeSceneFunc changeScene)
    {
        ...
        this.changeScene = changeScene;
    }
```

Change scene - Usage

We want to change the scene when every enemy is dead. To do that, we call the changeScene member as a function in the SceneMap's Update function :

```
class SceneMap
{
    ...
    public void Update(GameTime gameTime)
    {
        ...
        if(enemies.Count == 0)
        {
            changeScene();
        }
    }
    ...
}
```

Now, we have to implement the scene change mecanism. It will be located in the Game1 class, that hold the currentScene. First, we need to create a ChangeScene function.

```
class Game1
{
    ...
    void ChangeScene(SceneMap scene)
    {
        if (scene != null)
        {
            scene.Load(Content);
            currentScene = scene;
        }
    }
    ...
```

Note that we Laod the scene before changing it. If we do not, our new scene might be not loaded.

Then, we need to give to the SceneMap the function it will use when changing scene. We do that in the Initialize function.

```
    protected override void Initialize()
    {
        ...
        level01 = new SceneMap(tilemapData01, enemies01, "tileset", () => ChangeScene(level02));
        level02 = new SceneMap(tilemapData02, enemies02, "tileset", () => ChangeScene(null));
        currentScene = level01;

        base.Initialize();
    }
```

We create the first SceneMap with a strange argument :

```
() => ChangeScene(level02)
```

This argument means we pass a function with no parameter, that will execute the ChangeScene function with the leve02 argument. This kind of notation is called an "anonymous function".

We do the same in the second SceneMap, except we pass no level (null). This way, level02 cannot change scene. We don't want level02 to change scene because we have no level03.