Introduction to Game Programming

Session 7 – Enemies and shots

Nacho Cabanes
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

In this last session of the video game, we are going to make our main character fire his weapon and destroy the enemies that will be present at each level of the game. To do this, we need to add a specific weapon to each character type, and the corresponding sprites to move the shots in current main character's direction.

Besides, we will add some kind of enemy generators to each level, from which enemies will arise periodically, and will go to character's position.

1.1. About the solution for this session

If you download the solution to start with this session, you will see some new classes that were not present at previous session. These classes are:

- *Weapon* class, that represents every weapon for every character type. From this class, we define four concrete subclasses:
  - *Axe* weapon for the warrior
  - *Sword* weapon for the valkyrie
  - *Magic* weapon for the sorcerer
  - *Arrow* weapon for the dwarf

  Every class has its own sprite coordinates pre-loaded in the constructor, so you don't have to care about this issue.

Besides, there is a constant called STEP_LENGTH in *MainCharacter* class to determine how many pixels will he move in every movement, so that we can make it go faster or slower just by changing this value.

There are some other changes in *moveCharacter* method (to deal with this new constant and update the X and Y coordinates of the character and the map properly), and in *Level* constructor, to fix a bug when moving the main character to a start point which is out of the screen bounds. We've added an X character in the map text file to determine the initial map coordinates when starting the level, to make sure that starting point will be within the screen limits.

Besides these changes, you need to make visible the method *MovableSprite. UpdateSpriteCoordinates,* so set it as *public*.

2. Firing weapons

Let's make our main character fire his weapon whenever we press the space bar.

2.1. Changes in character hierarchy

To do this, we are going to add a list of weapons as part of our character attributes (*MainCharacter* class).

```
class MainCharacter: MovableSprite
{
    public const byte STEP_LENGTH = 2;
    public ushort Energy { get; set; }
    public ushort Points { get; set; }
    public List<Weapon> Weapons { get; }

    public MainCharacter()
    {
        Points = 0;
        Weapons = new List<Weapon>();
    }
```

Now, we need to add some methods to add weapons to the list, depending on the concrete character type. Let's start by defining our *MainCharacter* class as abstract, and add two new methods: an abstract method called *AddWeapon*, with no parameters, that will be overridden in the subclasses, and another method called *AddWeapon*, with a weapon as a parameter, that will be used by the subclasses, as we will see later. Besides, we add another method called *RemoveWeapon* to remove a weapon from the list given its index.

```
abstract class MainCharacter: MovableSprite
{
    ...
        public abstract void AddWeapon();

        public void AddWeapon(Weapon w)
        {
            w.X = this.X;
            w.Y = this.Y;
            w.CurrentDirection = this.CurrentDirection;
            w.UpdateSpriteCoordinates();
            Weapons.Add(w);
        }

        public void RemoveWeapon(int index)
```

```
        {
            Weapons.RemoveAt(index);
        }
    }
}
```

Then, we override the method in the different subclasses:

```
class Warrior: MainCharacter
{
    ...
    public override void AddWeapon()
    {
        Axe newAxe = new Axe();
        base.AddWeapon(newAxe);
    }
}
```

```
class Valkyrie: MainCharacter
{
    ...
    public override void AddWeapon()
    {
        Sword newSword = new Sword();
        base.AddWeapon(newSword);
    }
}
```

```
class Sorcerer: MainCharacter
{
    ...
    public override void AddWeapon()
    {
        Magic newMagic = new Magic();
        base.AddWeapon(newMagic);
    }
}
```

```
class Dwarf: MainCharacter
{
    ...
    public override void AddWeapon()
    {
        Arrow newArrow = new Arrow();
```

```
        base.AddWeapon(newArrow);
    }
}
```

2.2. Generating shots and moving them

We are going to create a new shot (weapon) every time we press the space bar. So we need to add this code in step #2 of the game loop (getting user input):

```
// 2. Move character from keyboard input
oldX = character.X;
oldY = character.Y;
oldXMap = level.XMap;
oldYMap = level.YMap;
moveCharacter();
if (hardware.IsKeyPressed(Hardware.KEY_SPACE))
{
    character.AddWeapon();
}
```

Then, we call this new method in step #3 of the game loop

```
// 3. Move enemies and objects
moveWeapons();
```

The code of this method is something like this (we add it in *GameScreen* class):

```
private void moveWeapons()
{
    foreach(Weapon w in character.Weapons)
    {
        switch (w.CurrentDirection)
        {
            case MovableSprite.SpriteMovement.DOWN:
                w.Y+=Weapon.STEP_LENGTH;
                break;
            case MovableSprite.SpriteMovement.LEFT:
                w.X -= Weapon.STEP_LENGTH;
                break;
            case MovableSprite.SpriteMovement.LEFT_DOWN:
                w.X -= Weapon.STEP_LENGTH;
                w.Y += Weapon.STEP_LENGTH;
                break;
            case MovableSprite.SpriteMovement.LEFT_UP:
                w.X -= Weapon.STEP_LENGTH;
                w.Y -= Weapon.STEP_LENGTH;
                break;
```

```
            case MovableSprite.SpriteMovement.RIGHT:
                w.X += Weapon.STEP_LENGTH;
                break;
            case MovableSprite.SpriteMovement.RIGHT_DOWN:
                w.X += Weapon.STEP_LENGTH;
                w.Y += Weapon.STEP_LENGTH;
                break;
            case MovableSprite.SpriteMovement.RIGHT_UP:
                w.X += Weapon.STEP_LENGTH;
                w.Y -= Weapon.STEP_LENGTH;
                break;
            case MovableSprite.SpriteMovement.UP:
                w.Y -= Weapon.STEP_LENGTH;
                break;
        }
    }
}
```

We also need to add the STEP_LENGTH constant in *Weapon* class:

```
class Weapon : MovableSprite
{
    public const byte STEP_LENGTH = 4;
}
```

2.3. Checking collisions with walls

Finally, let's add this method to our *Level* class to determine if any weapon collides with a wall. If so, the weapon will be automatically removed from the list.

```
public void CollideWeaponsWithWalls(MainCharacter character)
{
    int i = 0;
    while (i < character.Weapons.Count)
    {
        if (character.Weapons[i].CollidesWith(Walls))
            character.RemoveWeapon(i);
        else
            i++;
    }
}
```

We can call this method in step #4 of the game loop (just at the beginning, for instance).

```
// 4. Check collisions and update game state
level.CollideWeaponsWithWalls(character);
...
```

2.4. Avoiding too many shots per second

If you want to avoid the fact that, when you hold the space bar, an unlimited amount of fire shots arise from your main character, then you need to separate them in time. To do this, we can just add this new variable inside *GameScreen.Show* method:

```
public override void Show()
{
    short oldX, oldY, oldXMap, oldYMap;
    DateTime timeStampFromLastShot = DateTime.Now;
    ...
```

Then, whenever we press the space bar, we check if the number of milliseconds from the last time that we fired is greater than a given value, and if so, we fire again and update the timestamp to now:

```
// 2. Move character from keyboard input
...
if (hardware.IsKeyPressed(Hardware.KEY_SPACE))
{
    if ((DateTime.Now - timeStampFromLastShot).TotalMilliseconds > SHOT_INTERVAL)
    {
        timeStampFromLastShot = DateTime.Now;
        character.AddWeapon();
    }
}
```

The constant SHOT_INTERVAL can be defined inside *GameScreen* class:

```
class GameScreen: Screen
{
    const ushort SHOT_INTERVAL = 200;
    ...
```

3. Adding enemies

It's time to add enemies to our video game. In the original video game, enemies were generated from some kind of tomb, that we will call *enemy generator*. So let's define a generic, abstract class to deal with these generators.

3.1. Defining enemy generators

The general (parent) class for enemy generators can be like this one:

```
abstract class EnemiesGenerator : StaticSprite
{
    public byte TotalEnemies { get; set; }

    public EnemiesGenerator()
    {
        SpriteX = 226;
        SpriteY = 340;
    }

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

    public abstract Enemy NewEnemy();
}
```

From each one of these generators only one type of enemy can be created. So let's start by defining a generator for ghosts. This new class inherits from previous one, and overrides the abstract method, this way:

```
class GhostGenerator : EnemiesGenerator
{
    public GhostGenerator() : base()
    {
        TotalEnemies = 20;
    }

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

```
    }

    public override Enemy NewEnemy()
    {
        if (TotalEnemies > 0)
        {
            TotalEnemies--;
            return new Ghost();
        }
        else
        {
            return null;
        }
    }
}
```

We've added a total amount of 20 enemies to be generated by this generator. After all enemies are created, this generator is useless.

3.2. Defining enemies

Enemies are another type of *MovableSprite* objects, so let's start by defining the general class for all enemies...

```
class Enemy: MovableSprite
{
}
```

and then a specific subclass, such as *Ghost*:

```
class Ghost : Enemy
{
    public Ghost()
    {
        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.UP] = new int[] { 8, 442, 874 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.UP] = new int[] { 14, 14, 14 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.RIGHT_UP] = new int[] { 64, 496, 928 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.RIGHT_UP] = new int[] { 14, 14, 14 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.RIGHT] = new int[] { 118, 550, 982 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.RIGHT] = new int[] { 14, 14, 14 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.RIGHT_DOWN] = new int[] {172,604,1036};
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.RIGHT_DOWN] = new int[] { 14, 14, 14 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.DOWN] = new int[] { 226, 658, 1090 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.DOWN] = new int[] { 14, 14, 14 };
```

```
        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.LEFT_DOWN] = new int[] {280,712,1144};
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.LEFT_DOWN] = new int[] { 14, 14, 14 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.LEFT] = new int[] { 334, 766, 1198 };
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.LEFT] = new int[] { 14, 14, 14 };

        SpriteXCoordinates[(int)MovableSprite.SpriteMovement.LEFT_UP] = new int[] {388, 820, 1252};
        SpriteYCoordinates[(int)MovableSprite.SpriteMovement.LEFT_UP] = new int[] { 14, 14, 14 };

        UpdateSpriteCoordinates();
    }
}
```

As you can see, we've included all the sprites to animate the ghosts in every direction.

3.2.1. Moving enemies

Enemies will always move towards the main character. It doesn't matter if there is a wall between them, they will remain next to the wall, on the opposite side, waiting for the character to move.

Let's implement a simple approach to this algorithm in *Enemy* class. In order to do this, we need to add this code:

```
class Enemy: MovableSprite
{
    public const byte STEP_LENGTH = 2;

    public void Move(MainCharacter character)
    {
        short xDiff = (short)(character.X - this.X);
        short YDiff = (short)(character.Y - this.Y);

        if (xDiff < 0 && YDiff < 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.LEFT_UP;
            this.X -= STEP_LENGTH;
            this.Y -= STEP_LENGTH;
        }
        else if (xDiff < 0 && YDiff == 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.LEFT;
            this.X -= STEP_LENGTH;
        }
        else if (xDiff < 0 && YDiff > 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.LEFT_DOWN;
```

```
            this.X -= STEP_LENGTH;
            this.Y += STEP_LENGTH;
        }
        else if (xDiff > 0 && YDiff < 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.RIGHT_UP;
            this.X += STEP_LENGTH;
            this.Y -= STEP_LENGTH;
        }
        else if (xDiff > 0 && YDiff == 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.RIGHT;
            this.X += STEP_LENGTH;
        }
        else if (xDiff > 0 && YDiff > 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.RIGHT_DOWN;
            this.X += STEP_LENGTH;
            this.Y += STEP_LENGTH;
        }
        else if (xDiff == 0 && YDiff < 0)
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.UP;
            this.Y -= STEP_LENGTH;
        }
        else
        {
            this.CurrentDirection = MovableSprite.SpriteMovement.DOWN;
            this.Y += STEP_LENGTH;
        }

        this.Animate(this.CurrentDirection);
    }
}
```

We just check the difference between character's X and Y coordinates and enemy's X and Y coordinates to determine the direction to move, and then we use the STEP_LENGTH constant from *Enemy* class to move in that direction.

3.3. Adding enemies in map text file

In order to add enemies (ghosts) to our text file, we are going to place a G in every position in which we want to add an enemy generator (ghost generator). For instance:

```
XWWWWWWWWWWWWWWWWWWWWWW
W        W   T    W      PW
W   S    W        W       W
W        W                W
W        W                W
W                 W   W  W
WM                W   W  W
WWWWWWWWW         W   W  W
W   G             W   W  W
W                 W  TW EW
WWWWWWWWWWWWWWWWWWWWWWWW
```

Then, we add two more properties to our *Level* class: one to store all the generators in the level, and another one to manage all the enemies generated by the generators:

```
class Level
{
    public List<Sprite> Walls { get; }
    public List<Treasure> Treasures { get; }
    public List<Food> Meals { get; }
    public List<EnemiesGenerator> Generators { get; }
    public List<Enemy> Enemies { get; }
    ...
```

We initialize these lists at the beginning of the constructor...

```
public Level(string fileName)
{
    Walls = new List<Sprite>();
    Treasures = new List<Treasure>();
    Meals = new List<Food>();
    Enemies = new List<Enemy>();
    Generators = new List<EnemiesGenerator>();
```

and we add the generators to the list as soon as we detect a G in the text file:

```
for (int i = 0; i < lines.Length; i++)
{
    for (int j = 0; j < lines[i].Length; j++)
    {
        ...
        else if (lines[i][j] == 'G')
        {
            Generators.Add(new GhostGenerator((short)(j * Sprite.SPRITE_WIDTH),
                (short)(i * Sprite.SPRITE_HEIGHT)));
        }
    }
```

```
}
```

3.4. Changes in GameScreen class

Finally, we need to add some changes in *GameScreen* class in order to make the enemies appear and go towards our main character.

3.4.1. Timer to generate enemies

First of all, we are going to define another timer to generate a new enemy from every generator every 5 seconds. We can define it next to previous timer, in *Show* method:

```
var timer = new Timer(this.DecreaseEnergy, null, 1000, 1000);
var timerEnemies = new Timer(this.NewEnemies, null, 0, 5000);
```

The corresponding *NewEnemies* function can be defined in this class as well:

```
public void NewEnemies(Object o)
{
    foreach(EnemiesGenerator eg in level.Generators)
    {
        Enemy enemy = eg.NewEnemy();
        if (enemy != null)
        {
            enemy.MoveTo(eg.X, eg.Y);
            level.Enemies.Add(enemy);
        }
    }
}
```

3.4.2. Drawing enemies and generators

Next, we draw every generator and enemy in step #1 of the game loop:

```
// 1. Draw everything
hardware.ClearScreen();
...
foreach (Wall wall in level.Walls)...
foreach (Treasure t in level.Treasures)...
foreach (Food f in level.Meals)...
foreach (Weapon w in character.Weapons)...
foreach (EnemiesGenerator eg in level.Generators)
    hardware.DrawSprite(Sprite.SpriteSheet, (short)(eg.X - level.XMap), (short)
        (eg.Y - level.YMap), eg.SpriteX, eg.SpriteY, Sprite.SPRITE_WIDTH,
        Sprite.SPRITE_HEIGHT);
foreach (Enemy e in level.Enemies)
    hardware.DrawSprite(Sprite.SpriteSheet, (short)(e.X - level.XMap), (short)
        (e.Y - level.YMap), e.SpriteX, e.SpriteY, Sprite.SPRITE_WIDTH,
        Sprite.SPRITE_HEIGHT);
```

```
...
```

3.4.3. Move enemies

We can define a method called *moveEnemies* to move every enemy in the game loop:

```
private void moveEnemies()
{
    for(int i = 0; i < level.Enemies.Count; i++)
    {
        short oldX = level.Enemies[i].X;
        short oldY = level.Enemies[i].Y;

        level.Enemies[i].Move(character);

        if (level.Enemies[i].CollidesWith(level.Walls))
        {
            level.Enemies[i].X = oldX;
            level.Enemies[i].Y = oldY;
        }
    }
}
```

This method explores the enemies list and moves each enemy. If the enemy collides with any wall at the new position, then it turns it back to the old one. We just call this method from step #3 of the game loop:

```
// 3. Move enemies and objects

moveWeapons();
moveEnemies();
```

3.4.4. Destroying enemies

Now, let's check collisions between enemies and weapons. For every collision, both the enemy and the weapon must be destroyed, and some points must be added to our score. This last feature can be determined by a constant in *Enemy* class:

```
class Enemy: MovableSprite
{
    public const byte STEP_LENGTH = 2;
    public const byte DESTROY_SCORE = 50;
```

Then, we add a new method in *Level* class to determine the collision between enemies and weapons, and return the total score of these collisions (there can be more than one collision):

```
public ushort CollideWeaponsWithEnemies(MainCharacter character)
{
    ushort result = 0;
    int i = 0, j;
```

```
    bool weaponDestroyed;
    while (i < character.Weapons.Count)
    {
        j = 0;
        weaponDestroyed = false;
        while (j < Enemies.Count && !weaponDestroyed)
        {
            if (character.Weapons[i].CollidesWith(Enemies[j]))
            {
                character.RemoveWeapon(i);
                Enemies.RemoveAt(j);
                weaponDestroyed = true;
                result += Enemy.DESTROY_SCORE;
            }
            else
                j++;
        }
        if (!weaponDestroyed)
            i++;
    }

    return result;
}
```

This function can be called at the beginning of step #4 of the game loop (*GameScreen* class):

```
// 4. Check collisions and update game state
level.CollideWeaponsWithWalls(character);
ushort points = level.CollideWeaponsWithEnemies(character);
if (points > 0)
{
    character.Points += points;
    updatePoints();
}
...
```

3.4.5. Colliding with main character

If any enemy collides with the main character, then it must substract some energy from him. In order to do this, we add a new constant in *Enemy* class to determine the total damage of every collision:

```
class Enemy: MovableSprite
{
    public const byte STEP_LENGTH = 2;
```

```
    public const byte DESTROY_SCORE = 50;
    public const byte DAMAGE = 1;
    ...
```

If we allow that every enemy can substract DAMAGE points from our main character in every game loop, the energy decrease rate can be too fast. In order to slow down this, let's add another timestamp in our *Show* method:

```
public override void Show()
{
    short oldX, oldY, oldXMap, oldYMap;
    DateTime timeStampFromLastShot = DateTime.Now;
    DateTime timeStampFromLastDamage = DateTime.Now;
```

We need to pass this value as parameter to our *moveEnemies* method, so that damage will only be applied if time from previous damage is greater than a given constant (declared here in *GameScreen* class):

```
private void moveEnemies(ref DateTime timestamp)
{
    for(int i = 0; i < level.Enemies.Count; i++)
    {
        short oldX = level.Enemies[i].X;
        short oldY = level.Enemies[i].Y;

        level.Enemies[i].Move(character);

        if (level.Enemies[i].CollidesWith(level.Walls))
        {
            level.Enemies[i].X = oldX;
            level.Enemies[i].Y = oldY;
        }
        else if (level.Enemies[i].CollidesWith(character))
        {
            level.Enemies[i].X = oldX;
            level.Enemies[i].Y = oldY;
            if ((DateTime.Now - timestamp).TotalMilliseconds > DAMAGE_INTERVAL)
            {
                character.Energy -= Enemy.DAMAGE;
                timestamp = DateTime.Now;
            }
        }
    }
}
```

DAMAGE_INTERVAL can be like this one:

```
class GameScreen: Screen
{
    const ushort SHOT_INTERVAL = 200;
    const ushort DAMAGE_INTERVAL = 400;
```

Then, we update the call of this method from step #3:

```
// 3. Move enemies and objects
moveWeapons();
moveEnemies(ref timeStampFromLastDamage);
```

4. To think a little...

To finish with this session, you are asked to add a new, different type of enemy generator and enemy. You can choose among every enemy type provided to you in the sprite sheet. So, each level can have multiple generators of both enemy types, and all of the enemies will be stored in the same enemy list.