U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Final Report

Universe Bomb

UNIVERSE BOMB

Laboratório de Computadores - 2023/2024

Licenciatura em Engenharia Informática e Computação

Team T17_G04:

Supervisor: Nuno Cardoso

Regent Teacher: Pedro F. Souto

Students:

Amanda Silva (up202211647@fe.up.pt)

Leonardo Garcia (up202200041@fe.up.pt)

Manoela Américo (up202201391@fe.up.pt)

Marcel Medeiros (up202200042@fe.up.pt)

Index

1. User Instructions

1.1 Running The Game

Our game supports modes 0x14C and 0x115 of the Minix Virtual Machine, thus, the user can select which mode they want to play by running the game with differing commands:

1. Command “lcom_run proj”: will run the game in VBE mode 0x115.
2. Command “lcom_run proj [any character]”: will run the game in VBE mode 0x14C.

1.2 Main Menu

Upon starting the game the user will be presented with a menu screen, where they will be able to read the game instructions, start the game or exit. This menu screen serves as the gateway for closing the game, that is, whenever the user wins, loses or goes back, they will be directed to the menu.

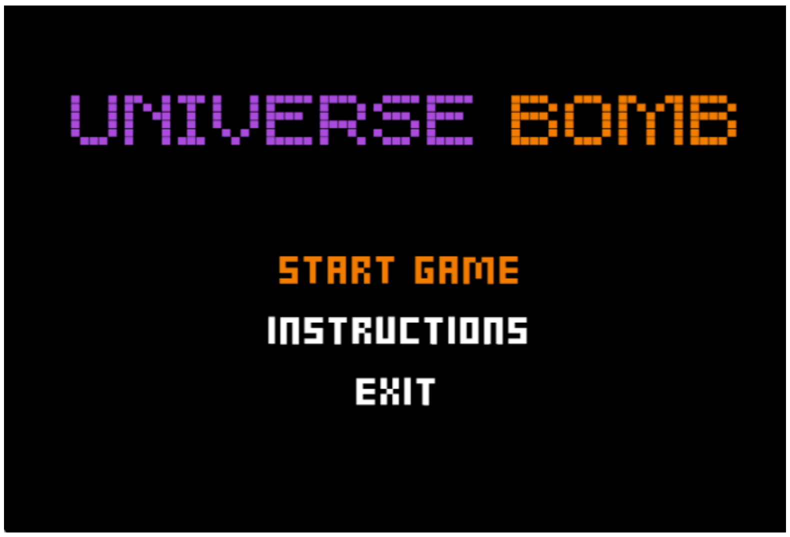


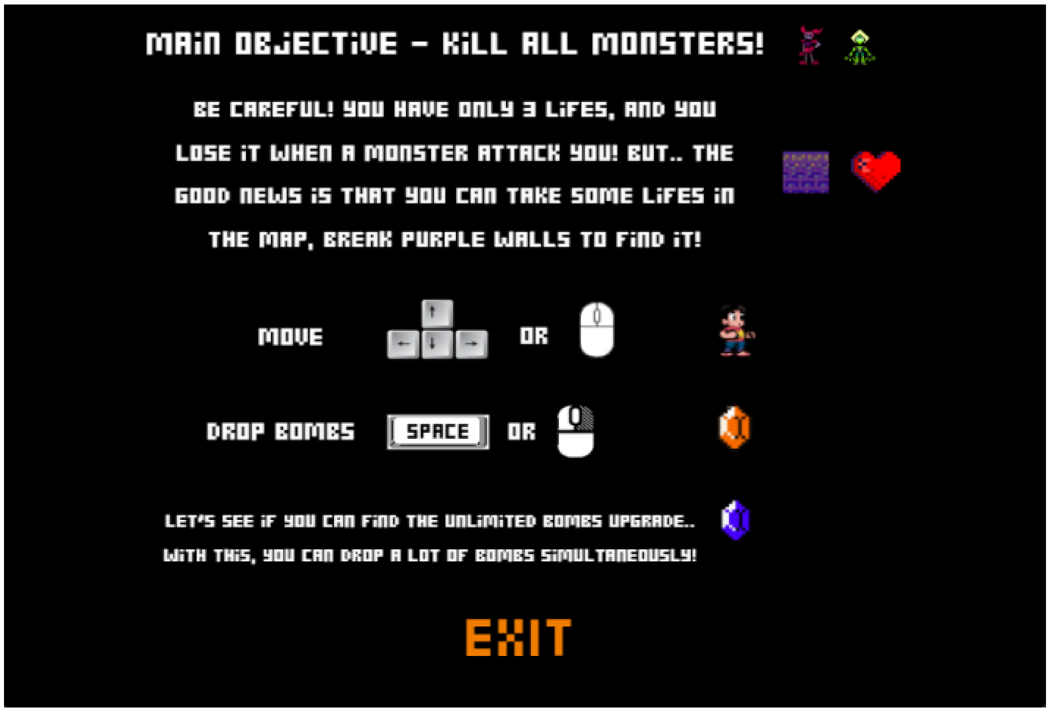
Figure 1. Main Menu

1.3 Instructions

In our game the user can go to the instructions menu and learn how to play it. We used both keyboard and mouse to allow the user to move and deploy bombs.

The controls are:

- Keyboard arrows **UP, DOWN, LEFT AND RIGHT** to move.
- **SPACE** bar deploys bombs.
- Steven (our hero) moves with the mouse.
- **Mouse-right-click** button to deploy bombs.
- **ESC** to escape (go back to menu)


MAIN OBJECTIVE - KILL ALL MONSTERS!

BE CAREFUL! YOU HAVE ONLY 3 LIFES, AND YOU LOSE IT WHEN A MONSTER ATTACK YOU! BUT.. THE GOOD NEWS IS THAT YOU CAN TAKE SOME LIFES IN THE MAP, BREAK PURPLE WALLS TO FIND IT!

LET'S SEE IF YOU CAN FIND THE UNLIMITED BOMBS UPGRADE.. WITH THIS, YOU CAN DROP A LOT OF BOMBS SIMULTANEOUSLY!

Figure 2. Instructions Menu

1.4 Game Screen

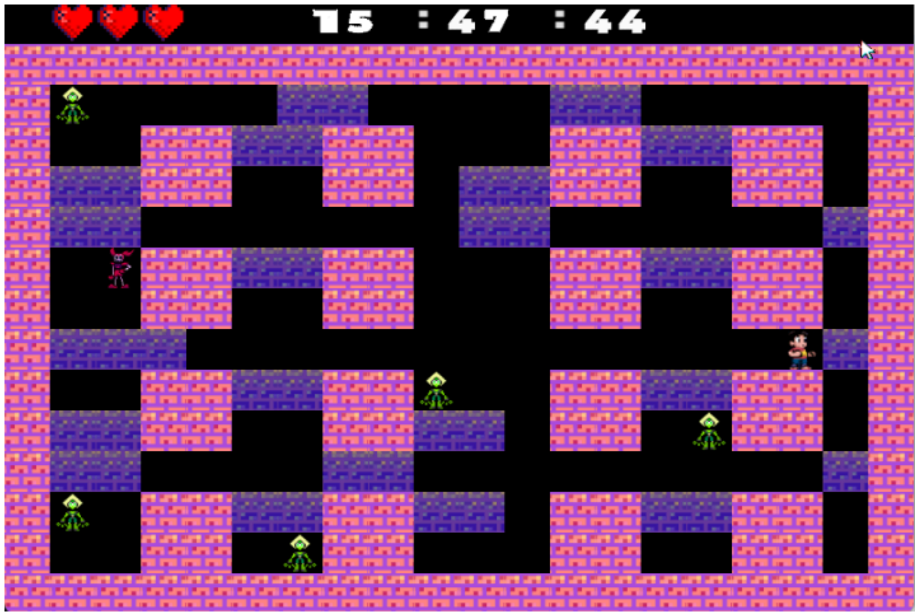
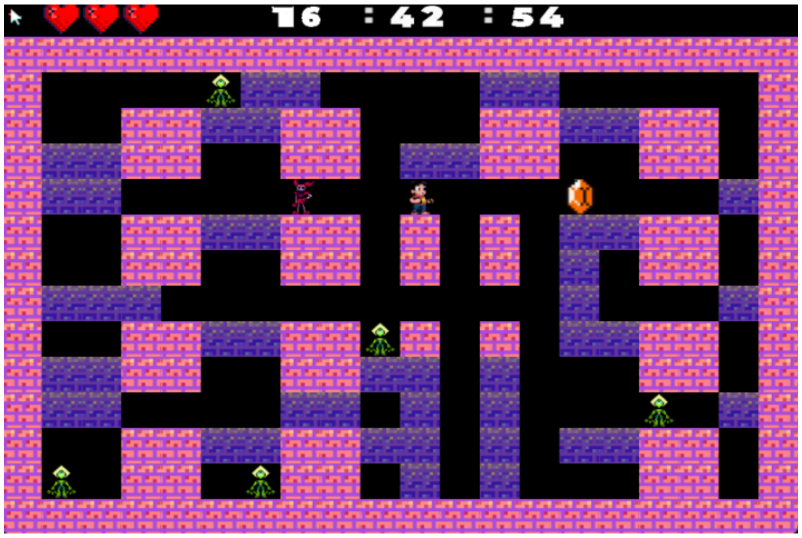


Figure 3. Game Screen

Upon entering the game, the user will be able to see the map (that changes its configuration considering the time of the day) and some of the functionalities of our game:

- **Steven**: our hero that moves upon user input.
- **Lives:** Steven starts with 3 lives that can be seen at any time on the screen.
- **Green Monsters**: monsters that move randomly at a rate of one position per second.
- **Purple Monsters:** monsters that move randomly at a rate of one position per 1/10 second.
- **Breakable Walls:** purple blocks that can break by an explosion of a bomb. Inside each of them there is a 10% chance of having an upgrade (additional life or unlimited bomb upgrade).
- **Unbreakable Walls:** pink blocks that stay the same throughout the game.
- **Clock:** the Real-Time-Clock device counts the time that is displayed on the screen.

1.5 Deploy Bombs



In the default setting of the game, Steven is allowed to deploy only one bomb at once. In the image, we can locate this bomb as the yellow diamond. After its release, the bomb waits 3 seconds for the explosion.

Figure 4. Steven deployed bomb

Once the explosion happens, it can be identified as a purple image of blocks that extends 2x at each position of a XY axis at the initial position of the bomb.

The explosion causes some effects considering the elements that it reaches:

1. Steven loses a life
2. Monsters die
3. Breakable wall break and may reveal upgrades.

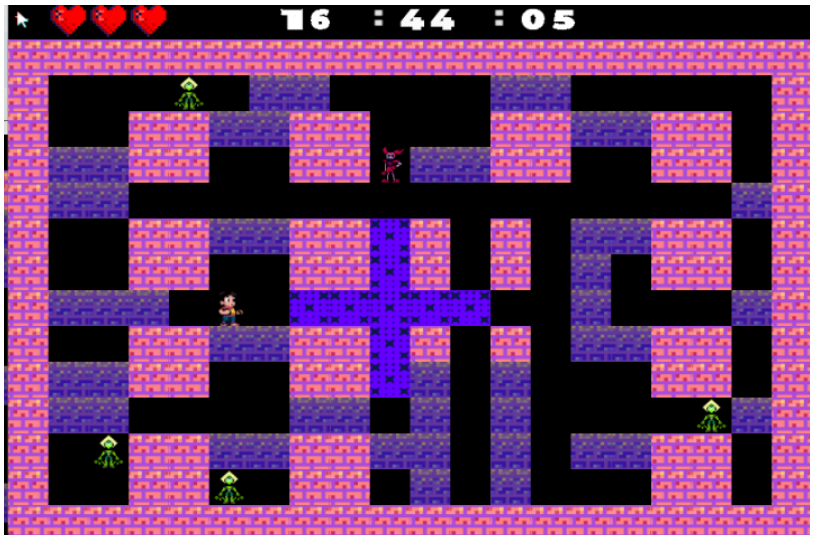
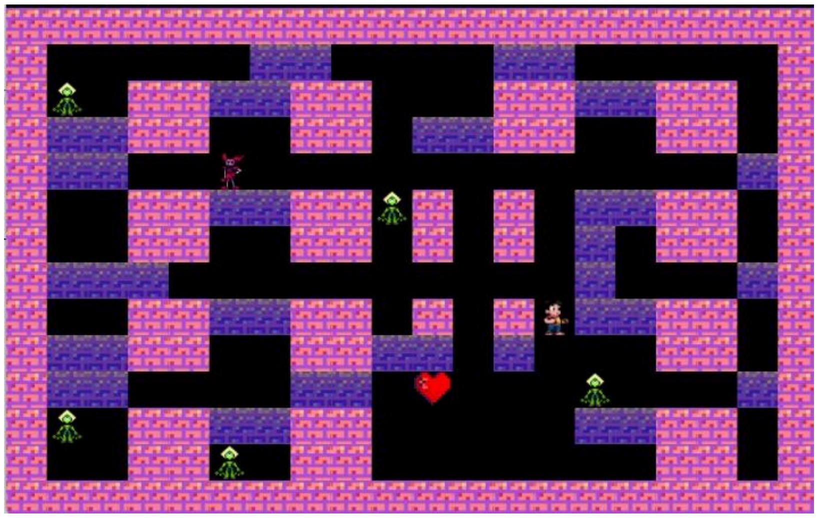


Figure 5.Bomb exploded

1.6 Upgrades

Whenever the user destroys a breakable well, it has a 10% chance of getting an upgrade.

Additional Life

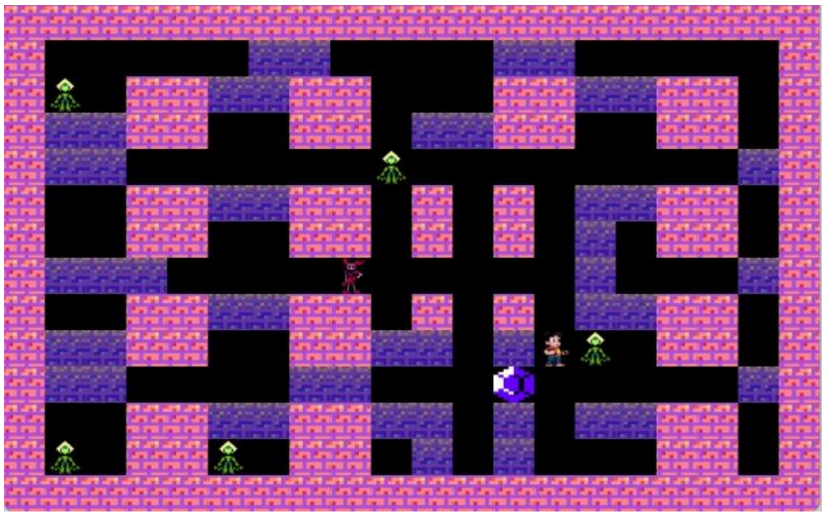


There two outcomes once you find an additional life:

1. **If Steven's lives are full**, he can collect more lives but there will be no effect (Max = 3)
2. **If Steven has less than three lives**, he can collect an additional life and have a better chance of winning the game.

Figure 6. Additional Life found when breaking walls.

Bomb Upgrade



Receiving a bomb upgrade means that, during five seconds, Steven will be able to deploy an unlimited amount of bombs at a time.

Figure 7. Unlimited Bombs upgrade

1.7 Win Screen

Once Steven defeats all monsters, he wins the game.

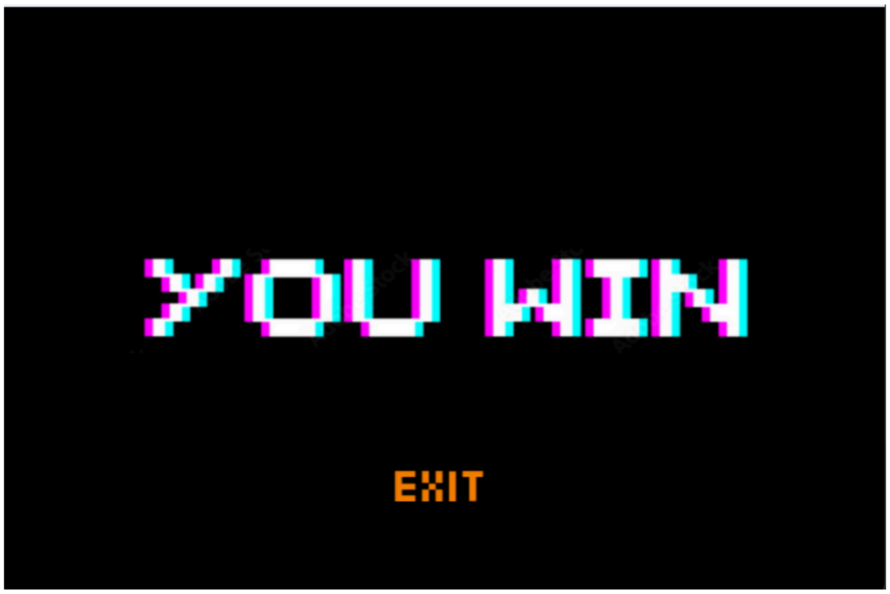


Figure 8. Game win Screen

1.8 Game Over

If Steven loses all his lives, he loses the game.

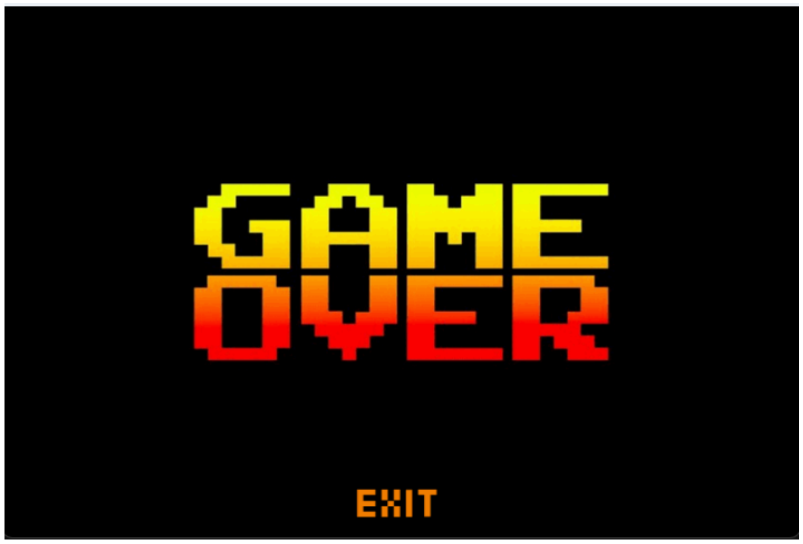


Figure 9. Game over screen

2. Project Status

Device	What for	Interrupts
Timer	Controlling frame rate and for timing game actions	Yes
Keyboard	Menu navigation, move the hero and release bombs	Yes
Video Card	Display user interface	No
Mouse	Move the hero and release bombs	Yes
RTC	Show the time on the game screen	Yes

2.1 Timer

The timer is essentially used in our game to control the frame-rate, making 60 updates to the frame buffer per second. In addition, the timer is also useful for updating the screen as certain actions take place in the game.

Every second, the hero's actions (hero_actions( )) are checked, i.e. if the hero has picked up an upgrade or a life, we update the map and the variables responsible for storing life and upgrade information. We also call the check_bomb( ) function, which checks the status of the current bomb and takes the necessary actions (explode, take the hero's lives, break walls, kill monsters, etc.). The other timer actions every second are: move the green monsters (move_monsters_green( )) and call the RTC interrupt handler (rtc_ih( )).

There are other actions that occur 6 times per second, the first one is to move the red monsters (which are faster than green ones), calling the function move_monsters_green( ). The other action is to check the unlimited bombs (check_unlimited_bombs( )), is it similar to check_bomb( ), but deals with a lot of bombs (max. 10) and it means that our hero is with the unlimited bombs upgrade.

2.2 Keyboard

The keyboard is responsible for giving the user the ability to control the game menu using the UP, DOWN, SPACE and ENTER arrows. Also, with the keyboard we can control the hero in the game, using the arrows to walk and SPACE to drop bombs. In addition, the ESC key during the game takes you to the menu, and in the menu you close the program if you press it.

All these keyboard responses are made in the gamepad_response( ) function, which checks what the game state is (MENU, GAME, GAMEOVER, etc.) and the scancode that has been received, in order to take the correct action.

2.3 Video Card

For our project, we used two graphic modes, where it is up to the user to choose which one to play, as mentioned in the user instructions (section 1).

The first mode used was 0x115, with 800x600 resolution, direct color mode, and 16.8M colors (8:8:8). The second mode used was 0x14C, with 1152x864 resolution, direct color mode and also 16.8M colors ((8):8:8:8).

We used the double buffering technique to improve the efficiency and visual quality of our game during frame updates. By allocating additional memory for a second buffer, we mainly draw on this double buffer and then copy its contents to the main frame buffer. This optimization allows us to draw each element more efficiently, as the frame buffer is only used for timer interrupts. For other device interrupts, the double buffer is utilized by calling the graphic_update() function.

To design our game, we used XPMs that we produced ourselves on the Canva website, and some that we found free on the internet, and turned them into sprites using the create_sprite( ) function. All the functions that configure the video card, as well as the functions that configure how to draw xpm's on the screen, are in the **graphics.c** file.

To move the objects during the game, we use a two-dimensional array to represent the 15x20 map, where our draw_map( ) function goes through this entire array and draws the respective xpm's.

The erase_screen( ) function clears the screen by setting each pixel to a specific color, effectively resetting the visual output. The draw_menu( ) function displays the game menu, highlighting the currently selected option and drawing each menu entry in the appropriate position. The functions draw_instructions( ) and draw_gameover( ) are responsible for displaying the instruction screen and the game over screen, respectively, by rendering the relevant sprites at predefined positions. The draw_gamewin( ) function similarly handles the game win screen display.

2.4 Mouse

The mouse allows the user to control the game movements, by moving the hero according to the mouse, in such a way that the hero follows the mouse movements. There is a tolerance rate to avoid the mouse being too sensible and the device can also be used to drop bombs, by pressing the right button.

To evaluate the mouse's most expressive movement direction (right, left, up, or down) and check if the right button is clicked, a function update_mouse() assigns a value to according to how the user interacts with the device.

The device responses are executed in the mouse_response() function, which receives the update_mouse() output as a parameter and takes the correct action in the game.

2.5 RTC

The Real-Time Clock (RTC) is used in our game to access and display the current hour, minute, and second in the user interface. Additionally, we implemented code within the game_start() function to display different game maps depending on the time of day.

In the file **rtc.c**, we implemented functions to configure the timer. The rtc_subscribe() and rtc_unsubscribe() functions handle interrupt calls, while the other functions like rtc_ih(), read_rtc(), conversion() and update_rtc_time() in the file read time units ranging from years to seconds, being called every one second. We also created a struct called **rtc_time** to store these values.

3. Code Organization / Structure

3.1 Main **(5%)**

File used for setting up the LCF configurations for the project, subscribing and unsubscribing all devices and creating the project main loop, using a drive_receive while loop that reacts with interrupts from each device and acts accordingly calling update functions.

3.2 Utils **(1%)**

File developed in labs that contains auxiliary functions used for handling device's commands.

3.3 Sprite **(2%)**

Used for creating and destroying all sprites in our project, as well as the struct Sprite. We decided to separate this into one file because we could include its header and easily access the sprites.

3.4 Timer **(6%)**

Contains the functions used for subscribing and unsubscribing timer interrupts, as well as handling timer interrupts. Aside from the functions developed in Lab 2, we also implemented the function timer_update(), that is called at every timer interrupt and handles game changes depending on the time.

3.5 RTC **(3%)**

This module contains functions to handle interrupts from the Real-Time Clock (RTC). We use the read_rtc() function to send commands to the RTC and read its output, which is essential for actions like checking if the device is busy and retrieving the current time. Once we collect all the information, we use the conversion() function to determine if the device is set to binary or BCD (Binary-Coded Decimal) time intervals. If it is set to BCD, we convert the time to binary.

3.6 Keyboard **(8%)**

File that contains code developed by the students in Lab 3. Used for subscribing and handling interrupts.

3.7 Mouse **(5%)**

File that contains code developed by the students in Lab 4 and specific game function for detecting the mouse movement direction and when a button is pressed. Used for subscribing and handling interrupts, processing mouse movement and disabling the mouse stream mode.

3.8 Graphics **(10%)**

Contains the functions used for setting mode, mapping the video RAM and drawing pixels developed in Lab 5. Besides this, we also implemented the functions print_xpm() and drawSpriteXpm(), that are used to render an XPM image and a Sprite, respectively. The file also features the alloc_double_buffer() function, which allocates memory for the double buffer to reduce flickering during rendering. Lastly, the file contains the function graphic_update() that updates the screen based on the mode of the game using the double buffer.

3.9 Setup_game **(5%)**

File that contains the functions setup_menu() and setup_game(), used for initializing the menu and game screens. Both initialize all variables used in other files in order to access positions, maps, counters and auxiliary variables. This is also the only time that we draw the pixels of the frame buffer, instead of copying it from the double buffer.

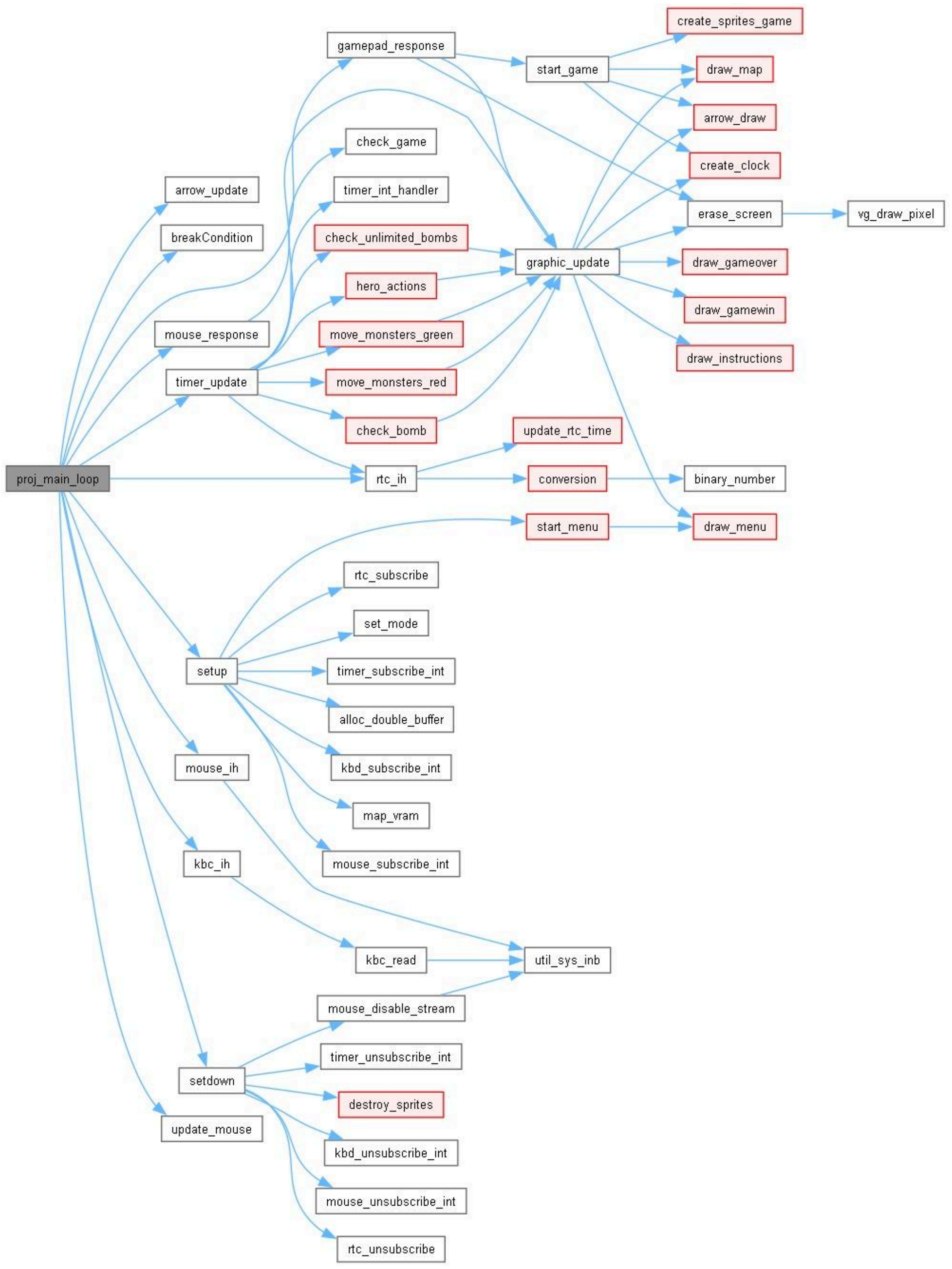
3.10 Draw **(15%)**

This file integrates various functions that handle drawing and rendering different elements on the screen for the game, like the map and the menu. In addition, it contains the function erase_map() and create_clock() that displays the current time with hour, minutes and seconds.

3.11 Controller **(30%)**

The controller file serves as the central hub for managing game interactions, particularly concerning movement mechanics. Among its pivotal functions, it has the mouse_response() that processes mouse inputs to update the game state based on the direction provided and additionally enables the placement of bombs in the game world, considering both limited and unlimited bomb scenarios. The gamepad_response() does the same but for processing the keyboard inputs. Furthermore, it includes the functions to move the monsters of the game and the hero's actions. At last, the function check_bomb() manages the detonation and aftermath of bombs in the game world and get_upgrade() places the upgrades on the map.

3.12 Function Call Graph



4. Implementation Details

The game implements a state-driven architecture through a state-machine enumerated in state_game declared in the file *setup_game.h*. The goal for using this approach aism on easing the use of different interrupt handlers tailored to the required actions within each state. This is noticeable on the mouse_response() and gamepad_response(), which will handle inputs differently depending on which state the game encounters itself. By organizing the game logic into distinct states, the codebase becomes more modular and maintainable, allowing for easier integration of new features and improvements. Moreover, the structured approach made it clearer to implement the graphic_update() function, which identifies the status of the application and executes the appropriate functions to draw and erase XPM's.

The implementation chosen for managing bombs and monsters in Universe Bomb relies on event-driven timer actions. For instance, when a monster changes its status, a timer kicks in to control how it moves. At each timer interrupt(1 time per second for green monsters and 6 per second for red monsters), the movement function is called and it's next movement is decided randomly. Similarly, timers are also used to manage the detonation of bombs. When a bomb is placed on the game grid, a timer is initialized to regulate its detonation time . When it expires, the explosion function is triggered, resulting in the destruction of nearby objects and potentially altering the game state.

Object-oriented programming was utilized as possible in the project, having classes be defined by the typedef struct expression, with their public methods declared in the corresponding header file. The structure *Gamepos*, which represents a position in a two-dimensional grid, was specially useful for the game as it was used for keeping track of the current position of the game elements after interrupt updates.

One of the challenges encountered when working with XPM's was how to work with it in the different Video Card modes and ensure that the map was still easily playable. The solution applied was to add the same XPM's with a 42.5% variance of size as it was identified as the resolution difference between modes. Also, one thing that helped us to solve it, was the creation of the variable XSize in draw_map( ), the value of it could be the division of the X resolution (vbe.XResolution) by the number of columns (int col) and it represents the size of the square XPM, which is very useful in the drawing loop.

5. Conclusion

The LCOM course has been a very enriching and difficult experience for us all. Creating this game helped us to use and gain perspective on many theoretical concepts learned across our journey with an emphasis on C programming and hardware interaction.

Creating the game required a deep understanding of various devices, such as the keyboard, mouse, timer, video, and RTC, and provided us with valuable practical experience in handling interrupts and implementing graphics. We utilized advanced techniques like double buffering to improve the efficiency and visual quality of the game, explored different graphic modes, and integrated multiple input devices. One aspect we aimed to implement but were unable to complete was the serial port functionality. Despite our efforts, we encountered challenges that prevented us from integrating this feature within the project timeline. This remains an area we would like to explore and improve upon in the future.

In the end, developing this project not only provided us with a practical opportunity to apply the knowledge we acquired but also encouraged us to think creatively and work as a team to overcome the challenges we encountered. The experience gained throughout this course will undoubtedly be a significant asset in our academic and professional careers.

We extend our gratitude to the professors and monitors for their collaboration and support throughout the semester, and we are proud of the final product we delivered. It was a great pleasure to participate in this course and develop such a comprehensive and engaging project.