SIT771 Object-Oriented Development

Credit Task 2.3: A Player Class

Focus

Make the most of this task by focusing on the following:

- Concept:

  Focus on exploring the ideas of constructors.

- Process:

  Consolidate your requirement analysis skills alongside strengthening your knowledge of how to create and build custom high-level programs using OOP concepts.

Overview

This is the start of a series of tasks in which you will develop a small program. These tasks are designed to help you explore the concepts being covered and to practice your programming skills.

The material in Course 1, Week 2 will help you with this task.

For this task, you will write the start of this program and implement one of its core classes.

Submission Details

Submit the following files to OnTrack.

- The Program and Player code (*Program.cs* and *Player.cs*)
- A screenshot of your program running.

You want to focus on the use of classes and objects within the program, and the ability to capture knowledge and behavior within the program's objects.

Instructions

The Robot Dodge game will have a player, represented by a bitmap on the screen, that can move around the screen and dodge incoming robots. The idea is to survive as long as you can. An example of this game running is shown below.

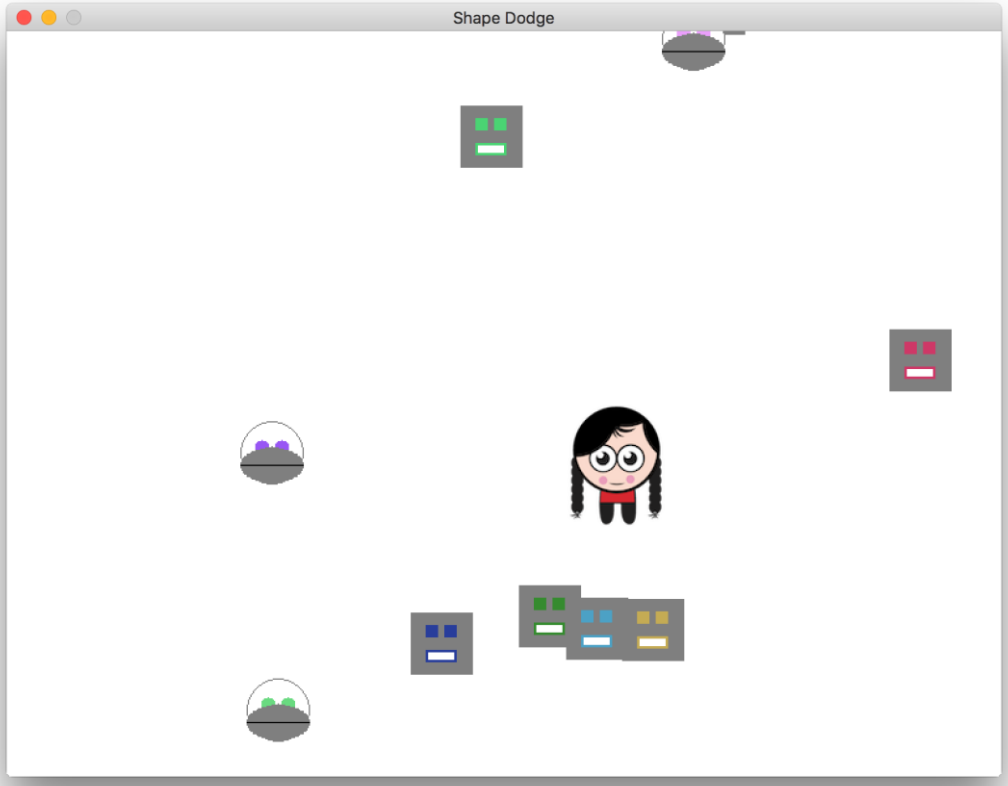

Figure: End game for Robot Dodge

When runs, the user will control the player. They can use the arrow keys to move the player around on the screen and must dodge the robots that appear and move toward them from off the screen. Each time the player is hit by a robot, the robot will disappear (with a bang), and the player will lose one life. The player will start with three lives, so the game is over when they have been hit by three robots.

To get this started we will build the code for the Player and give them the knowledge they need to be drawn to the screen.

1. Use skm and dotnet to start a new project for **RobotDodge**.
2. Download the resources for the game and extract (unzip) into the project's folder.

Robot Dodge First Design

Before we can start writing the code we need to understand what we are trying to build (analyse the requirements) and then come up with a plan to get something working quickly.

Read the description of the game, and study the screen shot. What kinds of objects do you think we will need?

I have a design in mind, but have a quick think before progressing to the next page.

`

Ok, hopefully you had some good ideas. The way I see this, we are going to need a **RobotDodge** game object, a **Player**, and a number of **Robot** objects, in addition to the **Program** which will house `Main`. These can be our candidate classes, and we can flesh out this design as we go.

When building software, you want to work in small steps. So rather than trying to get *all* of this working in one go let's focus on getting part of the player working.

Here is the UML Class diagram that shows what we need to build in this iteration (another name for step). Each of the rectangles in this diagram represent a **class** that you will need to create: so you will create a **Program** class and a **Robot** class. Within each of these boxes we describe the fields, properties, and methods that will be coded in these classes.

- Within the *Program* class you will need to code up a single **Main** method. This class will be responsible for coordinating the main steps of the program, such as creating the window and player and getting them drawing.
- Within the *Player* class you will need to add a **_PlayerBitmap** field, **X** and **Y** auto properties, **Width** and **Height** properties, as well as a **constructor** and a **Draw** method.

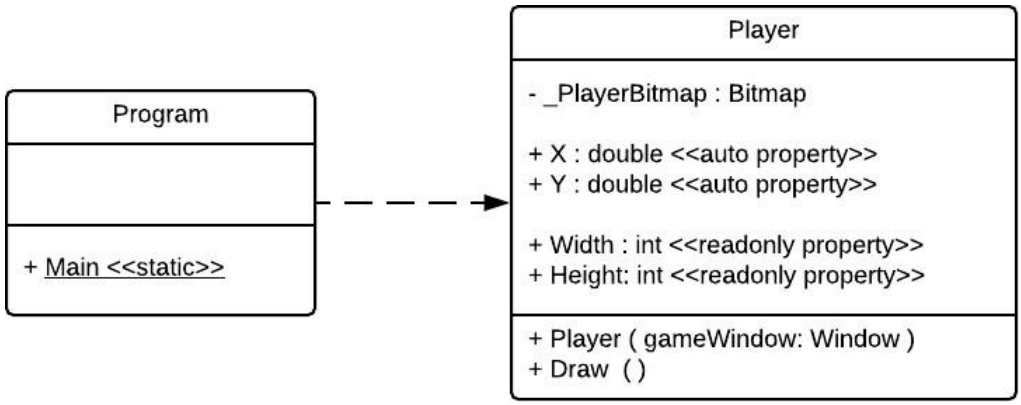

Figure: Iteration 1, showing the Program and Robot classes

The following instructions will guide you in creating these classes.

1. In the project, create a new file called **Player.cs**.

   We will use this file to contain all of the code for the Player class. For anything beyond a small program, it is best to separate the code into multiple files.

2. Add the code to use the `SplashKitSDK` namespace, and create a new `Player` class.
3. Within the player add the one `private` field for the `_PlayerBitmap`.

4. Add the auto properties for `X` and `Y`.

   An **auto property** is a shortcut provided by the C# compiler. This created a private field for you, along with a getter to read the field and a setter to writes to the field.

   You can optionally specify a scope on the get and set parts, allowing you to have a public get and a private set.

   ```
   public double X { get; private set; }
   ```

   We can use this to create the X and Y fields and properties for the Player.

5. Add in a read only `Width` property that you calculate from the width of the `_PlayerBitmap`. We can then use this in calculating where to position the player, and it will be available for others to check where the player is as well.

   ```
   public int Width
   {
       get
       {
           return _PlayerBitmap.Width;
       }
   }
   ```

6. Now add the Player's constructor. This will need to do three things: create a new Bitmap, set the Player X, and set the Player Y.

   The supplied resources include a file called "Player.png". Use this as the image for the player, and store a reference to it in the `_PlayerBitmap` field. Remember when creating the Bitmap you need to pass in a name for SplashKit to use, as well as the filename. You can check out examples in your scene drawing task from week 1.

   The constructor is passed a Window object. Use the passed in object to position the player in the centre of the screen. You can ask the `Window` object for its Width and Height. For example, we can set the X location of the Player using:

   ```
   X = (gameWindow.Width - Width) / 2;
   ```

7. Next, add the `Draw` method. In this you can tell the Bitmap object referred to by `_PlayerBitmap` to `Draw` and pass in the Player's `X` and `Y` values. This will draw the player bitmap on the current window.

8. Switch to your **Program.cs**. We can now make use of the new Player class. Add code to create a Window and a Player. Have the program perform the following steps:

   1. Clear the Window
   2. Draw the Player
   3. Refresh the Window (pass in 60 as the target framerate)

9. Switch to the Terminal, and compile and run your program.

   You should see your player appear in the centre of the screen.

   If they are slightly off, you need to adjust your calculation of the player's X and Y location. You need to take into consideration both the size of the Window and the size of the Bitmap. You can get the `Bitmap` width using `_PlayerBitmap.Width` (its a property of the Bitmap).

   Consider a case where you have a Window with a Width of 100, and a Bitmap with the Width of 10. In this case, the X value for the player should be 45. This keeps the center of the bitmap at the center of the screen.

Ensure that everything is centered correctly, then backup your work and submit the required material to OnTrack.