VRIJE
UNIVERSITEIT
BRUSSEL

Project Computersystemen

RUBBERDUCKER

Group G03

Abdullah Sabaa Allil & Lotte Laureys

December 26, 2021

Academic year 2021-2022

Computerwetenschappen

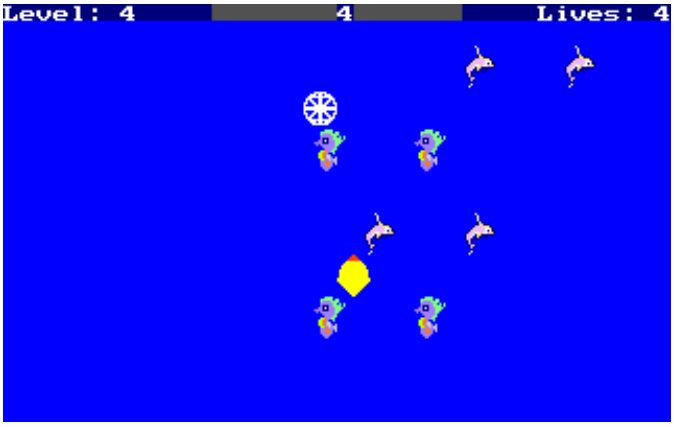

Figure 1: The game play of Rubberducker

1 Introduction

This document describes our project for Computersystemen. The project is a game called Rubberducker, inspired by Frogger. The main goal of the game is for the player to reach a high level. The player finishes a level once he successfully reached the top of the screen. When trying to do so, moving arrays of toys that can shoot bullets will try to prevent the player from reaching the top. The player has a limited time to finish each level.

2 Manual

To run the game, the player has to execute the executable *project*.

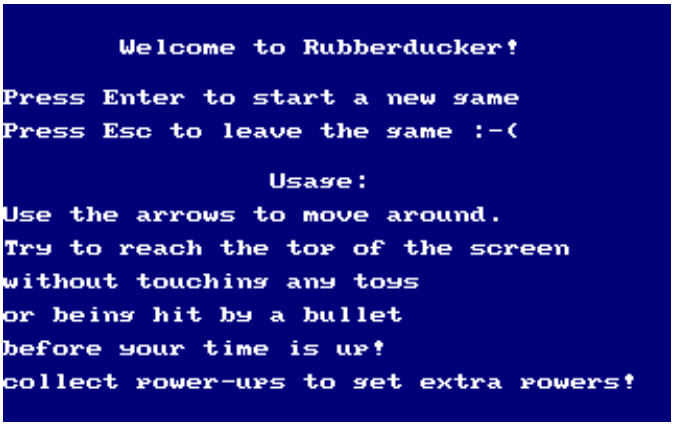

Figure 2: The start screen of Rubberducker

When starting the game, a manual is shown and the game can be started by clicking Enter. The game can be exited by clicking the escape button.
Starting at the bottom of the screen, the player can move around the screen by using the arrow keys. Moving outside the screen is not possible.
When hitting a toy or a bullet, or when the level time is up, the player loses a life and returns to the bottom of the screen.
The player finishes a level once he or she successfully reaches the top of the screen. A new level will then start and the player will return to the bottom of the screen.
The number of the successfully completed levels, as well as the number of the highest achieved level, are saved throughout the game.
The player has initially 5 lives. When the player has no more lives left, the game ends, and the player gets to see the number of the levels he or she successfully completed and whether he or

she achieved a new high level. At the end screen the player can also restart the game by pressing Enter or exit the game by pressing the Esc button.

Figure 3: The end screen of Rubberducker

3 Program features

Each new level gets harder than the previous one. This is achieved by increasing the amount and the speed of the enemies and the bullets. Each level has twice as much toys as the number of the current level. The speed gets increased each two levels.
There are multiple power-ups in the game that can help the player reach the top:

- freezing power-up: this power-up will freeze the timer for three levels long, making it easier for the player to reach the top at his own pace. When the freezing power-up is active, the timer bar will become red to indicate the timer is not running.

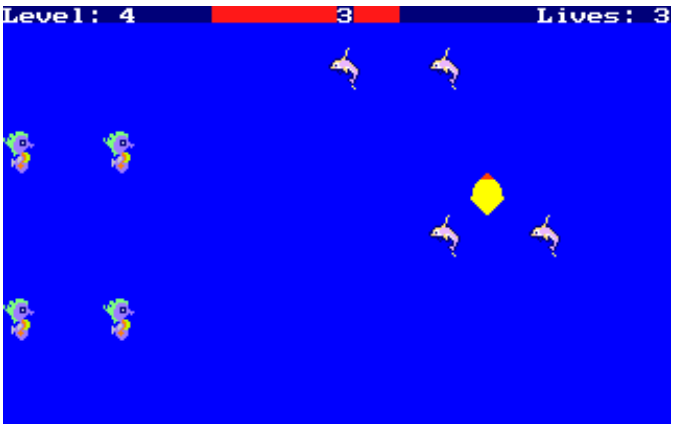

Figure 4: The freezing power-up is active, so the timer bar is red.

- health power-up: this power-up will give the player an extra life.

- shield power-up, a.k.a. the spy power-up: this power-up will help the player hide from the enemies and their bullets. When this power-up is active, the player will not lose any lives when hitting an enemy or one of its bullets. When this power-up is active, the duck will turn brown, as shown in figure 6, to indicate that it cannot be seen by the enemies.

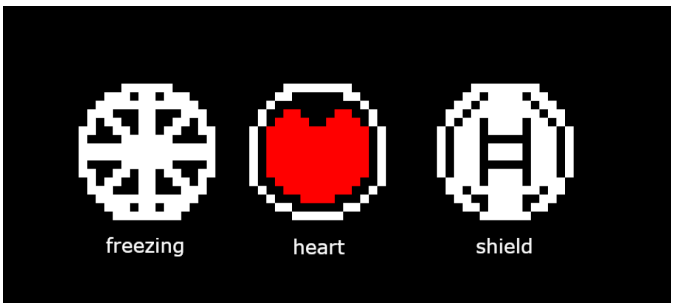

Figure 5: The power-ups icons.

When hitting a vertical side of the window, the enemy gets flipped and moves in the opposite direction towards the other side of the window.
There is a decreasing timer that is represented at the top of the screen by a number and a bar that shows the countdown.
The player is represented by a cute little duck that can be moved around the window using the arrow keys as described in the previous section.
There are multiple sprites in the game:

- a duck, which is controlled by the player. The duck can be either yellow, which is its normal state, or brown, to indicate that the shield power-up is active;

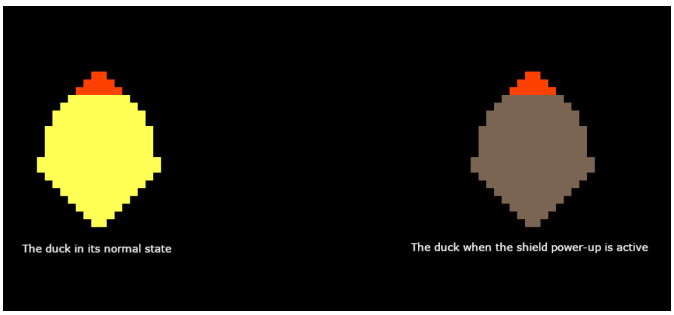

Figure 6: The sprites of the duck.

- a dolphin, which is an enemy;
- a seahorse, which is also an enemy;
- a bullet, which can be shot by the enemies;
- the different sprites of the power-ups.

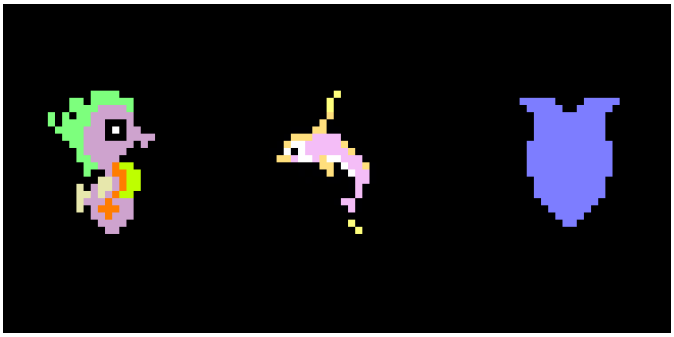

Figure 7: The sprites of the different enemies.

4 Program design

The program is segmented into different files:

4.1 gfx.asm

The *gfx.asm* module contains the procedures that are related to drawing different elements on the screen and the graphical interface of the game as well as the game menus. These procedures are responsible for updating the frame buffer when its contents are not being used (cfr. VBI), drawing things on the screen such as sprites, rectangles, strings , and numbers, changing the color palette, and drawing the start and the end screen. The sprites are assumed to have fixed dimensions of 16x20.

4.2 project.asm

The *project.asm* is the main module of the program. It contains all the procedures that are related related to the game logic, including the *main* procedure that has the game loop. The procedures in this module are responsible for starting and closing the game, loading the sprites into the memory, moving the game elements, the interaction between the different game elements, updating the various game variables, drawing the game and listening to input using the other modules described in this section.
The *main* procedure will change the video mode to *13h*, and update the colour palette so the colour represented by the black colour's index in the default colour palette become dark blue. This is done to make the background of texts and the start and end screens match the game theme. This procedure will also load all the binfiles of the sprites to the memory, and initialize the keyboardhandler. There are 3 'infinite' loops in the *main* procedure:

- the *start_menu_loop*, which is the loop of the start menu. The player will stay in this loop until Enter (to enter the game) or Esc (to leave the program) is pressed. The game usage is printed in this menu;

- the *game_loop*, which is the main loop of the game. This loop is responsible for drawing and updating the game elements according to the given input and/or the interactions between the game elements (e.g. a bullet hit the player). Before the *game_loop* starts, the different game values, such as the timer, the level counter, the lives counter, etc.. get initialized/reset. In the *game_loop* itself, the *drawGame* is responsible for drawing the

game elements, the *handleInput* procedure is responsible for listening and processing the input (the input that's relevant to the game play as well as the general input, e.g. pressing the Esc key for leaving the game), and the *updateGameStatus* is responsible for updating the game values such as the timer, activating powerups when they are collected, collision detection between the game elements, etc;

- the *end_menu_loop*, which is the loop of the end menu. The player will stay in this loop until Enter (to start a new game) or Esc (to leave the program) is pressed. In this menu, the player will get to see the last level he achieved as well as the high score and a congratulations string in case he achieved a new high score.

The game elements, such as the duck that is controlled by the player and the various enemy toys, are stored in strucs containing the position of the element, the sprite of the element, and other variables needed for the element to function properly.
The bullets and the enemies are represented by the same struc, and they are also stored in the same array. Each enemy struc saves the direction of the enemy, e.g. whether the enemy toy is moving left or right (which is the case for enemy toys) or down (which is the case for bullets). It also saves whether the enemy is valid or not, e.g. whether the enemy is currently active in the game and should be interacted with or not. This helps removing the bullets that hit the player or went out of the screen, but it makes it also easier to expand the game further with other features, such as a torpedo power-up that can hit the enemies. Sadly, we didn't have enough time to implement this power-up.
The *moveSprite* procedure is responsible for moving game elements. It is used to move the enemies as well as the player.
The *checkForCollision* procedure is responsible for checking whether two positions hit each other or not. The implemented algorithm in this procedure, which is called *rectangle rectangle collision detection*, checks whether there is a collision between two rectangles with the same dimensions as the sprites in the game. A collision between two objects happens when

$$x_{1st\ object} - width_{2nd\ object} < x_{2nd\ object} < x_{1st\ object} + width_{1st\ object}$$

and

$$y_{1st\ object} - height_{2nd\ object} < y_{2nd\ object} < y_{1st\ object} + height_{1st\ object}$$

In other words: when the x value of the first rectangle, which represents the first object, lies within the left and right edges of the second rectangle, representing the second object, and when the y value of the first rectangle lies within the top and bottom edge of the second rectangle, then a collision between the two objects has occurred.
This procedure is used for checking whether the player and an enemy (toys as well as bullets) hit each other and whether the player collected a power-up. This procedure is also used to check whether the game elements are within the screen borders.

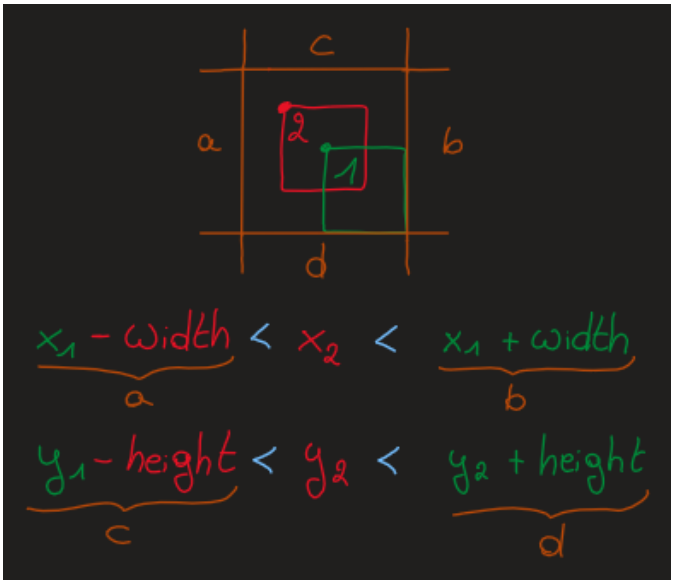

Figure 8: Example of collision detection

4.3 genl.asm

This module contains general procedures that are responsible for changing the video mode, terminating the program and calculating the length of a '$' terminated string.

4.4 fileh.asm

This module contains the procedures and variables that are responsible for opening a file, loading the file content into an uninitialized memory chunk and closing the file.

4.5 keyb.asm

This module contains the procedures that are responsible for listening to the keyboard input.

4.6 cons.asm

contains the constants that are used throughout the program.

4.7 rand.asm

contains the procedures that are responsible for generating random integers.

5 Encountered problems

- We didn't know how to get rid of redundant codes. We managed to implement general procedures (e.g. *moveSprite* and *checkForCollision*) to get rid of redundant code;

- We didn't know how to represent the various game elements. Luckily, strucs were introduced during the exercise sessions;
- We didn't know how to store the enemies and the bullets. During the feedback session, we were told to use an array of strucs, which resolved the issue;
- We didn't know how to make binary files to store the sprites. During the exercise sessions, a program called HxD was introduced;
- Our games was a little bit minimalist. We increased the complexity of the game by adding power-ups and making it possible for the enemies to shoot the player.

6 Conclusion

We managed to implement almost all the functionalities we intended to implement. We didn't have enough time to further optimize the code and to expand the program and introduce new functionalities. We wanted to make the game logic more complex by having various kinds of difficulties that the player has to avoid, and by adding extra power-ups to make the game more interesting, but time was not our friend.

We would love to thank David and Raees for patiently helping us throughout the whole project. Without them, this project wouldn't be possible.

7 Logbook

The following table describes how the work was divided in the project.

Logbook	
Abdullah	segmenting code into different files start and end screen movement of game elements the graphical functionalities binfiles of some sprites timer power-ups collision detection reading and loading files (sprites binfiles) listening and handling keyboard input
Lotte	implementing strucs for enemies and player making array of strucs for enemies ending game and making new level binfiles of soms sprites letting duck and enemies move collision detection shooting enemies

Table 1: Logbook.

8 Sources

- Collision detection algorithm:
  https://happycoding.io/tutorials/processing/collision-detection
- Handling the keyboard input:
  keyb.asm and *mykeyb.asm* in *examples*.
- Opening, reading and closing files:
  Dancer.asm from *examples*.
- Generating random numbers:
  rand.asm in *examples*.
- Displaying strings and integers:
  Displaystring and *printUnsignedInteger* procedures from the WPOs.
- The vertical blank interval:
  The assembly compendium from Canvas.
- *strlen*:
  http://www.int80h.org/strlen/