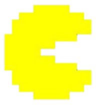
Introduction to Game Programming



Session 6 – Levels, objects and game state

Nacho Cabanes
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

This session we are going to deal with three important aspects of the game:

- First of all, we are going to define a file structure to create a level. It will consist in a text file with as many rows and columns as we want to add to our map. There will be walls, entry points, exit points, treasures and food all around the scene.
- Then, we will create several levels, and implement a way of adding new levels to the game without changing any piece of code
- Next, we are going to make the main character go from one level to the next one, by reaching the exit point of each level
- Finally, we will add some game state events:
  - Main character's energy level will be continuously decreasing
  - There will be some food in each level to let the main character recover some energy
  - There will be some treasures in the levels to increase our score.

1.1. Previous changes

Before going on, we need to add a piece of code to avoid drawing walls beyond the yellow line at the bottom of the screen. Let's add this method to our *Hardware* class:

```
public void ClearBottom()
{
    Sdl.SDL_Rect source = new Sdl.SDL_Rect(0, GameController.SCREEN_HEIGHT,
        screenWidth, (short)(screenHeight - GameController.SCREEN_HEIGHT));
    Sdl.SDL_FillRect(screen, ref source, 0);
}
```

What we do is draw a black rectangle below this line, and after drawing the walls, so that walls beyond this limit will be hidden by this rectangle.

Then, we call this method at step #1 of the game loop, just before drawing the yellow line:

```
// 1. Draw everything
hardware.ClearScreen();
...
hardware.ClearBottom();
hardware.DrawLine(...);
```

2. Defining levels with text files

To begin with, let's create a text file structure to define the elements of a given level. Then, we will load this text file from the *Level* class and use it to define the walls and other elements of the level itself. But, before this, **remember to remove the string array** that we used for defining the map in previous session.

2.1. The file structure

Add a new folder in your project called “levels”. You can use the example given in the resources for this session, called “level1.txt”. Copy and paste it inside this folder (remember to check the “Copy always” property).

The file contains the same structure that we used before in a string array to determine the location of each wall. For instance:

```
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
W       W             W                W
W       W             W                W
W       W             W                W
W       W             W                W
W                     W         W      W
W                               W      W
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
```

2.2. Loading the text file

In order to load this text file, we are going to change the constructor of class *Level*. It will receive a file name as a parameter, then load the contents of this file in a string array, and process this array:

```
public Level(string fileName)
{
    ...
    Floor = new Image("imgs/floor.jpg", 1196, 920);
    string[] lines = File.ReadAllLines(fileName);
    if (lines.Length > 0)
    {
        Width = (short)(lines[0].Length * Sprite.SPRITE_WIDTH);
        Height = (short)(lines.Length * Sprite.SPRITE_HEIGHT);
        for (int i = 0; i < lines.Length; i++)
        {
            for(int j = 0; j < lines[i].Length; i++)
            {
                if (lines[i][j] == 'W')
                {
                    AddWall(new Wall((short)(j * Sprite.SPRITE_WIDTH),
                        (short)(i * Sprite.SPRITE_HEIGHT)));
                }
            }
        }
    }
}
```

Note that we can determine the width and height of the level by using the number of lines in the file (height) and the length of each line (width).

Now, in our *GameScreen* class, we no longer need the *loadLevel* method, so we can remove it, and the method call from the *Show* method:

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

We also need to change the way we use the constructor for *Level* class in the *GameScreen* constructor:

```
public GameScreen(Hardware hardware): base(hardware)
{
    font = new Font("fonts/prince_valiant.ttf", 20);
    audio = new Audio(44100, 2, 4096);
    audio.AddMusic("sound/song_b.mid");
    level = new Level("levels/level1.txt");
    initTexts();
}
```

3. Allowing multiple levels

In order to allow multiple levels, we need a way of going from one level to the next. For now, we will just need to reach an exit point at each level to move to next one.

3.1. Adding entry and exit points to the maps

So let's create two new classes that inherit from *StaticSprite* class. One of them will represent our starting point at each level, and the other one will be our exit point.

```
class StartPoint: StaticSprite
{
    public StartPoint(short x, short y)
    {
        X = x;
        Y = y;
    }
}
```

We add the two corresponding properties to our *Level* class:

```
class Level
{
    public List<Sprite> Walls { get; }
    public Image Floor { get; set; }
    public ExitPoint Exit { get; set; }
    public StartPoint Start { get; set; }
    ...
```

In the map text file, we will represent with an S the starting point, and with an E the exit point, this way:

```
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
W   S   W             W                W
W       W             W                W
W       W             W                W
W       W             W                W
W                     W         W      W
W                               W   E  W
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
```

And, as we read the map file, we define the coordinates of each point:

```
public Level(string fileName)
{
   ...
```

```
        for (int i = 0; i < lines.Length; i++)
        {
            for(int j = 0; j < lines[i].Length; j++)
            {
                if (lines[i][j] == 'W')
                {
                    ...
                }
                else if (lines[i][j] == 'S')
                {
                    Start = new StartPoint((short)(j * Sprite.SPRITE_WIDTH),
                       (short)(i * Sprite.SPRITE_HEIGHT));
                    XMap = (short)(Math.Max(0, Start.X -
                        GameController.SCREEN_WIDTH / 2));
                    YMap = (short)(Math.Max(0, Start.Y -
                        GameController.SCREEN_HEIGHT / 2));
                }
                else if (lines[i][j] == 'E')
                {
                    Exit = new ExitPoint((short)(j * Sprite.SPRITE_WIDTH),
                        (short)(i * Sprite.SPRITE_HEIGHT));
                }
            }
        }
    }
}
```

When we read the start point, we change the map coordinates to focus that point in the screen, just in case it is beyond the screen limits.

At the beginning of the *Show* method of *GameScreen* class, before the game loop, we need to place the main character at the level's start point:

```
public override void Show()
{
    short oldX, oldY, oldXMap, oldYMap;
    character.MoveTo((short)(level.Start.X - level.XMap),
        (short)(level.Start.Y - level.YMap));
    audio.PlayMusic(0, -1);
    ...
```

3.2. Changing the game loop structure to use levels

Now let's change our game loop to allow multiple levels. First of all, we define a variable called *currentLevel* in *Show* method, initialized to 1. We also define a boolean called *gameOver* initialized to *false*:

```
public override void Show()
{
    short oldX, oldY, oldXMap, oldYMap;
    byte currentLevel = 1;
    bool gameOver = false;
    ...
```

At the end of step #4 of the game loop, we check if the main character collides with the exit point of current level. If so, we check if there is one more level to show. Then we load this level and place the main character at its start point, otherwise we set *gameOver* to *true*.

```
// 4. Check collisions and update game state
if (character.CollidesWith(level.Walls, level.XMap, level.YMap))
{
    ...
}
if (character.CollidesWith(level.Exit, level.XMap, level.YMap))
{
    currentLevel++;
    if (File.Exists("levels/level" + currentLevel + ".txt"))
    {
        level = new Level("levels/level" + currentLevel + ".txt");
        character.MoveTo((short)(level.Start.X - level.XMap),
            (short)(level.Start.Y - level.YMap));
    }
    else
    {
        gameOver = true;
    }
}
```

Remember to add the *gameOver* flag to the conditions of the *while* loop, and to draw the exit point in step #1 of this loop:

```
do
{
    // 1. Draw everything
    hardware.ClearScreen();
    ...
    hardware.DrawSprite(Sprite.SpriteSheet, (short)(level.Exit.X - level.XMap),
        (short)(level.Exit.Y - level.YMap), level.Exit.SpriteX,
        level.Exit.SpriteY, Sprite.SPRITE_WIDTH, Sprite.SPRITE_HEIGHT);
    hardware.DrawSprite(Sprite.SpriteSheet, character.X, character.Y ...);
    hardware.ClearBottom();
    ...
} while (!gameOver && ...);
```

4. Adding some game state

In this section, we are going to add some game state features to the video game:

- The main character is going to lose some energy periodically.
- There will be some treasures in each level, so that if we "collide" with them, our score will be increased.

4.1. Losing energy periodically

At the constructor of *MainCharacter* class, let's start by setting a total energy of 1000, and 0 points for the score:

```
public MainCharacter()
{
    Energy = 1000;
    Points = 0;
}
```

Then, we are going to create a *Timer* that periodically (every second) will decrease in 1 unit the energy. We start it before the game loop of *GameScreen* class:

```
var timer = new Timer(this.DecreaseEnergy, null, 1000, 1000);
do
{
   ...
```

The *DecreaseEnergy* method will just decrease the *Energy* property and update the corresponding text:

```
public void DecreaseEnergy(Object o)
{
    if (character.Energy > 0)
        character.Energy--;
    Sdl.SDL_Color red = new Sdl.SDL_Color(255, 0, 0);
    textEnergy = SdlTtf.TTF_RenderText_Solid(font.GetFontType(),
        "ENERGY: " + character.Energy, red);
}
```

The timer must be disposed at the end of the game loop:

```
} while (!gameOver && !hardware.IsKeyPressed(Hardware.KEY_ESC));
audio.StopMusic();
timer.Dispose();
```

The *gameOver* flag must be set to *true* whenever the energy level reaches 0 (step #4 of the game loop):

```
// 4. Check collisions and update game state
...
if (character.Energy <= 0)
```

```
    gameOver = true;
```

4.2. Collecting treasures

Let's create a new class to store the sprite of a given treasure that we may find in the scene. This class will inherit from *StaticSprite* class:

```
class Treasure : StaticSprite
{
    public const short TREASURE_SCORE = 500;

    public Treasure()
    {
        SpriteX = 1144;
        SpriteY = 340;
    }

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

We define a constant to determine the amount of points winned by the character when he takes the treasure.

Let's add a new property to our *Level* class to store a list of treasures:

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

    public Level(string fileName)
    {
        Walls = new List<Sprite>();
        Treasures = new List<Treasure>();
        ...
```

We will represent the treasures with a T in the map text file, this way:

```
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
W   S   W       T     W                W
W       W             W                W
W       W             W                W
W       W             W                W
```

```
W                     W         W      W
W                            T  W   E  W
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
```

Every time we read a T from the map text file, we add a new treasure to the list:

```
public Level(string fileName)
{
    ...
        for (int i = 0; i < lines.Length; i++)
        {
            for (int j = 0; j < lines[i].Length; j++)
            {
                ...
                else if (lines[i][j] == 'T')
                {
                    Treasures.Add(new Treasure((short)(j * Sprite.SPRITE_WIDTH),
                       (short)(i * Sprite.SPRITE_HEIGHT)));
                }
            }
        }
    }
}
```

We need to draw the treasure list in step #1 of the game loop, at *GameScreen* class:

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

Then, we need to check if the main character collides with any treasure of the list, then remove it and update the score. To do this, we add this method in the *Level* class:

```
public bool CollidesCharacterWithTreasure(MainCharacter character)
{
    int pos = 0;
    bool collided = false;
    while (pos < Treasures.Count && !collided)
    {
        if (character.CollidesWith(Treasures[pos], XMap, YMap))
        {
            collided = true;
            Treasures.RemoveAt(pos);
        }
        pos++;
    }
    return collided;
}
```

and call it from step #4 of the game loop:

```
// 4. Check collisions and update game state
...
if (level.CollidesCharacterWithTreasure(character))
{
    character.Points = (ushort)(character.Points + Treasure.TREASURE_SCORE);
    updatePoints();
}
if (character.Energy <= 0)
    gameOver = true;
```

Regarding the *updatePoints* method, it is a new, private method in *GameScreen* class to update the score:

```
private void updatePoints()
{
    Sdl.SDL_Color green = new Sdl.SDL_Color(0, 255, 0);
    textPoints = SdlTtf.TTF_RenderText_Solid(font.GetFontType(), "POINTS: " +
         character.Points, green);
}
```

5. To think a little...

To finish with this session, let's add some food and different energy levels depending on each character:

5.1. Energy level must be character-dependent

Adapt the constructors of each subtype of character to assign a different energy level at the beginning of the game, this way:

- Warrior must have an energy of 1500
- Valkyrie must have an energy of 1200
- Sorcerer must have an energy of 1000
- Dwarf must have an energy of 800

5.2. Adding food to the map

Let's add one more element to our level maps: there will be food at some places of each level. These places will be marked with an "F" in the map text file, and we need to load a food sprite on them. Every time the main character collides with these food objects, its energy will be increased in 200 points.

Try to distinguish at least two different types of food (meat and potion, for instance), and assign a different energy level to each one.