Documentation for the semester project

Programming 2

“MINI ACRADE GAME (INSPIRED BY HUNGRY HORACE)”

Ivan Turko,

13.07.22

1. ABOUT PROJECT (SHORTLY):   
The project is a mini arcade game (inspired by Hungry Horace and other so called ‘8-bit’ games from 90’s) represented by a Windows Forms application written with C# programming language. The application has a user interface accessible with a mouse, the game itself is played with the keyboard keys.

2. MORE DETAILED:

\* From the user’s perspective the task was to write a playable 8-bit styled arcade game inspired by Hungry Horace, as it was mentioned in the short description. There should have been different objects moving at the same time (e.g. ghosts and Horace), collectable “treasure” (coins), different levels, etc.

\* From the programmer’s perspective there have been two main goals:

1. To create a desktop application based on .NET’s Windows Forms using C# as a programming language; the application should have included game graphics and user interface.

2. To implement the main algorithm (idea) the game is based on.

3. ALGORITHM:

As the main algorithm, rather the main idea, entire game logic is based on, there was chosen the following idea basically using three main components:

First of all, there’s an object, let’s call it Map, which is responsible for representing and handling game processes (changes, moves, states, etc.) including such operations as displaying the game as well. Second of all, each moving object is represented as a different object, thus having its own properties such as its current coordinates (other data if needed). Finally, there’s a Ticker with a particular given interval (i.e. main loop), which carries out some operations each iteration of itself (according to the interval).

All together it works the following way: each iteration of the main loop the Map “moves” (if possible) all of the moving objects by calling their let’s say “MakeMove” function and using user’s input (e.g. key press), changes other parameters of a game state if needed and after all changes are done, prints out (displays) itself to the screen. That’s how a game process generally looks like from the perspective of a program.

What’s also important, let’s take a closer look at the object’s “MakeMove” function. As it was mentioned earlier, the entire game state is handled and represented by the Map. That means that each moving object doesn’t “know” anything except for its own coordinates.

For example: The Horace wants to make a move. Initially we know that from a certain point he can go only up, down, right and left. From the perspective of a Horace he can move to another cell only if it’s empty, he also definitely wants to avoid ghosts and so on. The Horace himself doesn’t know anything about the game state except for his own coordinates. So the only thing he can do is to ask the Map to check all he needs and then give a simple answer Yes or No (e.g. “was it empty?”) or (and) take some action (e.g. the Horace asks if the cell the gamer is about to move him to contains a coin, and if yes it also wants to eat it, so that asking the Map to do that as well. Eat a coin here means to remove it from a game plan, change the number of coins left, etc.). In the language of a program it means that the Map has many different bool functions which are responsible for detecting different objects, their states and so on.

4. PROGRAM:

Talking about the program itself and the ideas that were implemented, I’d like to mention few moments here:

* The application itself is a Windows Forms application created with common tools (e.g. buttons, labels) and corresponding functions (e.g. button\_click). This all stands for GUI.
* There are few examples of decomposing in the program. The first is there’re two .cs files: Game.cs with the “Engine” or logic of the game (Horace, Ghost and Map classes) and Form1.cs with all of the WindowsForms functions and properties. Except for initialization, Form1.cs itself consists of three main blocks: KEY INPUT HANDLING, RUNTIME and INTERFACE functions and properties.
  + KEY INPUT HANDLING: User’s input, i.e. pressed keys are detected by a

ProcessCmdKey() function and the pressed key is kept in a PressedKey (type) pressedkey variable (enum PressedKey {up, down, right, left, none}).

P.S. PressedKey.none state is not used in the game (I found the game more dynamic without it), though it can be turned on by switching the Form’s property KeyPreview to true.

* + RUNTIME: Main loop Ticker, two ghost’s states tickers (more later), StartNewLevel() function. Every time StartNewLevel() is called few important actions are carried out: a new instance of the map class (more simple: a new map) is created (reads a particular level from newplan.txt file according to a current level number), all needed runtime tickers are enabled.
  + INTERFACE: Functions which are responsible for (behind the scenes) interaction with the interface.
* Graphics (printing the game to the screen) are presented by System.Drawing namespace, also using bitmap images introduced from the images.png file. Map’s PrintOut function basically reads the character from a plan and displays a corresponding image to a certain position on the display according to the char’s position in the plan array.
* Game plan (location of objects, game map) is introduced as a text file, and then converted (while initializing a new map) into a two dimensional array of characters [width, height] (metadata like width, height, level number are also read from the file).
* The game itself is coordinated by different states it can be in inside of the main ticker (runtime) function (running, eaten, lost and so on).
* As it was mentioned many times earlier the game is presented a two dimensional character array. Mostly each character stand for a different object (empty cell, wall, coin, pill). The moving objects like ghosts and the Horace are handled and displayed as 2x2 cells objects. The “main cell” of this square is the up left corner (with a big char, e.g. ‘H’), the rest are the small once (e.g. ‘h’) which are not printed. All of the Map’s functions detecting objects, moving them and so on are written accordingly.

5. GHOSTS (ALORITHM):

One of the most important part of the game is ghosts, or more precisely their moves and states. Ghosts’ actions are the main barrier on the Horace’s path to pass the level and win the game, so that they have to be balanced to make the game neither extremely easy, nor as hard as unplayable. But at first, few words about how a ghost decides where to move to. The algorithm is inspired by the one used in PAC-MAN. Generally, the idea is that every time a ghost tries to make a move it checks all four directions: up, right, down, left (directions are prioritized according to the order, meaning that in case two directions are same “good”, a ghost will choose the more prioritized one). The “good” one here means the linearly closest to a particular target (and empty of course). The target is just a certain point on the map, which is chosen according to the ghost’s state. There are three main ghost’s states: CHILL, CHASE and FEAR. During the chill state, the target is such a point to make the ghost strand around and not to bother the Horace much: either the center of the map, or the doors. During the chase state the target is the Horace himself. To keep the balance I mentioned earlier, the chill and chase states are switched from one to another every N seconds according to the ChaseChillStates ticker’s interval (N is 5000ms in this version). After the Horace eats the red pill, the chase/chill still is disabled and the fear state is activated. During it ghosts chose the cell to move to randomly and the Horace can kill a ghost if he touches one. After the fear state is over (according to FearState ticker’s interval – 7000ms and 10000ms (3rd lvl)), it’s disabled, chase/chill is enabled and the dead ghosts are respawned (! there’s a bug (or a feature) with a respawn because of desync of kill ghost function and enabling fear state, if a player kills both ghosts within a very short period of time, they won’t respawn after so that the player can safely finish the level).

5.1 DISCUSSION (ALGORITHM):

When choosing an algorithm for ghosts, the first thing I tried to implement was simple BFS to find the shortest path from a ghost to the Horace, but the disadvantages where obvious very soon. Not only it was always chasing the Horace not giving him a chance to hide and think (as hard as unplayable), but it also had O (width\*height) time complexity each iteration which certainly loses comparing to the one which is used in the final version with O(1): each cell has it’s x and y coordinates and there’re only four directions and on target to find the linear distance to, which all are constant calculations.

6. USER PART:

Things like how to play the game, what are the available game modes and other information essential for any user can be found in the main menu so, as a user’s part of this documentation I’d just like to mention where it can be read.

7. WHAT ELSE COULD HAVE BEEN DONE:

One of the things I would have added are the animation of the objects during the game (for example two images per direction to make it look as it was moving) coordinated by an AnimationTicker.

I’d also add some other collectable objects and make more levels to make the game more entertaining.