SIT771 Object-Oriented Development

Credit Task 5.4: Many Robots

Focus

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

- Process

  Focus on crafting high-level code that employs diverse C# concepts and techniques, tailoring your approach to address distinct requirements and challenges presented throughout the task. Additionally, emphasize optimizing code efficiency as you explore methods to effectively handle numerous objects without introducing unnecessary code complexity.

Overview

This is the fourth 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 3, Week 1 will help you with this task.

In this task, you will make use of the `List` class so that your game can manage a number of `Robot` objects. This will require changes in the game so that Robots periodically spawn off-screen and move toward the player. Robots will then be removed from the game if they collide with the player, or they go off the screen.

Submission Details

Submit the following files to OnTrack.

- The program's code (*Program.cs*, *Player.cs*, *Robot.cs*, and *RobotDodge.cs*)
- A screenshot of your program running

You want to focus on how you can make use of the `List` class together with loops to easily work with many objects without needing lots of code.

Instructions

Get started by opening up the project and getting everything ready to start working on making the required additions.

1. Return to your **Robot Dodge** game project. Open it in Visual Studio Code, and have a Terminal open with that folder as the current working directory.

2. Open your **RobotDodge.cs**, and **Robot.cs** files.

Here is the design for the end of this iteration.

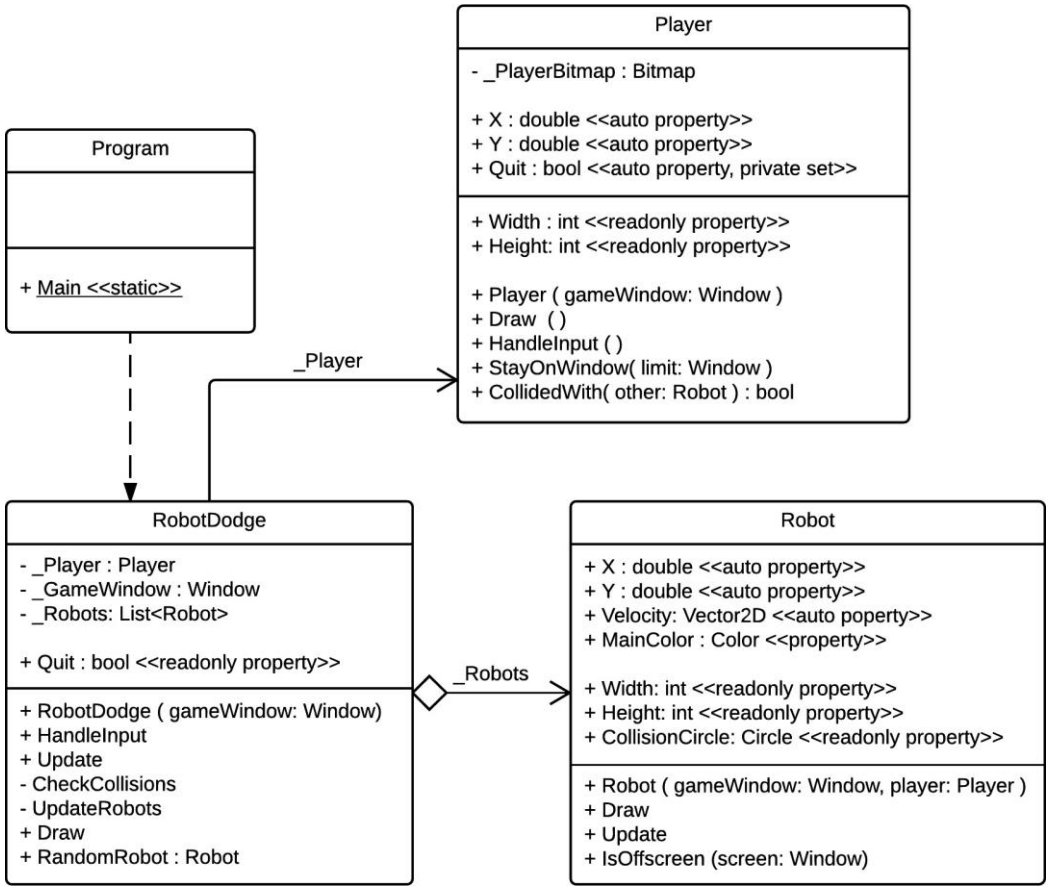

Figure: Robot Dodge iteration 4 design

Aiming for the _Player

To start with, lets get the one Robot we have starting off screen and moving toward the Player. Once we can do this with one object, doing it with many should just need us to add a list and a loop.

1. Switch to the **Robot.cs** file. Lets start by adding the code to have the Robot move toward the player.
2. Add a new private `Velocity` property that will store a `Vector2D` .

   A Vector2D can be though of as an arrow with both direction and magnitude (the length of the arrow). In this case, you can think of this as an arrow that point from the Robot in the direction it is heading. The length of the arrow then indicates the speed at which the robot is traveling.

   `Vector2D` comes from SplashKit.

   ```
   private Vector2D Velocity { get; set; }
   ```

3. We need to initialise the `Velocity` in the `Robot` constructor. Locate the constructor, and add the following code:

```
public Robot(Window gameWindow, Player player)
{
    // Lets test with starting at top left... for now
    X = 0;
    Y = 0;

    const int SPEED = 4;

    // Get a Point for the Robot
    Point2D fromPt = new Point2D()
    {
        X = X, Y = Y
    };

    // Get a Point for the Player
    Point2D toPt = new Point2D()
    {
        X = player.X, Y =  player.Y
    };

    // Calculate the direction to head.
    Vector2D dir;
    dir  =  SplashKit.UnitVector(SplashKit.VectorPointToPoint(fromPt,  toPt));

    // Set the speed and assign to the Velocity
    Velocity = SplashKit.VectorMultiply(dir, SPEED);
}
```

The comments in the above code indicate what we are doing in each step. This is using vector maths to calculate the vector that will store the direction and distance to travel.

Internally, the `Vector2D` is just storing an `X` and a `Y` offset. This represents the distance to travel each update.

4. To make use of the `Velocity`, we need to add a public `Update` method. This will then move the Robot by the amount in its `Velocity` property.

   To do this, add the value of `Velocity.X` to the Robot's `X`, and similarly with the `Velocity.Y` and the Robot's `Y` property.

   The Vector we created pointed to the Player, so the Robot will head toward the player. As we are not updating the Velocity, the Robot will not track the player if they move.

5. Now we need to switch to **RobotDodge.cs** to make use of these new Robot capabilities.

6. Locate the `RandomRobot` method and add an additional argument that passes in the `_Player`.

7. Now locate the `Update` method in RobotDodge. Have this call `Update` on the `_TestRobot`.

8. Build and run your program. The Robot should now appear at a random location on the screen

and move toward the player.

These kinds of edits represent the way in which we would work with object oriented programs. We add capabilitites to a class of object, then find its collaborators and change them to make use of these responsibilities.

Starting Off screen

Now lets get the Robot to start off screen.

1. Switch back to **Robot.cs** file and locate the constructor.
2. We can now make use of random numbers, together with some **if statements** to work out where to position the robot initially.

   Here is a start with this... code this in the constructor and add the missing logic of positioning the Robot just off the Left or Right of the screen.

   ```
   public Robot(Window gameWindow, Player player)
   {
       // Randomly pick... Top / Bottom or Left / Right
       if (SplashKit.Rnd() < 0.5)
       {
           // We picked... Top / Bottom

           // Start by picking a random position left to right (X)
           X = SplashKit.Rnd(gameWindow.Width);

           // Now work out if we are top or bottom?
           if (SplashKit.Rnd() < 0.5)
               Y = -Height; //Top... so above top
           else
               Y = gameWindow.Height; //Bottom... so below bottom
       }
       else
       {
           // We picked... Left / Right
           // ... add code here
       }

       // ... code to setup velocity continues from here
   }
   ```

3. Make sure you remove the old code that was setting the X and Y to a random screen point.

4. Build, run, and test your program. Make sure that the robot starts off screen and respawns when it hits the player.

   Don't move the player, and make sure you can see the Robot coming from each side.

   Then try dodging the robot... what happens when it goes off the other side of the window?

5. Lets fix the lost robot problem by adding a responsibility for the Robot to know if it is off screen.

Add a public `IsOffscreen` method to the Robot. This will accept the Window (`screen`) to check against, and will return a boolean: true when it is off screen, false when it is on the screen.

The Robot is offscreen when **any** of the following conditions are true:

- `X < -Width`
- `X > screen.Width`
- `Y < -Height`
- `Y > screen.Height`

> **Note:**
>
> See if you can code this without add any branching (**if** or **switch**) statements. You should be able to code this with a single **return** statement.

6. Switch to **RobotDodge.cs** and locate the `Update` method.
7. Change the if statement so that the robot also respawns if it is offscreen.
8. Build and run your program, test that you a new robot comes after the player when the test robot goes off the screen, as well as making sure that it still respawns when it hits the player.

Many Robots

We now have all of the required Robot capabilities, so we should now be able to quickly make this work with lots of robots.

1. Switch to **RobotDodge.cs**.
2. Remove the `_TestRobot` field.
3. Add the private `_Robots` field. This will store a `List` of `Robots`. Make sure that you initialise this to a new list, either in the declaration or in the constructor.
4. Remove the code from the constructor that initialised the `_TestRobot` field.
5. In `Update`, comment out the if statement and code that respawned the `_TestRobot` when it hit the player or went offscreen.

> **Note:**
>
> To *comment out code* means to convert that code to comments so it is ignored by the compiler.
>
> In VS Code, you can quickly comment out code by selecting it and hitting **crtl+/** or **cmd+/** on a Mac.

6. Now, lets change the code that asked `_TestRobot` to `Update`.

We now what to tell **all robots** in the list to `Update` themselves. Unfortunately, the computer can't do this. Remember that any time you want to do something to **all** objects/values in a list, you

need to rethink this in terms of what you want to do **for each** item in the list.

In this case we want to call `Update` on **each Robot** in the List.

Add the code to do this in Update, making sure you remove the code that was calling `Update` on `_TestRobot`.

7. Now, we also need to add some code to randomly create and add Robots to the game.

   In `Update`, add an **if statement** that randomly adds a `RandomRobot` into the `_Robots` list.

8. Finally, move to the `Draw` method and change it to draw **all** robots from the list.

   This should have removed all references to `_TestRobot` from the `RobotDodge` class.

9. Build and run your program.

   Play around with the probability that robots get spawned.

At this point you should be able to see lots of Robots moving on the screen. But they do not hit the player, and what happens when they go off the screen? If you keep running this for a while, what happens?

Removing Robots

The final step for this task is now to get the collisions working again, and to remove robots when they go off the screen.

One challenge with this task is that you cannot change a `List` while you are looping over it with a `foreach` loop. To get around this we can use a second list. When you loop through the robots, you can check which Robots we want to remove, and add them to this second list. Then in a separate loop, after looping through all Robots, we can loop over all of the Robots in the second list, and ask the first list to Remove each of those Robots.

1. Create a private `CheckCollisions` method in the `RobotDodge` class.

   This method will need to do the following things:

   - Create a new List, with the robots that need to be removed
   - Loop over all `_Robots`
     - If the current robot needs to be removed:
       - Add it to the new List
   - Loop over all Robots in the new List
     - Tell `_Robots` to `Remove` the current Robot

   Use the commented out logic from `Update` to determine if a Robot needs to be removed.

`CheckCollisions`

2. Switch back to `Update`, call , and remove the commented out code.

3. Build and run your program.

Once you have this working, save your code, back it up, and submit it to OnTrack.

Reflect on the process used to iteratively build this program, as well as think about how we made use of the List to easily work with many Robots.