VRIJE
UNIVERSITEIT
BRUSSEL

Project Computersystemen

SPACE BUBBELS

Group G04

Sam Dilmaghanian & Seppe Goossens

August 21, 2022

Academic year 2021-2022
Computerwetenschappen

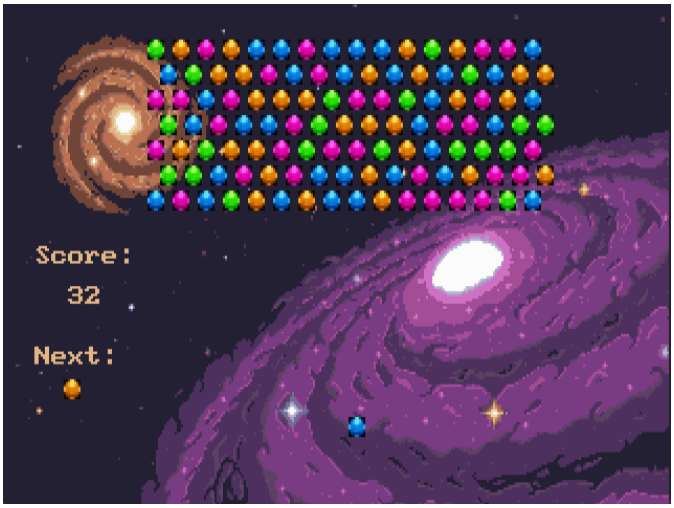

Figure 1: Example screenshot of the game.

1 Introduction

After failing to realize a functioning synthesizer in the first semester we've decided to recreate a simple graphical video game: Bubble Shooter. The goal was to recreate the game as close to the original. This includes: a graphical randomized play grid, interactive controls, a score counter and hit detection. The gameplay is very straight forward. A starting grid is displayed and using your mouse pointer you control the direction of a projected ball destroying similarly colored balls on the grid. Once all balls are destroyed the game is won. Losing is possible by having balls stack up towards the launching area.

2 Manual

As will be discussed in 5. Encountered Problems, the gameplay has some slight deviations from the original. The game is still successfully completed by destroying every ball with the corresponding ball color but losing now happens when you "run out of balls". A ball counter to the left of the playing area shows the amount of balls the player has to destroy the grid. If the player runs out of said balls the game is over, and the player loses. The launching of the balls has been altered. Using the arrow keys, the player decides precisely where the ball is launched. The launch of the ball is actuated by pressing the spacebar.

Using the escape key, at any time, the player can exit the game.

3 Program features

The game consists of a 16 by 16 randomized ball grid. With randomized balls (sprites) in 4 different colors. Thus, the program consists of a randomizer. And some way to hold this matrix clarified in 4. The randomizer is also used to chose your next ball. The next ball is displayed so that the player can make more thoughtful decisions when shooting the ball. You can move the shooting ball left and right. In total there are 32 positions where you can shoot the ball. This is done because the grid of balls are organized in a way that the rows are at an offset from each other. Comparable to a brick wall of a building (see fig1). You are only allowed to place the ball in the gap between the 2 upper balls. When you shoot the ball exactly under the upper ball, the ball will shift to the left or right depending on which side the ball is closest to. This is done so that the ball always ends up in the gap between the 2 upper balls.

(a) Shooting ball ends up just under upper ball

(b) Shifts right

Figure 2: Shift Operation

An important part of the game is the hit detection algorithm. When you shoot the ball, it goes straight up. Each time the ball moves, the algorithm will check if there was a hit or not. When the shooting ball hits another ball it will check what color it is. If the 2 balls are of the same color the algorithm will further check the surrounding balls of the ball that has just been hit. This way every ball of the same color as the shooting ball will be removed. When the shooting ball and the ball that has been hit are not the same color, the shooting ball will shift left/right or stay where it is depending on the relative position of the ball with its surrounding balls.

(a) Checks surrounding ball colors

(b) Remove balls of same color

Figure 3: Hit Detection

To make the game more challenging, we implemented a ball counter. You start with a limited amount of balls and you have to clear the grid with these limited amount of balls. If you run out of balls before the grid is cleared, you lose. If you clear the grid within the given amount of balls, you win the game. To show the score to the user the program implements a way to display a string of characters and a number. This number displays the ball counter which decreases by one after the launch of a ball. The text is displayed in separate game sections: “start the game”, “game over”. It is also used during the main game: “ball count: ”, ...

Figure 4: Ball Counter and Next Ball

To make the game easier to use we implemented a main menu screen and an end game screen. The main menu allows you to start and exit the game with the given instructions. The end game show you if you have won or not and how many balls you had left. You can also restart the game or exit it.

(a) Main Menu Screen

(b) Game Over Screen

Figure 5: Main Menu and Game Over Screen

4 Program design

4.1 Initialize

Let's with the main loop of our program. We start of by calling the `initialize` function. This function contains the random code initializer and the keyboard installer. We also set the video-mode to 13h and update the colour palette. After this we open all our files and pass them to the `readChunck` function which reads the chunck to the buffer (see DANCER). After the `initialize` function we set the start position of the starting ball. This position is a variable defined in the `DATASEG`. The value position is contained in a macro and we move this into the position variable. All the positions of the text and score attributes are also defined as macros. This makes it much easier because we only have to change the position in one place in the code.

Next we call the `fillArray` function. To better understand how this function works we need to explain how our game logic works. The playing grid is basically a 1D-array (We see it as a 2D-array but in reality it is a 1D-array). A 0 represents no ball and an integer between 1-4 represents a ball with a certain color. Because each row is offset by half a ball. 32 columns are needed to represent 16 balls. Thus, the game has a 32 by 16 grid instead of the perceived 16 by 16. Example of 2 rows of the array:

```
dd 1, 0, 2, 0, 1, 0, 1, 0, 3, 0, 3, 0, 2, 0, 4, 0, 1, 0, 4, 0, 2, ... ; row 1
dd 0, 2, 0, 1, 0, 1, 0, 1, 0, 3, 0, 4, 0, 4, 0, 3, 0, 2, 0, 4, 0, ... ; row 2
```

We chose to build our game like this because it makes drawing, updating, filling and checking the hit detection much easier than using 512 structs to represent your balls for example. The following functions are all based on checking, accessing, and editing this array. So now back to the `fillArray` function. As the name suggests, it fills the array. We start of with a zero filled `DWORD` array with a predefined length of512 . This function fills this array with the randomized balls represented by a 1, 2, 3 or 4 (signaling the color of the ball), 0 represents an empty space. Randomization is done by randomizing a 2-bit number. This is the design reasoning for displaying 4 different colors. This is done using a provided randomizer by the project organizers. This way we have a randomized starting grid that is different every time you launch the game.

4.2 Main Menu

After the initializing and filling the array we enter the main menu loop. We first draw our background. We will return on how we do this. After that we call the `refreshVideo` function. This function refreshes the video memory. After that we draw our strings on the screen with the `displayString` function. The x and y positions of these messages are defined as macros. The displaying of text and numbers is done by using provided code given by the project organizers. These implementations require a character or number (DWORD) input and a pixel specific location variables. These 3 strings make up our main menu. We now enter the keyboard loop. In here we check what the last pressed key is. This code is also provided by the project organizers. We simply have to campere the `al` register with the corresponding key. If space is pressed, we enter the gameloop. If esc is pressed, we exit the game. If nothing is pressed we keep looping the keyboard loop untill something is pressed.

4.3 Game Loop

We enter the game loop if the space bar is pressed in the main menu loop. We first call the `updateArray` function. This function is the hart of our game. It allows to place a ball of a given type/color at a given position and choose to check hit detection or not. So the function has 4 parameters: the array, the position, the ball type and a hit detection check, all of the `DWORD` type. Inside the function, we first load in the given parameters into the corresponding registers. One important thing to note is that the esi registers is only filled once with the array. After that we always refer to it as a pointer in other functions. This way we always have the latests and most up-to-date version of the array when using it in other functions. We place the given ball type at the right position in the array. After this we call the `decodeArray` function. We will shortly explain what this function does. After we placed the ball in the array we have to perform the hit detection. To keep track if we have a hit in our hit detection we use the `ball_hit` variable to check if we have a hit or not. This variable is also used in other functions. The hit detection function works as follows: we save the type of the ball that we have to check if it has hit a similar ball or not. After this we call the `hitDetection` function 4 times, each with a diferent orientation. There are 4 possible orientations: left, right, up left and up right. This function takes on 2 parameters: the position of the ball to check and the orientation. Let's for example say we have to check the right orientation first. We add 2 to the given position, check the type of ball at the +2 position and compare this to the original type. If they are not the same we jump out of this function and return to the other hit detection function calls for the other orientations. If they are the same we have to remove this ball. Before removing this ball we have to check if the ball also has neighbouring balls that are of the same type. To do this we again call the `hitDetection` in the right, up right and up left orientation. So we are performing recursion. This means we first propagate to the very end when there are no similar balls left and we gradually jump out of the recursive functions while removing the balls of the same type. This is done in all the corresponding directions. The `removeBall` function removes a ball at a given position. This is done by redrawing the background, placing a 0 at the given position and redrawing the shooting ball and the strings for the score and next ball.
Back to the `updateArray` function. There are 2 special occasions. When we are at the max left position or at the max right position of the grid. To check this we used the modulo operator on the position and checked what the remainer is. These are special because the hit detection function will check both the left and right orientations, also when we are at the edge of the grid. This is not something we want because in this way it's possible that your ball will be removed at the left edge when there is a ball of the same type one row higher at the right edge. This happens because our grid is a 1D-array but we pretend that is a 2D-array. So we

made some adjustments in the updateArray and hitDetection functions to compensate for this.

Back to the game loop, after that we updated the array with the starting ball, we call the showNextBall function. This shows the next ball that you will receive when playing. This function also uses the randomizer code to chose a random ball color. The type of the ball is saved in the nextBall_type variable so that we can use it in the next play. We also display the 2 strings for the score and the next ball. After this we enter the keyboard loop. This loop works in the same way as the keyboard loop in the main menu but just with different controls. We can move left, right, shoot or exit. When the left or right arrow key is pressed we enter the loop to move the shooting ball to the left or right. This is done with a special moveStartBall function. This function just moves the shooting ball left or right with 1 position using the updateArray function. We also limit the ball to go past the left and right edge. The moveStartBall has a parameter for the orientation so that we can use this for left and right movement. Each time we move the ball left or right we also have to redraw the strings. We also call the updateScore function. This function displays the score variabele at the right position. The other option is to shoot the ball. When we enter this loop we first decrease the score with the decScore function. This just subtracts the score with 1. Next we call the showNextBall and pass a 1 to the function meaning that it may generate a new random ball. After this we call the shootStartBall function. This function moves the ball up 1 row using the updateArray function meaning that we also check for hit detection every time we move up. Depending on the final position of the shooting ball we check if we have to shift left or right when hit another ball. After this we call the resetStartBall function. In here we reset the position and type of the shooting ball with the newly random generated ball type in the nextBall_type variable.
Every time we move trough the keyboard loop, we check if we the game has ended or not with the checkEndGame function. The game can end in 2 ways, either when you have 0 balls left or when the grid is empty. Depending on the outcome a corresponding variable will be set to 1.

4.4 End Game

When either of the 2 end game variables are set to 1 we jump in the end game label. In here we display the end game messages and either a well done message when you won or a message with how many balls you have left if you didn't complete the game. In here is also a keyboard loop in which you can choose to play again or to exit the game.

4.5 Drawing the assets

As said earlier, we still have to explain what the decodeArray function does. The decodeArray function itterates trough the array and checks every type of element in the array. Depending on the type it either doesn't draw anything or calls the drawBall function and passes the corresponding frame to the function. To draw these balls on the screen we use a struct to keep track of their x and y position. Each time we move trough the array, the x and y positions are adjusted as necessary. The drawBall function uses a .bin file as frame. It then scales the x and y positions with the frame width and length. The function loops trough this array and stores the bytes at the right place in the video memory. We also make use of a buffer. Instead of storing the bytes directly in the video memory, we first store them in a buffer. This makes the game smoother.
The making of the bin file takes a few different steps. We downloaded the background and resized it to make it 320x200 pixels. This way it fits the DosBox resolution. The 4 different colored balls were handmade using a sprite maker tool online and saved them as a .png file. They're all 8x8

pixels. Next we used InfranView to decrease the color depth of the images to 16 colors. This is done because DosBox only supports 256 colors at 320x200. In total we have 80 colors so we are below this threshold. We can read the exact RGB values from the color palette in IfranView. These are however 8 bit RGB codes (0-255). Because DosBox uses the VGA in the "classic" format, only 6 bits per color are used (0-64). This means that RGB codes that we read from the palette in IfranView have to be divided by 4 so that DosBox can read them. The python script handles the converting from .png to .bin file and the conversion of the color palatte. We load in the resized and decreased color depth .png files and the corresponding .bin files. Next, each pixel in the .png file is written to the .bin file by converting it to binary. After that we copy the 16 - 8 bit RGB values from the color palette to the corresponding arrays in the python script. These are all then converted to 6 bit and printed in the correct format so that DosBox can read them properly.

5 Encountered problems

One of the biggest reasons that we didn't replicate the original Bubble Shooter game is because we couldn't accomplish an oblique movement. There are a lot of advantages of using an array as the basis of the logic of your game but when you need to do complex physics or movements this is not very handy. Because our movement and graphics are directly linked to the array and more specific the resolution of the array, the complexity of the physics of our game is very limited. We tried to do oblique movement but it just wasn't smooth. The ball would visibly go 3 left, 1 up, 3 left and 1 up for example. On top of this we would have to hard code an oblique path for each end position of a ball. This just didn't work. A workaround would be to increase the resolution of our array so that we would achieve a more smooth oblique motion, but the problem that we have to hard code the oblique paths would still not be fixed. Because of this we decided to change the concept of the game a bit. Instead of oblique movement you could change the location of the starting ball and shoot up. Also instead of extra rows coming up from the top like in the original Bubble Shooter game we used a maximum ball counter to make the game as challenging as the original one.

Another aspect of the game that we struggled a lot with is the hit detection algorithm. It was quickly clear that we had to use recursion. But it was very complex to keep track of the different orientations and often we would be stuck in an infinite loop which made debugging a lot harder. We also had to limit the hit detection at the edges of the grid because otherwise the algorithm would think you hit a ball that is on the other side of the grid. To fix this we always check the position of the shooting ball and check if we are at the sides or not. When you are at the side you go to a special part in the algorithm that skips a part of the hit detection check. You will see that the ball goes visibly slower at the edges because of this. However we didn't find a fix to make the ball go smoother at the edges.

Another visible problem is the flickering. This problem is inherent in the way we draw our assets on the canvas. When a ball moves we redraw everything. So first draw the background, the balls and then the text. Because of this you can see the text flicker sometimes. We mostly resolved this by adding a buffer that waits a few frames before performing another action. This is the `timer` function. Especially when you shoot the ball up, this is noticeable. We mostly fixed this by placing the `timer` function at different locations in the code and also by playing with the `TIMER_CTE` macro which changes the amount of frames to wait.

The most frustrating problem however was the error we would get when opening certain files. When opening the "bg.bin" and "blueball.bin" files everything worked fine. But when trying to open the "greenball.bin" or "yellowball.bin" we would always get the error "couldn't open file". This was of course just the message defined in the DATASEG. The weird thing was that these files were created in the exact same way as the background of blueball files. Because we initially didn't find the bug we just continued and only used the "blueball.bin" file for the other balls. This way we could continue with our game logic. However this problem haunted us for weeks and we just didn't find the cause of this problem. We tried everything from remaking the .bin file multiple times, modifying them, checking the open file procedure, checking certain registers for error codes, changing the order of opening the files and so on. We just didn't find the problem. Until one day we tried to change the name of the files and miraculously it worked. So apparently there is a limit on the amount of characters a file name can contain. In retrospect this of course makes total sense to check this but we didn't expect it to be something this minor.

6 Conclusion

We succeeded in the main goal of the project. We've created a playable game that closely resembles the original. Additionally, we learned that making certain design decisions during the starting phases can have unexpected impacts on the full completion of the project. Therefore, we've also learned to be flexible and come up with viable alternative game-play solutions to the previously made design decisions. The completion of the program is considered mostly successful.