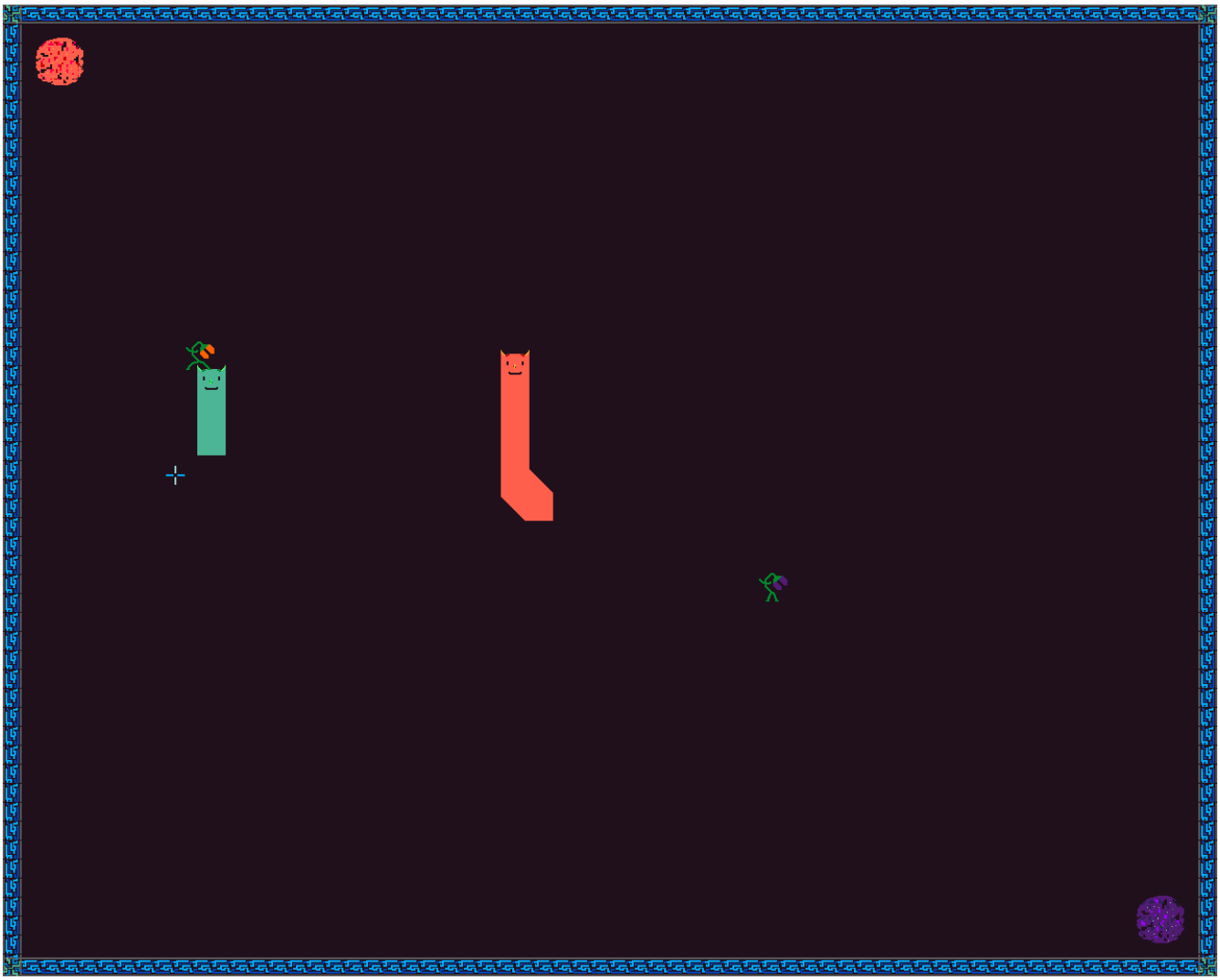
Skane Royale Edition

FEUP MIEIC 2019/2020

Final Project for the LCOM course



Authors:

João de Jesus Costa - up201806560

João Lucas Silva Martins - up201806436

Table of Contents

1. User’s instructions

1.1. Main menu

Figure 1: Skane Royale Edition's main menu

Once you open the program, you are greeted by the game’s main menu. Here you can see an image mixing art from the game and from another game, that served as inspiration for this one, called **Skane** (mentioned in section 6).

You have 3 options available to you: “singleplayer”, “multiplayer” and “quit”. The “singleplayer” option ‘throws’ you right into a game where you’ll have to fend of

enemies by yourself. The “multiplayer” option leaves you waiting for a second player to join your game. Once a second player joins, you can both engage in battle until death while fending off small enemies.

Both game modes will be discussed later in the report.

The “quit” options just quits the game restoring “MINIX’s” default configurations.

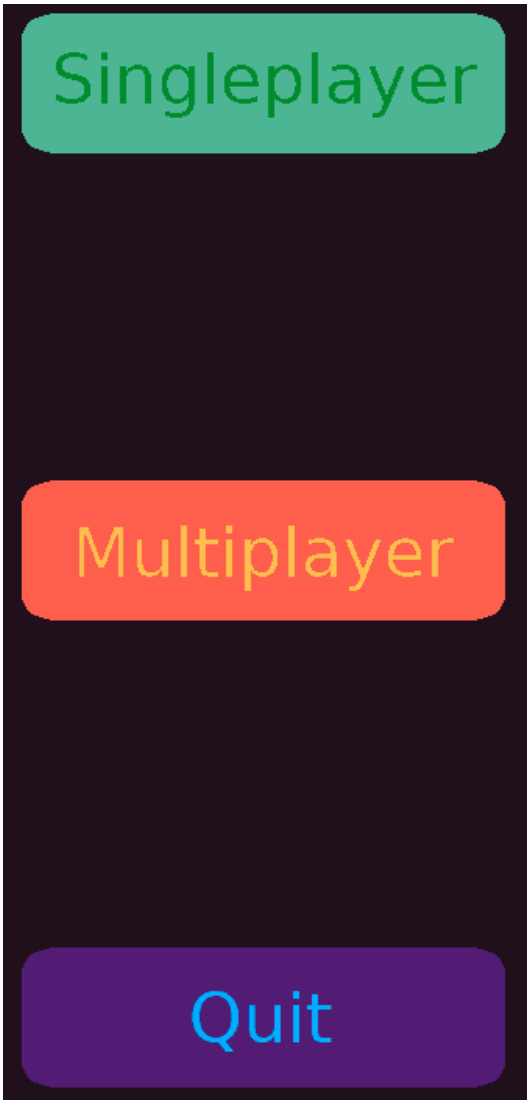


Figure 2: Menu buttons

1.2. Singleplayer Mode

If you chose the “singleplayer” option on the “Main Menu”, you’ll enter the “Singleplayer Mode”.

In this game you control a green snake called **Skane**.

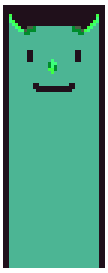


Figure 3: Skane

In this mode you must run away from enemies while shooting them down before they kill you. To do this, you have the ability to move using the standard ‘WASD’ keys and the ability to shoot projectiles (called **missles**), at the enemies chasing you, using your left mouse button.

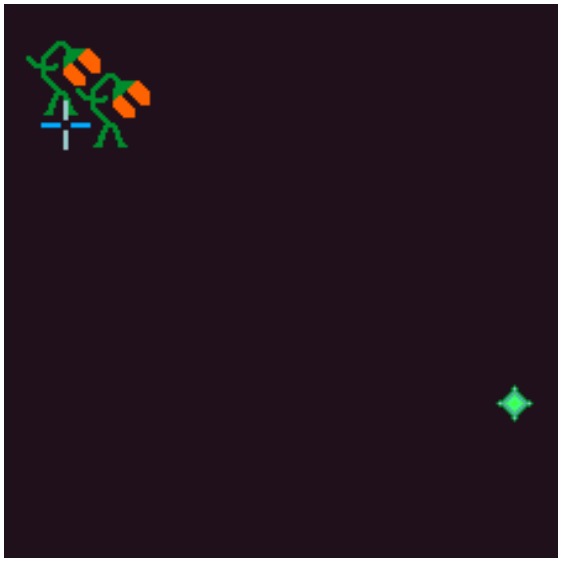


Figure 4: Missle and two enemies

The enemies will only try to bite your head. If they manage to bite you, you’ll take damage and the enemy will have to wait a bit before starting to chase you again.

If you shoot an enemy enough times, it will die and drop a delicious **strawberry**. When you touch a strawberry, you **eat** it and **grow longer**. The longer you are, the more health you have, so be careful you don't become so small that you die. You should also be careful when using your shooting ability because **missles** have a cost. You lose a bit of your health when shooting them.

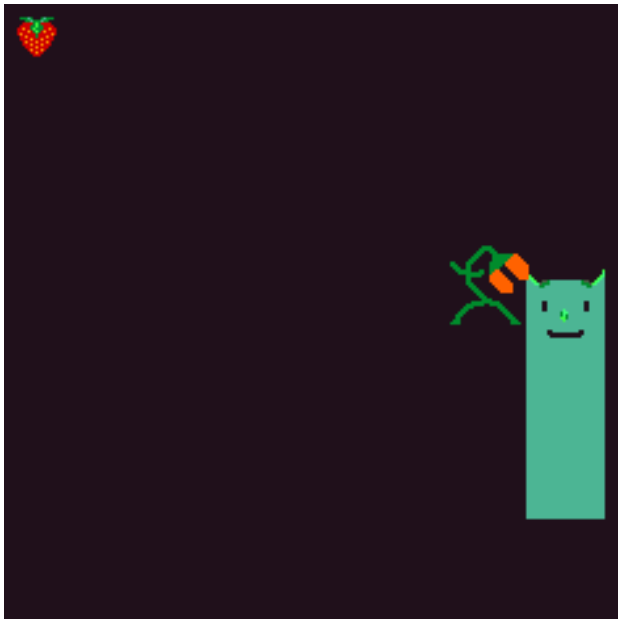


Figure 5: Strawberry, Skane and enemy

Even though you lose health when shooting, you shouldn't worry too much about it, because the strawberry dropped by an enemy **heals** you for more than what is needed to **kill** the enemy.

It should also be noted that shooting has a cool-down period and that **your** enemies always spawn from the same point (although the amount of spawned enemies is randomized).

In this game mode, the **difficulty of the game scales over time**: the enemies spawn in **bigger groups** and move **faster**. Although this seems unfair, you can use this to your advantage. Enemies are dumb so when they collide with each other, they push themselves around. With this, you can use your movement to make the enemies chasing gather in a big group and consequently start moving slower in your direction.

Every few rounds of game play, you'll level up. When you level up, your **missles'** speed and your rate of fire are increased.

Each time a group of enemies spawns, a new **round** starts. When a new round starts the game's colors are optionally randomized (**"Singleplayer mode" only**) (options are talked about in section 1.4).

Now that you know the basic game **mechanics**, you can roam around this map and see how big you can get before the enemies manage to kill you.

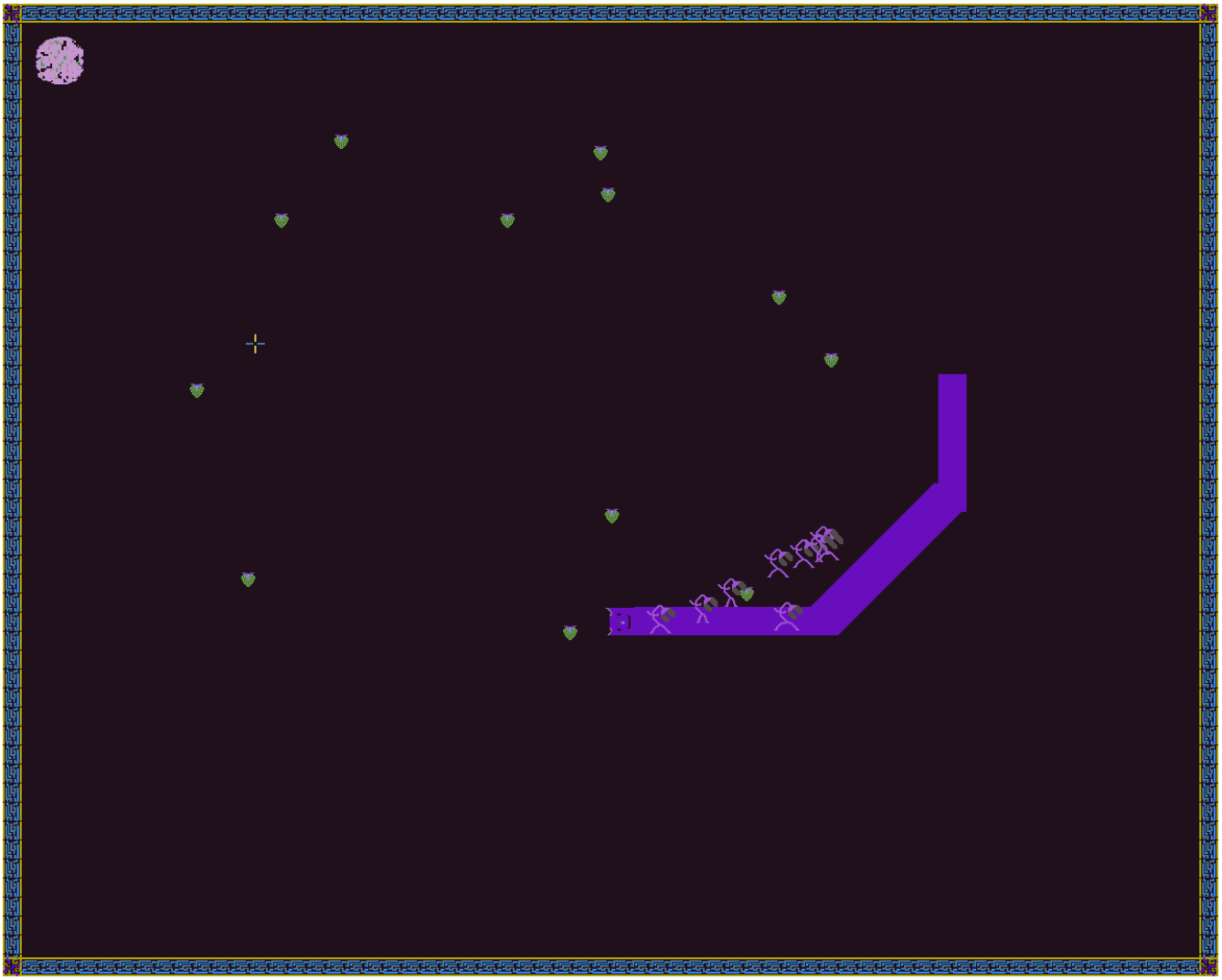


Figure 6: Game screen with Skane, enemies and random colors.

1.3. Multiplayer Mode

If you chose the “multiplayer” option on the “Main Menu”, you’ll find yourself in a loading screen. This screen signals that you’re waiting for an opponent to join you for a battle. You can’t escape from this loading screen, but it will quickly timeout if no other player connects.

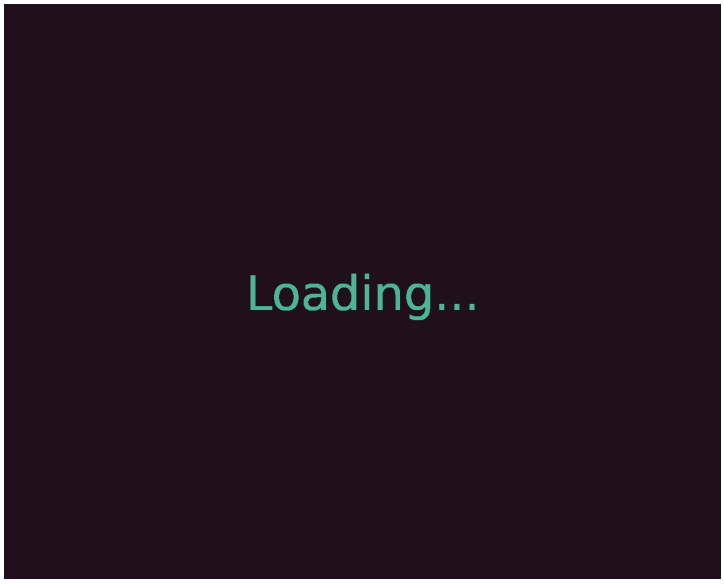


Figure 7: Loading screen

If you find someone to play with, you’ll enter the “Multiplayer Mode”.

This mode is very similar to the, already discussed, “Singleplayer Mode” but with three

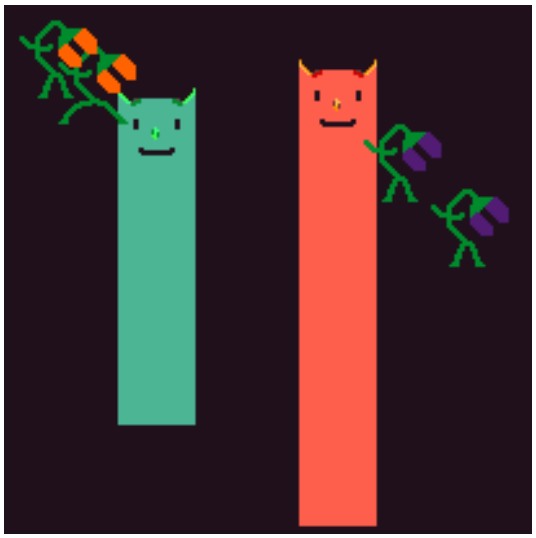


Figure 8: Both players and respective enemies

noticeable differences: you have an **enemy Skane** on screen, there’s **no enemy scaling** and you have **allies** (explained in a few paragraphs).

Your objectives in this game mode should be the same as in the “Singleplayer Mode” with the addition of having kill the enemy **Skane**. To achieve this, you’ll need to shoot him and dodge his shots.

If you manage to kill your opponent, the game ends and you win!

There’s a few multiplayer mechanics you should know about: your opponent’s enemies are your **allies** (there’s **no ‘friendly fire’**), shooting your opponent’s head deals double damage (**‘headshots’**), and colliding with your opponent has some special effects.

Colliding with your **head on your opponent’s body** will kill you if you’re smaller than your enemy. Nothing will happen if you’re both the same size or if you’re bigger.

A **head-on-head collision** with your opponent will damage the both of you.

Remember, for you to collide with your opponent, you have to be moving. You can use the ability to **stop moving** to your advantage.

1.4. Game options

If you feel especially adventurous, you can customize almost all of the game options and some mechanics by tweaking the "*include/game_opts.h*" file. Most options that you can use here are safe but you might get some undesired effects/buggy game-play if you tweak some things to the extreme (e.g.: making objects faster than half of a screen per frame might break some collisions).

If you're looking for an option that is not in this file, it was probably because it was considered unsafe but you'll most likely be able to find it in its respective module header file.

```
/** @brief Skane 2 attack sprite */
#define SKA2_ENEPATH_ATK "/skane2/skane2Ene_Atk.bmp"
/** @brief Skane 2 spawner sprite */
#define SKA2_SPAWNER "/skane2/skane2Spawner.bmp"

/* skane defaults */
#define SKA_X             100 /**< @brief Skane default start X location */
#define SKA_Y             50  /**< @brief Skane default start Y location */
#define SKA_X_2           100 /**< @brief Skane default start X location (2nd) */
#define SKA_Y_2           250 /**< @brief Skane default start Y location (2nd) */
#define SKA_S             5   /**< @brief Skane default start speed location */
#define SKA_HP            30  /**< @brief Skane default start health location */
#define SKA_SHOOTING_COST 1   /**< @brief Skane default shooting cost */
#define SKA1_SPAWN_DIREC  N   /**< @brief Skane 1 start spawn direction */
#define SKA2_SPAWN_DIREC  S   /**< @brief Skane 2 start spawn direction */

/* missle defaults */
#define MIS_SPEED 10 /**< @brief Missile speed. */
#define MIS_DMG   10 /**< @brief Missile damage. */
#define MIS_CD    30 /**< @brief Shotting cooldown (in frames). */
```

Figure 9: Some of the game options available

We advise you take a look at the option below, because it randomizes the game's colors and some people might be sensitive to that (only affects the "Singleplayer mode").

```
/** @brief Set to something between [0, 256] (the higher, the safer) */
#define YOUGOTEPILEPSY  1
```

Figure 10: Color randomizing option

2. Project Status

2.1. Table of I/O devices

Device	Used for	Interrupts
Timer	Controlling frame rate (Update method)	Yes
Keyboard	Player movement and game mode quitting	Yes
Mouse	Cursor movement, menu option selection and shooting	Yes
Video card	Menus and game screen display	No
RTC	Enemy and ally spawning scheduling	Yes
UART	Multiplayer game mode communication	Yes

2.2. Timer

The timer 0 is used to obtain an interrupt when a new frame is should be rendered (timer 1 and 2 aren’t implemented).

Time 0 interrupts are subscribed exclusively at the start of the program in the function **sub_interrupts()** (file *ev_disp.c*).

No configuration of the timer is required, due to the fact that the game was tuned to work at 60 frames per second, timer 0’s default configuration .

All of the interrupts are caught in the **mainloop()** function (file *ev_disp.c*) which calls the timer interrupt handler (**timer_ih(),** defined in *timer.c*).

2.2.1 Most relevant timer functions

File *ev_disp.c*:

- **mainloop() -** calls the interrupt handler;
- **sub_interrupts()** - Subscribes interrupts;
- **quit()** - Unsubscribes interrupts.

File *timer.c*:

- **timer_ih() -** timer 0 interrupt handler;
- **timer_set_freq(uint8_t timer, uint32_t freq) –** attempts to a set a given frequency to a given timer;

2.3. Keyboard

The keyboard is used to get user input. With this device, the user can control his **skane** and exit from a game mode.

Keyboard interrupts are subscribed exclusively and the default MINIX's keyboard controller configuration is restored when the game is closed.

By using interrupts, the game analyzes and updates an **input_array** (a *container* that stores the state of all keyboard keys and mouse buttons) for each key press/release.

In each frame all relevant keys to the game are checked by accessing the input array, which in turn triggers the associated event, e.g.: the Escape key exits to the main menu, the W and A keys, when combined, move the **skane** diagonally (North and West).

As stated previously, each keyboard interrupt updates the game's **input_array**. This is achieved by checking the validity of the new keyboard byte, followed by its scan code assembling and finally updating the **input_array**.

2.3.1 Most relevant keyboard functions

File *ev_disp.c*:

- **mainloop() -** Calls the interrupt handler, contains and updates the **input_array** keyboard information;
- **sub_interrupts()** - Subscribes interrupts;
- **quit() -** Unsubscribes interrupts.

File *kbd.c*:

- **kbd_ih() -** The keyboard interrupt handler;
- **kbd_scancode(bool* make, uint8_t* bytes) –** Assemble a scan code;
- **kbd_restore_conf() -** Restores MINIX's default keyboard controller configuration.

2.4. Mouse

The mouse was used to update the cursor's **position** on the screen, to select menu options (**buttons**) and to shoot projectiles from the player's position to the cursor.

Identically to the keyboard, the mouse also updates the game's **input array** in the same fashion, that is, it is updated when a new mouse packet is ready.

2.4.1 Most relevant mouse functions

File *ev_disp.c*:

- **mainloop() -** Calls the interrupt handler, contains and updates the **input_array** mouse information;
- **sub_interrupts()** - Subscribes interrupts;

- **quit() -** Unsubscribes interrupts.

File *mouse.c*:

- **mouse_ih()** - Mouse interrupts handler;
- **mouse_sync_packet(struct packet *pp)** – Assembles a full mouse packet;
- **mouse_en_data_report(void) –** Enables mouse data reporting;
- **mouse_set_stream_mode(void) –** Set mouse to streaming mode;

2.5. Video Card

The video card was used to render all game's objects and menus.

The project designed for the super **VGA mode 0x107** (**indexed/packed pixel, 1280x1024 pixels, 256 colors**) in particular, although **other indexed video modes are also supported** and static objects positions should be adjusted automatically (you can specify the desired super VGA mode via command line arguments, which will be discussed at the end of this document).

The game was developed with an indexed/packed pixel mode because the copying of image data was faster (less data per sprite) and we were able to **alter the game's color palette** to one defined by us.

The game will also work with direct color modes, but to achieve that the game sprites have to be converted to the desired RGB mode (this process should be fairly straight forward since the **GIMP** projects (.xcf files) for all sprites are provided).

The game uses two video buffers for rendering (**double buffering**) with **page flipping**, if there is enough video memory to allocate both of them (there is enough video memory for two video buffers for all the super VGA modes described in VBE 2.0 standard). Optionally, the game also uses **vsync**. This will be explained in more detail in section 4.

Additionally we also fully implemented (discussed in section 4):

- **Changed color palette DAC format** - (set **truecolor** mode);
- **Collision detection**;
- **BMP** (bitmap v5) file reading;
- **Transform operation** on sprites;

2.5.1. List of VBE functions implemented in the game (*vg_utils.c* file)

- VBE function **0x00** (Return VBE Controller Information) – get controller information (not used);
- VBE function **0x01** (Return VBE Mode Information) – get a given super VGA mode information (used to set current mode information);
- VBE function **0x02** (Set VBE Mode) – set a given super VGA mode (used to set the mode information);
- VBE function **0x06** (Set/Get Logical Scan Line Length) – check if it is possible to extend the logical scan line enough have two buffers and set that scan line length (used to allocate the second video buffer);
- VBE function **0x07** (Set/Get Display Start) – get the display's first scan line and pixel starting point (used for **page flipping**);
- VBE function **0x08** (Set/Get DAC Palette Format) – get the current palette format and set **truecolor** format, if possible (used).
- VBE function **0x09** (Set/Get Palette Data) – get or set the colors in the current color palette (used to set the color palette we want/a random one).

2.5.2. Most relevant video card functions

File *obj_handle.c* (will be discussed later in more detail):

- **render_objects()** - draws all objects in the current buffer handler (each object has its own definition of the draw() virtual function);
- **update_objs_collisions()** - draws all the colliders for all the objects currently in game (each object, that has collisions, has its own definition of the updateCollision() virtual function)

File *skane.c*:

- **skane_diff(Skane_t* ska, bool reset)** function – randomizes the video graphics color palette when the game difficulty increases (reset is **false**) or returns it to the game's default (reset is **true**).

File *vg_utils.c*, which has all the VBE functions described above;

File *vg.c*:

- **vginit(uint16_t mode, bool v_sync)** – Set a given super VGA video mode;
- **next_buffer()** – switches between buffers (page flipping);
- **set_color_palette_file(const char* const filename)** – Sets a color palette from a file;
- **draw_sprite(Sprite_t* sprite, const uint16_t x, const uint16_t y, const uint32_t transp)** – Draws a sprite on the buffer that is currently not shown (page flipping)

2.6. RTC

Although the **RTC** has been fully implemented to work with alarm, periodic and update interrupts, and reading the current data/time, we only use the alarms interrupts and current time reading in the game.

The **RTC** alarm interrupts are used to schedule the enemy spawn.

2.6.1 Most relevant RTC functions

File *ev_disp.c*:

- **mainloop() -** Calls the interrupt handler, contains and updates the current time so it doesn't need to be read more than once;
- **sub_interrupts()** - Subscribes interrupts;
- **quit()** - Unsubscribes interrupts.

File *rtc.c:*

- **rtc_ih()** – RTC interrupt handler;
- **rtc_get_date(rtctime_t* ret)** – get current date;
- **rtc_get_time(rtctime_t* ret)** – get current time;
- **rtc_get_datetime(rtctime_t* ret)** – get current date and time;
- **time_ff(rtctime_t* curr, uint32_t secs)** – moves a time point further in time by a given amount of seconds (takes care of BCD and binary conversions);
- **rtc_set_alarm_ff(uint32_t secs)** – set an alarm a given number of seconds form now;
- **rtc_set_alarm_ff_curr(curr_time, ENEMY_SPAWN_RATE)** – set an alarm a given number of seconds form a given time point;

2.7. UART

The UART was used to communicate between two players when in multiplayer mode. To accomplish this, we used and fully implemented the **serial port's interrupts** and **FIFO** capabilities (64 byte **FIFO** with 1 byte trigger).

In addition, **handshake** functions and a **queue** 'class' were also developed.

2.7.1. Information exchange frequency:

A new packet is sent on the following events:

- A handshake is occurring;
- A **missle** is fired;
- Enemie(s) are spawned;
- A **snake** dies;
- If no information has been sent at the end of a frame, we send a synchronization packet to make sure the other player didn't disconnect;

This mean that, in the median case, we send around 80 bytes per second to each side (160 bytes in total).

2.7.2. Communication parameters:

The project uses the standard **8 bit per character, 1 stop bit and no parity** for communication. The **divisor latch** value is **0x01** (LSB: 0x01 and MSB: 0x00) so we get the maximum **baud rate**.

2.7.3. Most relevant UART functions

File *ev_disp.c*:

- **mainloop()** – calls serial port interrupt handler and related functions;
- **com_handler()** – receives and treats incoming packets;
- **multiplayer_handshake()** - subscribes serial port interrupts, configures it and then waits for another player to try to connect and selects which player will be player 1 and player 2;
- **exit_to_main_menu()** - Unsubscribes interrupts if a multiplayer game was occurring;

File *obj_handle.c*:

- **spawn_enemy(gamestate gamest)** – spawns enemies and sends how many enemies were spawned to the other player (in case the game is multiplayer);
- **transmit_skane_info()** - Transmits the new **skane** information to the other player (only transmits information that changed since the last frame);

File *serial.c:*

- **serial_send_push(uint8_t data)** – Push the given byte into the serial port send queue.
- **serial_send_push_float(float data)** – Push the given float into the serial port sending queue.
- **serial_send_all()** - Sends all bytes in the send queue.
- **serial_get_data()** - Read all data from the controller’s queue.
- **serial_ih()** - Serial port interrupt handler.
- **serial_check_lsr()** – Checks the line status register for errors and handles them. Also sets the transmission available flag if the “transmitter is holding empty register”. Returns 1 if there’s data available for reading.

- **serial_restore_conf()** - Restore the MINIX's default serial port configuration;

3. Code Organization/Structure

3.1. Timer (*timer.c* file) – 2%

Where all timer related functions reside. Most functions were kept and/or adapted from lab2. It contains functions to set/read a timer's configuration, timer 0's interrupt handler and functions to get the current number of **ticks** (timer 0 interrupts) and reset the **tick** counter.

Implemented by both members simultaneously.

3.2. KBC Utilities (*kbc_utils.c* file) – 2%

Where all **KBC** controller functions reside. Most functions were kept and/or adapted from lab3 or lab4. It includes functions to write commands to the **KBC**/mouse controller and to set/restore the **KBC's** configuration.

Implemented by both members simultaneously.

3.3. Keyboard (*kbd.c* file) – 3%

Where all timer related functions reside. Most functions were kept and/or adapted from lab3. It contains functions to read/assemble the keyboard's scan codes and the keyboard interrupt handler.

Implemented by both members simultaneously.

3.4. Mouse (*mouse.c* file) – 4%

Where all mouse related functions reside. Most functions were kept and/or adapted from lab4. It contains functions to read/assemble the mouse's packets (using **lcf's packet struct),** to set the mouse configuration (such as enabling/disabling stream mode or data reporting) and the mouse interrupt handler.

Implemented by both members simultaneously.

3.5. Bitmaps (*bmp.c* file) – 5%

Handles the creation of sprites by importing a **BMP** file's data (bitmap v5). It also contains sprite shearing and rotation functions (any number of degrees or number of 90º steps).

3.5.1. Data structures

- **BMPFileHeader_t** – bitmap file format header.
- **BMPV5Header_t** – bitmap image information.

Implemented by João Costa.

3.6. Video graphics (*vg.c* file) – 8%

Some functions were kept and/or adapted from lab5. It contains a set of global static variables associated with the current video mode (horizontal/vertical resolution, pointers to video buffers, etc...).

All the functions to render sprites are defined here. All higher-level functions that change the color pallete or perform page flipping are also within this file.

3.6.1. Data structures

- **Sprite_t** – A sprite object. Has a width, height and data fields. The data field is an array that contains all the pixels information from left to right, top to bottom;

Implemented by both members simultaneously.

3.7. Video graphics utilities (*vg_utils.c* file) – 5%

Where all **VBE 2.0** controller functions reside. Some functions were kept from lab5. It only contains functions that call the **sys_int86(reg86_t*)** function to interact with the **VBE 2.0** functions such as changing the color palette, getting controller information, set/ get video mode, set/get display start, etc…

3.7.1. Data structures

- **VbeInfoBlock_t** – Used to read and store a graphics controller info;

Implemented by João Costa.

3.8. RTC (*rtc.c* file) – 4%

Where all **RTC** related functions reside. This module contains functions for: enabling/disabling all types (alarm, update and periodic) of **RTC** interrupts (either individually or globally); The **RTC** interrupt handler; Get the current date and time (either individually or at the same time); Move a **rtctime_t** struct a given number of seconds forward in time, and set alarms to a given number of seconds in the future or a given number of seconds from a given **rtctime_t** struct in the future.

3.8.1. Data structures

- **rtctime_t** - Used to store a date and time point by the RTC;

Implemented by João Costa.

3.9. Serial port (*serial.c* file) – 10%

Where all serial port related functions reside as well as the two queues: **send_queue** and **receive_queue**. It contains wrappers to control data in said queues (send all data, read data, etc ...), functions to configure the **UART** and its **FIFO**, to perform an '**handshake**', check data errors, read/pushing data from/into the **UART** and the serial port interrupt handler.

Implemented by João Costa.

3.10. Vector (*vector.c* file) – 2%

Where the '**C++ stl vector class**' like implementation of our vector struct/object is found. It contains all functions to manipulate (insert, delete, modify, push_back, pop_back, etc…) a vector.

3.10.1. Data structures

- **vector** – Vector struct/object.

Implemented by João Costa.

3.11. Queue (*queue.c* file) – 2%

Where the 'C++ stl vector class' like implementation of our queue class is found. It contains all functions to manipulate (push, pop, etc…) a queue.

3.11.1. Data structures

- **queue** - Queue struct/object;

Implemented by João Lucas.

3.12. Object (*object.c* file) – 9%

Where the object base class is implemented.

It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **updatePos(void*)**, **render(void*)** and **updateColl(void*)** virtual functions. All derivations of the object ‘class’ have a defined instance of all these structs , except for **Derived_obj_t**. This aspect will be discussed further in the report.

3.12.1. Data structures

- **Object_Identifier_t** – Used to identify the **type** and **id** of an object;
- **Object_Vtable_t** – Contains pointers to all **virtual functions** of an object
- **Object_t** – Contains all data-members related to all objects.
- **Derived_obj_t** – Used to cast a void pointer (of a derived ‘class’) to extrapolate either its **object pointer** or **virtual table**.

Implemented by João Lucas.

3.13. Collisions (*collisions.c* file) – 3%

Where all possible object collisions are defined. In it the **collision_dispatcher(void*, void*)** is defined, which identifies two given objects and calls the correct collision function for their interaction.

Implemented by both members simultaneously.

3.14. Cursor (*cursor.c* file) – 2%

Where the cursor object, derived from the object base class, is defined. It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **render(void*)** and **updateColl(void*)** virtual functions. It also has the **updateCursor(Cursor_t* cursor, int16_t delta_x, int16_t delta_y)** function, which updates a cursor with the given deltas (usually from mouse packets).

3.14.1. Data structures

- **Cursor_t** – Cursor object;

Implemented by João Lucas.

3.15. Missile (*missile.c* file) – 2%

Where the missile object, derived from the object base class is defined. It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **updatePos(void*)**, **render(void*)** and **updateColl(void*)** virtual functions.

Other types of projectiles could be created by simply instantiating this object with a different sprite/default values (no changes to the code needed).

3.15.1. Data structures

- **Missle_t** – Missle object;

Implemented by João Lucas.

3.16. Enemies (*enemies.c* file) – 2%

Where the enemy object, derived from the object base class is defined. It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **updatePos(void*)**, **render(void*)** and **updateColl(void*)** virtual functions.

3.16.1. Data structures

- **Enemy_t** – Enemy object;

Implemented by both members simultaneously.

3.17. Food/Strawberries (*food.c* file) – 1%

Where the food object, derived from the object base class is defined. It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **updatePos(void*)**, **render(void*)** and **updateColl(void*)** virtual functions.

Other types of food could be created by simply instantiating this object with a different sprite/default values (no changes to the code needed).

3.17.1. Data structures

- **Food_t** – Food object;

Implemented by João Lucas.

3.18. Walls (*wall.c* file) – 2%

Where the wall object, derived from the object base class is defined. It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **updatePos(void*)**, **render(void*)** and **updateColl(void*)** virtual functions.

3.18.1. Data structures

- **wall_type** – Enumerator that defines the possible types of walls (defines the collision directions);
- **Wall_t** – Wall object;

Implemented by João Lucas.

3.19. Menus/buttons (*menu.c* file) – 2%

Where the menu object, derived from the object base class is defined. It contains a set of macros that define the position of each menu/button. It also contains its own definitions of the **destroy(void*)**, **print(void*) and render(void*)** virtual functions.

3.19.1. Data structures

- **Menu_t** – Menu button/image object;

Implemented by João Lucas.

3.20. Skane (*skane.c* file) – 7%

Where the **skane** object, derived from the object base class is defined. It contains functions designed to perform a specific action in a given snake (move, take damage, fire **missle** and eat food). It also contains its own definitions of the **destroy(void*)**, **print(void*)**, **updatePos(void*)**, **render(void*)** and **updateColl(void*)** virtual functions.

3.20.1. Data structures

- **direc** – Enumerator that defines the possible **Skane** states;

- **seg** – Represents a segment of a **skane's** body (segments are delimited by direction changes);
- **ska_sprt_t** – Holds all the sprites needed for a **skane**, its **missles**, its enemies and enemies dropped food (in this case a **strawberry**);
- **enemy_diff** – Holds the current difficulty scaling options;
- **Skane_t** – A **skane** object;

Implemented by both members simultaneously.

3.21. Object Handler (*obj_handle.c* file) – 7%

Maintains and controls all objects in the game as well as the game's collision matrix. It contains functions to add/remove objects, to instantiate/free the project's containers (the objects vector and the collision matrix) and some wrappers to access some object's proprieties (e.g.: to check if a snake can shoot or to get the cursor's position).

The menu objects are also instantiated and contained here.

This is used to abstract the event dispatcher from the object internals.

Implemented by both members simultaneously.

3.22. Event dispatcher (ev_disp.c file) – 13%

Maintains and controls the game state, manages the interrupts and the interrupt handlers to be called, calls the functions to handle each event, and manages the multiplayer communication.

The menu objects collisions are handled here.

Implemented by both members simultaneously.

3.23. Utilities (*utilities.c* file) – 1%

Contains general-use utility functions such as the scalar product, wrappers to subscription/unsubscription of interrupts and **sys_inb(uint8_t, uint32t*)**, macros to convert between decimal and **BCD**, etc…

Implemented by both members simultaneously.

3.24. Error handling/feedback utilities (*err_utils.c* file) – 1%

Includes functions to write warnings and errors to a log file and other more general user communication/feedback functions.

Implemented by João Costa.

3.25. Program main function/command line arguments parser (*skane_royale.c* file) – 1%

Parses commands line arguments (if any) and calls the game **init()** functions if everything seems alright.

Implemented by João Costa.

3.26. Main loop function (calls driver_receive()) call graph (cut down version)

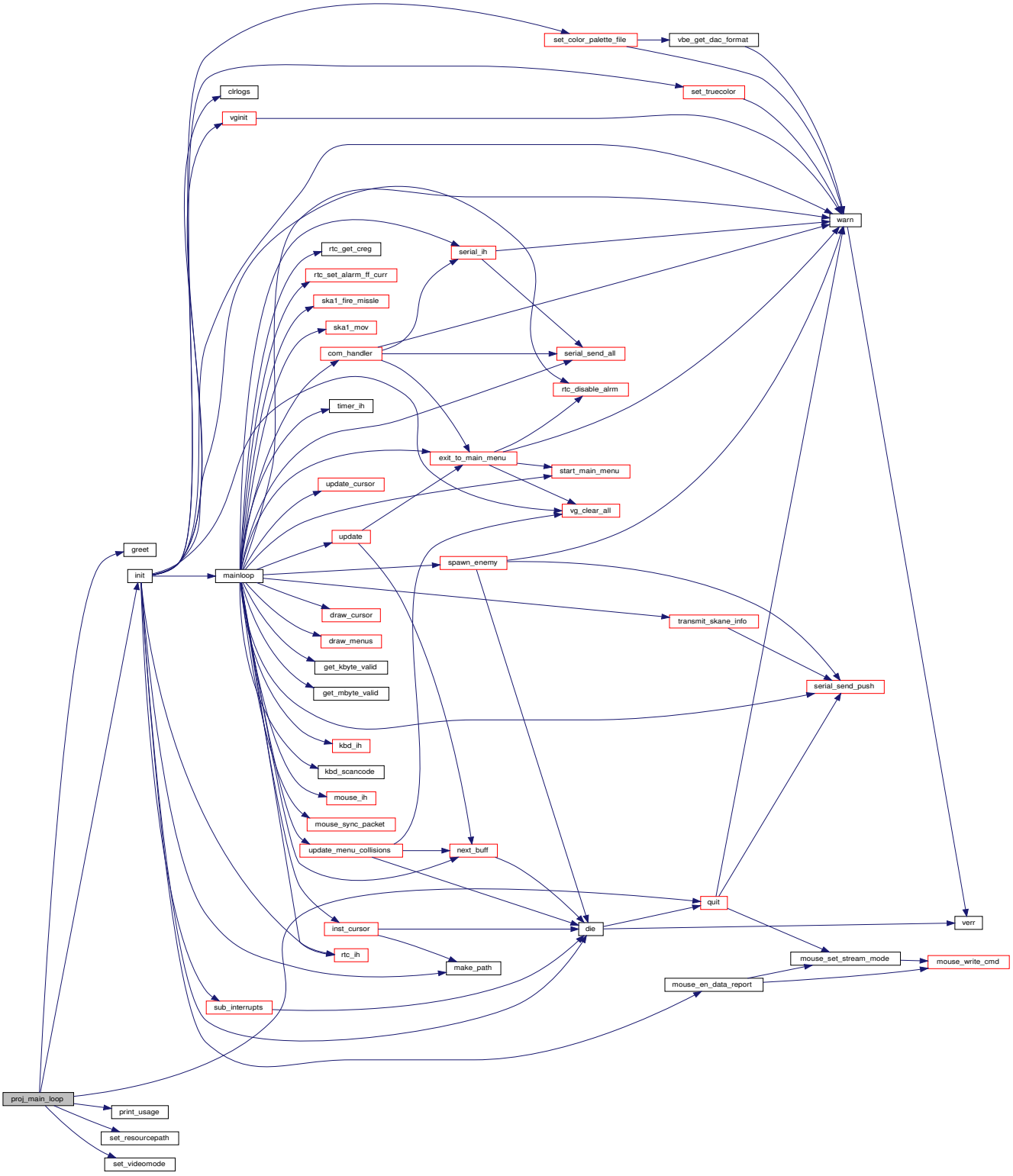


Figure 11: Project function call graph

3.26.1. Main functions short description

- **init()** - Initializes the game routines;
- **sub_interrupts()** - Subscribes all interrupts (except serial port becomes serial isn’t always needed)
- **vginit(uint16_t mode, bool v_sync)** – Set a given super VGA video mode (with **vsync** optionally);
- **quit()** - Safely quit the game (unsubscribes all interrupts, frees all dynamically allocated memory, restores MINIX’s default configurations and sets the system back to text mode);
- **mainloop()** - Receives interrupts from the **driver_receive()** function and calls the correct **interrupts handlers** and game handling functions;
- **com_handler()** - Parses information read from the serial port (stored on the *serial.c* **receive_queue**) until an end of frame signal is received or the game times out. Also tries to send the packets queued for sending on the *serial.c* **send_queue**;
- **next_buff()** - Switches currently shown video buffer (**page flipping**);
- **update()** - Updates object’s colliders, checks for collisions, updates object’s positions, renders them and switches video buffers (**next_buff()**);

4. Implementation details

4.1. Generic containers (vector and queue)

A C++ like implementation of the vector class was made to better maintain multiple game objects, using dynamic memory allocation (*vector.c*). This allows for the game to theoretically have an unlimited number of objects, while also making the addition/removal of new objects and access to old ones easy.

Likewise, a queue module (*queue.c)* was added to avoid streamline the reading and transmission of data when using the UART.

4.2. Layering

The objects vector, defined in *obj_handle.c*, is made up of several vectors, one for each type of game object, which in turn are composed of void pointers to objects of their respective object type. The order of the types (layer) is specified in the header file *object.h* by the **obj_type** enumerator.

This approach aids us when deciding which type of objects should be drawn on top of others. For instance, if we wanted food to be drawn always on top of snakes, we could easily accomplish this by simply setting the FOOD type to be higher than the SKANE type. In this way the food objects would be rendered after the skane objects.

```
typedef enum OBJ_TYPE {
  WALL       = 0, /**< Wall type */
  SKANE      = 1, /**< Skane type */
  FOOD       = 2, /**< Food type */
  ENEMY      = 3, /**< Enemy type */
  MISSILE    = 4, /**< Missle type */
  SKANE_BODY = 5, /**< Skane body type */
  CURSOR     = 6, /**< Cursor type */
  NOT_SET    = 7, /**< @brief Undefined type (should be the highest tag set) */
  MENU       = 8  /**< @brief Menu type always (above NOT_SET)*/
} obj_type;
```

Figure 12: obj_type enumerator type

4.3. Object Oriented Programming, Polymorphism and Garbage Collection

4.3.1 Object Oriented Programming

As to help the creation and maintenance of game objects, the group decided to take an **OOP** approach to the game. To achieve this we implemented the module *object.c* as a ‘base class’, where all generic game object ‘data-members’ reside. Each derived object has a pointer to an instance of an **Object_t**. This allows access to the base class data-members and data-functions (virtual functions present in the **vtable** which will be explained further).

Furthermore, the identification of an object is possible through **Object_Identifier_t**, containing the **id** and **type** of the object. An id is unique for an object type. This is used later when handling collisions or checking for objects flagged for deletion (‘garbage collection’).

The following list contains all the derived objects developed for the project:

- Cursor
- Enemy
- Food
- Menu
- Missle
- Skane
- Wall

4.3.2 Polymorphism

The inclusion of the **vector** module allowed us to implement virtual functions to all objects and apply polymorphism in the event dispatcher module using a **vector** of void pointers.

This was done using a **vtable**, **Object_Vtable_t**, a struct of pointers to virtual functions defined in *objects.h*. The file also contains a group of ‘wrapper functions’, one for each virtual function, which call the correct virtual function of a given void pointer.

This causes a massive simplification of the event dispatcher’s code. For instance, in order to render all objects we only need to iterate through all objects and call the **render(void*)** function (as long as the ‘wrapper function’ for rendering has been correctly defined for that object).

The following list contains all virtual functions of the game objects:

- **draw(void*)** - Draws the object sprite in its current state (position, rotation, animation state, etc...) on the video buffer;
- **updatePos(void*)** - Updates a given object’s position (coordinates);
- **destroy(void*)** - Destroys a given object (frees all its memory);
- **updateCollision(void*)** - Updates a given object’s collider on the collision matrix;
- **print(void*) -** used for debugging purposes;

Each derived object has its own definition of the object virtual table with its corresponding virtual functions. This table is then assigned in each creation of said derived object. Some of the derived virtual functions may call the base virtual function, by accessing and calling the respective virtual function of its base object pointer.

4.3.3 Garbage Collection

During the development of the project the group decided that the deletion of the game's objects should be the last operation in each frame, to avoid undefined behavior. Thus, when the game decides to free an object it tags its ID to 0.

Afterwards, in the end of each frame, the garbage collector detects all objects with ID set to one and frees them from memory.

4.4. Collisions

Handled by a collision matrix, each element of the matrix representing a pixel on the screen and pointing to the object that is present in that pixel. The **updateCollision()** virtual function updates said collision matrix with a given object and calls the **collision_handler()** (defined in the *collisions.c file*), which identifies both objects and calls the function that should handle the collision.

4.5. Error utilities

To better maintain and debug the code the group decided it would be better to fully implement a set of utilities that write error/warning messages to a log file (similar to the **lcf's trace.txt**). We quickly found, when calling a lot of functions, the **trace.txt** was often too convoluted with function calls irrelevant to what we were trying to see.

For this, 3 functions were developed to better detect the causes of problems in our code and terminate the execution if necessary:

- **warn(const char* fmt, ...)** - Writes the given arguments as "warnings" to the log file.
- **die(const char* fmt, ...)** - Writes the given arguments as "fatal errors" to log file and **safely** terminates the execution of the program.

- **clrlogs()** - Clears the log file.

By default, the log file is located in the ***/tmp*** directory of the root file system (meaning it will be deleted on each reboot). The log file location can be changed on the *err_utils.h* header file.

4.6. Graphics and rendering details

4.6.1. Bitmap file reading

We implemented a function to read bitmap files (**bitmap v5**) with no compression. It supports both positive and negative image heights (negative height images mean the sprite should be read from the top to the bottom).

This function was used to read bitmap files in **8 bit indexed mode** for our project but it will work with any bitmap v5 files with no compression.

The reading of bitmap files (binary files) prompted the creation of 2 new utility functions (*utils.c* file):

- **fskip(FILE* fp, uint32_t num_bytes)** – Skips a file pointer by a given number of bytes (or until **EOF** is reached);
- **fskip_until(FILE* fp, char search_byte)** – Skips a file pointer until a given byte is found (or until **EOF** is reached);

4.6.2. Sprite transforms – Sprite shearing

There are two functions that apply shearing transformations to sprites in our project:

- **shearX_sprite(Sprite_t* ori_sprite, float shear)** – Shears a sprite by a given amount in the horizontal direction;

- **shearY_sprite(Sprite_t* ori_sprite, float shear)** – Shears a sprite by a given amount in the vertical direction;

By combining a shear in X, followed by a shear in Y and, lastly, following that by a shear in X, it is possible to rotate a sprite. The "**ImageMagick**" software suit uses this method for rotating sprites in multi-threaded applications (not our case) so we stocked with the more 'traditional' rotation methods.

4.6.3. Sprite transforms – Sprite rotation

There are two functions that apply rotation transformations to sprites in our project:

- **rotate_sprite(Sprite_t* ori_sprite, float angle)** – Rotates a sprite by a given number of degrees (radian units);
- **rotate_sprite_intPI(Sprite_t* ori_sprite, int8_t num_turns)** – More efficient than the previous one but can only rotate by a given number of 90º turns (clockwise: negative **num_turns** or counter-clockwise: positive **num_turns**);

4.6.4. Sprite copying

We have a function that copies a sprite (allocating the memory needed for it):

- **sprite_cpy(Sprite_t* orig)** – Copies the given sprite.

4.6.5. Page flipping and vsync

As mentioned before, the project uses **page flipping.** To do this, the game extends the screen's logical scan line to double its length and uses the VBE function **0x07** ("Set/get display start function") to change the starting pixel of the scan lines. This way, we can switch video buffers almost instantly each frame.

Optionally, the game uses **vsync** by calling this VBE function using the option **0x80** (Set display start during vertical retrace) instead of option **0x00** (Set display start).

The most relevant functions are:

- **next_buff()** - switches the current videos buffer (*vg.c* file);
- The “draw” set of functions – draw sprite/line/rectangle on the ‘hidden buffer’ (*vg.c* file);
- **vbe_set_scanline_psize(uint32_t* psize)** – sets a new logical scan line size;
- **vbe_set_display_start(uint32_t ipixel, uint32_t iscanline, bool vsync)** – sets the new start scanline, start pixel and optionally does the operation with **vsync** (*vg_utils.c* file);

4.6.6. Color palette and DAC format

Since we used an indexed/packed pixel super VGA mode, we decided to change the color palette in order to have access to all the **RBG** colors.

We started by setting **truecolor** mode (using **VBE 2.0 function 0x08 with option 0x00**) to have access to the full **RGB** range.

After that, we use **VBE 2.0 function 0x09 with option 0x00** to set a new color palette for the game. The function **set_color_palette_file(const char* const filename)** reads color palette information from a file and sets it for the game.

The color palette file is a binary file with the following format:

- the first byte is the number of colors to read (N)
- the second byte is the index of the first color to change (I)
- the following N integers represent each color of the palette (both default and **truecolor** modes are supported).

To help with the creation of this file, we provided two python scripts and one shell script (to connect the other two) that converts a color palette created in **GIMP** to the format used by our program.

There's also a function that randomizes the color palette based on a game setting and state (talked about later). The colors are randomized (if the user wishes them to) when in single player mode and a new **round** starts:

- **set_random_color_palette(uint8_t palette_size, uint8_t first_color_ind)** – Randomizes color palette (*vg.h file*).

4.6.7. Sprite animations

```
/** @struct SKA_SPRT_T
 *  Group of sprites relating to a skane.
 */
typedef struct SKA_SPRT_T
{
  Sprite_t h_sprite; /**< Skane's head sprite. */
  Sprite_t b_sprite; /**< Skane's body sprite. */
  Sprite_t t_sprite; /**< Skane's tail sprite. */
  Sprite_t m_sprite; /**< Skane's missle sprite. */
  Sprite_t f_sprite; /**< Skane's enemies dropped food sprite. */
  Sprite_t ene_sprite[ENE_ANIMCYCLE]; /**< Skane's enemy sprites. */
} ska_sprt_t;
```

Figure 13: Container for the sprites related to a Skane

For animated sprites, the game uses an array of sprites and constant macro values to know when to change sprites (a 'mini state machine').

In this project the enemies are the only object with animations, besides the real-time **Skane's head sprite rotation.**

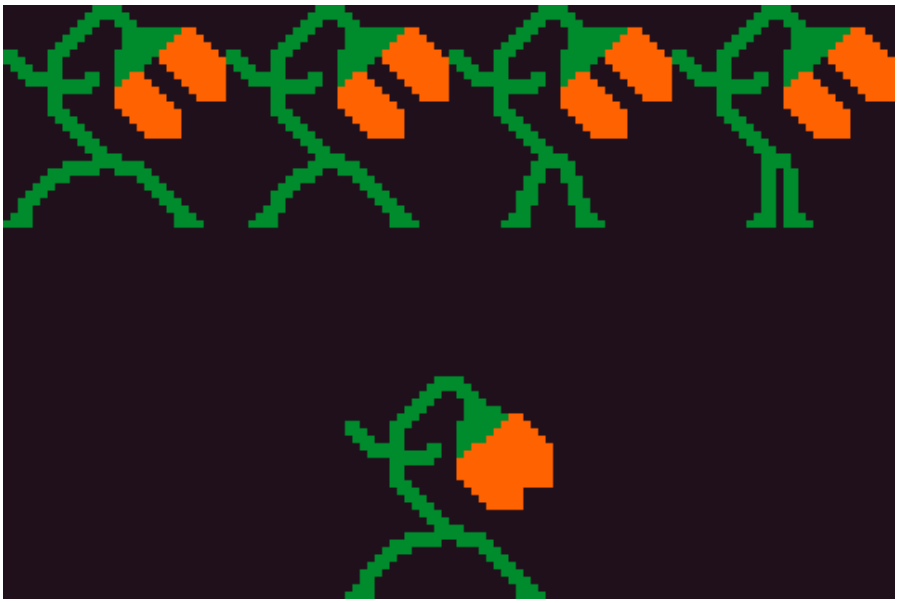


Figure 14: Enemy animation cycle

The upper 4 sprites represent the enemy **walking cycle** (the enemy goes from the **first** sprite to the **fourth** and then from the **fourth** to the **first**).

The enemy reaches the fifth sprite when he **bites/damages** a **Skane**. When he reaches that state, he'll stay immobile, in that state, for a number of frames, **n**. After **n/2** frames, the enemy will return to his first sprite and after **another n/2**, the enemy will restart his **walking cycle**. This is the **biting/attacking animation of the enemy object**.

```
#define ENE_ANIMCYCLE     5 /**< @brief Number of skane enemy sprite */
#define ENE_ANIMCYCLE_T   3 /**< @brief Frames between enemy sprite changes */
#define ENE_ATK_ANIMCYCLE 2 /**< @brief Number of enemy attack sprites */
/** @brief Frames between enemy attack sprite changes */
#define ENE_ATK_ANIMCYCLE_T 10
```

Figure 15: Constants that determine the behavior of the enemy's animation cycles

4.7. Input Array

The **input_array** type is a **Boolean** array used by the game to store the user's pressed keyboard keys and mouse buttons. In the reception of both full and valid mouse **packets**

and/or keyboard **scan codes** (1 or 2 byte **scan codes**) the input_array is updated for the respective key.

The **input_array** is aided by an enumerator type **input** with matches the value of an **hash** function like macro to an index of the array.

The indexes of the **input_array** are set when receiving a **make code** from the keyboard or a pressed button **Boolean** from the mouse packets and unset when receiving a **break code** from the keyboard or a released button **Boolean.**

4.8. Skane state machine

A **Skane's** state determines the direction of the movement on the current frame. To help selecting, identifying and transitioning between each state, we make use of an enumerator **direc** type. Each state in this enumerator has been carefully given an integer value so that the sum of multiple directions/states always results in a valid state.

```
/** @enum SKANE_DIREC
 *  Direction that the skane will move on the next frame (state of the skane).
 */
typedef enum SKANE_DIREC {
  E    = 2,  /**< East direction */
  N    = -3, /**< North direction */
  W    = -2, /**< West direction */
  S    = 3,  /**< South direction */
  NE   = -1, /**< North-East direction */
  NW   = -5, /**< North-West direction */
  SE   = 5,  /**< South-East direction */
  SW   = 1,  /**< South-West direction */
  STOP = 0   /**< Indicates the skane didn't move */
} direc;
```

Figure 16: Skane's direc enumerator type

For example, if the Skane, when analyzing the input array, verifies that both the D key (move right) and W key (move up) are pressed, its direction will be the result of the sum: **STOP + E + N = 0 + 2 – 3 = -1 = NE**, which is the correct state we wanted.

```
ska->curr_state = STOP;

if (input_array[d])
  ska->curr_state += E;
if (input_array[w])
  ska->curr_state += N;
if (input_array[a])
  ska->curr_state += W;
if (input_array[s])
  ska->curr_state += S;
```

Figure 17: Calculate new state

In another case, if the D key, the W key and the A key (move left) are all pressed, the resulting sum will be: **STOP + E + N + W = N**, which means opposite directions cancel each other.

With this information we can easily **rotate the Skane's head sprite** in the direction it'll move on the current frame (using the **sprite rotation** functions discussed above).

4.9. Skane body position

```
/** @struct SKANE_SEGMENT
 *  Segment of a skane.
 */
typedef struct SKANE_SEGMENT
{
  size_t len; /**< Length of the skane's segment. */
  direc dir;  /**< Direction of the skane's segment. */
} seg;
```

Figure 18: Skane's seg struct type

Each **Skane** has a vector of **seg** type structs (commented in section 3). Each **seg** contains a skane state/direction, **direc**, and a length (number of frames moving in that direction).

Each frame, the game iterates over that vector and starts drawing (from the head) block of each segment length and direction. When drawn one after another, the blocks form the body of the **Skane**.

Each frame we decrement the length of the last segment by 1 (and pop it from the vector if the length reaches 0) and increment the first segment by 1 if the **Skane** didn't change

states. However, if the **Skane** changed states, we insert a new segment of length 0 as the new first segment of the body.

This method of storing and drawing each **Skane's** body is really efficient, even for big **Skanes**, but makes working with varying speeds a bit more complex (we would have to store the length moved instead of the number of frames moved in a given direction).

4.10. Game state machine

The game can be in one of 4 states:

- **SINGLE** – User is in singleplayer mode.
- **MULT1** – User is in multiplayer mode and is the player 1.
- **MULT2** – User is in multiplayer mode and is the player 2.
- **MENUST** – User is within the main menu.

The main menu (**MENUST**) is the game's default state when launched. This state is also reached when a game ends (a snake dies, user presses **Escape key** or the enemy snake dies, in case of multiplayer).

The singleplayer state (**SINGLE**) is reached when the "singleplayer" is selected on the main menu. After the game initialization process (object instantiation, etc...), the **mainloop()** function analyses the current game state and behaves according to it (calling the correct functions to handle the game).

When both users select the "multiplayer" option in the main menu, an 'handshake' process will be triggered. If successful (doesn't timeout/fail), the player ID will be determined and represented by the **MULT1** or the **MULT2** game state, otherwise the game will remain in the **MENUST** state.

4.11. RTC reading date/time

Many functions for the **RTC** have been implemented and tested for our project but ended up not being used. None of the 'not used' functions are very important or complex so their details can be found on the **doxygen documentation** for the **RTC module**.

The only functions that should be discussed are the ones that read the date and/or time, e.g.: **rtc_get_datetime(rtctime_t* ret)**.

To store the current date and/or time in a format that could easily be used with the **RTC**, we defined the **rtctime_t** mentioned in section 2.

This type is a **struct** that can contain the following information:

```
/** @struct RTCTIME_T
 *  Struct used to store a date and time point by the RTC. */
typedef struct RTCTIME_T
{
  uint8_t secs;  /**< @brief Current number of seconds [00-59]. */
  uint8_t mins;  /**< @brief Current number of minutes [00-59]. */
  uint8_t hours; /**< @brief Current number of hours [00-23]. */
  uint8_t dow;   /**< @brief Current day of the week [01-07]. */
  uint8_t dom;   /**< @brief Current date of the month [01-31]. */
  uint8_t month; /**< @brief Current month [01-12]. */
  uint8_t year;  /**< @brief Current year [00-99]. */
} rtctime_t;
```

Figure 19: rtctime_t definition

When attempting to the current date and/or time, we do the following (*rtc.c* file functions):

- Check is the **RTC** is currently updating its date/time information register by reading **RTC's A register** and checking bit 7 - function **rtc_is_update_in_progress()**;
- If this fails, we repeat the first step after 61 microseconds. We repeat this step a maximum of 4 times - maximum of 244 microseconds of waiting.

- If after 4 tries, there's still an update in progress, we return from the function signaling a failed operation (so we don't get potentially stuck in case of a problem);
- After checking that there's no '**update in progress**', we inhibit **RTC** updates by setting the bit 7 of the **RTC B register** – function **rtc_inhibit_update()**;
- Now we read all the information requested (either date, time or both) from the respective **RTC** registers;
- At the end of this progress, we allow the **RTC** to update again by unsetting the bit 7 of the **RTC B register** – function **rtc_allow_update()**;

4.12. UART communication packet specification

For communication, we use data packets with a header. The header represents the type of packet we're getting, it's size (in bytes) and whether or not that was the last packet on a given frame.

1	0	1	S	X	X	X	X

Bits of a packet header.

If the bit S is 1, i.e. bit 4 is set, the next packet is the last packet to read and parse in the current game frame.

The X frames denote the type of packet received (4 least significant bits) :

- 0001 (0x1) – **Skane** changed states (followed by 1 byte representing the new state)
- 0010 (0x2) – **Skane** shot a **missle** (followed by 8 bytes representing the cursor position when shooting)

- 0011 (0x3) – Enemies spawned (followed by 1 byte representing the enemy group size)
- 1101 (0xD) – Skane died (followed by 0 bytes)
- 1110 (0xE) – Empty packet (used to confirm ‘handshakes’) (followed by 0 bytes)
- 1111 (0xF) – Sync packet (used for starting ‘handshakes’ and ending frames that have no data) (followed by 0 bytes)

5. Conclusion

5.1. The bad

We found the documentation and theoretical classes about the serial port to be of lower quality when compared to the rest of the courses’ materials. The group also had some problems related to lab5 (video card) as we often found contradictory information coming from different teachers and classes and the LCOM’s MINIX updates were extremely delayed for some.

We also had some issues with the practical test as we didn’t have access to the MINIX’s image during the first hour of the test, the path to the folder to mount wasn’t explicit and there were very misleading errors on the test html page.

5.2. The good

During the labs and especially during the development of the project, we learned a lot of new concepts about I/O devices and had the opportunity to study some concepts C programming language.

6. Sources/Inspiration

6.1. Sources

All the game sprites have been drawn by us.

We haven’t copied any code from the internet for the project (or the labs), besides what was given to us for the LCOM’s course by the teachers, but we’ve read articles/books about some of the course’s topics and feel like they deserve to be mentioned:

- http://www.brackeen.com/vga/index.html - 256-Color VGA Programming in C;
- http://www.phatcode.net/res/224/files/html/index.html - Michael Abrash's Graphics Programming Black Book Special Edition;
- https://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming - 8250 UART Programming;
- https://www.codeproject.com/articles/108830/inheritance-and-polymorphism-in-c - Inheritance and Polymorphism in C;

6.2. Inspiration

The base inspiration for the game was an old project developed by a member of the group: **Skane, a snake clone in bash**. That game can be found in the scripts directory of the project’s repository.

The sprites have been inspired by a game called **GONNER** which can be found on:

- https://www.gonnergame.com/

7. Appendix

To compile the game it often is enough to call:

```
make clean && make
```

on the game's **src/** directory.

If you changed the default resources folder path to the path you put the game directory on, you can just start the game by calling:

```
lcom_run proj
```

With this, the game also accepts some command line arguments (optionally):

- The first argument is a string containing the full path to the game's resource directory, e.g.:

  ```
  lcom_run proj "/home/lcom/labs/proj/src/resources/"
  ```

- The second argument is an integer (in hexadecimal format) defining an alternative super VBA mode to run game at, e.g.:

  ```
  lcom_run proj "/home/lcom/prj/src/resources/ 0x107"
  ```

It just be noted that trying to pass the second argument (video mode), presuppose that the first argument (game's resource folder full path) was also passed.
If the passed resources folder path is incorrect (doesn't exist or doesn't contain all necessary files), the game will quit and warn the user about the problem.

The game might behave unexpectedly if the game sprites used don't correspond to the video mode used, e.g.: sprites using 8:8:8 RBG image format and the game set on an index/packed pixel video mode.