

FIREMI & WATERNIX

Uma versão *street* do jogo do
“Fireboy e Watergirl” para dois jogadores



Sérgio Rodrigues da Gama, up201906690

Lucas Calvet Santos, up201904517

Índice

Instruções de utilização.....	3
Estado do projeto.....	4
Organização/Estrutura do código.....	7
Documentação.....	19
Detalhes de implementação.....	19
Conclusões.....	20
Apêndice – Instruções de instalação.....	21
Apêndice – Diagramas das funções principais.....	22
Lista de Figuras.....	25

Instruções de utilização

O jogo inicia com um menu inicial, onde se tem acesso a dois botões distintos. O controlo destes é feito através do uso do rato, movendo o para cima do botão que se pretende pressionar e clicar no botão do lado direito do rato.

A tecla ESC encontra-se reservada para sair do programa a qualquer momento da sua execução.

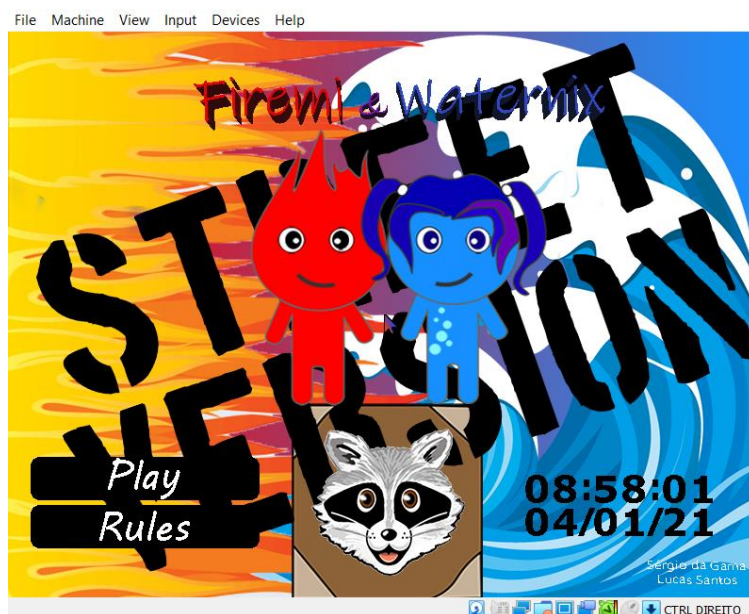


Figura 1 Menu Principal

- **Opção Play**

Quando este botão é pressionado, os jogadores podem, efetivamente, começar a jogar, encontrando-se no ecrã o primeiro nível do jogo. Existe uma totalidade de 4 níveis, sendo a sua dificuldade variável. De modo a completar o nível, as personagens devem chegar ambas à sua respetiva porta, identificadas pela cor, sendo a vermelha a do *firemi* e a azul a da *waternix*.

Contudo, para alcançarem as suas portas, estes têm de ultrapassar alguns desafios colaborativamente.

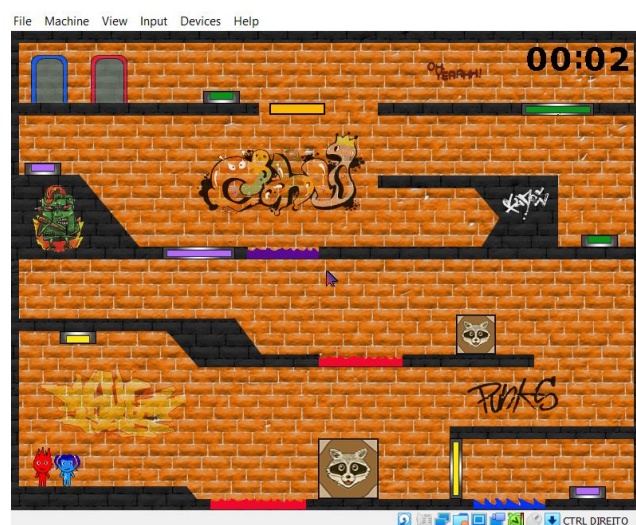


Figura 2 Nível 1

- **Opção Rules**

Após selecionada esta opção, o utilizador é direcionado para uma página informativa, onde se encontram as regras do jogo e as instruções de como controlar as duas principais personagens, entre outras indicações. Desta forma, o *firemi* é controlado através das teclas A, W e D e a *waternix* pela seta superior, esquerda e direita. A tecla A e a seta esquerda induzem um movimento para a esquerda, já a tecla D e a seta direita proporcionam um deslocamento para a direita, nas respetivas personagens. A tecla W e a seta superior levam as personagens a saltar uma vez, se clicadas uma vez, e a saltar repetitivamente, se deixadas pressionadas.



Figura 3 Menu das Regras

Estado do projeto

Na tabela seguinte encontram-se os periféricos que utilizamos, tal como o propósito da sua utilização e a forma como foram aplicados, se com interrupções ou não. Aquelas que utilizam interrupções são os periféricos que são controlados através do *loop* criado pela função *driver_receive()*, que notifica cada um deles se ocorreu um evento ou não.

Periférico	Utilização	Interrupções
Timer	Controlar o <i>frame rate</i> e um relógio no jogo	Sim
Keyboard	Mover as personagens	Sim

Mouse	Escolher opções nos menus e controlar as alavancas no jogo	Sim
Video Card	Mostrar os menus e o jogo no ecrã	Não
RTC	Mostrar a data e hora atual no menu principal	Sim
Serial Port	Não aplicada	Não

• Timer

O timer é utilizado tanto para controlar o *frame rate* do jogo, tal como o relógio que aparece no decorrer do mesmo.

Na implementação deste periférico foi necessário o desenvolvimento de certas funções, tais como as que subscrevem e deixam de subscrever as interrupções do timer (*timer_subscribe_int()* e *timer_unsubscribe_int()*). Para além disso foi criado um *handler* (*timer_int_handler()*), cujo propósito é tratar cada interrupção feita pelo timer.

• Keyboard

O keyboard é utilizado no projeto para mover as personagens do jogo, sendo este controlado via kbc, tal como o mouse.

A sua utilização, assim com o timer, requeria o desenvolvimento das funções de subscrição, mas neste caso para o keyboard. Sendo assim, foram criadas as funções *keyboard_subscribe_int()* e *keyboard_unsubscribe_int()* para subscrever a *irq_line* to keyboard. Em adição, também foi criado um *handler* para o este periférico, *keyboard_ih()*, que será posteriormente abordado, mais detalhadamente, na secção do estado do projeto, no tópico do keyboard.

• Mouse

O mouse é utilizado para interagir com os menus e para ativar as alavancas que existem no jogo.

Tal como os outros periféricos já mencionados, também este necessitou das suas funções de subscrição correspondentes, *mouse_subscribe_int()* e *mouse_unsubscribe_int()*. Contudo, no caso do mouse não é o suficiente, tendo sido também criadas as funções *mouse_enable_data_report()* e *mouse_disable_data_report()*, dado que o *data reporting* do mouse se encontra por *default* desligado no sistema (minix).

O mouse também tem o seu *interrupt handler*, *mouse_ih()*, sendo que a única tarefa que este tem de executar é ler um *byte* do *output buffer* do kbc, dado que a construção dos pacotes do mouse, que têm 3 bytes de tamanho, são construídos noutra função, *build_packet()*;

- **Video Card**

O jogo tem uma resolução de 800x600, sendo utilizado o modo 0x115 do VBE (macro *VBE_DIRECT_800_MODE*). Este tratar-se de um modo direto, tendo as core 24 bits no total, 8 bits para cada uma das componentes do RGB (RGB 8-8-8). A utilização deste modo possibilitou uma ampla variedade de cores ($2^{24} = 16777216$ cores possíveis).

No início do projeto começamos por criar uma função *restore_background()*, que voltava a imprimir uma pequena área do *background*, evitando ter que imprimir a sua totalidade. Por consequência, foram obtidos resultados bastante satisfatórios com esta função, pois parecia evitar muito bem o *overprocessing* da máquina. Contudo, mesmo sendo legível e jogável, era notável que alguns objetos eram parcial e momentaneamente apagados pela mesma.

Deste modo, numa fase mais madura do projeto, recorremos ao método do *double buffering*, uma vez que este resolvia o problema por completo e evitava o uso excessivo da função anteriormente referida.

Foram desenvolvidas então duas funções específicas para a implementação desta solução, *copy_buffer_to_vram()*, *copy_to_buffer()*. A primeira para copiar o *buffer* secundário para *vram* e a segunda para copiar um *pixmap* para o *buffer* secundário. Para além disso, também no âmbito da implementação desta solução, foi trocado na função *draw_pixel()* o endereço da *vram* para o do *buffer* secundário. No entanto, esta função e muitas outras, tal como as que efetuam o tratamento de colisões serão abordadas com mais detalhe neste relatório, na secção da Organização/Estrutura do Código.

- **RTC**

O RTC (*real-time clock*) foi utilizado para imprimir a data e a hora atual no menu principal.

Para este efeito, criou-se também uma função que subscrevesse as interrupções correspondentes a este periférico, *rtc_subscribe_int()*, e também um *handler* próprio, entre outras funções que são abordadas na secção da Organização/ Estrutura do Código.

Organização/Estrutura do Código

Todos os módulos apresentados têm à frente do seu título a indicação do autor de cada um, estando o S a simbolizar autoria do Sérgio e L autoria do Lucas. De modo a manter a legibilidade do relatório, foi criado um apêndice com todos os diagramas principais funções do nosso projeto.

Módulos de cada periférico

A maioria destes módulos foram, efetivamente, realizados no decorrer das aulas práticas. Contudo, muitos foram posteriormente melhorados e reorganizados.

Timer

Módulo timer [S e L]

Este módulo contém as funções que manipulam o timer diretamente, sendo as principais abordadas de seguida.

`timer_subscribe_int()`: subscreve as interrupções na *irq_line* 0, correspondente ao timer;

`timer_unsubscribe_int()`: retira a subscrição previamente feita ao timer (*irq_line* 0);

`timer_set_frequency()`: altera a frequência a que o timer trabalha;

`timer_int_handler()`: incrementa um contador sempre que existe uma interrupção, gerada pelo timer.

Módulo 'i8254'

Neste módulo encontram-se todas as macros (constantes), que são necessárias para a manipulação do timer, sendo que nos foi fornecido durante o decorrer da segunda aula laboratorial. Assim, este módulo não foi realizado por nós.

KBC

Módulo KBC [S e L]

Tanto o PS/2 mouse e o keyboard são controlados pelo KBC. Deste modo, foi desenvolvido um módulo comum aos dois periféricos, onde se encontram as funções

necessárias para comunicar e trabalhar com o KBC diretamente. As principais funções que se encontram neste módulo são as seguintes apresentadas.

`kbc_write_cmd()`: envia um comando para o *control register* (porta 0x64) do KBC, tendo o controlador de ter a porta livre. Caso não esteja livre, a função tenta um determinado número de tentativas enviar o comando, e se, de facto, verificar que a porta se encontra vazia, envia;

`kbc_read_data()`: lê os dados que se encontram no *output buffer* do KBC. Tal como a função anterior, esta também faz uma verificação antes de ler os dados, tendo o *output buffer* (porta 0x60) de estar cheio para os dados serem lidos. Caso não esteja, também esta função repete um determinado número de vezes a verificação;

`kbc_write_arg()`: semelhante à primeira função, conquanto, neste caso envia um argumento de uma das funções do KBC, sendo este enviado para o *input buffer* (0x60) do KBC, ao contrário da `kbc_write_cmd()`, que envia um comando para o *control register*.

Keyboard

Módulo keyboard [S e L]

Neste módulo encontram-se as funções que manipulam o teclado, tirando partido do módulo do KBC. As principais funções do módulo estão especificadas no seguimento deste parágrafo.

`keyboard_subscribe_int()`: subscreve as interrupções relativas ao keyboard (irq_line 1);

`keyboard_unsubscribe_int()`: cancela a subscrição efetuada para as interrupções do keyboard;

`keyboard_ih()`: lê os dados do *output buffer* (0x60) do KBC, utilizando o módulo do mesmo e constrói os *scancodes* do teclado. Assim, sempre que existe uma interrupção do keyboard, é lido do *output buffer* um *byte*, se o *scancode* apenas for constituído por um *byte*, ou dois *bytes*, se este último tiver, de facto dois *bytes* de tamanho.

Módulo i8042 [S e L]

Neste módulo encontram-se as constantes relativas ao uso do KBC. No entanto, optou-se por colocar este módulo no tópico do keyboard, dado que também contém os códigos dos *scancodes* do teclado.

PS/2 Mouse

Módulo mouse [S e L]

Aqui estão presentes as funções que manipulam o rato PS/2, tirando partido do módulo KBC, concebido anteriormente, visto que é através deste controlador que se interage com o mouse. De seguida, encontram-se as principais funções deste módulo abordadas.

`mouse_subscribe_int()`: subscreve as interrupções do mouse (`irq_line 12`);

`mouse_unsubscribe_int()`: cancela a subscrição das interrupções do mouse;

`mouse_ih()`: lê um *byte* do *output buffer* do KBC;

`mouse_write_cmd()`: envia um comando para o mouse. Na prática, envia o comando de escrita no mouse (`0xD4`), através da função `kbc_write_cmd()`. De seguida, envia o comando que quer enviar para o mouse, através da função `kbc_write_arg()`, onde o comando a enviar é, de facto, um argumento da função representativa do comando anteriormente enviado, através da `kbc_write_cmd()`;

`mouse_enable_data_report()`: ativa o *data report* do mouse, enviando o comando (`0xF4`), através da função `mouse_write_cmd()`.

`mouse_disable_data_report()`: desativa o *data report* do mouse, enviando o comando (`0xF5`), através da função `mouse_write_cmd()`.

`build_packet()`: constrói o *packet* do mouse, composto por 3 *bytes*, tendo o primeiro informação relativa ao estado dos botões, entre outros, e os dois últimos, informação relativa à posição do cursor, *x* e *y*, respetivamente.

Módulo PS2 [S e L]

Neste módulo encontram-se as constantes relacionadas com o uso do mouse. Deste modo, estão incluídas as constantes dos comandos passados como argumento do comando de escrita no mouse (`0xD4`), os *acknowledgment bytes* do mouse, entre outras constantes utilizadas na construção do pacote do rato.

Video Card

Módulo video_gr [S e L]

Este módulo diz respeito às funções e certas constantes que englobam a parte de manipulação da VBE e dos buffers de vídeo, o principal e o secundário (*back_buffer* ou *auxiliar_buffer*). As suas principais funções:

`get_vbe_mode_info()`: obtém toda informação acerca do modo passado como argumento. Crucial para o funcionamento da função seguinte `vg_init()`;

`vg_init()`: inicia o video card com a o modo passado como argumento. Para esse efeito, obtém as informações desse modo com a função anterior, mapeia a memória de vídeo e aloca espaço para o buffer secundário, utilizado para a implementação do *double buffering*;

`color_assembler()`: constrói os *bytes* de uma cor de um *pixmap*, através de uma dada posição passada em argumento e do valor dos *bits per pixel*, obtidos anteriormente pela `vg_init()`;

`convert_BGR_to_RGB()`: esta função é imprescindível no que toca à leitura dos nossos *xpm's*, dado que ao utilizarmos vários *websites* de conversão de formato *png* para *xpm*, concluímos que todos adotavam o formato BGR. Assim, criamos esta função que converte do formato BGR (8-8-8) para o RGB (8-8-8);

`vram_get_color_by coordinates()`: tal como o nome assim o indica, obtém a cor na *vram*, numa coordenadas específicas. Uma função útil para algumas verificações de colisões que efetuamos;

`pixmap_get_color_by coordinates()`: praticamente a mesma função que a anterior, no entanto, ao contrário de ser na *vram* é num *pixmap* passado em argumento;

`draw_pixel()`: altera a cor de um *pixel* no *buffer* secundário, numa posição passada em argumento, com as componentes x e y;

Nota: as funções `copy_buffer_to_vram()` e `copy_to_buffer()` já foram explicadas na secção do estado do projeto, no tópico da video card. Daí não serem abordadas detalhadamente aqui.

RTC

Módulo RTC [L]

Neste módulo encontram-se as funções e constantes que manipulam o RTC, de modo a obter a data e hora atual. Os seus principais constituintes estão de seguida abordados.

`rtc_subscribe_int()`: subscreve as interrupções do RTC (*irq_line* 8);

`rtc_unsubscribe_int()`: cancela a subscrição das interrupções do RTC;

`rtc_enable_update_int()`: ativa o *update ended interrupt* do RTC;

`rtc_ih()`: lê o registo de dados do RTC, após ter enviado o devido comando para identificar o registo a ser lido;

`rtc_get_time()`: acede ao registo de cada parâmetro e guarda a data e a hora numa *struct* do tipo `time`, em formato de 24 horas e em binário. Caso este esteja no formato `bcd`, é convertido para binário. Caso as horas estiverem no formato de 12 horas, também são convertidas para o formato de 24.

Módulos próprios do jogo

Módulo `sprite` [S e L]

Neste módulo encontram-se definidos os `sprites` de forma orientada a objetos. Os `sprites` são todo o tipo de objetos que tem como base uma representação gráfica e que têm posições variáveis, consoante aquilo que se pretende. Contudo também era necessária uma definição de um *animated* `sprite`, ou seja um `sprite` em que a sua imagem base vai alterando, mediante certas condições. Deste modo, de forma a evitar criar outro módulo e simplificar significativamente esta parte crucial da definição dos `sprites`, desenvolvemos apenas este módulo `sprite` com a seguinte definição:

```
typedef struct {
    int x, y; //current position
    uint32_t transparency_color; //transparency_color
    enum xpm_image_type xpm_type; //xpm image type
    int width, height; //dimensions
    int xspeed, yspeed; //current speed
    uint8_t *map; //current pixmap
    uint8_t *xpm[11]; //array of xpm's
} Sprite;
```

Figura 4 Definição de `Sprite`

Com esta definição evitamos a criação de dois módulos distintos para a manipulação de objetos gráficos animados e não animados. Para isso, foi apenas necessário adicionar um *array* de *pointers* para *pixmap*s, (*xpm*s), onde estão todos os *pixmap*s para um determinado `sprite`. Se apenas fosse preciso um `sprite`, o *array* só teria esse mesmo. O *pointer map* aponta para o mapa atual, logo, sempre que é preciso animar o `sprite`, basta alterar esta *pointer* para um dos *pixmap*s do *array xpm*s.

Ainda neste módulo, estão definidas algumas constantes que são utilizadas na função *sprite_keyboard_move()*, onde é replicado um ambiente com gravidade e fricção.

Para além destas definições, neste módulo, encontram-se as funções que manipulam estes últimos. Algumas das principais serão cobertas seguida.

```
//environment constants
#define FPS 30
#define V_STEP 1
#define JUMP_STEP 27
#define GRAVITY 3
#define MAX_V 8
#define FRICTION 1
#define SLOPE_STEP 8
#define ON_LIMIT_HELP_SPEED 7
```

Figura 5 Macros relacionadas com o comportamento dos sprites

de

draw_sprite(): imprime um sprite no seu *pixmap* atual e nas suas coordenadas atuais;

draw_sprite_at_angle(): imprime um sprite num determinado ângulo passado em argumento, através de um determinado ponto âncora;

restore_background(): restaura o background numa determinada região passada como argumento;

sprite_keyboard_move(): cria o movimento das personagens, consoante as teclas que são pressionadas. Para além disso, ao longo que estas se vão movendo, verifica se existe colisão entre as mesmas e a cor 0x0 (preto), cor utilizada para definir as paredes e os limites dos objetos;

collision_two_rects(): verifica se há colisão entre dois sprites, tratando cada um deles como retângulos;

collision_one_rect(): verifica se há colisão entre dois sprites, tratando um deles como retângulo e outro com a sua forma real, definida pelos pixéis com cores diferentes da *transparency color*. Esta demonstrou-se uma função bastante importante, no sentido em que é utilizada pelos *handlers* das caixas, dos botões e das barras, para detetar colisões entre os vários objetos;

check_collision_sprite_at_angle(): faz o mesmo que a função anterior mas para um sprite que se encontra num determinado ângulo;

check_sprite_collision_by_color(): verifica se há colisão entre um sprite e uma cor na *vram*, ou num *pixmap*, sendo assim que verificamos as colisões das personagens com as três cores das lavas e a cor preto, que define as paredes. Para isso, é passado como argumento um *pixmap* de cada nível criado de propósito para a deteção de colisões;

check_sprite_xpm0_collision_by_color(): verifica colisão entre um sprite e uma cor na *vram*, ou num *pixmap*. Faz então o mesmo que a anterior, mas com a diferença de que efetua a verificação sempre com o primeiro *pixmap* e não com o atual apontado pelo *pointer map*.

Módulo `game_handler` [S e L]

Este módulo e o seguinte complementam-se, dado que um deles é responsável pela definição dos componentes do jogo, o `game_handler()`. Já o outro, `game_engine()`, é responsável pela criação e encapsulamento de todos os componentes por nível e a conexão entre os mesmos através da função `handle_level()`.

Principais componentes deste módulo fortemente orientado a objetos:

- **Cursor:** cursor visível no ecrã, que segue o movimento do rato;
- **Clock:** relógio que pode ser impresso onde se pretender no ecrã, controlado pelo timer;
- **Game_button:** botão controlado pelas personagens, que pode ser pressionado para mover uma barra ao qual esteja conectado. Existem quatro tipos diferentes de botões, sendo definidos pela sua orientação, NORTH, SOUTH, EAST e WEST. Para além dos botões do tipo SOUTH, todos os outros têm de se manter premidos para continuarem premidos. Já o SOUTH basta premir uma vez e ele fica pressionado até voltar a ser premido;
- **Game_bar:** barra que pode ser controlada vários botões, ou alavancas, estando a conexão entre os componentes identificada através de cores iguais no centro dos seus gráficos;
- **Game_lever:** alavanca que é controlada pelo utilizador para mover uma barra, mas apenas quando uma das personagens se encontra próxima o suficiente dela. Esta ainda muda de cor na ponta, de vermelho para verde, sendo a primeira cor indicativa de que está inativa e a segunda de que está ativa.
- **Wind:** apesar de não ter uma classe associada a ele este tem um *handler* correspondente, podendo com o sprite correto ser criado um local com as características do vento, na direção vertical, aplicado às personagens.
- **Lava:** também não tem uma classe associada a ela. No entanto, tal como o wind, com o sprite correto, pode ser criada uma lava que se vai constantemente movendo para a esquerda, utiliza a função do `restore_background()` para cortar as suas extremidades. A utilização desta última, evita terem de ser criados vários *pixmaps* iguais com tamanhos diferentes sempre que queríamos uma lava com tamanho diferente. Assim, apenas existem três *pixmaps* com uma largura de 240 *pixels*, um para a lava vermelha, outro para a azul e outro para a roxa.

Todos os componentes referidos anteriormente têm então um *handler* associado a si, que trata do seu comportamento, mediante as mudanças que vão acontecendo ao longo do *runtime* do jogo. Em adição a esses *handlers*, também se encontra um `handle_characters_move()` que utiliza o `sprite_keyboard_move()` para controlar o movimento das personagens.

Módulo `game_engine` [S e L]

Aqui encontram-se tratados todos os quatro níveis que criamos, tal como outros estados do jogo, como o menu principal, o das regras e o de pausa. Assim, era necessário saber em que estado se encontrava o jogo num determinado momento, tendo sido criada esta *enum* para o definir:

```
//enum with the possible game states
enum game_state {
    MAIN_MENU, RULES_MENU, PAUSE, LEVEL_1, LEVEL_2, LEVEL_3, LEVEL_4
};
```

Figura 6 Possíveis estados do jogo

Neste módulo existem três funções fundamentais:

- `create_level()`: cria um nível com todos os seus componentes;
- `handle_level()`: chama os *handles* dos objetos criados na função anterior;
- `delete_level()`: apaga e restaura a memória alocada pelos objetos criados para o nível.

Nota: apesar das funções terminarem todas no seu nome com *level*, também funcionam para os outros *game states*: `MAIN_MENU`, `PAUSE`, `RULES_MENU`.

Proj.c

Este foi o ficheiro fornecido para colocarmos o código referente ao ciclo do *driver receive* e os componentes que surgem acompanhados deste mesmo. Assim, devido a uma boa organização do código da nossa parte, obtivemos um `proj.c` com apenas o essencial. Desta forma, não sobrecarregamos este ficheiro com muito código mantendo uma fácil leitura e interpretação do mesmo. Com efeito, não retrata um módulo nosso, mas é onde o *main loop* do nosso programa se encontra.

Design Gráfico

O desenvolvimento de gráficos foi uma parte bastante grande do nosso, projeto, sendo a maioria dos gráficos dinâmicos da nossa autoria. Estes foram criados através do *Adobe Illustrator* e *Adobe Photoshop*, sendo a maioria delas gráficos vetoriais, posteriormente convertidos para *pixmaps* com as dimensões desejadas. Apesar de ser, teoricamente, mais dispendioso em utilização de memória, optou-se por colocar todos os *pixmaps* necessários em módulos devidamente separados, pois foi como obtivemos melhores resultados. Para além disso, também se refletiu numa vantagem a nível de

organização do trabalho, dado que tornou a utilização dos gráficos mais prática. Assim, os seguintes módulos constituem a totalidade dos gráficos utilizados, tendo todos o prefixo *xpm*.

Módulos *xpm_firemi* e *xpm_waternix* [S]

Para ambos estes módulos foram criadas 3 personagens paradas (*idles*), que iriam trocando, quando este não se desolacava, contudo devido a um pequeno deslinhamentos acabaram por não ser aplicados. Também a personagem com o braço de fora, para simulação do empurro da caixa não foi aplicado, dado que só foi criado um, não sendo possível simular o movimento das pernas com eles. Assim, são, efetivamente, utilizados 7 *pixmap*s, 6 para o movimento e 1 para quando este se encontra parado.

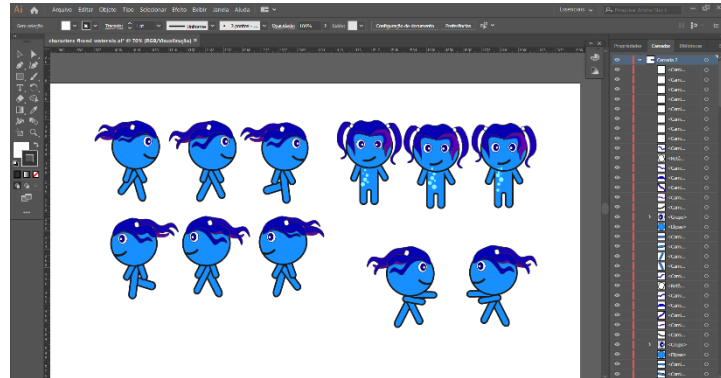


Figura 7 Criação da personagem Waternix

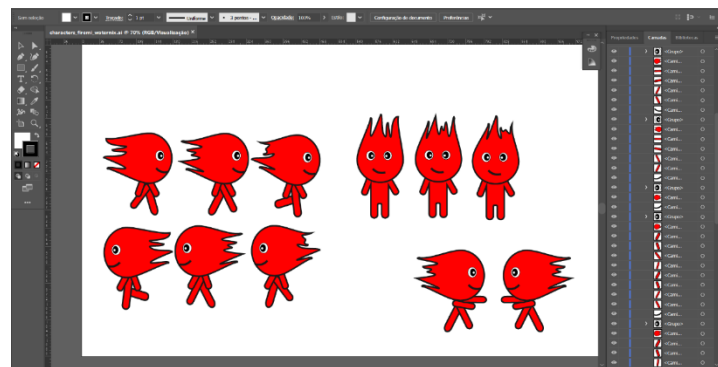


Figura 8 Criação da personagem Firemi

Módulo *xpm_menus* [S e L]

Neste módulo encontram-se todos os componentes gráficos relativos aos menus, o *main menu*, o *pause menu* e o *rules menu*. Para estes, foram desenhados botões devidamente rotulados, em duas versões com cores diferentes. Assim, um delas aparece quando se posiciona o rato dentro do mesmo, dando a ilusão de que estes trocam de cor.

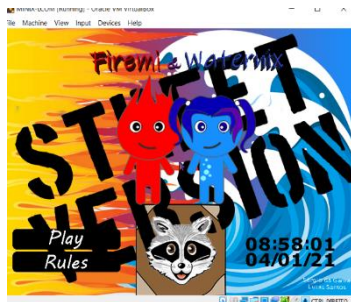


Figura 10 Main menu



Figura 11 Menu das Regras

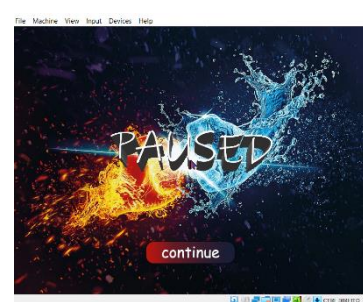


Figura 9 Menu de pausa

Módulo xpm_titles [S]

Este módulo contém as frases ou expressões textuais que utilizamos, tais como *Game Over*, *Level 1 Completed*, etc.

Estas têm todos a cor branca para haver contraste entre as mesmas e o sítio onde estão a ser impressas.

Módulo xpm_levels [S]

Todos níveis seguiram o mesmo modelo de desenvolvimento gráfico que vai ser abordado de seguida, utilizando como exemplo o nível dois.

Em primeiro lugar, dávamos lugar a um processo criativo, uma vez que todos os níveis são da nossa autoria. Logo, o processo consistia em ir desenhando e apagando objetos dinâmicos e estáticos até obter algo que fosse minimamente desafiante e funcionasse, mediante as conformidades que tinhas definido no código do nosso jogo. Desta forma, no final deste primeiro processo era obtido um nível com todos os elementos (dinâmicos ou não) que o nível final iria apresenta.

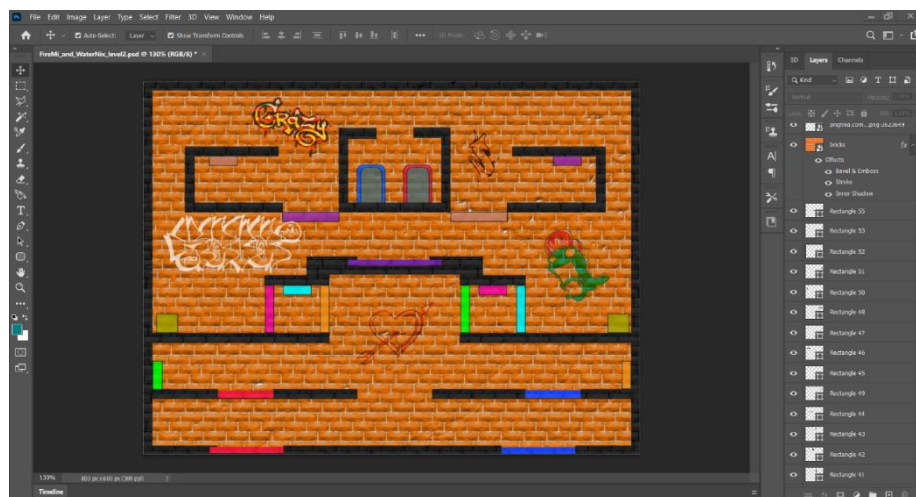


Figura 12 Criação do nível 2

De seguida, eram removidos todos os elementos dinâmicos, de modo a obter o nível que ia, efetivamente, ser impresso estaticamente no ecrã.



Figura 13 Nível dois, mapa impresso no ecrã

Na terceira etapa é feito o mapa ou máscara de colisões. Isto é, as texturas eram retiradas, de modo às paredes ficarem com a sua cor preta original, através da qual as colisões são detetadas. Para além disso, apesar de serem elementos dinâmicos, as lavas também voltavam a ser simbolicamente representadas pela sua cor, neste mapa. Este mapa nunca é impresso no ecrã, mas é utilizado para verificar as colisões das personagens com as paredes e com as lavas.

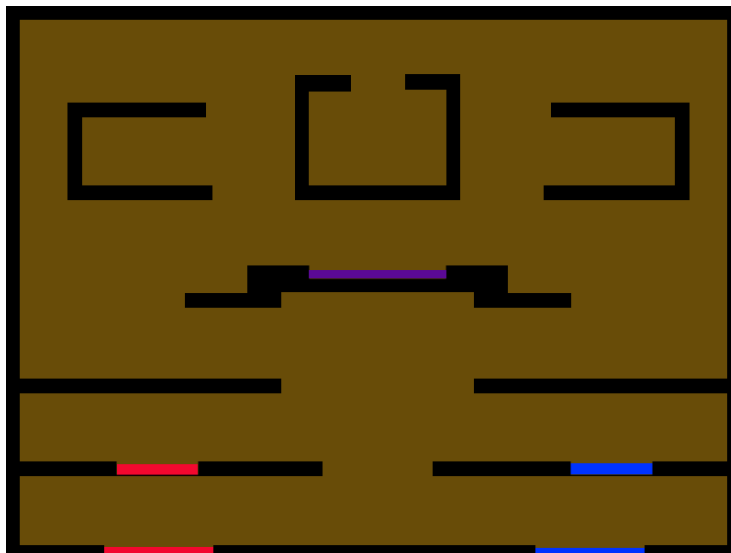


Figura 14 Nível dois, mapa utilizado para deteção de colisões

Por fim, era efetuada a criação de todos os objetos dinâmicos, complementando o mapa estático em *runtime* com os mesmos. Nesta fase, efetuamos alguns testes e

verificamos se os níveis são possíveis de completar ou não, fazendo as alterações necessárias.



Figura 15 Nível dois no jogo em runtime, completo

Módulo `xpm_game_elements` [S e L]

Neste módulo todos os elementos, excluindo o cursor do mouse (`xpm_mouse`) e os números (`xpm_numbers`), foram concebidos com um contorno preto, devido à forma como efetuamos a verificação de colisões. Deste modo, foram desenvolvidas duas caixas quadradas, com tamanhos distintos, com 50x50 e outra com 76x76, ambas com o logótipo do Minix, como referência ao mesmo. Depois foram criados os protótipos dos botões e das barras, onde a cor do meio indica quais estão conectados a quais, sendo que todos os botões controlam no máximo uma barra, mas uma barra pode ser controlada por vários e até alavancas. Estes dois elementos depois de convertidos para o formato de `xpm` podiam ser modificados manualmente, trocando então a cor central para a desejada.

Tanto a caixa, como os botões e as barras são controladas pelos movimentos das personagens, enquanto que as alavancas são controladas pelo mouse, quando uma personagem se encontra próxima o suficiente da mesma.

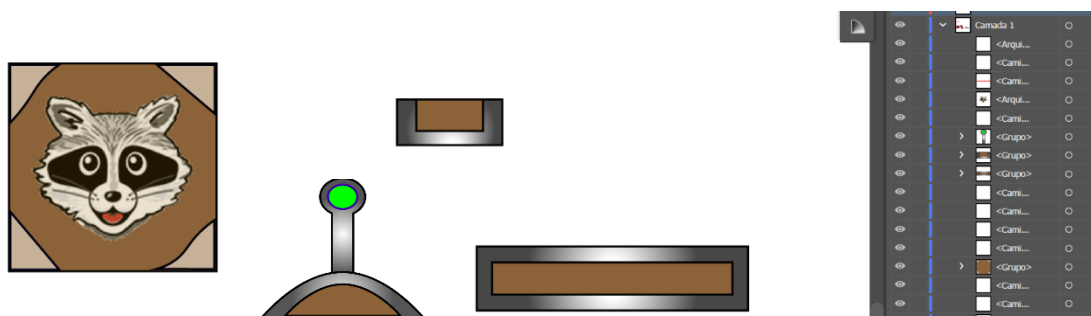


Figura 16 Criação dos elementos dinâmicos do jogo

Ainda neste módulo temos um slider, que apenas é utilizado uma vez, dado que foi um gráfico que criamos mais preliminarmente e optamos por manter. Encontram-se também gráficos do cursor e dos números utilizados pela objetos tipo *Clock*, tal como os botões, as barras e as alavancas correspondem a objetos do tipo *Game_button*, *Game_bar* e *Game_lever* do módulo *game_handler*.

Documentação

Todos os módulos, excluindo os módulos gráficos, foram documentados utilizando o Doxygen. Em adição, também foram efetuados comentários no interior do próprio código sempre que necessário, para uma melhor compreensão posterior do mesmo.

Através da documentação doxygen foi gerado um diagrama de chamada de funções, que se encontra na pasta *doc* do projeto.

Detalhes de Implementação

O nosso jogo tem um elevado número de componentes dinâmicos, o que tornou certas tarefas, significativamente, mais complicadas. Desta forma, tivemos muitas vezes de recorrer a soluções alternativas às inicialmente pensadas, de modo a otimizar o jogo e proporcionar uma experiência de jogabilidade muito melhor.

Todo o projeto foi desenvolvido com o paradigma da orientação a objetos em mente, dado que se reflete em programas melhor organizados e muito mais fáceis de interpretar. Assim, a maioria dos componentes que se encontram no jogo foram criados utilizando *structs*, com os atributos de cada objeto no seu interior. Sendo assim, os métodos de cada objeto são as funções que aceitam o objeto como argumento e têm o seu nome semelhante a *handle_object()*, onde o *object* é substituído pelo objeto correspondente.

No ficheiro *proj.c* temos o ciclo do *driver receive*, onde cada interrupção é tratada devidamente. Neste ciclo, é chamado o *handle_level()*, que funciona como uma máquina de estados, sendo cada um dos *game_states* correspondentes a um estado da nossa máquina de estados.

Por fim, devido à elevada complexidade da implementação de todos os componentes necessários, não conseguimos, de facto, ter tempo para implementar a porta série, que estava nos nossos planos iniciais. Contudo, obtivemos um excelente projeto final com o qual estamos completamente satisfeitos.

Conclusões

A nível global a cadeira de Laboratórios de Computadores foi das cadeiras mais desafiantes até ao momento, na medida em que, se realmente não houver dedicação e esforço, não é possível aprender os tópicos retratados na mesma. No início, apresentava-se como algo extremamente confuso e era complicado saber por onde começar a codificar uma determinada solução. Conquanto, ao longo do decorrer da cadeira foi se maturando o melhor método de aprendizagem, tornando-nos cada vez mais eficientes a cada laboratório que passava. De facto, trata-se de uma cadeira exigente e com uma curva de aprendizagem grande. Contudo, é um percurso que vale a pena, dado que acabamos com um conhecimento muito maior acerca dos vários periféricos que foram estudados e muito mais. Conhecimentos acerca de verificação de colisões, de máquinas de estado, de desenvolvimento de jogos, de programas *event driven* e até de design gráfico. Para além disso, os docentes demonstraram-se sempre disponíveis, tanto nas aulas, como remotamente, dado que o nosso grupo até chegou a efetuar uma chamada via *teams* com o docente que nos acompanhava nas aulas, para esclarecer algumas dúvidas do projeto.

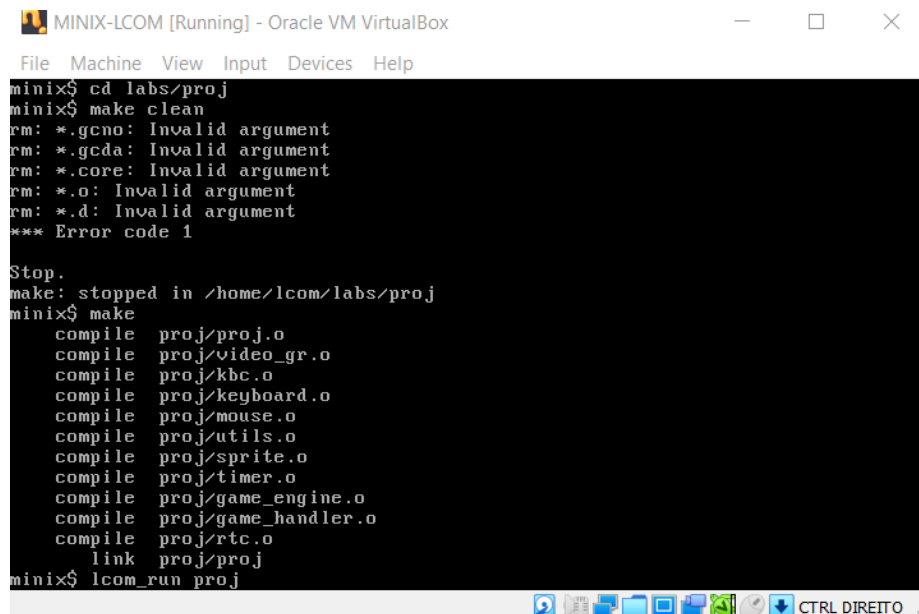
Apesar de todos aspetos positivos, ainda temos algumas sugestões de melhoria para nos anos seguintes se tornar em algo ainda melhor:

- Na nossa opinião, seria muito útil haver pelo menos uma aula, ou um bocado de uma aula, para abordar alguns tópicos acerca do tratamento de colisões. Trata-se de um tópico incontornável no que toca à programação de um jogo com objetos dinâmicos.
- Para além disso, seria bastante útil para nos primeiros contactos com a disciplina não ficarmos um bocado perdidos, nas primeiras aulas ser nos dada uma *overview* global do que se pode fazer com os conhecimentos obtidos em LCOM.

Apêndice – Instruções de Instalação

Para instalar o jogo e corrê-lo, basta dar clone do repositório do *gitlab*, onde se encontram todos os componentes do projeto para uma pasta associado ao Minix. De seguida, num terminal do Minix, ou num terminal com acesso ao Minix através de *ssh* apenas se tem de fazer *make clean*, por precaução, *make* e *lcom_run proj*, que utiliza a *framework* de LCOM para executar o programa. O *make clean*, *make* e execução só podem ser efetuados no mesmo diretório da pasta para onde se fez clone. Este processo encontram-se demonstrado nas figuras X.

No nosso repositório do *gitlab* temos tudo guardado de modo devidamente rotulado. Desta forma, está tudo organizado com pastas intituladas com os recursos correspondentes. Assim, este nosso arquivo incluía todos os ficheiros *do Adobe Photoshop* e *do Adobe Illustrator*, tal como todas as imagens utilizadas e não utilizadas, de versões antigas. Contudo, como demorava algum tempo a fazer o clone do nosso repositório, retiramos os ficheiros *Adobe* e deixamos apenas as imagens.



```
MINIX-LCOM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
minix$ cd labs/proj
minix$ make clean
rm: *.gcn: Invalid argument
rm: *.gda: Invalid argument
rm: *.core: Invalid argument
rm: *.o: Invalid argument
rm: *.d: Invalid argument
*** Error code 1

Stop.
make: stopped in /home/lcom/labs/proj
minix$ make
  compile proj/proj.o
  compile proj/video_gr.o
  compile proj/kbc.o
  compile proj/keyboard.o
  compile proj/mouse.o
  compile proj/utils.o
  compile proj/sprite.o
  compile proj/timer.o
  compile proj/game_engine.o
  compile proj/game_handler.o
  compile proj/rtc.o
  link proj/proj
minix$ lcom_run proj
```

Figura 17 Terminal do Minix pronta para correr o jogo

Apêndice – Diagramas das funções principais

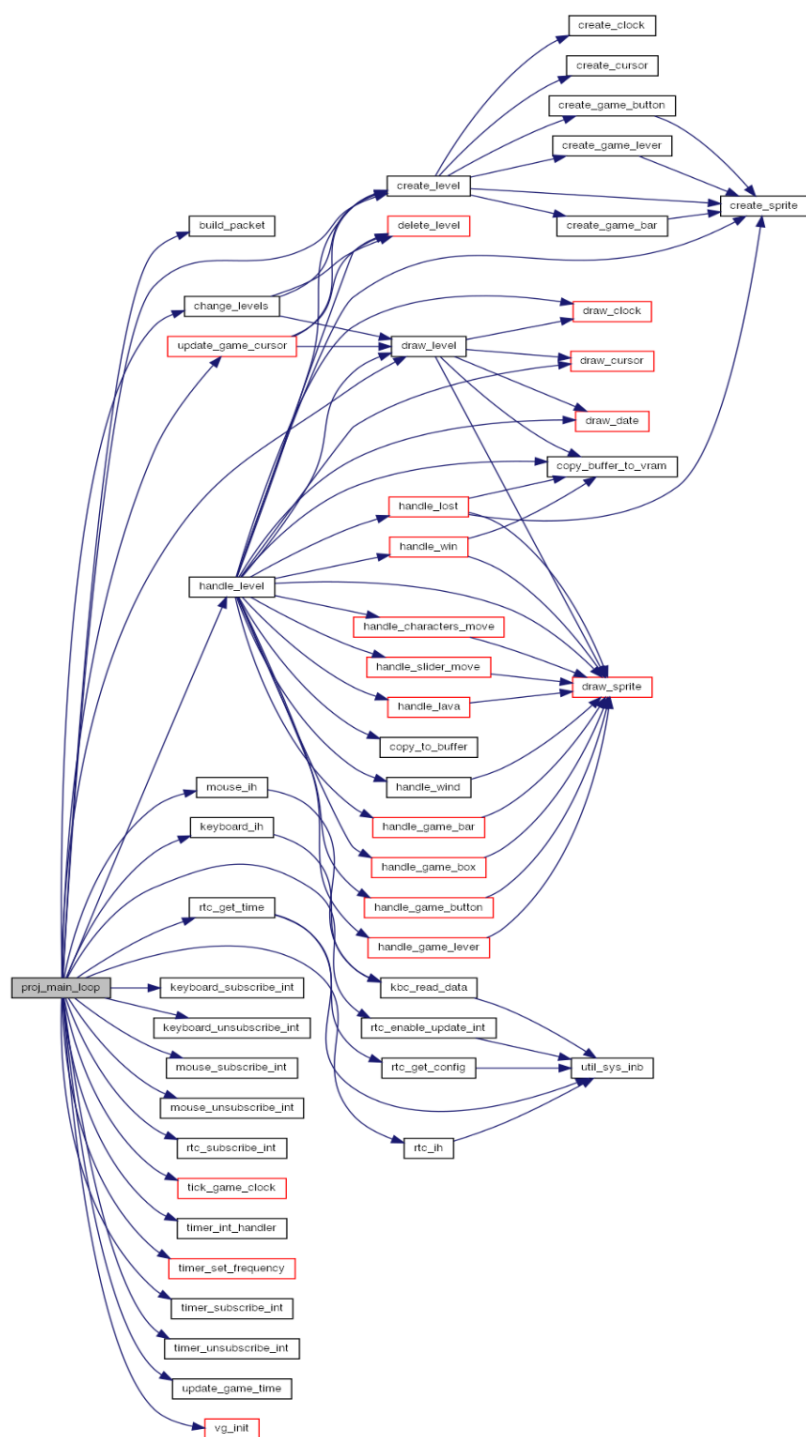


Figura 18 Diagrama da função principal do proj.c

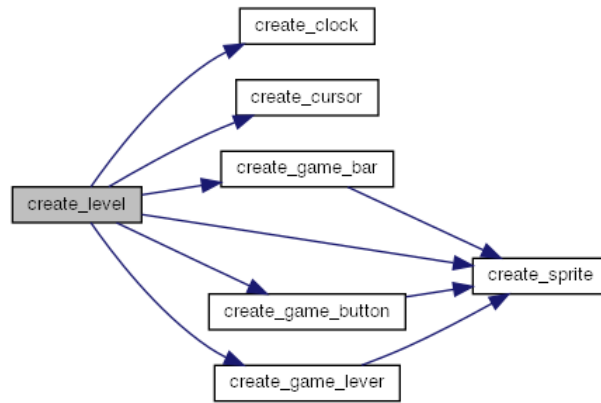


Figura 20 Diagrama da função `create_level` do `game_engine`

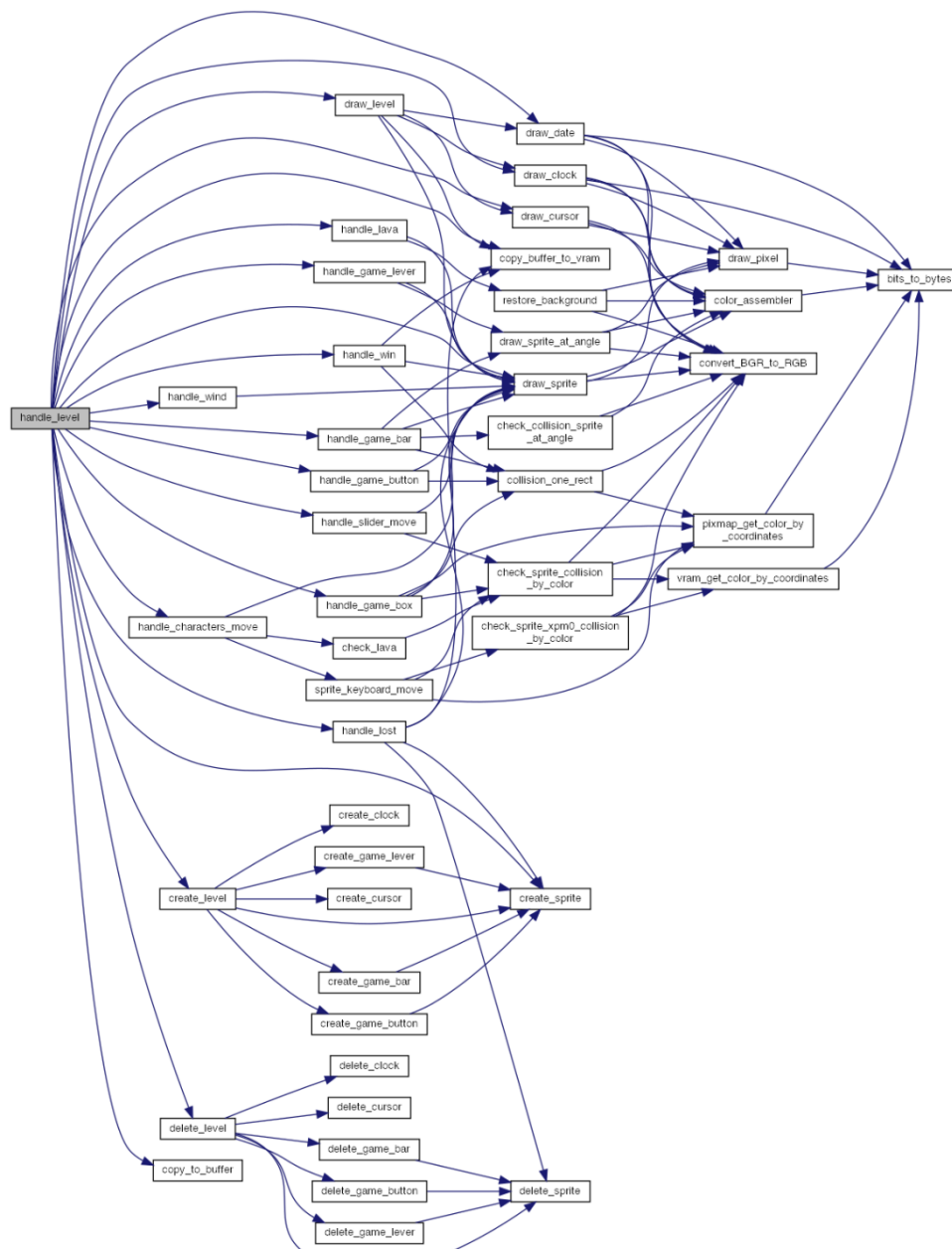


Figura 19 Diagrama da função `handle_level` do `game_engine`

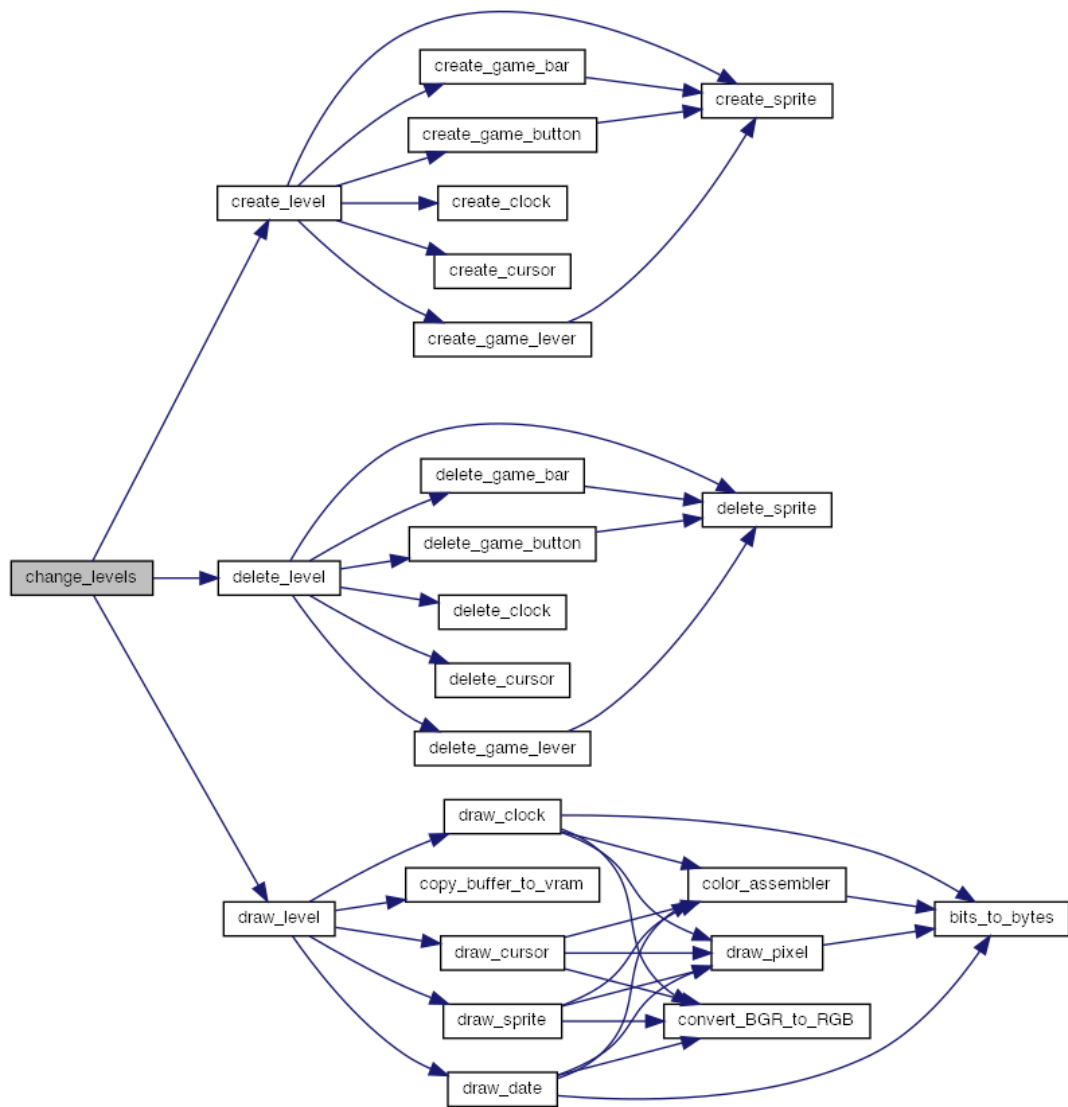


Figura 21 Diagrama da função `change_levels` do `game_engine`

Lista de Figuras

Figura 1 Menu Principal	3
Figura 2 Nível 1.....	3
Figura 3 Menu das Regras	4
Figura 4 Definição de Sprite	11
Figura 5 Macros relacionadas com o comportamento dos sprites.....	12
Figura 6 Possíveis estados do jogo	14
Figura 8 Criação da personagem Waternix	15
Figura 7 Criação da personagem Firemi.....	15
Figura 9 Menu de pausa.....	15
Figura 10 Main menu	15
Figura 11 Menu das Regras	15
Figura 12 Criação do nível 2	16
Figura 13 Nível dois, mapa impresso no ecrã	17
Figura 14 Nível dois, mapa utilizado para deteção de colisões	17
Figura 15 Nível dois no jogo em runtime, completo.....	18
Figura 16 Criação dos elementos dinâmicos do jogo.....	18
Figura 17 Terminal do Minix pronta para correr o jogo.....	21
Figura 18 Diagrama da função principal do proj.c	22
Figura 19 Diagram da função handle_level do game_engine	23
Figura 20 Diagrama da função create_level do game_engine.....	23
Figura 21 Diagrama da função change_levels do game_engine.....	24