ENSIIE

MOOC Report

Beginning Game Programming with C#

Student : Anthony Barbier

December 22, 2017

Sommaire

1 MOOC presentation **1**
- 1.1 Course introduction . . . 1
  - 1.1.1 Teacher . . . 1
  - 1.1.2 Learning objectives . . . 1
  - 1.1.3 Course structure . . . 2
- 1.2 MOOC validation . . . 3
  - 1.2.1 Monitoring . . . 3
  - 1.2.2 Final grade . . . 3

2 Modules **4**
- 2.1 Module 1 : Introduction . . . 4
- 2.2 Module 2 : First C# Program . . . 5
  - 2.2.1 Executing our code . . . 5
  - 2.2.2 Comments . . . 5
  - 2.2.3 Coding standards . . . 6
- 2.3 Module 3 : Data Types, Variables, and Constants . . . 7
- 2.4 Module 4 : Classes and Objects . . . 7
- 2.5 Module 5 : XNA Basics . . . 8
  - 2.5.1 General ideas for game development . . . 8
  - 2.5.2 XNA for game development . . . 9
  - 2.5.3 Simple drawing example . . . 10
- 2.6 Module 6 : Strings . . . 11
- 2.7 Module 7 : Selection . . . 12
- 2.8 Module 8 : XNA Mice and Controllers . . . 13
  - 2.8.1 Mouse location processing . . . 13
  - 2.8.2 Mouse button processing . . . 14
  - 2.8.3 Controller thumbstick and button processing . . . 15
- 2.9 Module 9 : Arrays and Collection Classes . . . 16
  - 2.9.1 Arrays . . . 16
  - 2.9.2 Collection classes . . . 17
- 2.10 Module 10 : Iteration . . . 18
  - 2.10.1 For Loops . . . 18
  - 2.10.2 Foreach loops . . . 20
  - 2.10.3 While loops . . . 20
- 2.11 Module 11 : Class Design and Implementation . . . 21
- 2.12 Module 12 : XNA Audio . . . 22
- 2.13 Module 13 : XNA Text IO . . . 23
  - 2.13.1 XNA keyboard input . . . 23
  - 2.13.2 XNA text output . . . 25

3 MOOC project **26**

4 Personal project **32**
- 4.1 Introduction . . . 32
- 4.2 Development . . . 33
  - 4.2.1 World map . . . 33
  - 4.2.2 Player : Pacman . . . 36
  - 4.2.3 Ghosts . . . 40
  - 4.2.4 Collision between Pacman and ghosts . . . 41

4.2.5 End of the game 41
4.2.6 Music 42

5 Conclusion **43**

1 MOOC presentation

1.1 Course introduction

The MOOC I followed this semester is an online course on Coursera, and is free of charge. Here's the link to the course : `https://www.coursera.org/learn/game-programming/`.

1.1.1 Teacher

This course is given by Tim *"Dr. T"* Chamillard, an associate professor in the Computer Science Departement at the University of Colorado. He teaches game design and development courses.

Figure 1: Dr. T, from University of Colorado

He also spent 5 and a half years as an indie game developer in a compagny that he started with his two sons.

1.1.2 Learning objectives

There are 3 main course learning objectives for this course.

- The first is learning basic programming concepts, independent of game development. Since there are complete beginners who have never programmed before, Dr. T has to teach us (or rather, them) the ideas behind programming developing software.
- The second learning objective is about basic object oriented concepts. C# is an object oriented language, and this paradigm is really useful for game development because in games, we have lots of interacting game entities that make our game run and which can be represented as classes and objects.
- The last learning objective is about using the XNA framework to actually build games. Since Microsoft stopped adding additional functionality to XNA several years ago, we're going to be working with MonoGame, which is a cross-platform, open-source implementation of the XNA framework.

1.1.3 Course structure

This course is an on-demand course, so we can work through it at our own pace, but it is recommended that we spend 12 weeks doing it.
I started the course on October 9 and finished it on December 17, at the end of week 10 for the recommended pace (it should have ended on January 01), which means I am two weeks ahead the recommended pace. I made sure to complete the course earlier so that I could write this report having studied the whole course.

For this course, there are 7 weeks of new material, 4 recovery weeks, and then there's a final week, where a final exam is to be taken (a multiple-choice questionnaire).

A new material week is composed of some lectures and videos. Then, after learning new things, there is an exam graded by other students. Finally, we have to review at least 5 other students, grading them and giving them advice and feedback on their code.
After completing all these tasks, the week is considered *completed*.

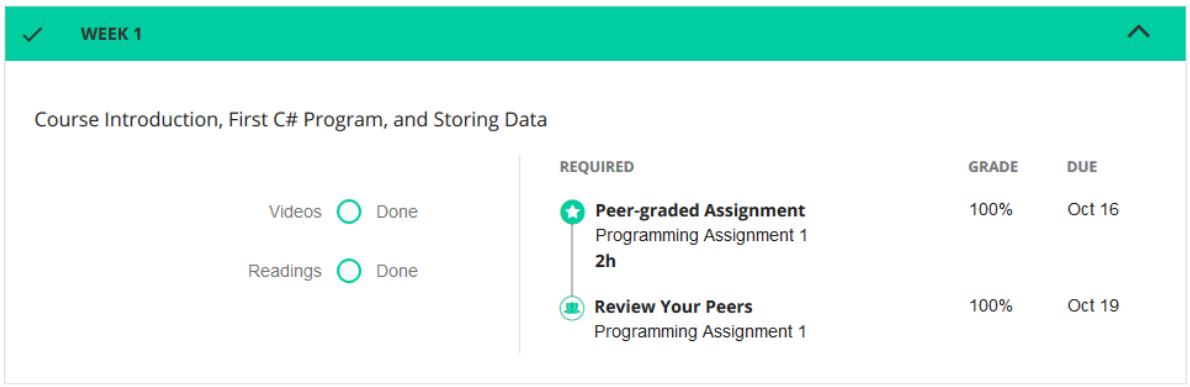


Figure 2: A new material week

Course project

There is a game to build throughout this course, divided into 5 project increments.
A recovery week consists in using what we learned the week(s) before in order to complete one of the project increments.
After completing the 5 increments, we have a full game we can play, how minimalist it is.
We'll talk about this optional project throughout this report.

1.2 MOOC validation

1.2.1 Monitoring

As I mentioned earlier, I completed the course on December 17. Here's a screenshot of my MOOC monitoring on Coursera :

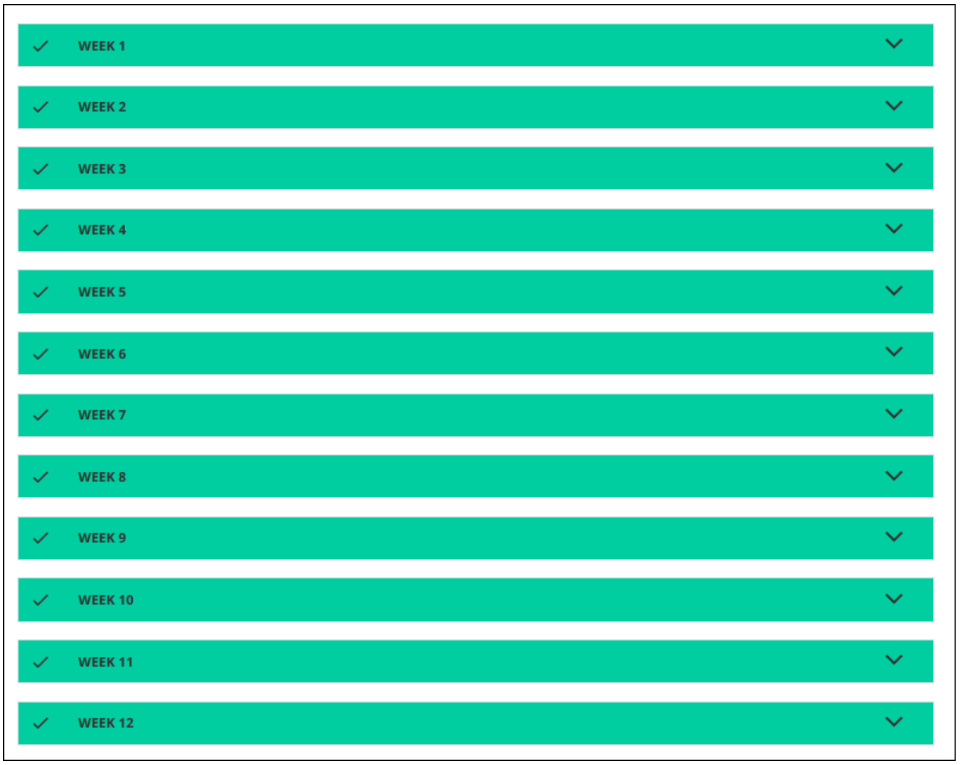


Figure 3: MOOC monitoring

1.2.2 Final grade

The final grade is made up of 3 components : 6 programming assignments (weight : 8% each), 5 project increments (6% each), and a final exam (22%).

I passed every one of the 12 weeks with a 100% grade, so my final course grade is 100/100 :

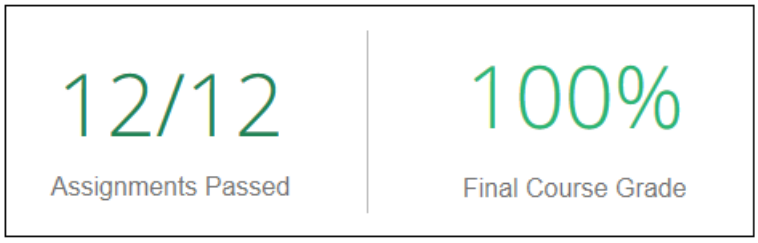


Figure 4: My final grade after completing the course

2 Modules

There are 13 modules in this course. During the new material weeks, there is at least one module, sometimes more.
In this section, we are gonna see what we learned in each module.
During the recovery weeks, there is no module to study. Let's take a peek at how these modules are organized in the schedule :

- Module 1 : week 1
- Module 2 : week 1
- Module 3 : week 1
- Module 4 : week 2
- Module 5 : week 2
- Module 6 : week 3
- Module 7 : week 3
- Module 8 : week 5
- Module 9 : week 5
- Module 10 : week 7
- Module 11 : week 9
- Module 12 : week 11
- Module 13 : week 11

2.1 Module 1 : Introduction

In this module, Dr. T tells us all we need to know about this course, which I summed up in this first section of this report (cf. MOOC presentation).

We also learn how to install Visual Studio (2017 in my case, the most recent version), and to add MonoGame to it, in order to be able to use it directly within Visual Studio.

Dr. T wants us to feel reassured, since MonoGame allows us to build real games, it is not just a tool for kids. For example, I found that famous games such as Dust: An Elysian Tail, FEZ, and Stardew Valley were made with MonoGame.

2.2 Module 2 : First C# Program

2.2.1 Executing our code

How to run our code has been one of the key point of this course. We just have to write ‘`Console.WriteLine("Hello world !");`‘, compile it and execute it.

2.2.2 Comments

Another key point of this module is about commenting. We should comment our code for many reasons : we want to communicate with other progrmmers through comments, we want to be able to come back to our code and remember quickly what a particular code line is for ...

In C#, there are two ways of commenting.

Documentation comments

The first way is with the documentation comments that we write at the top of classes and methods. They start with three slashes, and they have XML-like summary. We use those comments to actually generate documentation. For example, in this code we commented a constructor (which is in fact a method) :

```
/// <summary>
/// Constructs a burger
/// </summary>
/// <param name="contentManager">the content manager for loading content</param>
/// <param name="spriteName">the sprite name</param>
/// <param name="x">the x location of the center of the burger</param>
/// <param name="y">the y location of the center of the burger</param>
/// <param name="shootSound">the sound the burger plays when shooting</param>
public Burger(ContentManager contentManager,
    string spriteName,
    int x,
    int y,
    SoundEffect shootSound)
{
    // some code
}
```

Figure 5: Documentation comments, here at the top of a method

And so, I would be able to generate a documentation of my classes and methods, and give it to other programmers. I could give away DLLs to other programmers, but I would have to provide a documentation so that they can actually use my engine code.

Line comments

The other way we communicate with other programmers (including ourselves) is through line comments. These comments start with two slashes, and are always green (compared

to some code written in white in documentation comments). Unlike the documentation comments, they are not used to generate documentation, but are used to understand what's going on our code, what we try to do in a particular place.

For example, the '`// some code`' in the screenshot above is a line comment.

Also, we have to be careful when using line comments : we must not write really stupid comments, because they don't help. We have to comment wisely, not stupidly. It means that we should only put line comments that are useful to help programmers understand our code.

2.2.3 Coding standards

There are coding standards that we are going to be used in this course, about capitalization, commenting, indentation, white space, variable declarations, string variables, and statement length.
I won't give details of those standards, because it is not very interesting, but Dr. T wants us to follow them throughout the course.

2.3 Module 3 : Data Types, Variables, and Constants

In this module we learn basic programming concepts such as data type, variables, constants, integers, float/double, other value types, and the difference between bits and bytes.

There is nothing new that I hadn't already learned as part of my training at ENSIIE. Here, we just learn how to declare variables, and that to declare a constant we use the '`const`' keyword.

2.4 Module 4 : Classes and Objects

Module 4 is about classes and objects. We are going to design a class, and we will also be using existing classes provided by the teacher.

The object oriented paradigm is very well suited to game development, because in a game there are lots of entities that interact with each other, so modeling game worlds in our software as interacting objects is a really powerful technique that we can and will use.

Here, the foundational ideas behind the object oriented paradigm are explained : software objects have state, behavior, and identity.
When designing a class, we have different parts :

- The fields : things that we are going to make internal to the class.
- The properties for the class : they will be used to provide external consumers of the class knowledge about the fields. The properties are the getters and setters we talk about in object oriented languages.
- The methods : the behavior stuff we talked about.

We also learn what is a constructor and that it should have the same name as the class.

Nothing new in this module, we already knew it all thanks to our classes at ENSIIE.

2.5 Module 5 : XNA Basics

In this module, we talk about how to use XNA/MonoGame to develop games, and basic principles about developing games.

2.5.1 General ideas for game development

Ignoring XNA, we'll also learn a number of general things that we do in any game that we build :

- The first thing we do is initializing the game. It includes initializing some information that we are going to need throughout the game, setting score to zero...
- We also load content, it means loading the graphical assets (in 2D games, those assets are typically called *sprites*, in 3D games we have models and textures), the audio... We also need to load the levels, but we don't load them all at once, because it is too long. Usually, we do some up front, early on and regularly we will load more content as we go through a game. For example, we might load the initial level, and when the player has completed it, we need to load the next level.
- We then enter what is typically called the *game loop*, which basically runs until the player quits the game. This game loop updates the game, and changes the game world. Within the game loop, after updating our game world (all our entities), we need to draw it. We update then draw, then update then draw, etc... until the player quits the game. Dealing with menus and this kind of things is also a part of the update/draw loop.

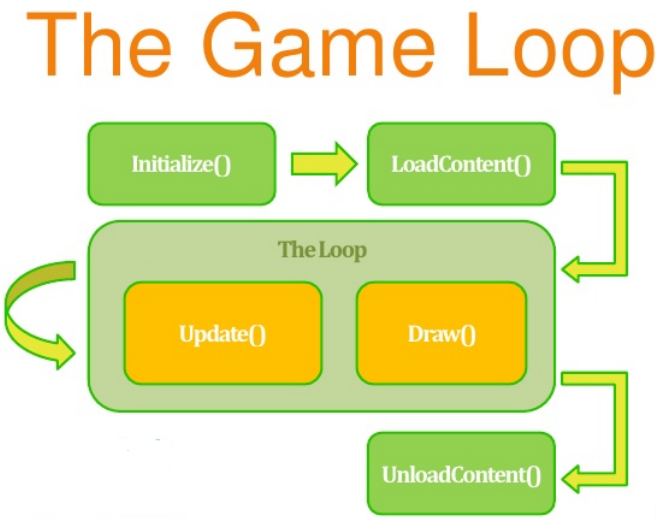


Figure 6: The XNA Game Loop

According to Dr. T, those are standard ideas that happen no matter what game development framework we're using.

2.5.2 XNA for game development

How does XNA support those things ?

When creating a new MonoGame project, a file `Game1.cs` is created : everything we mentioned above is contained in the `Game1` class. Let's have a look at the `Game1` class :

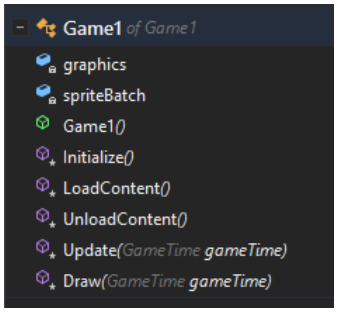


Figure 7: `Game1` class created by MonoGame

We have a `Game1` class, with a constructor `Game1`.

The sequence of events for XNA games is the constructor gets called to create a game object, then `Initialize` gets called to initialize the game world.
Then, `LoadContent` gets called to load the content we talked about : graphics, sound effects, this kind of things.
And then XNA enters the game loop.

So these first three things, that is to say constructor `Game1`, `Initialize` method, and `LoadContent` method are those pre-game loop activities that we do in pretty much every game we build.

During this course, Dr. T will not be using the `UnloadContent` method, because we just do not need it. For larger games, we would unload content so that we don't have everything in memory as we play.

We will add all our world updating logic in the `Update` method. In fact, this method will get called every frame, so whenever we need to update the game world, we will do it in this method. When generating the `Game1` class, MonoGame creates a pre-built `Update` method :

```
/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
        Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // TODO: Add your update logic here

    base.Update(gameTime);
}
```

Figure 8: `Update` method, generated by MonoGame

2.5.3 Simple drawing example

Let's draw a picture onto our screen using XNA.

First, we have to add the picture to our project in Visual Studio. We should add `.xnb` files to a content folder, built with the MonoGame Pipeline from the actual image. A tutorial about building content with MonoGame Pipeline is provided by the teacher in this course, but isn't very interesting, so I will not enlarge on it.
`.xnb` files represent a content for XNA, it can be a picture, an audio file, etc ...

In XNA, we draw a texture onto a rectangle, so we need to declare 2 variables in the `Game1` class.

```
public class Game1 : Game
{
    // bears drawing support

    private Texture2D bear;
    private Rectangle drawRectangle;
```

Then, we load the sprite (the image) in the `LoadContent` method (note that the coordinates of the top-left corner of the window are (0;0)) :

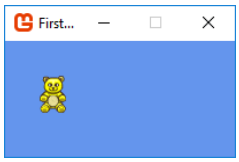
```
protected override void LoadContent()
{
    // load teddy bears and build draw rectangles

    bear = Content.Load<Texture2D>(@"graphics\teddybear0");
    drawRectangle = new Rectangle(150, 100, bear.Width, bear.Height);
```

Rectangle.Rectangle(int x, int y, int width, int height) (+ 2 overloads)
Creates a new instance of Rectangle struct, with the specified position, width, and height.

Finally, we draw the bear ('`Color.White`' means that the image should be drawn using its actual colors):

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // draw bear
    spriteBatch.Begin();

    spriteBatch.Draw(bear, drawRectangle, Color.White);

    spriteBatch.End();

    base.Draw(gameTime);
}
```

The result is what we expect, a bear drawn at the position we specified :



2.6 Module 6 : Strings

Using strings is very important when developing a video game. They are used to display information on the screen, in menus, etc ... In this module, we learn how to manipulate strings in C#.

We learned some useful things we can do to manipulate strings. First, we can concatenate 2 strings using the `+` operator.
We can use instance methods such as `IndexOf` and `Substring`. These instance methods are called like this : ‘`myStringVariable.IndefOf()`‘.
For example, the following lines would result in `commaLocation` equal to 5 :

```
// find comma location
string myString = "Hello, world !";
int commaLocation = myString.IndexOf(',');
```

We also learned how to get user input, reading a line the user types, thanks to ‘`Console.ReadLine()`‘ :

```
// prompt for and read in gamertag
Console.Write("Enter gamertag : ");
string gamertag = Console.ReadLine();
```

Nothing very difficult for this week.

2.7 Module 7 : Selection

In this module we learn how to make decisions in our programs. Regarding video games, we will be able to control what happens in our game thanks to selection control structure. For example, we might decide, based on user input, which direction we should move the player's avatar.

The exampled used by Dr. T in this module is bouncing bears whenever they get outside the window. To check if a bear is outside the game window, we have to check the sprite's draw rectangle position (see Simple drawing example for a reminder about sprites and draw rectangles).

Here, we also learn that when creating a class, for example a Bear class, we can add a `Update` public method to the class, so that these objects can update themselves :

```
/// <summary>
/// Updates the teddy bear's location, bouncing if necessary
/// </summary>
public void Update()
{
    if (active)
    {
        // move the teddy bear (velocity is a Vector2 object with a X and Y component)
        drawRectangle.X += (int)(velocity.X);
        drawRectangle.Y += (int)(velocity.Y);

        // bounce as necessary (these two methods are private methods within the class)
        BounceTopBottom();
        BounceLeftRight();
    }
}
```

Of course, we have to tell these entities to update themselves in the `Game1 Update` method :

```
/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed ||
        Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // update game objects
    bear0.Update();
    bear1.Update();
```

So, on every frame, we can tell game entities to update themselves.

2.8 Module 8 : XNA Mice and Controllers

Module 8 is a very exciting one for learning an incredibly important piece of game development : how to get and use player input. In this week we focus on using the mouse and an Xbox 360 controller. Keyboard input will be covered later in this course.

2.8.1 Mouse location processing

First, we are going to focus on using the mouse location in games.
We are going to see how to use mouse location by drawing a character onto our screen, making this character follow the mouse.

To do so, we need a `Rectangle drawRectangle` and a `Texture2D characterSprite` that we declare inside our `Game1` class.

We load the content, here the character sprite, in the `LoadContent` method, and give appropriate width et height to its draw rectangle :

```
protected override void LoadContent()
{
    // load character sprite
    characterSprite = Content.Load<Texture2D>(@"graphics\character0");

    // start character in top left corner
    drawRectangle = new Rectangle(
        0,
        0,
        characterSprite.Width,
        characterSprite.Height);
```

Then, we update our draw rectangle object using the mouse coordinates so that the character in centered on the mouse. We also make sure that the character is clamped inside the game window, because we do not want it to be able to get out of the game window (note that the Left, Right, Top and Bottom properties of the `Rectangle` object automatically update when we change the rectangle's X or Y properties (which are the coordinates of the top left corner of the rectangle)) :

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

    // make character follow mouse
    MouseState mouse = Mouse.GetState();
    drawRectangle.X = mouse.X - characterSprite.Width / 2;
    drawRectangle.Y = mouse.Y - characterSprite.Height / 2;

    // clamp character in window
    if (drawRectangle.Left < 0)
        drawRectangle.X = 0;
    if (drawRectangle.Right > WindowWidth)
        drawRectangle.X = WindowWidth - drawRectangle.Width;
    if (drawRectangle.Top < 0)
        drawRectangle.Y = 0;
    if (drawRectangle.Bottom > WindowHeight)
        drawRectangle.Y = WindowHeight - drawRectangle.Height;
```

And so, when moving the mouse around, the character (more precisely, the center of the draw rectangle) will follow the mouse, but never go outside the game window.

2.8.2 Mouse button processing

Now, we want to use a mouse button for input to change the character to another one. Actually, we're just going to change the sprite of the character.

When we click on the left button of the mouse, we want a character to be randomly picked out of 4 possible characters. To do so, we need to introduce randomness to our games.
We obviously need to load the 4 sprites in the `LoadContent` method first.

The `Random` class

Brief point about randomness : we will be using the `Random` class to generate random numbers. We just have to create a new `Random` object (usually named `rand`) at the top of our `Game1` class (only once for the entire game), and then we use the Next method by calling `rand.Next()`.
This method has 2 overloads : we can provide either a maximum int values, or both a minimum and a maximum int values.

Checking if a button is pressed

We can check if a button is pressed using the `mouse.LeftButton` property. This property gives us one out of two values of an enumeration : either `ButtonState.Pressed` or `ButtonState.Released`.

We can then compare the current state of the left button thanks to those 2 values.

Processing a click

Since we want to change to another random character after every click, we have to be able to actually detect a click.

A click is pressing the button followed by releasing it.
So the way we detect a click is : we check if the button is currently released **and** that it was being pressed just before.

It appears that we need to store in a variable the previous state of the mouse left button, so we have to declare such a variable.
After every update of the game (every frame), we will update the previous state of the left button.

Also, we want to store the current character in a variable, so that we can update our draw rectangle's properties (width and height) after changing the current character.
All of this is done in the `Update` method of our `Game1` class, as shown is the image below :

```
// change character on left mouse click (NOT LEFT MOUSE PRESS!)
if (mouse.LeftButton == ButtonState.Released &&
    previousButtonState == ButtonState.Pressed)
{
    // change to random character (between 0 and 3 inclusive, so 0 to 4 exclusive)
    int characterNumber = rand.Next(0, 4);
    if (characterNumber == 0)
    {
        currentCharacter = character0;
    }
    else if (characterNumber == 1)
    {
        currentCharacter = character1;
    }
    else if (characterNumber == 2)
    {
        currentCharacter = character2;
    }
    else
    {
        currentCharacter = character3;
    }
    drawRectangle.Width = currentCharacter.Width;
    drawRectangle.Height = currentCharacter.Height;
}
previousButtonState = mouse.LeftButton;
```

2.8.3 Controller thumbstick and button processing

We also learned how to process input with a Xbox 360 controller thumbstick and button.

Since it is extremely similar to what we do with a keyboard, and that we will talk about keyboard input later in this report, I prefer to shorten this part.

2.9 Module 9 : Arrays and Collection Classes

2.9.1 Arrays

Arrays are very easy to understand, and particularly since we learned how to use them in many programming languages at ENSIIE.
I won't detail how to use them precisely, but I will show that we learned how to use them to be actually more logical when developing games.
We certainly need to know that an array should be created knowing how many elements we want to store in it, and that it can hold only one data type, not more.

Here's an example of how to get the same result as in the example where when we wanted to change the current character sprite based on a random number on every mouse left button click.

First, we need to declare an array of length 4 in our `Game1` class :

```
public class Game1 : Game
{
    Texture2D[] characters = new Texture2D[4];
```

Then, in the `LoadContent` method, we load the 4 different sprites for our characters directly inside the array :

```
// load character sprites
characters[0] = Content.Load<Texture2D>(@"graphics\character0");
characters[1] = Content.Load<Texture2D>(@"graphics\character1");
characters[2] = Content.Load<Texture2D>(@"graphics\character2");
characters[3] = Content.Load<Texture2D>(@"graphics\character3");
```

Finally, in the `Update` method, we do not need to have an if selection statement anymore, we just need to select a random number inside the array of characters :

```
// change character on left mouse click (NOT LEFT MOUSE PRESS!)
if (mouse.LeftButton == ButtonState.Released &&
    previousButtonState == ButtonState.Pressed)
{
    // change to random character
    currentCharacter = characters[rand.Next(4)];
    drawRectangle.Width = currentCharacter.Width;
    drawRectangle.Height = currentCharacter.Height;
}
previousButtonState = mouse.LeftButton;
```

2.9.2 Collection classes

In C#, we need to know how many variables we need when we create an array object.
Is many cases, we do not know how many variables we will be using : the number of elements we need to store changes dynamically. For example, in a game where we spawn and kill stuff, that's a problematic limitation.

Collection classes do not have that limitation. They can grow and shrink dynamically as we need them to as the game runs.
As for arrays, they can only hold one data type.

We already studied collections at ENSIIE, during the course about Object Oriented Paradigm (ILO), so I won't explain in details how do they work, we will simply use them.

We're going to use the `List` collection class to do the same example we did with arrays.

First, we declare the list we will use :

```
public class Game1 : Game
{
    List<Texture2D> characters = new List<Texture2D>();
```

We then load the 4 different sprites for our characters in the `LoadContent` method, adding them to our list :

```
// load character sprites
characters.Add(Content.Load<Texture2D>(@"graphics\character0"));
characters.Add(Content.Load<Texture2D>(@"graphics\character1"));
characters.Add(Content.Load<Texture2D>(@"graphics\character2"));
characters.Add(Content.Load<Texture2D>(@"graphics\character3"));
```

And finally, we have to pick a random character and update the variable storing the current character. Since the code used to do this with the List collection class is exactly the same as with array, I won't put a screenshot for this.

2.10 Module 10 : Iteration

In this module, we learn another important control structure : iteration, also called looping. We regularly have to update and draw arrays or collections of game entities, that this will really be useful for developing our games !

When looping, we can use 3 types of loops : the for loop, the foreach loop, and the while loop.

2.10.1 For Loops

Now that we know how to use collection classes (such as lists) to store our game entities, we will use them and iterate over them in our games.

Example using a for loop

For example, let's say we want to have a game in which we spawn a new bear every one second.
To do so, we need to store our bears (the class provided by Dr. T is called `TeddyBear`) in a collection class (List here), and spawning support variables : one will be a constant equal to the spawn delay between two bears, and another one will be the total milliseconds elapsed since the last spawn.

```
// spawning support
private const int TotalSpawnDelayMilliseconds = 1000;
private int elapsedSpawnDelayMilliseconds = 0;

// game objects
private List<TeddyBear> bears = new List<TeddyBear>();
```

Now, in the `Update` method, we update the timer, and if we reached the spawn delay, we create a need bear and reset the timer. Then, we update our bears with a for loop :

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

    // spawn bears as appropriate
    elapsedSpawnDelayMilliseconds += gameTime.ElapsedGameTime.Milliseconds;
    if (elapsedSpawnDelayMilliseconds >= TotalSpawnDelayMilliseconds)
    {
        // reset the delay since last spawn
        elapsedSpawnDelayMilliseconds = 0;

        // add a new bear
        // GetRandomTeddyBear() creates a new TeddyBear object with a random location
        bears.Add(GetRandomTeddyBear());
    }

    // update bears
    for (int i = 0; i < bears.Count; i++)
    {
        bears[i].Update(gameTime);
    }
```

Finally, in the `Draw` method, we have to draw every bear. We created an `Update` method in the `TeddyBear` class, so we just need to tell each bear to draw itself with a for loop :

```
spriteBatch.Begin();

// draw teddy bears
for (int i = 0; i < bears.Count; i++)
{
    bears[i].Draw(spriteBatch);
}

spriteBatch.End();
```

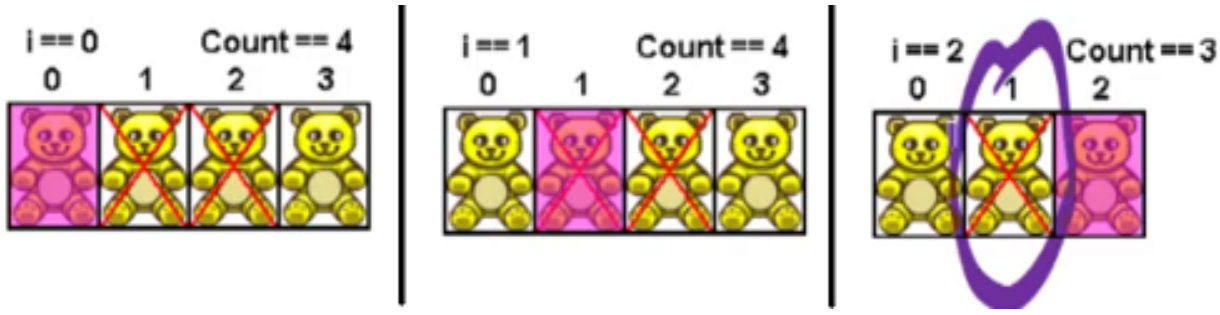
And so, every second a new bear will be added to our collection. On every frame, each bear will update itself then draw itself.

For loop and `Remove` method

When we do not need some of our bears in our bear collection, we could for example make them inactive. We want to remove those inactive bears from our bear collection, because we don't need to update and draw them, because it wastes time and CPU cycle processing.

We have to be careful when using a for loop when removing entities from a collection, because going front to back of the collection is not the way we should do it (and it is the way we usually do for any other for loop !).
The reason is that when we use a remove method (for example, the `RemoveAt` method), the end of the collection is shifted over.
And so, we will miss elements in our collection !

For example, if we have an array of 4 bears in this order : active (0), inactive (1), inactive (2), active (3), when iterating from 0 to Count (the number of bears in the array) in order to remove the inactive bears, here's what happens (with a screenshot from a video in the course) :



We missed one bear which was inactive, and so we could not remove it from the collection !

The solution to this issue is to iterate over the collection from back to front, because when bears will be shifted over to the front of the collection, we do not care anymore

since we already dealt with them.

When iterating over a collection when removing items from it, we should do it from back to front :

```
// remove dead teddies
for (int i = bears.Count - 1; i >= 0; i--)
{
    if (!bears[i].Active)
    {
        bears.RemoveAt(i);
    }
}
```

2.10.2 Foreach loops

When iterating, we can also use a foreach loop.
We cannot use the foreach loop when iterating a collection in which we will remove items, for the same reasons as with a for loop from front to back.

Here's the foreach loop syntax :

```
// update bears
foreach (TeddyBear teddyBear in bears)
{
    teddyBear.Update(gameTime);
}
```

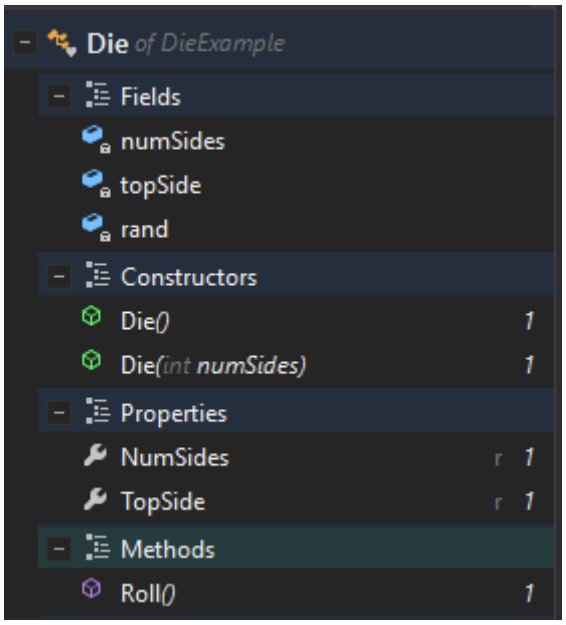
2.10.3 While loops

There was nothing very interesting when we learned how to use while loops.
We use them as we would in any programming language.

2.11 Module 11 : Class Design and Implementation

In this module, we learn how a class is designed : with fields, constructors, properties, and methods.
Nothing new compared to what we learned at ENSIIE, but is was necessary for Dr. T to explain this notions to people who did not know them yet.

Here is a quick look at what a Die class can look like :



In our games, we are usally going to add `Update` and `Draw` methods to our custom classes, to tell these entities to update and draw themselves easily.

2.12 Module 12 : XNA Audio

In video games, a very important thing to entertain the player is music.
In this module, we learn how to play a sound effect (for example, when taking damage), and a background music.

Playing a sound effect

A sound effect is, just like an image is, a game content.
We need to build this content using the MonoGame Pipeline, in order to create a `.xnb` file.

First, we declare a `SoundEffect` variable in the `Game1` class :

```
// collision sound effects
private SoundEffect destroy;
```

In the `LoadContent` method, we load the sound effect :

```
// load a sound effect
SoundEffect destroy = Content.Load<SoundEffect>(@"audio\zombie");
```

Now, every time we want the sound to play, we just use its `Play` method in the `Update` method :

```
// play the sound effect
destroy.Play();
```

Playing a background music

A background music is here considered to be a music that plays on a loop.
Since the `LoadContent` method is called once and only once in the program, we can declare the background music variable, load the content, and play it on a loop there.

To tell the music to play on a loop, we have to create an instance of it, and then its `IsLooped` property to true :

```
// load and start playing background music
SoundEffect backgroundMusicEffect = Content.Load<SoundEffect>(@"audio\backgroundMusic");
backgroundMusic = backgroundMusicEffect.CreateInstance();
backgroundMusic.IsLooped = true;
backgroundMusic.Play();
```

Now, our music will play on a loop when the program is launched.

2.13 Module 13 : XNA Text IO

We are going to look at text IO. IO stands for input and ouput.
We will use the keyboard input to control our character in the game, and we will do text output to display important game information, like score and health for example.

2.13.1 XNA keyboard input

Let us see how to use the keyboard input to control our character in the game.
The principle is very similar to what we do with the mouse for checking is a button/key is down. The difference lies in moving the character.

With the mouse, we have to update the coordinates of our draw rectangle so that they match the mouse's coordinates.
With the keyboard, as long as a movement key is pressed, we have to move the character in the appropriate direction.

This time, we do not want to retrieve a mouse state in our `Update` method, but a keyboard state. We pass it as an argument to the bear's `Update` method :

```
/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)

{
    KeyboardState keyboard = Keyboard.GetState();

    // update the bear
    bear.Update(keyboard);
```

Now, when updating the bear in its own `Update` method, we check if a directional key is pressed, and move accordingly our bear by a constant amount of pixels (here, I chose 5 pixels) :

```
/// <summary>
/// Updates the bear's location
/// </summary>
/// <param name="keyboard">current keyboard state</param>
public void Update(KeyboardState keyboard)
{
    // move the bear based on the keyboard state
    if (keyboard.IsKeyDown(Keys.Right))
        X += BearMoveAmount;

    if (keyboard.IsKeyDown(Keys.Left))
        X -= BearMoveAmount;

    if (keyboard.IsKeyDown(Keys.Up))
        Y -= BearMoveAmount;

    if (keyboard.IsKeyDown(Keys.Down))
        Y += BearMoveAmount;
}
```

When several directional keys are pressed (for example, both Right and Left), the bear does not move : its X property won't change because the first two 'if' conditions are true.

Beware ! When moving horizontally or diagonally, I'm moving at 5 pixels per update. But when moving diagonally, I'm moving at $5 \times \sqrt{2}$ pixels per update, which is faster than moving in a single direction !

This is a bug that can be fixed, and there is pretty famous bug in the game Doom that people called 'Strafe 40 bug' which essentially happens because of a similar idea to this one.

We have to be careful when dealing with keyboard input to move our characters.

2.13.2 XNA text output

To display text in our XNA games, we need a SpriteFont, which is a font that will get used within our games.
A SpriteFont is a content, just like images and audio. We need to build that content using the MonoGame Pipeline and create a `.xnb` file.

In order to display text, we need to have a font, a string (in which we will put our text), and a position.
The position will be stored as a vector with 2 coordinates (a `Vector2`).

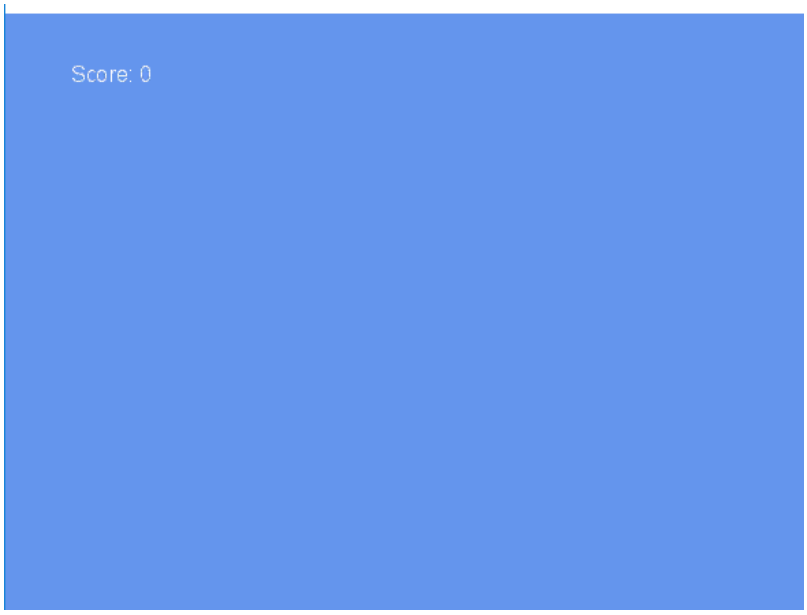
```
// text display

private SpriteFont font;
private Vector2 scorePosition = new Vector2(WindowWidth / 12, WindowHeight / 12);
private string scoreString;
```

In the `Update` method, we change the `scoreString` to whatever we want to display on the screen, and then we draw the string in the `Draw` method, at the position we want :

```
// draw score
spriteBatch.Begin();

spriteBatch.DrawString(font, scoreString, scorePosition, Color.White);

spriteBatch.End();
```

```
void SpriteBatch.DrawString(SpriteFont spriteFont, string text, Vector2 position, Color color) (+ 5 overloads)
Submit a text string of sprites for drawing in the current batch.
```

The result is :



3 MOOC project

At the end of recovery weeks, and at the end of week 11, we had to do increments for a project, guided by Dr. T.
The project is broken up into 5 increments, and is meant to give us some experience developing a simple game using the concepts covered in the course.

The game we developped is a simple 2D shooter in which you try to feed french fries to bears until they explode.
The bears explode after eating one serving of french fries. The bears fight back, of course.

Dr. T provided us some code, and in every project increment we had to modify it so that at the end of programming increment 5, we would have a complete game !

I will present the steps followed during these project increments, and give a link to a video we had to upload at the end of each project increment. When there is a technique we didn't elaborate on in the modules, i'll explain them.

Project increment 1

Link to te video of this project increment : `http://barbier.iiens.net/Cours/MOOC/pi1.mp4`.

In this increment we display a stationary burger and a moving bear.

Steps followed :

- Add burger
- Add bear
- Move bear

To create the bear at a random location and with a random speed, here's how I did it :

```
// generate random location
int x = GetRandomLocation(GameConstants.SpawnBorderSize,
    graphics.PreferredBackBufferWidth - 2 * GameConstants.SpawnBorderSize);
int y = GetRandomLocation(GameConstants.SpawnBorderSize,
    graphics.PreferredBackBufferHeight - 2 * GameConstants.SpawnBorderSize);

// generate random velocity
float speed = GameConstants.MinBearSpeed +
    RandomNumberGenerator.NextFloat(GameConstants.BearSpeedRange);

float angle = RandomNumberGenerator.NextFloat(2 * (float)Math.PI);

Vector2 velocity = new Vector2(
    (float)(speed * Math.Cos(angle)),
    (float)(speed * Math.Sin(angle)));

// create new bear
TeddyBear newBear = new TeddyBear(Content, @"graphics\teddybear", x, y, velocity, null, null);
                    TeddyBear.TeddyBear(Microsoft.Xna.Framework.Content.ContentManager contentManager, string spriteName, int x, int y, Vector2 velocity, SoundEffect bounceSound, SoundEffect shootSound)
                    Constructs a teddy bear centered on the given x and y with the given velocity
bears.Add(newBear);
```

Project increment 2

Link to te video of this project increment : `http://barbier.iiens.net/Cours/MOOC/pi2.mp4`.

In this increment the player will now be able to move the burger with the mouse and make it shoot. The bears will shoot automatically too.

Steps followed :

- Move burger
- Have burger shoot
- Have projectiles move
- Control burger firing rate
- Deactivate projectiles that have left the screen
- Have bears shoot

To control burger firing rate, here's what we had to do : if the player keeps the mouse left button pressed, he can only shoot every 0.5 second, but if he spams left button clicks, there is no limit and he can shoot as fast as he's clicking.

Here's how I implemented this :

```
// update shooting allowed
if (!canShoot)
{
    elapsedCooldownMilliseconds += gameTime.ElapsedGameTime.Milliseconds;
    if (elapsedCooldownMilliseconds >= GameConstants.BurgerTotalCooldownMilliseconds ||
        mouse.LeftButton == ButtonState.Released)
    {
        canShoot = true;
        elapsedCooldownMilliseconds = 0;
    }
}
```

Project increment 3

Link to te video of this project increment : `http://barbier.iiens.net/Cours/MOOC/pi3.mp4`.

In this increment we blow up bears, spawn multiple bears, and have them bounce off each other.

Steps followed :

- Have bears collide with projectiles
- Have bears blow up
- Remove finished explosions
- Include multiple bears
- Bounce bears off each other

To check for collisions (to blow up bears or to bounce them), we just have to check if the draw rectangles of every couple of entities in the game intesect.

To check and resolve collisions between bears and projectiles (explode them), I did it directly like that :

```
// check and resolve collisions between teddy bears and projectiles
foreach (TeddyBear bear in bears)
{
    foreach (Projectile projectile in projectiles)
    {
        if (projectile.Type == ProjectileType.FrenchFries &&
            bear.Active &&
            projectile.Active &&
            bear.CollisionRectangle.Intersects(projectile.CollisionRectangle))
        {
            bear.Active = false;
            projectile.Active = false;
            explosions.Add(new Explosion(explosionSpriteStrip, bear.Location.X,
                bear.Location.Y));
        }
    }
}
```

To check and resolve collisions between bears (bounce them), Dr. T provided us a `CollisionResolutionInfo` class and told us exactly how to use it. It allows us to check if a rebound has led to an out of bounds bear. If the rebound happened correctly, we change at least one bear's velocity.

Here's the code for this collision resolution :

```
// check and resolve collisions between teddy bears
for (int i = 0; i < bears.Count; i++)
{
    for (int j = i + 1; j < bears.Count; j++)
    {
        if (bears[i].Active &&
            bears[j].Active)
        {
            CollisionResolutionInfo cri = CollisionUtils.CheckCollision(
                gameTime.ElapsedGameTime.Milliseconds,
                GameConstants.WindowWidth,
                GameConstants.WindowHeight,
                bears[i].Velocity, bears[i].DrawRectangle,
                bears[j].Velocity, bears[j].DrawRectangle);
            if (cri != null)
            {
                // resolve collision
                if (cri.FirstOutOfBounds)
                {
                    bears[i].Active = false;
                }
                else
                {
                    bears[i].Velocity = cri.FirstVelocity;
                    bears[i].DrawRectangle = cri.FirstDrawRectangle;
                }
                if (cri.SecondOutOfBounds)
                {
                    bears[j].Active = false;
                }
                else
                {
                    bears[j].Velocity = cri.SecondVelocity;
                    bears[j].DrawRectangle = cri.SecondDrawRectangle;
                }
            }
        }
    }
}
```

Project increment 4

Link to te video of this project increment : `http://barbier.iiens.net/Cours/MOOC/pi4.mp4`.

In this increment we spawn new bears as appropriate, blow up bears, add burger health, and add burger collisions with bears and bear projectiles.

Steps followed :

- Spawn new bear when one is destroyed
- Only spawn new bear into a collision-free location
- Add burger health
- Add burger collisions with bears
- Add burger collisions with projectiles

We make sure new bears only spawn into a collision-free location inside a method (in the `Game1` class) called `SpawnBear`.
We generate a random velocity, a random location, and then create the bear. Then, we change the bear's coordinates until he spawns into a collision-free location, that is to say that he doesn't collide with another game entity when spawning.
Here's the `SpawnBear` method :

```
/// <summary>
/// Spawns a new teddy bear at a random location
/// </summary>
private void SpawnBear()
{
    // generate random velocity
    float speed = GameConstants.MinBearSpeed +
        RandomNumberGenerator.NextFloat(GameConstants.BearSpeedRange);
    float angle = RandomNumberGenerator.NextFloat(2 * (float)Math.PI);
    Vector2 velocity = new Vector2(
        (float)(speed * Math.Cos(angle)), (float)(speed * Math.Sin(angle)));

    // generate random location
    int x = GetRandomLocation(GameConstants.SpawnBorderSize,
        graphics.PreferredBackBufferWidth - 2 * GameConstants.SpawnBorderSize);
    int y = GetRandomLocation(GameConstants.SpawnBorderSize,
        graphics.PreferredBackBufferHeight - 2 * GameConstants.SpawnBorderSize);

    // create new bear
    TeddyBear newBear = new TeddyBear(Content, @"graphics\teddybear", x, y, velocity,
        null, null);

    // make sure we don't spawn into a collision
    List<Rectangle> collisionRectangles = GetCollisionRectangles();
    while (!CollisionUtils.IsCollisionFree(newBear.CollisionRectangle, collisionRectangles))
    {
        newBear.X = GetRandomLocation(GameConstants.SpawnBorderSize,
            graphics.PreferredBackBufferWidth - 2 * GameConstants.SpawnBorderSize);
        newBear.Y = GetRandomLocation(GameConstants.SpawnBorderSize,
            graphics.PreferredBackBufferHeight - 2 * GameConstants.SpawnBorderSize);
    }

    // add new bear to list
    bears.Add(newBear);
}
```

Project increment 5

Link to te video of this project increment : `http://barbier.iiens.net/Cours/MOOC/pi5.mp4`.

In this increment we add sound effects and health and score displays to the game.

Steps followed :

- Add health display
- Add scoring system and score display
- Change burger control scheme(Mouse to Keyboard)
- Add burger and teddy bear shooting sound effects
- Add teddy bear bounce sound effects
- Add burger damage sound effects
- Add explosion sound effects
- Add losing sound effects

That's it ! The game is complete now.

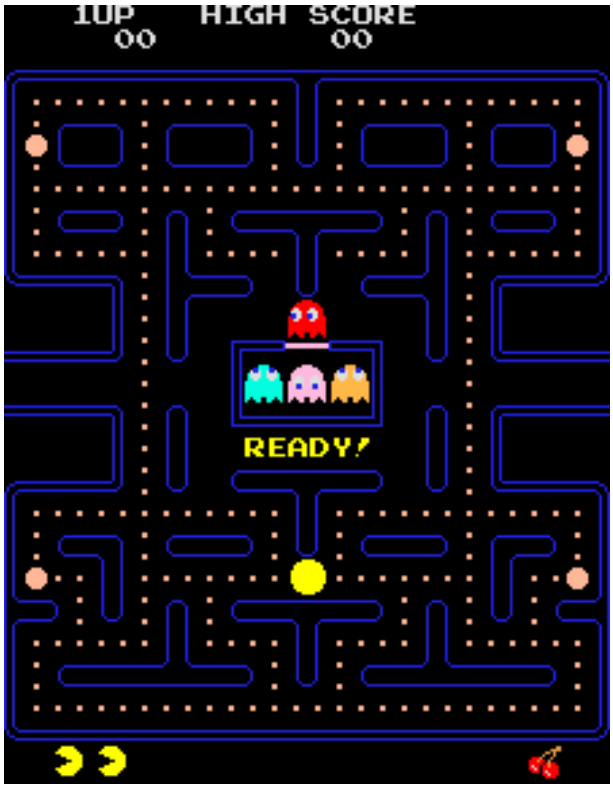
The sources can be found in the `Anthony_Barbier_MOOC_project` folder (or here : `http://barbier.iiens.net/Cours/MOOC/Anthony_Barbier_MOOC_project.zip`).

4 Personal project

4.1 Introduction

We had to do a personal project for the MOOC oral presentation.
This section has been added to my report in January 2018, after I finished the game I developed.

The game I developed is a simplified Pac-Man-like game.
Here's a screenshot of the original complete Pac-Man game :



Here's what I simplified compared to the original game :

- Only one game is played. It means that when you win or die, you are stuck and cannot play again, you have to exit and relaunch the game to do so.
- There are no candies to eat ghosts, so you just cannot eat ghosts, you will have to avoid them.
- The tunnels are not working. The entrance of each tunnel is coded as a wall.
- The ghosts do not start in the center rectangle : since there are 4 ghosts, I made them start at each corner of the map.
- Ghosts are not intelligent : they don't chase Pacman, but rather choose their path at random.
- There is no animation when Pacman dies.

In my game, the player's goal is to collect all the dots (I also call them "points") while avoiding the ghosts.

4.2 Development

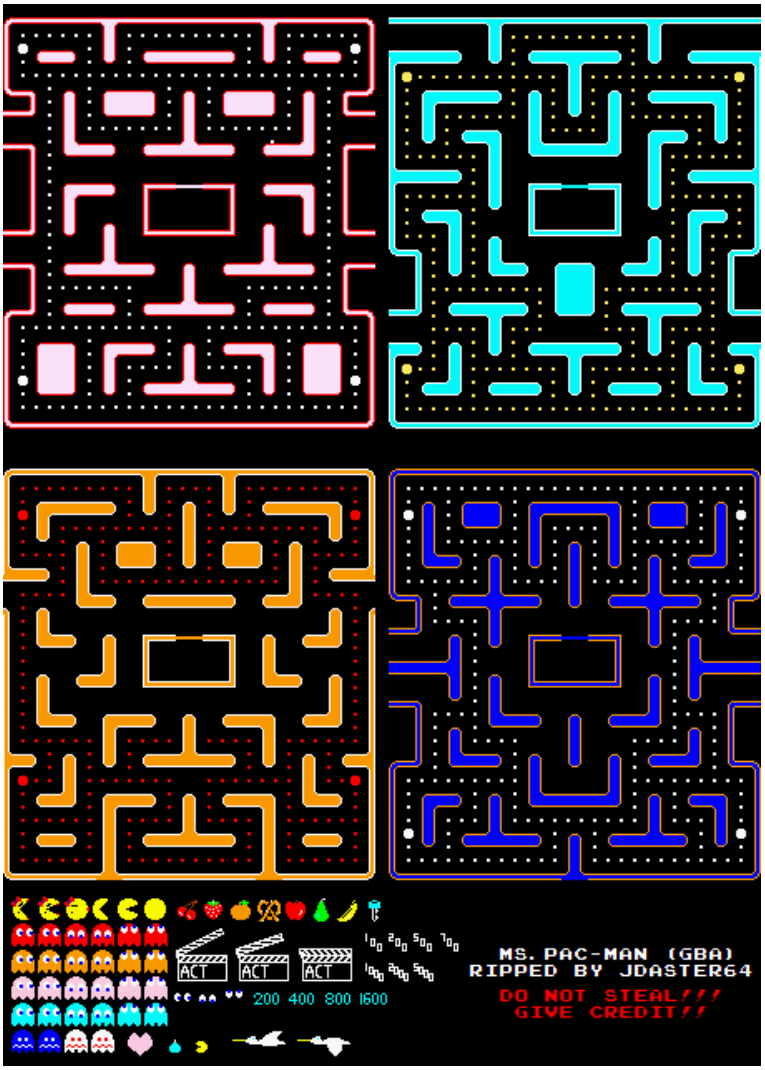
In this subsection, I will give some details about the steps I followed when developing my game.
Of course, I won't comment every line of code I wrote, but only the parts that are interesting.

4.2.1 World map

The first thing to think about is how will the world map be coded in Monogame.

The world map has its own class in my code.

I decided that I would use this sprite from The Spriters Resource :



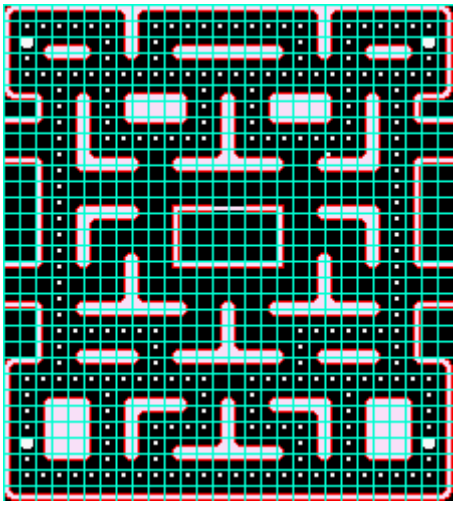
There are four world maps that I can use, but I only tried to implement two of them, because the steps to add one more map would exactly be the same as I did with the first two.

About the **dots** : when Pacman eats a dot, it has to disappear, so I just cannot use the map "as is" !

I had to remove the dots from the image, and then draw them myself within the code. We'll get back to that later on.

How to actually **code the world map** ? This Pacman game is totally compatible with the use of a grid : Pacman can move up, down, left and right in a limited amount of space.

I divided the world map in a grid in which each tile (square) is 8 pixels large. For example, for the first world map :



Then, I wrote a script in Node.js that would convert my world map with a grid to a matrix in C#. I used Node.js over C# because it is only a quick script that would help me code the world map, not the actual game, and I already used Node modules to analyze images, so I knew how to do it.
You can find this script in the annex.

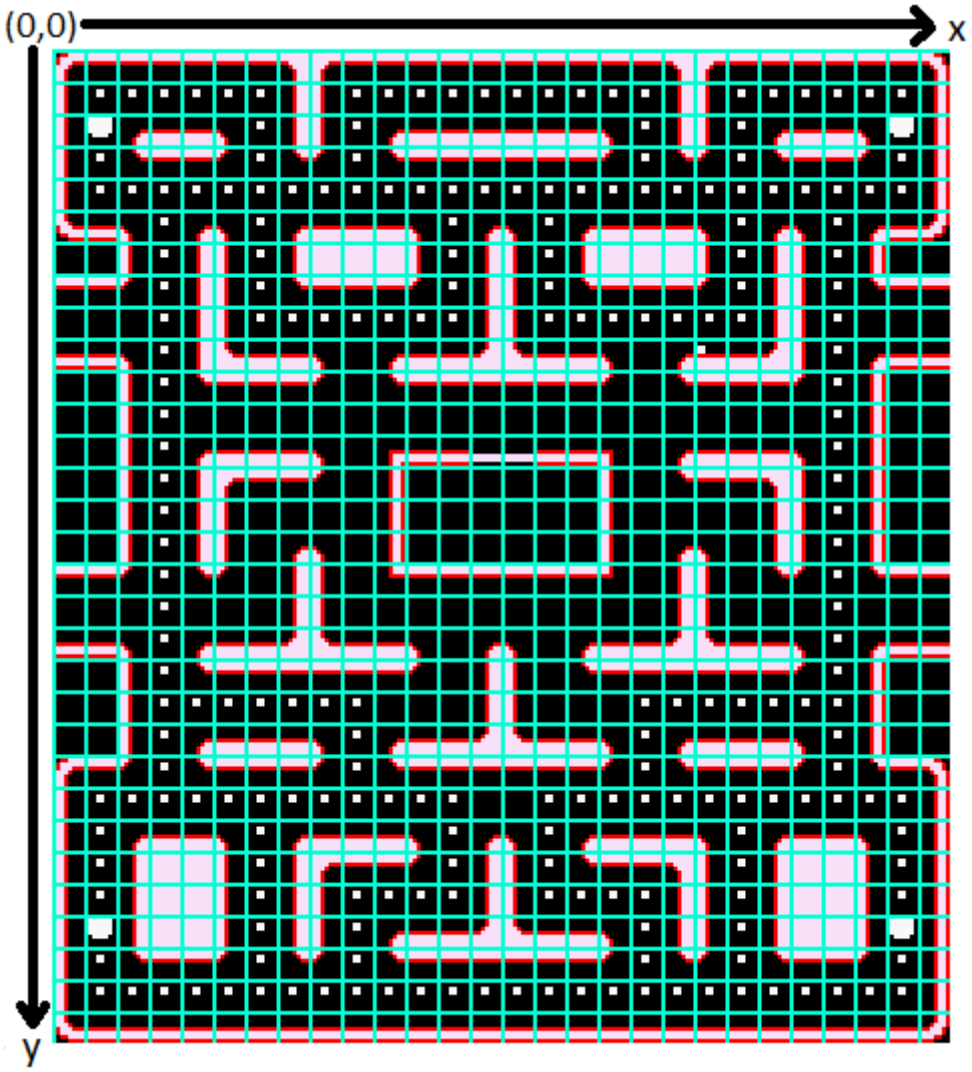
This script will check in every tile if there is a dot, checking the color at the appropriate pixel. Else, if at least one corner of the tile is the same color as walls, then it's a wall. Otherwise, the tile is empty.
This works almost perfectly : the wall from where the ghosts initially spawn are detected as "empty" by this script, so we have to be careful about these two tiles :

Each tile of the grid can either be empty, be a wall, or contain a dot :

```
// grid definition
public enum Tile
{
    EMPTY = 0,
    WALL = 1,
    POINT = 2
}
```

When there is a candy in a tile, I decided to consider and code it as a point.

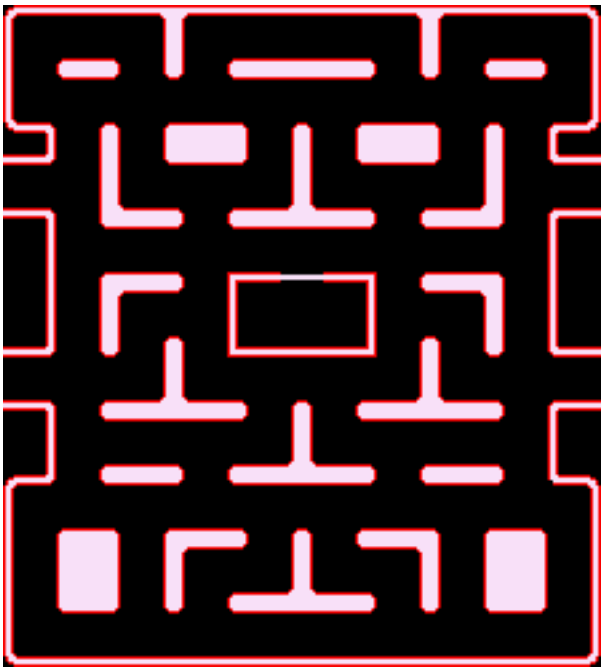
I had a bit of trouble accessing correctly my tiles in my code.
Indeed, a tile can be represented by its (x, y) coordinates :



The fact is I coded my grid as a matrix : I access the tile at (x, y) with : `matrix[y, x]`.
That's a tricky point, so to not be confused I'm using this method I wrote whenever accessing a tile (I wrote a similar method when setting a Tile) :

```
/// <summary>
/// Get the value of a tile
/// </summary>
/// <param name="x">the x-th tile from left to right</param>
/// <param name="y">the y-th tile from top to bottom</param>
/// <returns>The value of the tile : Tile.EMPTY, Tile.WALL, or Tile.POINT</returns>
public Tile GetTile(int x, int y)
{
    return matrix[y, x];
}
```

Finally, to draw the world map, we first draw the "empty image" of the map without the dots, that is to say an image like this one :



And then we check for every tile if it is a point, to draw it accordingly.

4.2.2 Player : Pacman

To begin with, Pacman and the ghosts will share lots of properties and methods. I chose to use inheritance of a `Living` class, even if we did not learn this principle during the course. I'm using it with very basic notions, so nothing hard here since I already learned inheritance in Java and C++ at ENSIIE.

Move Pacman

My first goal was to be able to move Pacman anywhere in the screen where there are no walls, and of course clamped inside the game window.

Since my game is based on a grid, I chose to put Pacman in a tile at the beginning of the game, and then he would only be able to go to the 4 nearby tiles (up, down, left, right).

At first, whenever the player pressed a key to move to another tile, I just changed the Pacman's location, but of course it was very abrupt and not enjoyable.

To make the movement smoother, I decided that when a key to move is pressed, I'd rather set the tile we want to move to as a target for Pacman, and then Pacman will move smoothly toward that target.
Of course, the player is unable to change Pacman's direction until he has reached its target.
This way, it's way more smooth. We are also able to change in the code Pacman's speed by deciding how often he will update its position, thanks to a timer.

I implemented this inside two methods in the `Pacman` class, `Move` and `UpdatePosition`

```
public void Move(GameTime gameTime, KeyboardState keyboard, World world)
{
    // if not moving yet
    if (!targetSet)
    {
        if (keyboard.IsKeyDown(Keys.Z))
        {
            // we want to move up
            direction = PossibleDirections.UP;

            // if we can move up
            if (tileY > 0 &&
            world.GetTile(tileX, tileY - 1) != World.Tile.WALL)
            {
                SetTarget(world, tileX, tileY - 1);
            }
        }

        else if (keyboard.IsKeyDown(Keys.Q))...

        else if (keyboard.IsKeyDown(Keys.S))...

        else if (keyboard.IsKeyDown(Keys.D))...
    }
}
```

```
public void UpdatePosition(GameTime gameTime, World world)
{
    // update timer
    msSinceLastUpdate += gameTime.ElapsedGameTime.Milliseconds;

    if (msSinceLastUpdate >= msBetweenTwoUpdates)
    {
        // reset timer
        msSinceLastUpdate = 0;

        if (targetSet)
        {
            // update position
            position.X += (int)velocity.X;
            position.Y += (int)velocity.Y;

            // update collision rectangle
            UpdateCollisionRectanglePosition();

            // if we finished to move
            if (world.GetTileCenterX(tileXTarget, tileYTarget) == getCenterFrameX() &&
                world.GetTileCenterY(tileXTarget, tileYTarget) == getCenterFrameY())
            {
                ClearTarget();

                // we reached the target
                tileX = tileXTarget;
                tileY = tileYTarget;

                // check if it was a point case
                if (world.GetTile(tileX, tileY) == World.Tile.POINT)
                {
                    score += 1;
                    world.SetTile(tileX, tileY, World.Tile.EMPTY);
                    world.NumberPointsLeft -= 1;
                }
            }
        }
    }
}
```

As you can see in the `UpdatePosition` method above, I also check if the tile we arrived in contains a point/dot : if so, we increment the player's score by 1, and we set the tile to "empty".

Sprite animations

Now that Pacman can move all around the screen, I had to make him change sprites.

In the sprite I downloaded, there are 3 images for Pacman, and 3 images for Ms. Pacman. I chose to use the images of Ms. Pacman.

Basically, when moving in the same direction, Pacman will alternate between 3 images. To that extent, I created a `png` file with 12 square of length 14x14 pixels with Pacman's stances, and then we just have to pick the appropriate square/stance in this file when we want to :

We change the animation image only when the player wants to move (i.e. is pressing a key) or when Pacman is moving (having a target set).

Also, we won't change Pacman's image every frame of the game, it would be too fast. I use a timer to control the time we change the frame, and I put it by default to 100 ms.

```
public void UpdateFrame(GameTime gameTime, KeyboardState keyboard)
{
    // update time
    msOnScreen += gameTime.ElapsedGameTime.Milliseconds;

    // update only if the player wants to move
    bool wantsToMove = keyboard.IsKeyDown(Keys.Z) ||
        keyboard.IsKeyDown(Keys.Q) ||
        keyboard.IsKeyDown(Keys.S) ||
        keyboard.IsKeyDown(Keys.D);

    if (targetSet || wantsToMove)
    {
        if (msOnScreen > frameTime)
        {
            // reset timer
            msOnScreen = 0;

            switch (direction)
            {
                case PossibleDirections.UP:
                    if (frameIndex == FramesIndexPlayer.UP_1)
                    {
                        frameIndex = FramesIndexPlayer.UP_2;
                    }
                    else if (frameIndex == FramesIndexPlayer.UP_2)
                        frameIndex = FramesIndexPlayer.UP_3;
                    else
                        frameIndex = FramesIndexPlayer.UP_1;
                    break;

                case PossibleDirections.LEFT:
                    ...

                case PossibleDirections.DOWN:
                    ...

                case PossibleDirections.RIGHT:
                    ...
            }
            // update source rectangle
            sourceRectangle.X = (int)frameIndex * frameWidth;
        }
    }
}
```

Now, when we want to update Pacman's position and image in the `Game1` class, we just call them in this order :

```
// move player
player.Move(gameTime, keyboard, world);
player.UpdateFrame(gameTime, keyboard);
player.UpdatePosition(gameTime, world);
```

Player's score and lives

Pacman's score is just an int. The player starts with a score of 0 and earn one point each time they eat a dot.

Likewise, pacman's lives are just an int, and represent the "lives in reserve". It means that if I give you 3 chances to beat the game, you will have 2 lives in reserve.
When you have died once, you'll have 1 live in reserve, and when you have died twice you have 0 live in reserve, but you are still alive. When you have no more lives in reserve and you die, it's game over !

We display those 2 data at the bottom of the screen, so our window's height is now equal to the height of our world map image plus some pixels to display data.

We draw those data in the `Draw` method in `Game1` :

```
// score at the bottom
spriteBatch.DrawString(font, "Score : " + player.Score,
    scoreTextLocation, Color.White);

// player lives in reserve
spriteBatch.DrawString(font, "Lives : ", lifeTextLocation, Color.White);
for (int i = 0; i < player.NumberLivesInReserve; i++)
    spriteBatch.Draw(lifeSprite, new Rectangle(
            (int)lifeTextLocation.X + 52 + (i * 14),
            261,
            lifeSprite.Width,
            lifeSprite.Height),
        Color.White
        );
```

4.2.3 Ghosts

In my game, ghosts cannot die (cannot be eaten by Pacman), so it is pretty simple to initialize 4 of them once and for all at the beginning of the game.

Move Ghosts

Ghosts movement mechanics is almost exactly the same as Pacman's.
The only difference is that they won't move based on the user's input, but they will make they own way through the world map.

In the original Pac-Man game, they have different behavior like turning around or chasing Pacman, but I didn't do an IA this type in my game.

So, whenever ghosts arrive on a tile, they will choose a destination at random among those possible, barring the tile from which they are coming from.
The only exception to this is when a ghost is blocked in a dead end and needs to do a U-turn.

To do so, we create a list of the possible new directions the ghost can go to.
If there are at least 2 directions (junction), we pick one at random. Else if there is only one direction we choose that one (corner or straight line). Else there is no new direction, so that's when we need to do a U-turn.

Sprite animations

We only have 2 different images provided by the source sprite for our ghosts, unlike Pacman which had 3 images.

Except from that, the animation follows the same logic.

4.2.4 Collision between Pacman and ghosts

Since we learned collisions using rectangles in the MOOC, I quite naturally decided to use Collision Rectangles for the living entities.

The collision rectangle is exactly the same size as the living's sprite size, and is updated whenever the living entity moves.

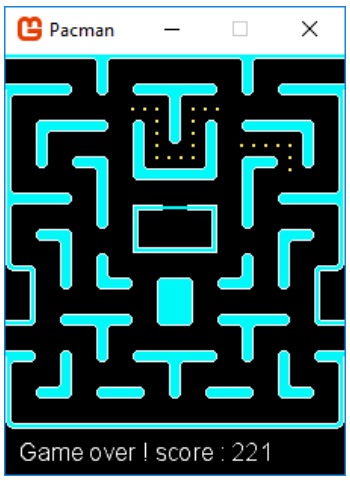
On every update of the main game loop (every 1/60 seconds), we check if Pacman's collision rectangle intersects with (at least) one of the ghost's collision rectangle.

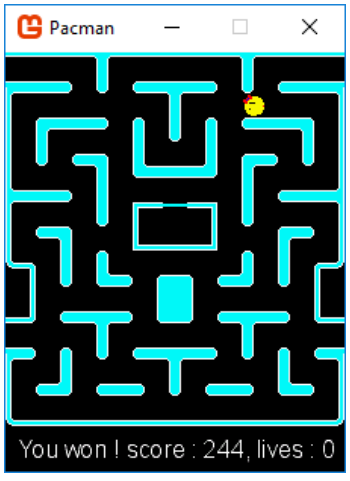
When the player hits a ghost, he looses one life and all the living entities (Pacman and the 4 ghosts) get back to their original position.

4.2.5 End of the game

The game ends whenever the player has used all of their available lives, or when the player succeeded to collect all the dots in the map.

In these cases, we draw the final score and an appropriate message ("You won/lost").





4.2.6 Music

I added 3 sounds effect : a sound that will play when Pacman is moving, a sound when it dies, and a background looped music.

So that there are not tons of sound whenever Pacman moves by a pixel, I only played the sound when it moves whenever I changed the animation (which is limited with a timer as previously explained).

When the game ends, I stop the background music.

Moreover, when starting the game, I added 3 seconds where everything is frozen, so that the player has got enough time to put his fingers onto his keyboard. During these 3 seconds, the background music won't be played.
Also, these 3 seconds where no music is played and everything is frozen is also here when the player dies (and everybody is teleported back to its original location).

5 Conclusion

To conclude, I would say that despite this MOOC being pretty easy to pass, I learned some concepts of developing a video game.

After finishing the MOOC, we are able to develop a simple game on our own, like my simplified Pacman game.

I would recommend other students at ENSIIE to take this MOOC, even if it is easy, because on the one hand we wrote a lot of C#, and on the other hand we were introduced to game development.
At ENSIIE, students who are not sure whether they want to ask for JIN or not, precisely because they have no idea if they would like to develop games, can definitively benefit from this MOOC.