VRIJE
UNIVERSITEIT
BRUSSEL

Project Computersystemen

FLAPPY BIRD

Group G28

Tibo Martens & Ebe Coomans

December 23, 2021

Academic year 2021-2022

Ingenieurswetenschappen

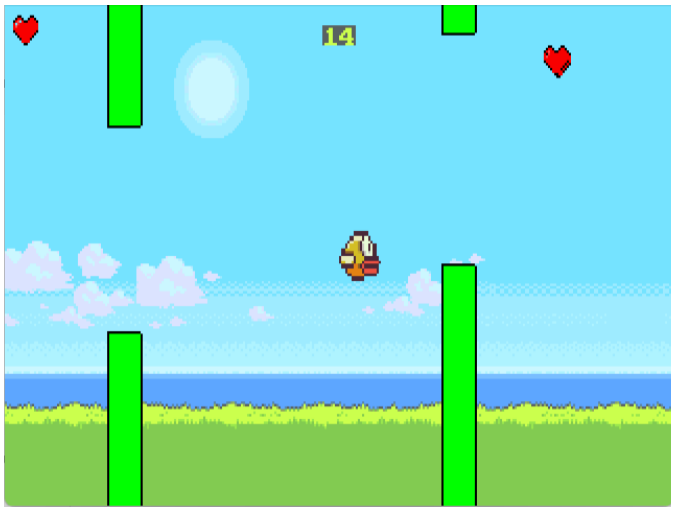

Figure 1: Example screenshot of the game.

1 Introduction

We decided to make a game very similar to the popular mobile game called flappy bird. This is a game where a bird needs to dodge tubes. This is accomplished by moving the bird upwards using a simple touch on the screen or in our case a press on the spacebar. It is as though every press on the spacebar is a wing stroke of the bird. It is not possible to move the bird downward yourself. The downward movement of the bird is controlled by the game in a way which ressembles gravity. The holes of the tubes where the bird needs to get trough differ in size and position. All of the above makes flappy bird a simple, fun and addicting game.

Our goal was to replicate the game as good as possible and on top of that add some extras to the game. It was also important to us that the game is userfriendly and fun and easy to play. Finally we wanted to recreate the oldschool feeling of the original game, but add a modern touch to it.

The rules of this game are quite simple. Just try to dodge as much tubes as you can without dying and get the highest score. The difficulty of the game increases as you progress. This means the tubes move faster towards you and the hole size decreases. In our game we added moving tubes in certain places, which adds another degree of difficulty. You can also gain extra lives in our version. The last thing we added is a boost. This is a very brief overview of the functionalities of our game, but everything will be explained in more detail in the following sections.

2 Manual

The controls of the game are very simple. The game starts when the enter key is pressed. During the game the space bar is used to move the bird up. As expained before you can't control the downward movement yourself. You can play again after dying by pressing the enter key, or you can stop playing using escape.

3 Program features

As explained briefly before the game increases in difficulty as you progress. This first of all means the tubes start moving faster towards the bird. More specifically this means every time the score is a multiple of 10 the speed is increased by one. There is a maximum speed, so the game remains possible to play. Secondly, the hole size decreases when you progress in the game. This happens at the same points where the speed is increased. Important to note is that the hole sizes are randomly generated, so to decrease the hole size we decrease the minimum hole size. This also means you won't notice a sudden change in hole size, but generally the holes will be smaller the further you get.

To add to the increasing difficulty there are score intervals where the tubes start to move vertically up and down. This happens at the intervals [20,30], [40,60],... The game is developped in such a manner that the holes of the tubes never move too far up or down.

A new feature in the game compared to the original game is the possibility to gain lives. These lives appear randomly and they move up and down at random. This is self explanatory, but everytime you hit a moving life one life is added to your total. Your total amount of lives is displayed in the top right corner of the screen. There is a 1 in 150 chance of generating a life every game loop. This is further explained in the next paragraph.

There also is a second special feature which is a boost. Like a life it moves up and down at random. When you hit a boost you basically skip 4 tubes without losing any lives. A boost is also interesting because it decreases the speed back to a value close to the initial value at the start of the game. So when you hit a boost later on in the game, the game also becomes less difficult. Because of this feature a boost is more rare and only has a 1 in 350 chance to be generated every gameloop.

4 Program design

First of all we use two different types of structs in our code. One is a struct called flappy which is used for the bird, the lives and the boost. This struct has 3 properties: an x-and y-coordinate and a flag that tells if the element needs to be drawn or not (this only applies to the lifes and the boosts). The other struct is used for the tubes. This struct has 5 properties: an x-and y-coordinate, a width, a height and a flag to indicate wether a tube is moving up or down (this is for the vertically moving tubes feature). Our game uses two tubes which move from right to left. Their parameters (hole size, position of the hole,...) are updated when jumping from the left edge back to the right edge. But basically the same two tubes are appearing every time. Now this is explained we can move on to the structure of the game itself.

Everything starts with setting the video mode to 13h. Then the colourpalette is updated with the colors we use for the game. In total 44 colours are used. How we get these colours is explained later. Next all the used bin files are opened and read. These bin files contain the information to draw the background and the sprites.

At the beginning of the game the background is displayed and a message is shown to press enter to start the game. In a small loop a readkey procedure is called which checks if the enter key is pressed. This loop is stopped when a flag which indicates if the game is being played or not is set. It offcourse becomes set when the enter key is pressed. After that the game loop is entered.

The first thing that happens in this gameloop is checking if the bird has hit a tube or not. This is done first because the positions of the moving elements in the game haven't been updated yet here. If a tube is hit one life is taken away. It is important to note that this procedure is called once for each of the two tubes. Then the procedure which draws every element of the game is called. This procedure draws the background, the flappy, the amount of lives and the two tubes. (the lives and boosts are drawn in a procedure which deals with the specials). How we display the images on screen is explained at the end of this paragraph. Next the procedure which updates all of the positions is called. More specifically the position of the flappy and the two tubes. This procedure also calls the specials procedure. This is a very large procedure which deals with all the specials. It updates the speed of the game, decreases the hole sizes of the tubes, deals with the vertically moving tubes and deals with the extra lives and the boosts (generating them and calling their draw funtions). As mentioned before the appearance of these specials is based on chances, so now is a good time to explain how the concept of chances and randomness is added to the game.

We use a procedure called random which generates a large random number. Then we divide this large number by another number and use the remainder to have a random number between 0 and the number we divided by. To implement the idea of a chance we generate a random number and then compare this random numer with another one. So for example if we want a chance of 1 in 300 we generate a random number between 0 and 299 and then compare it with 298. In the game itself you will notice the chance of generating a life is 1 in 300, but since we call the specials procedure once for every tube, this means there is a 1 in 150 chance of creating a life. The same applies to the boosts, but there the chance is even smaller.

After updating the positions the procedure is called which checks if certain keys are pressed. The keys that will have an effect are the space bar to move the bird upwards, the enter key to (re)start the game and the escape key to exit the game. Next up the score is calculated, this is done using a procedure which checks if a tube has been passed or not. This procedure is also called once for each of the two tubes.

After all of this the time procedure is called. This procedure introduces a timing mechanism in order to make sure that the game renders and processes at a constant speed, independent of the machine it runs on. Then at last the amount of lives is checked and if it is equal to zero the game loop is ended.

When the game loop is ended the background is displayed once more, this time with a message saying to press enter if you want to play again or to press escape if you want to exit the game. Afterwards all the parameters of the game elements are restored to the default values. Then finally the readkey procedure is looped again to see if a key is pressed. Depending on which key is pressed the game will start all over or the game will be exited. This concludes the main structure of our game.

Finally we will go into more detail on how we display the images. It is a process with a lot of different steps. First of all an image is chosen from the internet. Then this image is resized to the desired size using an online tool. For the background for example this was 320x200 pixels. Next up the colour dept of the image is decreased using an image-editing program called Irfanview. The background and the flappy image are each reduced to 16 colours. The life and boost are reduced to 6 colours. This leaves us with 44 colours in total. These colours can be read from the palette in Irfanview, but they are 8 bit RGB codes. Which means each value is between 0 and 255. The assembly palette however uses 6 bit RGB codes. So the original 8 bit RGB values are put in python where they are reduced to 6 bit by simply dividing them by 4. Now we just need to make bin files of the images. In these bin file each pixel is represented by a byte. The value of the byte refers to the correct colour in the colourpalette. This means the bin file can be read in assembly and the bytes can be stored one by one to the video memory since they represent the correct colour that needs to be drawn.

The making of the bin file is quite easy in python. The image is opened en then every pixel is accessed one by one. Each of these pixels is represented by an integer between one and 11. These integers are converted to binary and are then written to a bin file. This is done by opening a file in the write binary mode and then using the file.write() function. For the next images we add the amount of colours we already used in the previous images to the integers that represent the coulour value of the pixels. This way the bytes in the bin file remain consistent with the colour palette. Next up we use these bin files in assembly. There the files are opened and read. After they are read they have been stored in an array. To display the image we just run over this array and store the bytes to the right place in the video memory. A last thing that is interesting to mention is that we don't immediately store the values to the video memory. Instead we first write them to a buffer. This is done because our drawing method is large and complicated so sometimes it wouldn't be possible to perform all necessary updates during the time of one game loop. During the time procedure this buffer can be copied extremely fast to the frame buffer. This ensures our game looks continuous and not choppy.

5 Encountered problems

A first difficulty in our project was managing the timing of the game. We needed to make sure that some procedures only took place once and others multiple times. For example after touching the tube and losing a life it does not have to be checked again. Situations like this were pretty common in our game and we resolved this by using flags.

Another problem that we faced was the smooth appearance of the tubes at the borders. At first, we resolved this by drawing black borders at the left and right sides of the screen so that the tubes disappeared smoothly under the borders. Since, this is not the most beautiful solution we decided to use an alternative where we decrease and increase the width of the tubes with the speed of the game when they reach the edges.

Furthermore, when drawing the game the upper side of the screen was flickering. The reason behind this was that we could not update the entire video memory in the provided time. This was resolved by using a buffer. In the procedures we stored everything to this buffer. As explained before this buffer can then be copied to the video memory very fast and thus the flickering problem was solved.

Another problem we encountered was while drawing the images. We first wrote python code to make an array of the used colours in an image. When we ran this code it became apparant the images contained way too much different colours. The 320x200 background for example used around 750 different colours. This is when we decided we needed to decrease the colour depth of our images.

We also had a problem with the drawing of the image of our boost. This gave some errors which we did not understand. After a lot of debugging we found out it was because of the fact that we did not put the edx register on zero before making a division. This was however in a totally different procedure.

Finally we almost always had some problems when trying to put the different asm files we each wrote together, but in the end we always found the right solution, sometimes only after a long search though.

6 Conclusion

In the end we managed to code a working flappy bird game, with all the new features we wanted to add. There are still some small improvements possible. For example when a life and a boost appear at the same time they move in a very similar way. The learning process for assembly is quite a bumpy road and we encountered our fare share of problems, but overall we are pretty happy with the results and we think we achieved our main goals.