LCOM

Minix Arcade

(Road Bomber)

Afonso Carvalho Pereira de Sá **up201604605**

Nelson Alexandre Saraiva Gregório **up200900303**

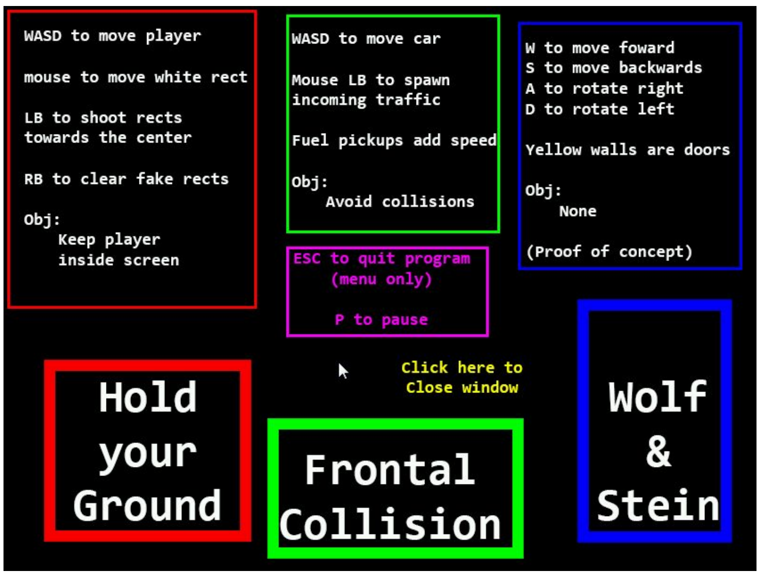
Introduction

In this project we have made three arcade style games for Minix:

- Hold your ground – Player 1 must keep within the screen boundaries whilst player 2 shoots boxes that push the player off screen.
- Frontal collision – Player 1 controls a car that is trying to avoid hitting cars that are being placed by player 2.
- Wolf & Stein – The player is put in a maze and can explore a Wolfenstein like game (this game is a proof of concept, there are no objectives).

User's instructions

The main menu allows you to choose between the three games by pressing their boxes or see the rules for each game. It also has a clock that displays the current time and date.

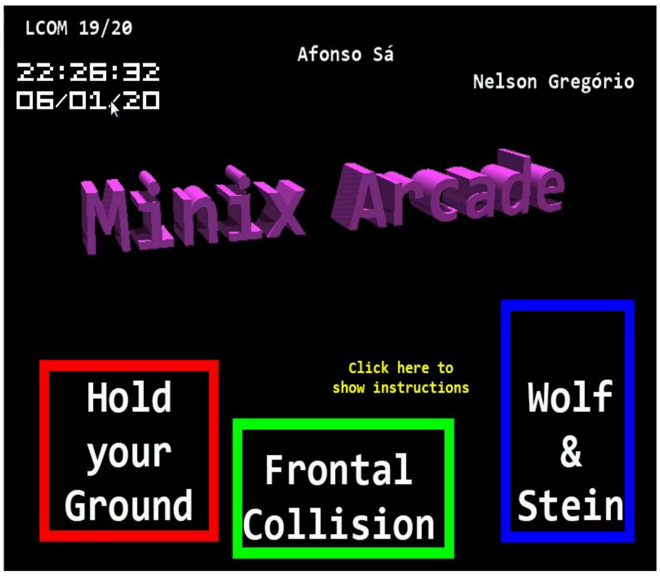


```
WASD to move player

mouse to move white rect

LB to shoot rects
towards the center

RB to clear fake rects

Obj:
    Keep player
    inside screen
```

```
WASD to move car

Mouse LB to spawn
incoming traffic

Fuel pickups add speed

Obj:
    Avoid collisions
```

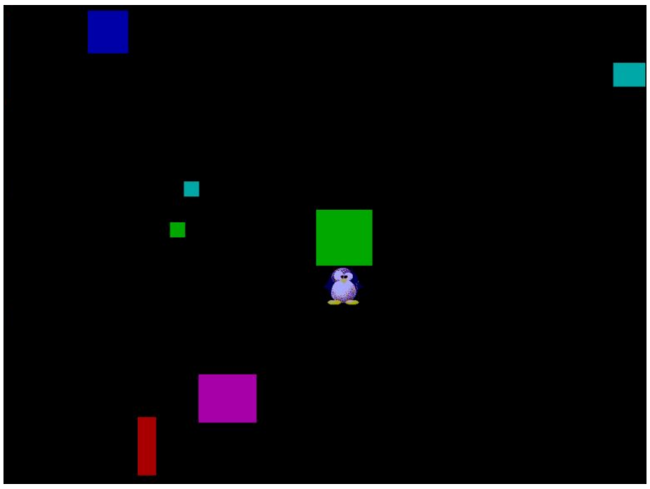
```
W to move foward
S to move backwards
A to rotate right
D to rotate left

Yellow walls are doors

Obj:
    None

(Proof of concept)
```

```
ESC to quit program
    (menu only)

    P to pause
```



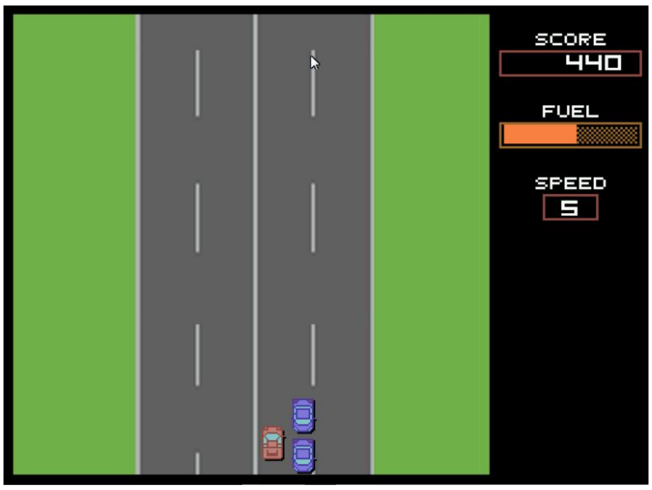
Hold you ground:

Player 1 must avoid being pushed by the boxes that are being placed by player 2 in order to keep himself inside the screen area. Player 1's movement is controlled by WASD and Player 2 must click the screen in order to shoot boxes.



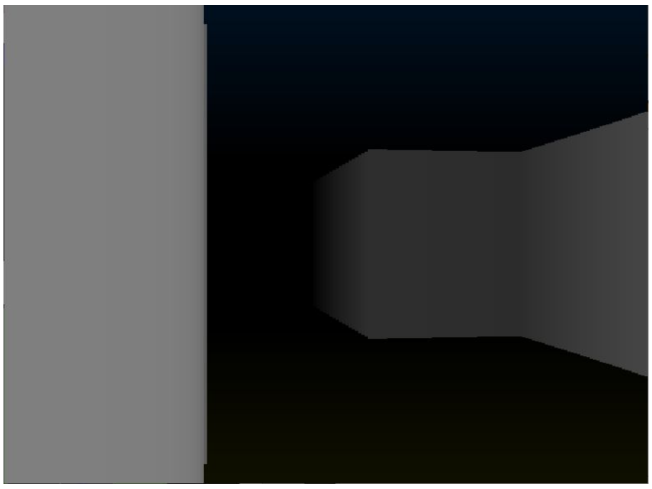
Frontal Collision:

Player 1 must avoid incoming traffic by using WASD and must catch all the jerrycans to obtain more fuel. Player 2 can place cars on the road in order to cause a collision with player 1 by clicking the screen.



Wolf & Stein:

In this game mode there is only one player. The movement is controlled by WASD and the player can explore a maze-like map and see the 3D ray cast simulation. Although there is no objective in this game the main interest was to show this game mode as proof of concept.

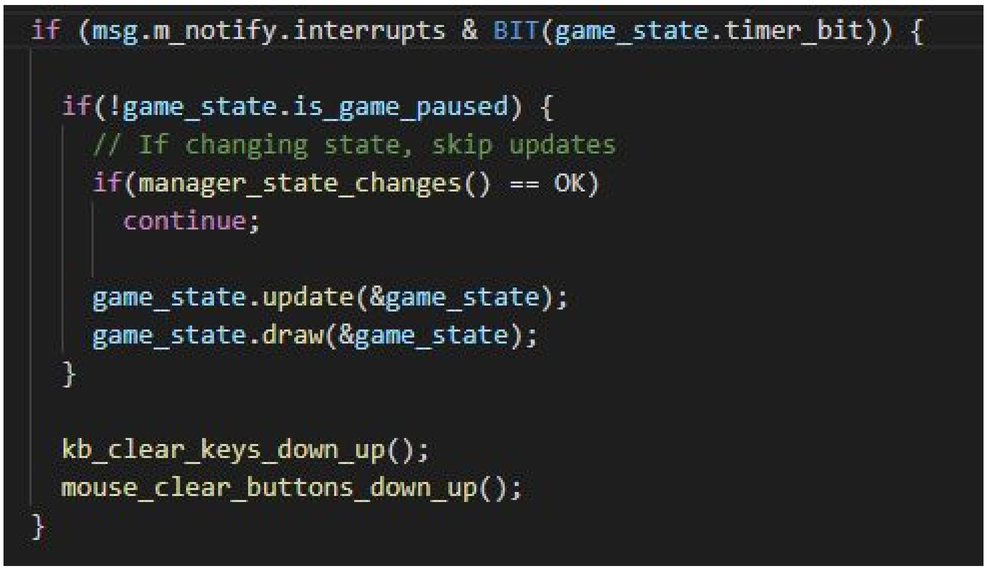


Project Status

Device	What for	Int.
Timer	Event based logic, frame rate	Y
Keyboard	Player movement and menu navigation	Y
Mouse	Player interaction and menu navigation	Y
Video Card	Drawing images on screen	N
Real Time Clock	Show current time on menu screen	N

Timer

Handles event based logic, keeps count of time passed since entering a state. Call state update and draw based on selected frame rate.



```
if (msg.m_notify.interrupts & BIT(game_state.timer_bit)) {

  if(!game_state.is_game_paused) {
    // If changing state, skip updates
    if(manager_state_changes() == OK)
      continue;

    game_state.update(&game_state);
    game_state.draw(&game_state);
  }

  kb_clear_keys_down_up();
  mouse_clear_buttons_down_up();
}
```

Keyboard

Handles keyboard based input logic. Allows for detection of press, release and hold keys. Device manager calls **kb_update_keys()** with every keyboard interrupt.

Keyboard interaction is composed of a **kb_manager()**, **keyboard** and **kbc**.

kb_manager uses bitmasks to keep track of currently held keys, as well as keys pressed and released in the current frame.

WASD is used to move players.
ESC to exit to start menu, and in the start menu to quit the program.
P to pause the game everywhere.

1,2,3 to switch maps in Wolf & Stein

(Only these keys are tracked, but the system is modular and easy to extend to a complete key list)

An API developed for handling the keyboard.

kb_init() and **kb_destroy()** start and stop the manager.

kb_is_key_held(), **kb_is_key_down()**, **kb_is_key_up()** to check keys state

kb_clear_keys_down_up() to reset bitmask every frame (presses and releases)

Mouse

Mouse follows a similar structure to the keyboard.

Handles mouse based input logic. Allows for detection of mouse buttons down, up and hold. Device manager calls **mouse_update_state()** with every keyboard interrupt.

It also has an API to start, stop the manager and several functions to check current position and button states

Mouse position is used to draw a cursor and detect clicks inside boxes (clickable graphics) and Left/Right buttons are used to spawn objects in the various games. (Middle mouse is also tracked but not used).

Video Card

The video card is used to display images either via bitmaps, XPM or by drawing individual pixels.

Different states use different video modes.

Hold your ground (initial prototype and testbed) uses indexed mode 256-color palette @ 800x600 (0x103)

Front Collision and start menu both uses direct mode 5:6:5 @ 800x600 (0x114)

Wolf & Stein uses direct mode 8:8:8 @ 800x600 (0x115)

Double buffering is implemented and used in every state

Hold your ground and Front Collision both have collision detection of moving objects (AABB).

Hold your ground also allows moving boxes to push the player box around

(One of the main gameplay features of this game)

Several functions were made for drawing (solid color/XPM/bitmap) graphics in different ways.

A function is provided to clear a previously drawn XPM (to be used after copying frame_buffer to the GPU). **vg_clear_xpm()**

To enable font drawing (or any type of spritesheet usage) a function was created to draw only a selected area of a bitmap. **vg_draw_bitmap_area()**

RTC

RTC is used to fetch the current time and date (to be displayed in the start menu). **rtc_get_param()**

Code Organization / Structure

box.c: Handles intersections/collisions between box_t (AABB structure). It is a key part of our project as two of our games have collisions as a backbone. Also has a function to allow one box to physically push another. Relative weight is: **10%**.

device_manager.c: Manages all devices (inits and cleanups), holds the interrupt loop and inside it calls the current state update/draw functions. Performs state changes (reassign state function pointers). Relative weight is: **9%**.

game_state.h: Struct that hold pointers to the current state functions. Also keeps the bits used for device interrupts and some flags for time and state switching. Relative weight is: **13%.**

gs_hold_your_ground.c: Holds game logic for "Hold your ground". Contains functions to create and display randomly sized boxes on Left mouse click and adjust their initial velocity based on the vector between mouse and screen center. Deals with the logic of the player's movement and how the boxes collide and push him. Contains enter/exit/draw/update functions assigned to the **game_state** at runtime. Relative weight is: **9%.**

gs_inf_runner.c: Holds game logic for "Frontal Collision". Contains all the game information, such as player's speed and amount of fuel, spawn rates and cooldowns for fuel and cars. Has mouse logic for the second player to place enemy cars. Draws game variables such as score, amount of fuel and player speed using a font bitmap spritesheet. Relative weight is: **13%.**

gs_menu.c: Contains all the code for the start menu, such as the functions to display time and date and mouse click in areas (for clickable graphics) to launch each game. Relative weight is: **3%.**

gs_wolfenstein.c: Holds game logic for "Wolf & Stein". Prototype implementation of a fake 3D display for 2D top down maps, inspired by Wolfenstein 3D. Holds the maps as strings. Has a function for ray casting (horizontal/vertical intersections) along a grid. Has a function to sweep an angle "casting rays" and finding the distance to the wall. Draw vertical rectangle with a height based on the distance calculated. Draws floor and ceiling for the whole screen (with walls in front). Does color lerping to simulate depth (using distance). Handles player movement and collisions inside the map. Relative weight is: **7%.**

kb_manager.c: Ease of use module that uses keyboard.c and kbc.c created in lab3 to read incoming scancodes and update bitmasks for keys being held, or pressed/released in the current frame. As explained above, provides an API to check a given key state. Relative weight is: **12%.**

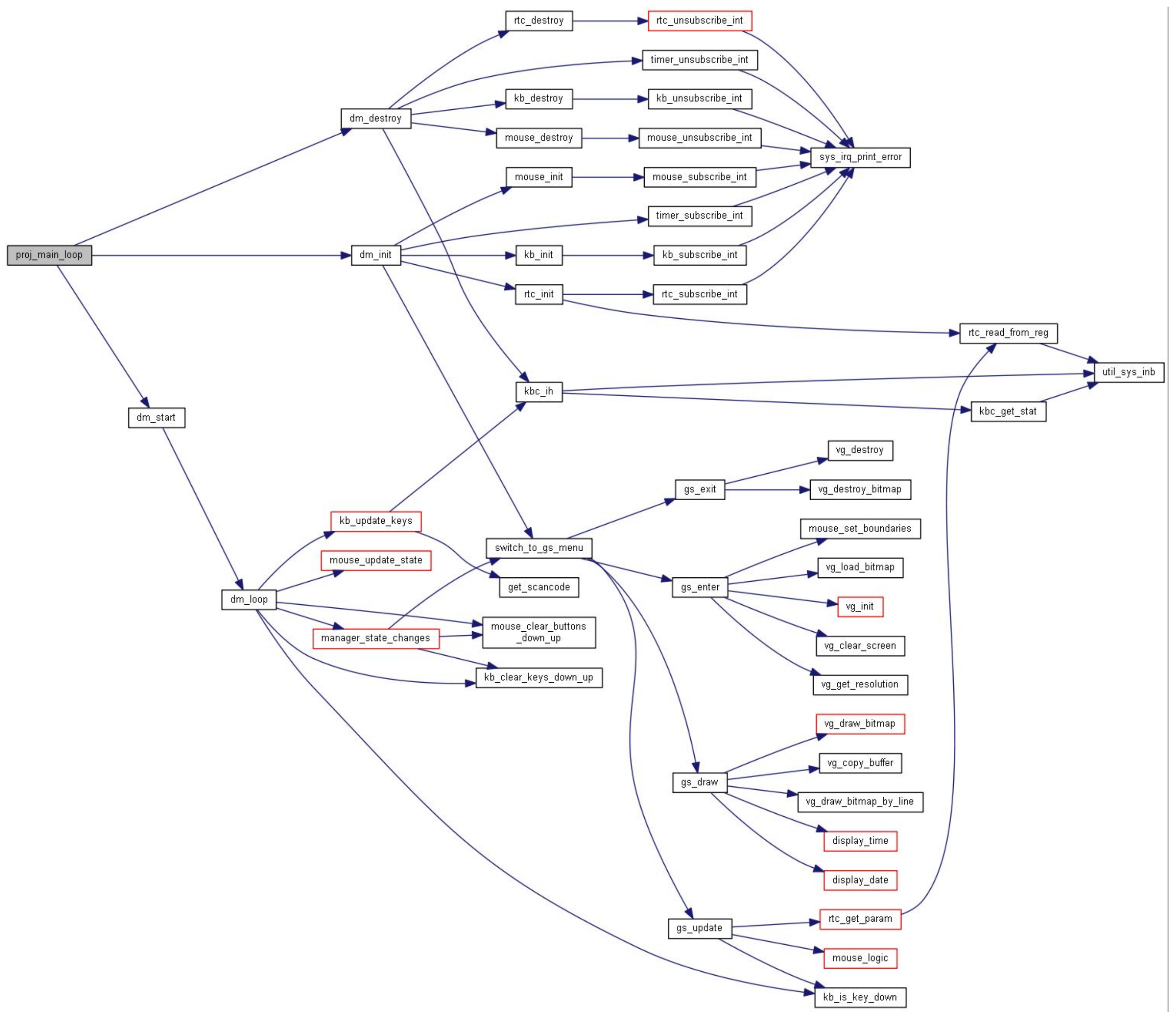
mouse_manager.c: Ease of use module that uses mouse.c and kbc.c created in lab4 to read incoming mouse packets and update bitmasks for buttons being held, or pressed/released in the current frame. Also keeps track of the mouse position (and locks it inside the screen). As explained above, provides an API to check a given button state and mouse position. Relative weight is: **7%.**

proj.c: Initiates the device manager (thus launching all the setup functions), and starts the interrupts cycle. Upon exit from the cycle call for cleanup of the device manager (unsubscribe devices, free memory). Relative weight is: **2%.**

rtc.c: Module to handle interaction with the RTC, from initialization to cleanup. Provides a function to try and read from a particular register (second, day, etc). Relative weight is: **2%.**

video.c: Handle all the video mode switching and drawing implemented in lab5. Support was added to allow loading, drawing in several ways and disposing of bitmaps. Double buffering was also implemented. All drawing is done to a buffer which is only copied to the GPU upon calling **vg_copy_buffer()** Relative weight is: **13%.**

Call graph


rtc_destroy
rtc_unsubscribe_int
timer_unsubscribe_int
kb_destroy
kb_unsubscribe_int
dm_destroy
mouse_destroy
mouse_unsubscribe_int
sys_irq_print_error
mouse_init
mouse_subscribe_int
timer_subscribe_int
proj_main_loop
dm_init
kb_init
kb_subscribe_int
rtc_init
rtc_subscribe_int
rtc_read_from_reg
util_sys_inb
kbc_ih
kbc_get_stat
dm_start
vg_destroy
gs_exit
vg_destroy_bitmap
kb_update_keys
mouse_set_boundaries
switch_to_gs_menu
mouse_update_state
vg_load_bitmap
get_scancode
gs_enter
dm_loop
vg_init
mouse_clear_buttons_down_up
manager_state_changes
vg_clear_screen
kb_clear_keys_down_up
vg_get_resolution
vg_draw_bitmap
vg_copy_buffer
gs_draw
vg_draw_bitmap_by_line
display_time
display_date
rtc_get_param
gs_update
mouse_logic
kb_is_key_down

Implementation details

The games and menu were implemented with code bundled in four functions, Enter, Exit, Draw, Update. A state machine like approach was used, where a state implements those four functions. A global game_state struct keep pointers to those four functions based on which state is currently active. When switching states, the Exit function is called, the function pointer are switched (**switch_to_gs_<insert_state>**) and the new Enter function is called. On every timer interrupt the Update function is called followed by the Draw function.

This system allows for code separation between states, and provides clear entry/exit points for initialization/cleanup and keeps an orderly flow of events.

The program launches in the **menu** state. On-screen buttons allow switching to the three games available. **ESC** exits the program if pressed. In each game, **ESC** takes the user back to **menu**, and **P** pauses the game.

We developed collision detection for the first two games, therefore we had to use extra curricular knowledge in order to implement collisions between boxes. For the first game we also developed a form of collision resolution that pushes the player's box when there is a collision with the bouncing rectangles.

To cycle the box structs used to move enemy car down the screen, we came with a method inspired by TCOM, to use a string and a marker to know which boxes are out of screen (done with the animation) and available to be launched by the player. The string marks 'Y' or 'N' if the box is being used or not.

To draw numbers/letters a monospaced spritesheet was used, together with the vg_draw_bitmap_area to easily display any element of the spritesheet. This method was used to display the time and date given by RTC and several game variables in Frontal Collision.

Conclusions

In our project we would have liked to implement the serial port, but due to time constraints we could not. The functions for XPM and bitmaps could have been removed from the video module for a more modular code. Also, some of the games could be polished, such as scenery dressing in Frontal Collision. In Wolf & Stein, with it being a proof of concept, not enough time was given to it, therefore no game mechanics were implemented. However the drawing method alongside the modules developed for the other games could easily expand it into a game.

Our project allows for a quick integration with new games, similar to a game engine, without much effort.