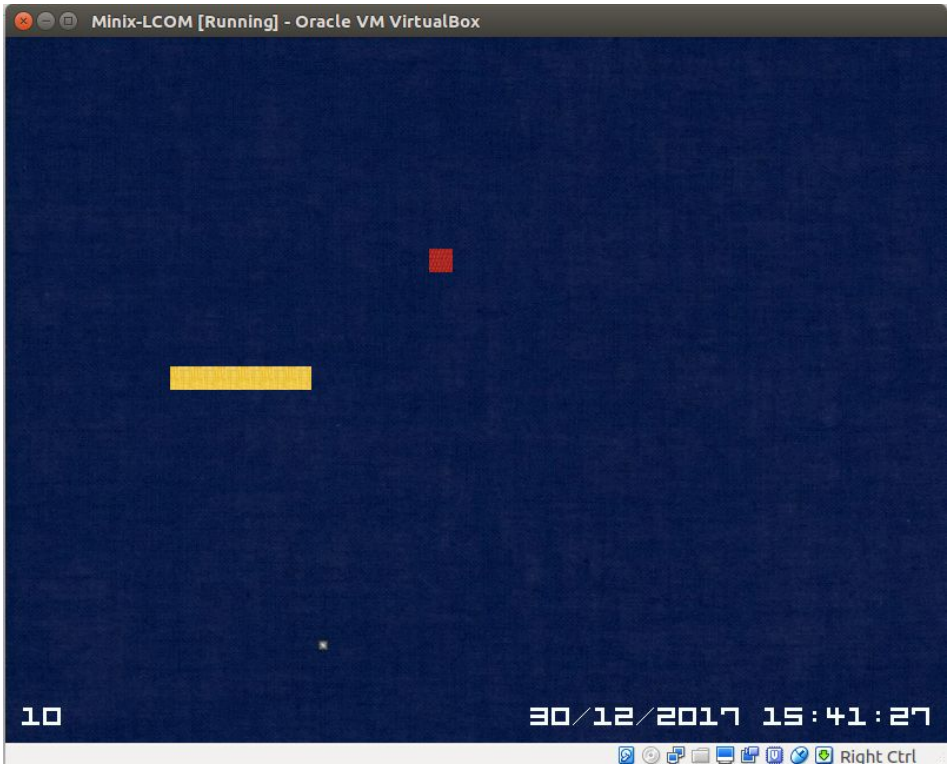
SnakeyNix

O “Jogo da Cobra” para múltiplos jogadores

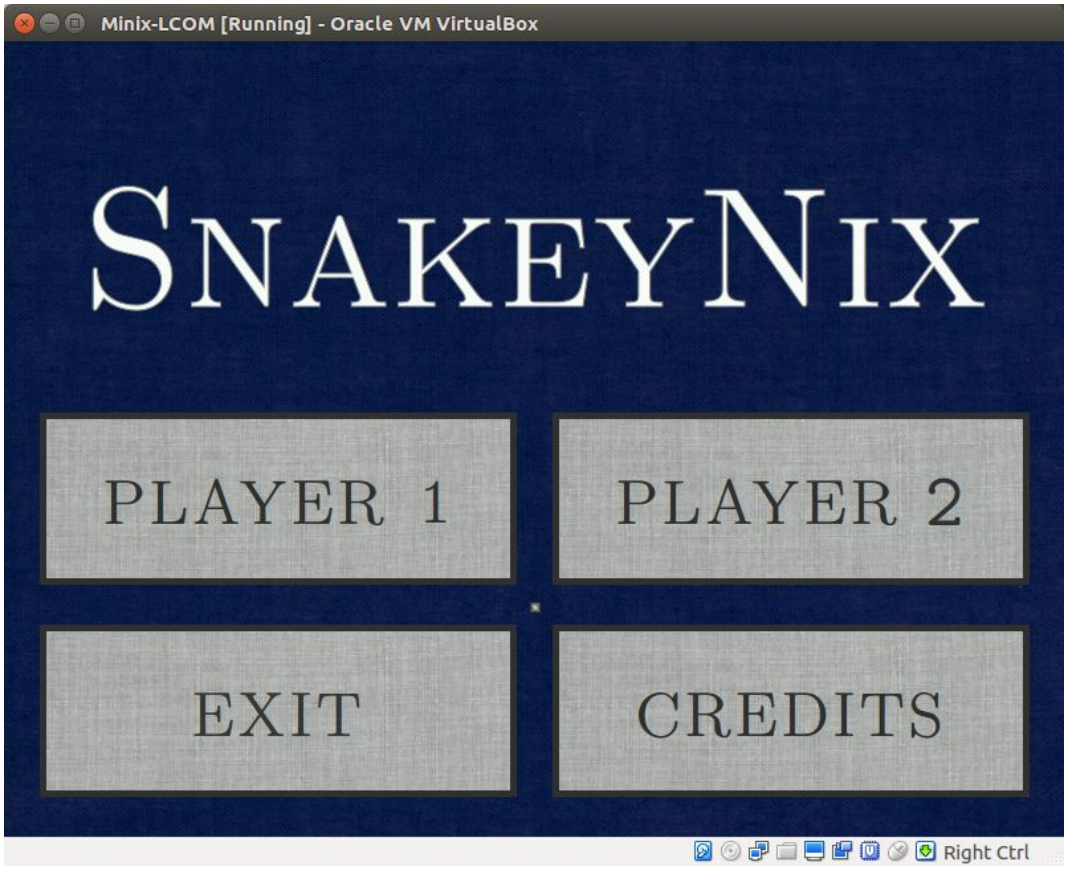


Rui Pedro Moutinho Moreira Alves, up201606746
Filipa Manita Santos Durão, up201606640

Índice

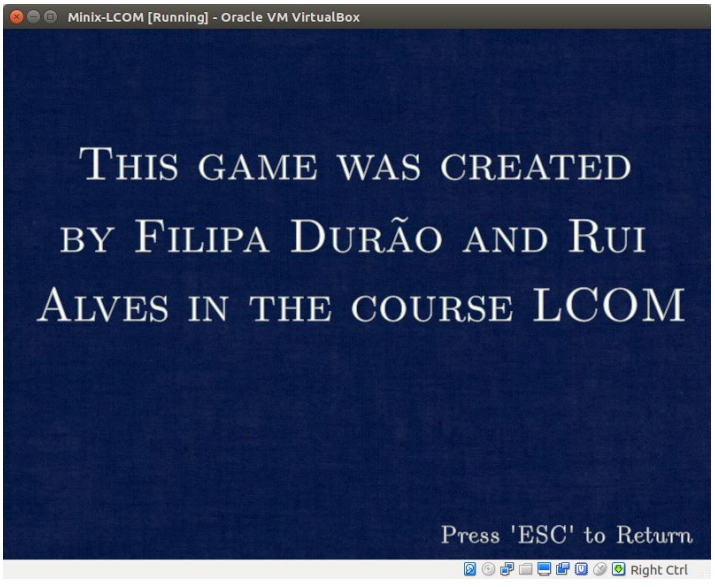
Instruções de Utilização

O programa inicia com um menu, em que as escolhas possíveis são apresentadas em botões. A escolha entre as várias opções é efetuada com o rato, clicando no botão correto.



- Opção “Credits”

Quando é selecionada a opção Credits, o utilizador é redirecionado para o ecrã dos créditos, onde é mencionado o nome dos criadores do jogo e onde é identificado o nome da Disciplina. O utilizador pode volta para o menu principal após premir a tecla ‘ESC’.

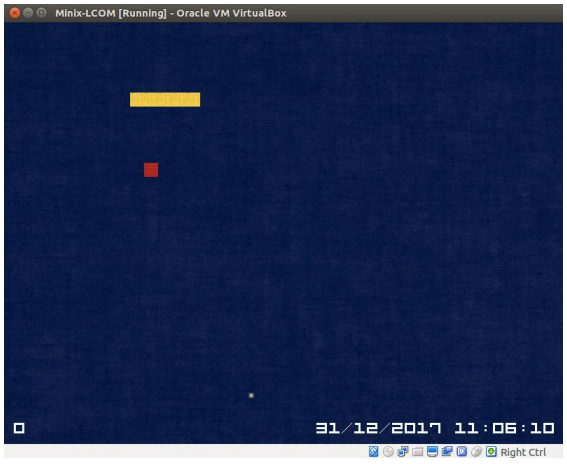


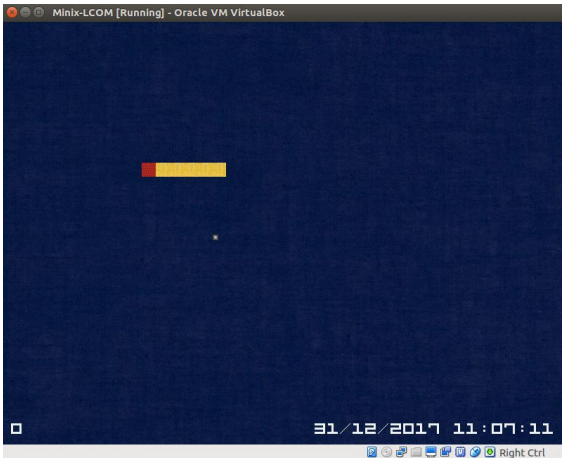
- Opção “Player 1”

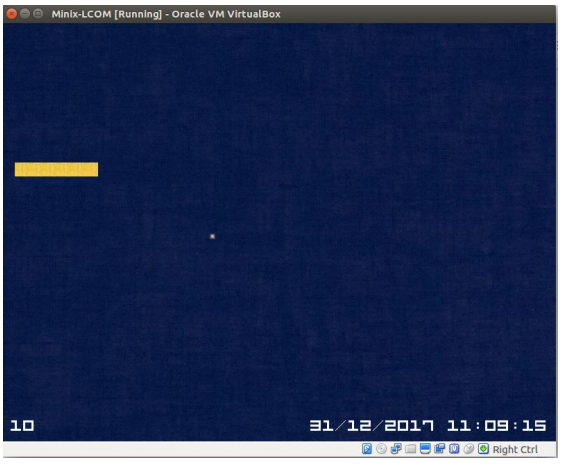
Quando é selecionada a opção “Player 1” o jogo principal é iniciado, ou seja, surge no ecrã a cobra, uma maçã, a pontuação do jogador (inicialmente 0), e a data e hora atual. Para mudar a direção da cobra, utilizam-se as teclas ‘w’, ‘a’, ‘s’ e ‘d’. A cobra vira 90º a cada tecla premida. Se a tecla premida for equivalente a uma direção de 180º nada se altera (por exemplo, se ela se estiver a movimentar para a direita e se tentar ‘virar’ para a esquerda).

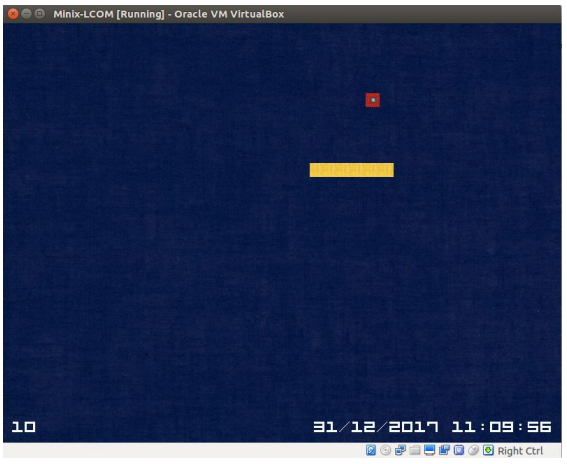
O objetivo do Player 1 é simples, **apanhar a maçã** presente no ecrã e nunca colidir com o próprio corpo. Ao apanhar a maçã a cobra cresce e a maçã desaparece. Para uma nova maçã aparecer no ecrã, o Player 1 deve **utilizar o rato** e clicar no local onde deseja que uma nova maçã apareça.

O jogador pode, a qualquer altura, sair do jogo, tendo unicamente de carregar na tecla ‘Esc’.



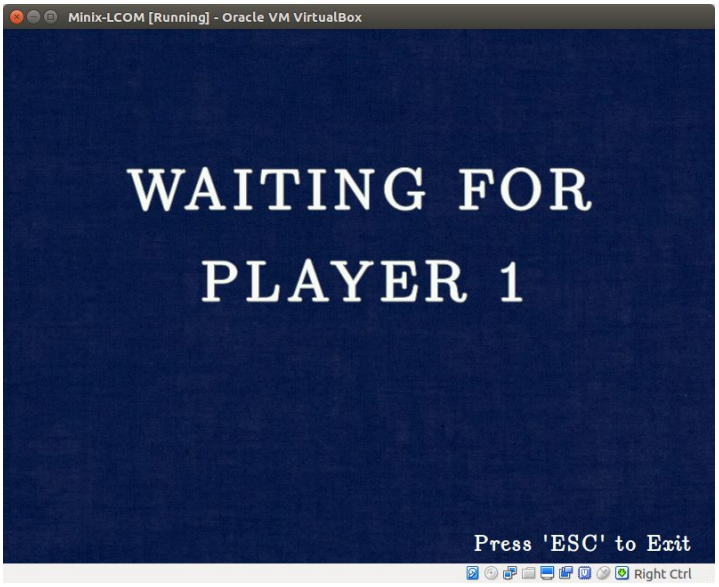






- Opção “Player 2”

Quando é selecionada a opção “Player 2” é iniciado o jogo “secundário”. Se na outra máquina não estiver a correr o jogo com o “Player 1”, o jogo fica em modo de “Aguardar pelo Jogador 1”, até ao Jogador 1 se juntar na outra máquina e a comunicação ter sido estabelecida.



Caso o Jogador 1 já se encontre a jogar na outra máquina , o Jogador 2 é automaticamente redirecionado para o modo de jogo. A forma como esta comunicação é efetuada está explicada no capítulo “Detalhes de Implementação” do presente relatório. No ecrã surgem 4 botões com setas que apontam para 4 direções diferentes. O objetivo do Player 2 é dificultar ao Player 1 o apanhar da maçã, movendo-a. O jogador seleciona a direção em que quer mover a maçã clicando no botão com a seta da direção desejada. A maçã no ecrã do Player 1 irá mover-se.

Se no decorrer do jogo o Jogador 1 perder ou sair do jogo voluntariamente, o "Jogador 2" é notificado e a comunicação entre as duas máquinas é quebrada. O Jogador 2 é imediatamente redirecionado para o estado representado na figura a seguir:

- Opção "Exit"

Sai do programa.

No fim do jogo, quando o Player 1 perde aparece no ecrã um "losing screen", que apresenta a pontuação final do Player. O "losing screen" desaparece ao fim de 10 segundos (utilizando um alarme do RTC) ou quando o jogador clica no botão "EXIT".

Estado do Projeto

A tabela a seguir representada contém a informação resumida de que periféricos utilizamos, como os utilizamos e se utilizamos ou não interrupções. O "ciclo de interrupções" (função onde é chamada a função **driver_receive()** e os *handlers* de cada periférico) localiza-se na função **update_game()** do módulo Game.

Periférico	Utilização	Interrupções
Timer	- Controlar o *frame rate*	Sim
KBD	- Mover a Cobra no ecrã	Sim
Mouse	- Escolher as opções no menu - Criar novas maçãs no ecrã (Player 1) - Mover a maçã no ecrã do adversário (Player 2)	Sim
Video Card	- Mostrar no ecrã os menus e o jogo	Não
RTC	- Obter a data e hora atuais - Alarmes	Sim
Serial Port	- Comunicação entre computadores durante o jogo	Sim

TIMER

O timer é utilizado para o controlo do *frame rate* do jogo.

Para esse controlo utilizamos as interrupções do timer (usando a sua frequência base, 60 interrupções por segundo). Foram implementadas funções para subscrição de interrupções (no decorrer do Laboratório 2), sendo essas funções **timer_subscribe_int()** e **timer_unsubscribe_int()**. O seu *Interruption Handler,* **timer_int_handler()**, foi feito utilizando *assembly* e está definido no ficheiro assembly_timer.S. Outra função importante é a função **handle_timer()**, que chama a função previamente mencionada e gera quaisquer eventos relevantes.

KBD

No projeto, o teclado é utilizado para controlo da Cobra, ou seja, para controlo dentro da aplicação de elementos do jogo, no modo "Player 1".

São utilizadas interrupções para verificação das teclas premidas. Foram implementadas funções para subscrição de interrupções (no decorrer do Laboratório 3), sendo essas funções **kbd_subscribe_int()** e **kbd_unsubscribe_int()**. O seu *Interruption Handler*, **kbd_int_handler()**, está definido em assembly no ficheiro assembly_kbd.S. Outra função importante é a função **handle_keyboard()**, que chama a função previamente mencionada e gera quaisquer eventos relevantes (Tecla 'W' pressionada, tecla 'ESC' pressionada, …).

MOUSE

O mouse é utilizado tanto nos menus, como dentro do jogo (quer no modo Player 1, quer no modo Player 2). Em ambos os casos, o mouse utiliza tanto os botões como o movimento, seja para selecionar opções no menu como para colocar novas maçãs no jogo (Player 1) e clicar em botões para mover a maçã (Player 2).

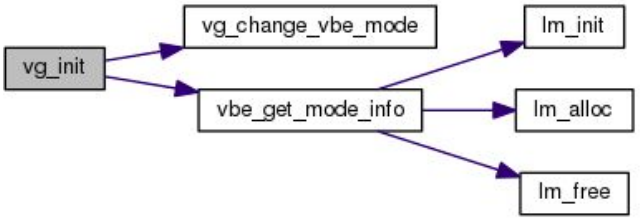
Funciona por interrupções, logo, sempre que há mudança de posição ou um clique de botão é gerada uma interrupção que é devidamente tratada pelo seu handler, **ps2_int_handler()**, que está definido em assembly no ficheiro assembly_ps2.S. Foram implementadas funções para subscrição de interrupções (no decorrer do Laboratório 4), sendo essas funções **ps2_subscribe_int()** e **ps2_unsubscribe_int()**. Outra função crucial é a função **ps2_enable_dataReporting()** que permite receber packets vindos do mouse. A função **handle_mouse()** chama o *interruption handler* previamente mencionado e gera quaisquer eventos relevantes (Botão Esquerdo pressionado, Botão Esquerdo largado, Rato Moveu-se).

VIDEO CARD

Para o nosso jogo usamos uma **resolução 800x600**, com 16 bits por pixel (o que perfaz um total de 2^16 = **65536 cores**), utilizando **RGB 5-6-5** (5 pixels para as cores vermelho e azul e pixels para a cor verde). O número do modo utilizado é **0x114** (276 em decimal).

No projeto é utilizado *double buffering,* assim como *fonts* e objetos que se movimentam.

No projeto é utilizado ***double buffering****,* sendo a alocação de memória para o **buffer principal** e para o **back-buffer** efetuada na função **vg_init()**. Esta função também é responsável por trocar para o modo de vídeo, chamando a função **vg_change_vbe_mode()**.



Para fazer a troca de buffers, não é utilizada qualquer função da VBE, sendo utilizada a função **flipBuffer()**, por nós implementada, que utiliza função memcpy() para copiar diretamente todo o conteúdo do back-buffer para o buffer da memória de vídeo. No fim do jogo é utilizada a função **vg_exit()** para libertar a memória alocada para os buffers e para retornar ao modo de escrita *default* do Minix.

Os objetos do jogo são representados no ecrã por bitmaps de 16 bits, utilizando a classe **Bitmap**, que se encontra devidamente explicada no capítulo "Organização/Estrutura do Código", sub-secção "Módulos da Placa de Vídeo" do presente Relatório.

Para mostrar texto no ecrã, recorremos tanto a imagens como a *fonts* por nós implementados (no módulo "font", devidamente explicado mais à frente, o capítulo "Organização/Estrutura do Código", sub-secção "Outros Módulos" do presente Relatório). Recorremos a imagens para texto estático (invariável) e a fonts para texto dinâmico (variável). No entanto, apenas tínhamos necessidade de carateres numéricos e de alguns carateres especiais, pelo que não utilizamos *fonts* de letras.

Utilizamos também **deteção de colisões** dos objetos de jogo**,** quer do rato com botões (feita no *event handler* do clique do rato, **mouse_lb_pressed_event_handler()**, do módulo "StateMachine". Note-se que esta função não se encontra documentada por ser *static*), quer da cabeça da cobra com a maçã, quer da cabeça da cobra com o seu próprio corpo (funções **check_snake_head_apple_collision()** e **check_snake_self_collision()** do módulo "Snake").

RTC

Do periférico Real-Time Clock são usadas as funcionalidades de **leitura da data e hora** e as **interrupções de alarme.**

A data e hora atuais são apresentadas no ecrã de jogo do Player 1, e são obtidas através da função **get_date()** do módulo "utilities", que utiliza os getters do rtc para retornar a

data e hora atuais sob a forma de uma string. Foram implementadas funções para subscrição de interrupções, sendo essas funções **rtc_subscribe_int()** e **rtc_unsubscribe_int()**. O seu *interruption handler* é a função **handle_rtc()** que verifica que tipo de interrupção ocorreu e gera os respetivos eventos relevantes.

A **funcionalidade de alarme** é utilizada no final do jogo, para limitar o tempo que o *losing screen* permanece no ecrã, utilizando a função **rtc_set_alarm()**.

SERIAL PORT

Para efetuar a comunicação entre duas máquinas para o nosso jogo (entre o Player 1 e o Player 2) utilizamos a porta série com interrupções para a receção e com polling para o envio (sendo que a leitura de bytes denominados "acknowledgement bytes", função **uart_get_acknowledgement_byte()**, efetuada com base no nosso protocolo é feita utilizando polling, ao contrário da leitura de mensagens que é feita utilizando interrupções).

A configuração da porta série que utilizamos é de 8 bits por carater, com 2 stop bits e com paridade ímpar. Utiliza interrupções de Dados Recebidos e de Erro, sendo o tratamento destas interrupções efetuada pela função **handle_uart()**. O bit-rate utilizado é de 9600 bits por segundo. Toda esta configuração é realizada na função **configure_uart_SnakeyNix()** do módulo "utilities", que por sua vez chama as funções necessárias para cada parte da configuração.

Para tornar a comunicação mais robusta e mais à prova de falhas, foi implementado um protocolo de comunicação, que é explicado a fundo mais à frente no capítulo "Detalhes de Implementação" do presente relatório. O envio de carateres e de mensagens completas é feita utilizando as funções **uart_write_character()** e **uart_write_message()**, sendo a sua receção e parsing efetuada pela função **uart_parse_message()**.

O envio das mensagens é efetuado com frequência variável (depende do *input* do Jogador). O conteúdo das mensagens é maioritariamente de eventos (Seta direita pressionada, Seta esquerda pressionada, …), sendo também enviadas mensagens de sincronização para notificar um jogador quando o outro se juntou ao jogo (Jogador 1 juntou-se ao jogo, Jogador 1 saiu do Jogo, Jogador 2 juntou-se ao jogo, …)

Organização/Estrutura do Código

No desenvolvimento do projeto, dividimos o código em diversos módulos, de forma a tornar o código mais estruturado e legível. Os diagramas de chamada de funções foram gerados com o Doxygen, pelo que estão contidos na documentação gerada todos os gráficos de chamadas de funções, estando apenas no presente relatório os diagramas "mais importantes".

Módulos Relacionados com Periféricos

Os módulos a seguir apresentados são relacionados com o "manuseamento" direto dos periféricos, tendo sido grande parte deles realizado no decorrer dos vários laboratórios no decorrer do semestre (módulos relativos ao timer, kbd, mouse e placa de vídeo).

Módulos do Timer

Módulo 'timer'

Neste módulo encontram-se funções para manipulação direta do timer (handler do timer, funções de subscrição de interrupções). Este módulo foi desenvolvido no decorrer do Laboratório número 2, tendo sido realizado igualmente por ambos os membros.

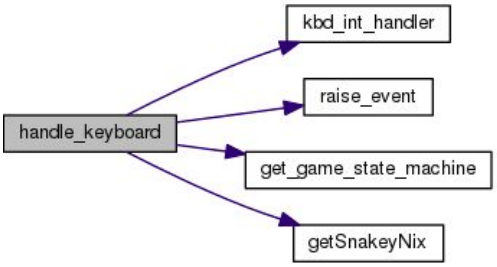
Módulo 'i8254'

Neste módulo encontram-se todas as constantes simbólicas para manipulação do timer. Este módulo foi-nos fornecido para a realização do Laboratório 2, não sendo, portanto, da nossa autoria.

Módulos do Keyboard

Módulo ‘kbd’

Neste módulo encontram-se funções para manipulação direta do teclado (handler do keyboard, funções de subscrição de interrupções, funções de escrita e leitura em registos do kbc, …). Este módulo foi desenvolvido no decorrer do Laboratório número 3, tendo sido realizado igualmente por ambos os membros.



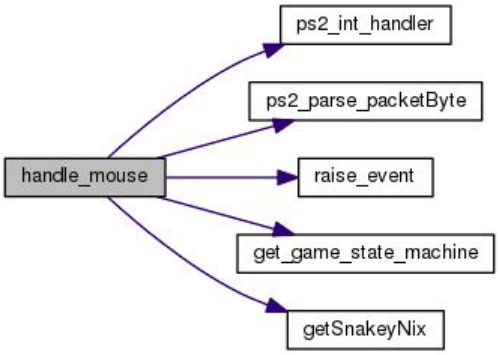
Módulo ‘i8042’

Neste módulo encontram-se todas as constantes simbólicas para manipulação do keyboard e também do rato (ps2), visto que ambos utilizam o *KBC Controller*. Este módulo foi desenvolvido no decorrer do Laboratório número 3, tendo sido realizado igualmente por ambos os membros.

Módulos do Rato

Módulo ‘ps2’

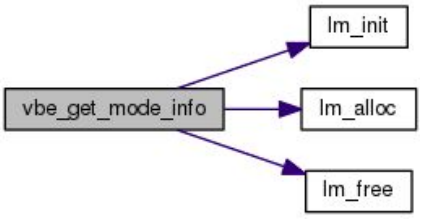
Neste módulo encontram-se funções para manipulação direta do mouse (handler do mouse, funções de subscrição de interrupções, funções de escrita e leitura em registos do kbc, …). Este módulo foi desenvolvido no decorrer do Laboratório número 4, tendo sido realizado igualmente por ambos os membros.



Módulos da Placa de Vídeo

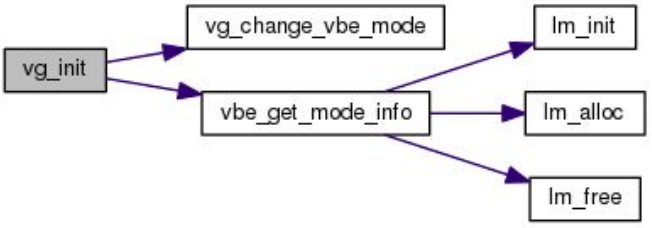
Módulo ‘vbe’

Neste módulo encontram-se funções para manipulação da VBE. Este módulo foi-nos fornecido parcialmente completo para a realização do Laboratório número 5, tendo sido completado igualmente por ambos os membros.



Módulo ‘video_gr’

Neste módulo encontram-se funções para manipulação direta dos buffers de vídeo (*back-buffer* e buffer da memória de vídeo). Este módulo foi-nos fornecido parcialmente completo para a realização do Laboratório número 5, tendo sido completado igualmente por ambos os membros.



Módulo ‘video’

Neste módulo encontram-se todas as constantes simbólicas para manipulação de serviços da BIOS e, mais especificamente, da VBE. Este módulo foi desenvolvido no decorrer do Laboratório número 5, tendo sido realizado igualmente por ambos os membros.

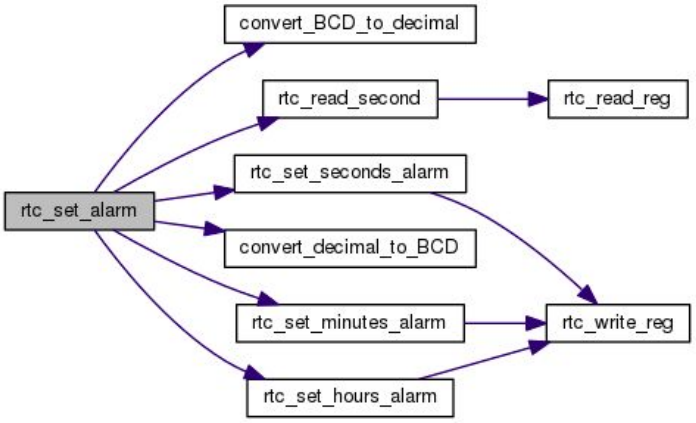
Módulo ‘lmlib’

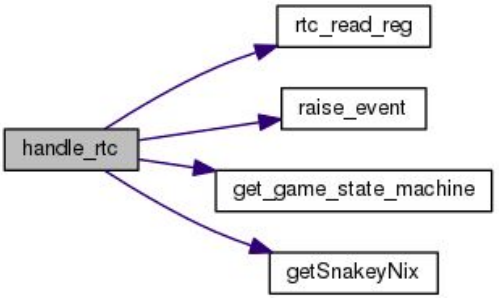
Neste módulo encontram-se as funções para manipulação de memória no primeiro Mega Byte de memória física (necessário para a BIOS). Este módulo foi-nos fornecido para a realização do Laboratório 5, não sendo, portanto, da nossa autoria.

Módulos do RTC

Módulo ‘rtc’

Neste módulo encontram-se funções para manipulação direta do RTC (handler do RTC, funções de subscrição de interrupções, funções de escrita e leitura em registos do RTC, funções para programação de alarmes, …). Também se encontram neste módulo todas as constantes simbólicas para manipulação do RTC. Este módulo foi desenvolvido pela Filipa Durão.

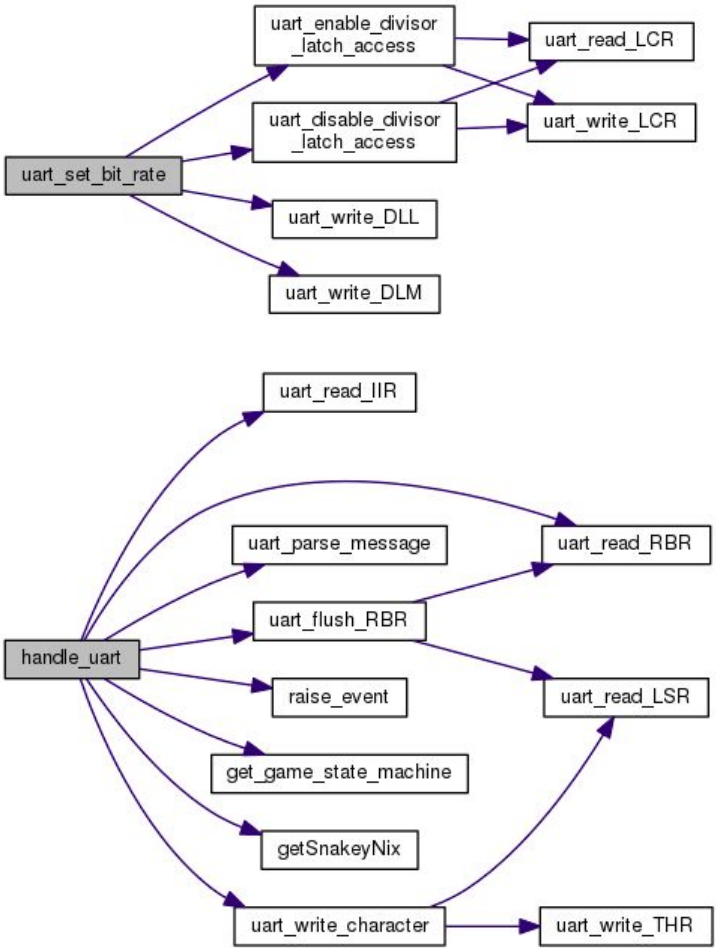




Módulos da Porta Série

Módulo ‘portaserie’

Neste módulo encontram-se funções para manipulação direta do UART (handler do UART, funções de subscrição de interrupções, funções de escrita e leitura em registos do UART, configuração do UART, …). Encontram-se também neste módulo algumas funções relacionadas com o protocolo de comunicação. Este módulo foi desenvolvido pelo Rui Alves.



Módulo ‘spProtocol’

Neste módulo encontram-se todas as constantes simbólicas e dados relativos ao funcionamento do protocolo desenhado para a comunicação utilizando a porta série. O funcionamento em concreto deste protocolo está explicado no capítulo “Detalhes de Implementação” do presente relatório. Este módulo foi realizado pelo Rui Alves.

Módulo ‘uart’

Neste módulo encontram-se todas as constantes simbólicas para manipulação do UARTe de todos os seus registos. Este módulo foi desenvolvidos pelo Rui Alves.

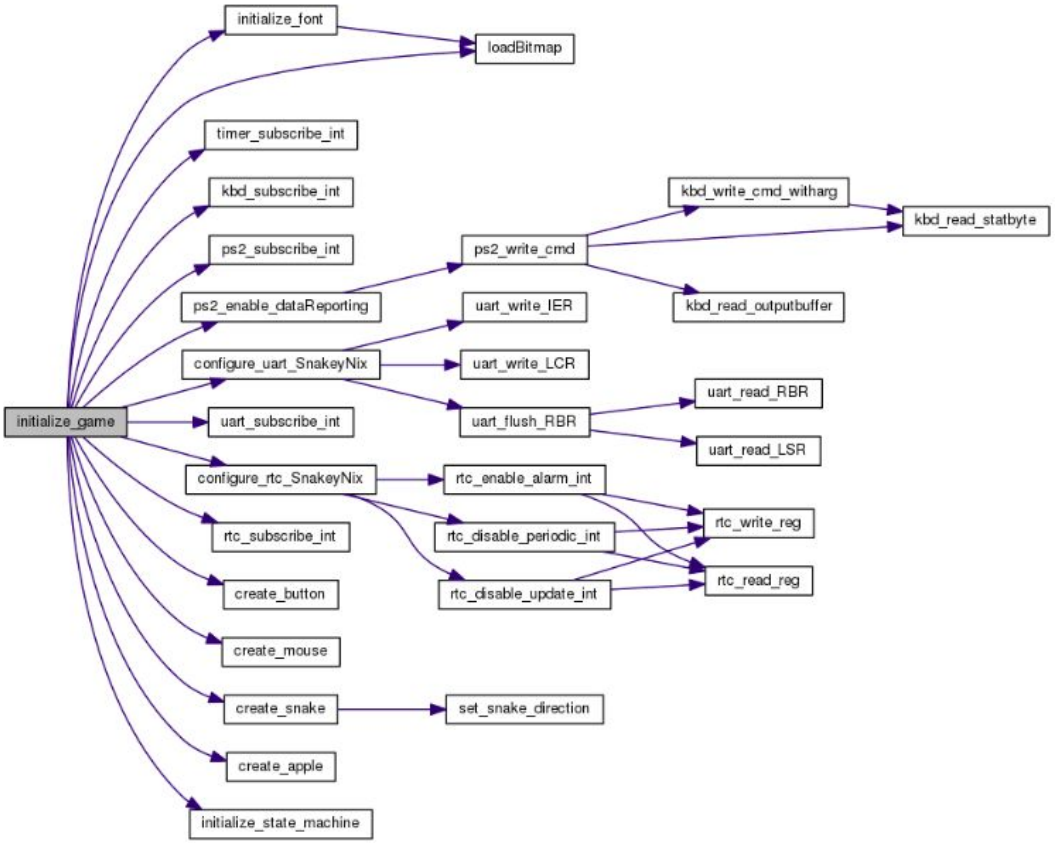
Módulos Relacionados com Classes do Jogo

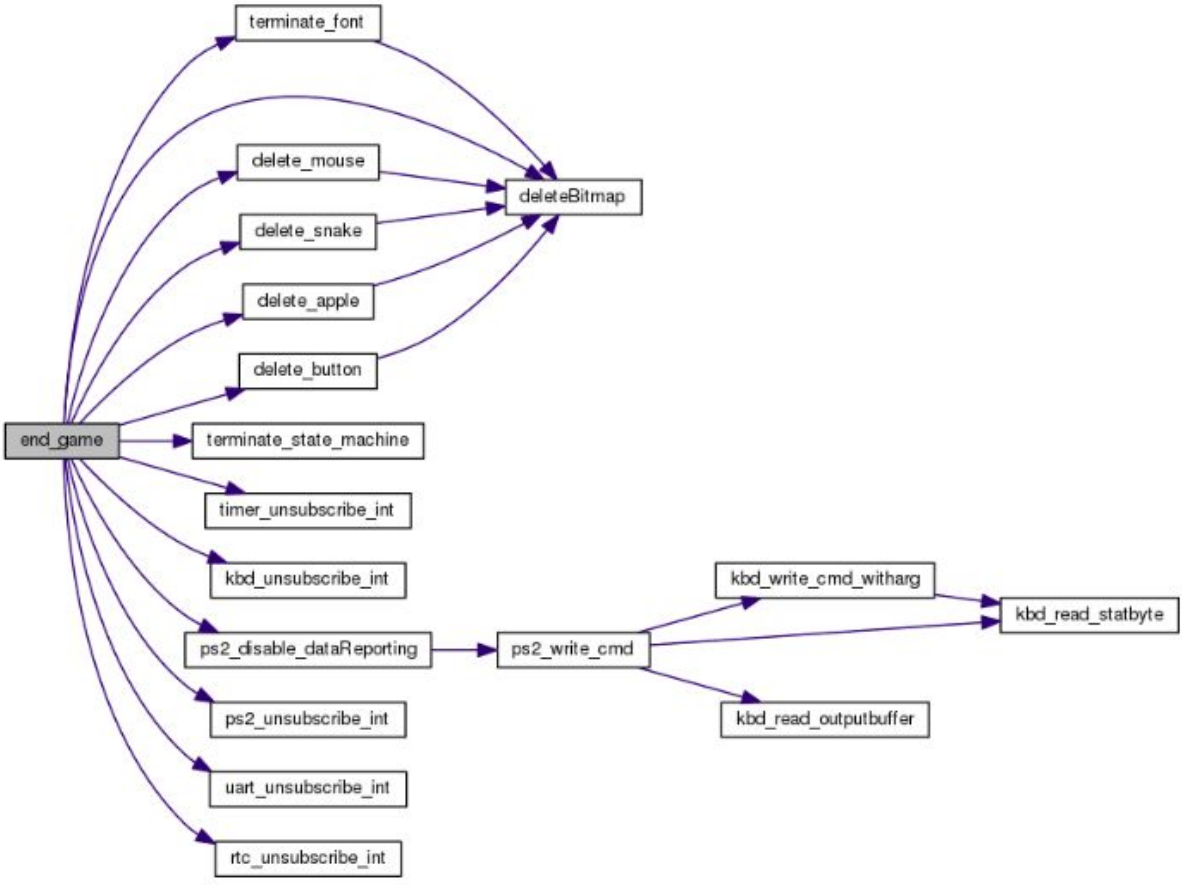
Os módulos a seguir apresentados são relacionados com o as classes implementadas a fim de representar objetos no jogo, seguindo uma paradigma “orientado a objetos”, sendo este tópico explorado mais a fundo no capítulo “Detalhes de Implementação” do presente relatório.

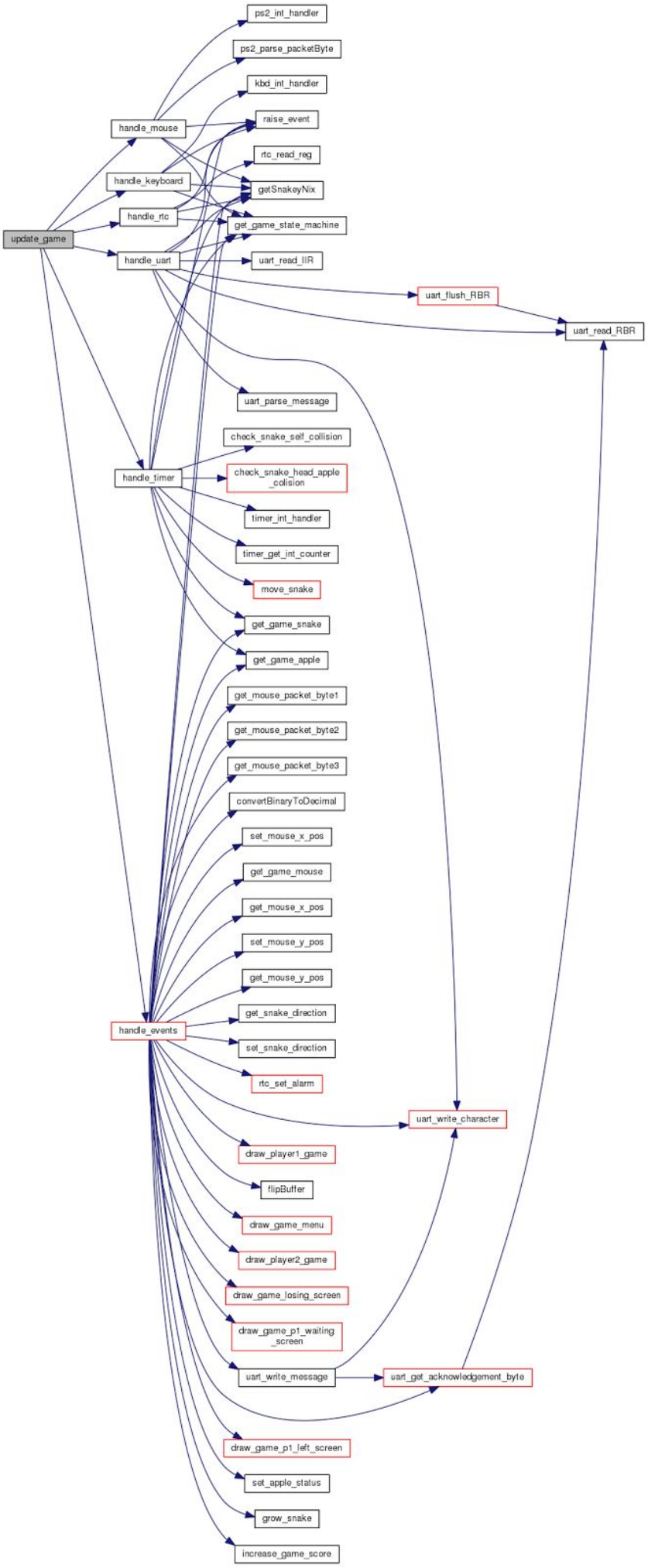
Módulo ‘Game’

Neste módulo encontra-se a implementação da classe Game, que representa em si o nosso jogo, sendo um dos “módulos principais”. Esta classe classe contém todos os objetos de jogo, sendo de certa forma o motor de jogo do nosso projeto. Este módulo foi realizado pelo Rui Alves. Os diagramas de chamada de funções mais relevantes do Jogo apresentam-se nas duas páginas seguintes.

É na função **update_game()** deste módulo que se realiza a verificação da ocorrência de interrupções (chamada à função **driver_receive()** e onde são chamados todos os *interrupt handlers*.


initialize_font
loadBitmap
timer_subscribe_int
kbd_subscribe_int
ps2_subscribe_int
ps2_enable_dataReporting
ps2_write_cmd
kbd_write_cmd_witharg
kbd_read_statbyte
kbd_read_outputbuffer
configure_uart_SnakeyNix
uart_write_IER
uart_write_LCR
uart_flush_RBR
uart_read_RBR
uart_read_LSR
initialize_game
uart_subscribe_int
configure_rtc_SnakeyNix
rtc_enable_alarm_int
rtc_disable_periodic_int
rtc_disable_update_int
rtc_write_reg
rtc_read_reg
rtc_subscribe_int
create_button
create_mouse
create_snake
set_snake_direction
create_apple
initialize_state_machine


terminate_font
delete_mouse
deleteBitmap
delete_snake
delete_apple
delete_button
end_game
terminate_state_machine
timer_unsubscribe_int
kbd_unsubscribe_int
ps2_disable_dataReporting
ps2_write_cmd
kbd_write_cmd_witharg
kbd_read_statbyte
kbd_read_outputbuffer
ps2_unsubscribe_int
uart_unsubscribe_int
rtc_unsubscribe_int


update_game
handle_mouse
handle_keyboard
handle_rtc
handle_uart
handle_timer
handle_events
ps2_int_handler
ps2_parse_packetByte
kbd_int_handler
raise_event
rtc_read_reg
getSnakeyNix
get_game_state_machine
uart_read_IIR
uart_flush_RBR
uart_read_RBR
uart_parse_message
check_snake_self_collision
check_snake_head_apple_colision
timer_int_handler
timer_get_int_counter
move_snake
get_game_snake
get_game_apple
get_mouse_packet_byte1
get_mouse_packet_byte2
get_mouse_packet_byte3
convertBinaryToDecimal
set_mouse_x_pos
get_game_mouse
get_mouse_x_pos
set_mouse_y_pos
get_mouse_y_pos
get_snake_direction
set_snake_direction
rtc_set_alarm
uart_write_character
draw_player1_game
flipBuffer
draw_game_menu
draw_player2_game
draw_game_losing_screen
draw_game_p1_waiting_screen
uart_write_message
uart_get_acknowledgement_byte
draw_game_p1_left_screen
set_apple_status
grow_snake
increase_game_score

Módulo ‘Apple’

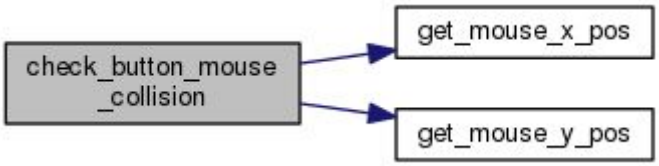
Neste módulo encontra-se a implementação da classe Apple, que representa um maçã (alimento da cobra, o seu objetivo é apanhar as maçãs). Este módulo foi realizado pelo Rui Alves.

Módulo ‘Bitmap’

Neste módulo encontra-se a implementação de uma classe para manipulação de imagens no formato .bmp. Este módulo não foi desenvolvido pelo nosso grupo, mas por um ex-aluno da FEUP, **Henrique Ferrolho**, que a disponibilizou no seu blog online (http://difusal.blogspot.pt/2014/09/minixtutorial-8-loading-bmp-images.html). Foi pedida autorização via Moodle aos docentes da unidade curricular para a utilização do código em questão. No entanto, **efetuamos algumas alterações** na classe por ele implementada: Adicionamos à função “drawBitmap()” um parâmetro extra, que consiste na base de memória do buffer no qual se pretende desenhar (foi necessário visto usarmos double buffering no projeto). Implementamos também uma nova função, “drawBitmapWithTransparency()”, que ao fazer o desenho do Bitmap num buffer, ignora uma cor em específico, permitindo assim utilizar transparência.

Módulo ‘Button’

Neste módulo encontra-se a implementação da classe Button, que representa um botão, sendo os objetos desta classe utilizados quer no menu principal, que no modo “Player 2”. Esta classe é um exemplo de uma classe que utiliza verificação de colisões. Este módulo foi realizado pelo Rui Alves.



Módulo ‘Mouse’

Neste módulo encontra-se a implementação da classe Mouse, que representa o ponteiro do rato utilizado em todo o jogo. Esta classe foi implementada pelo Rui Alves.

Módulo ‘Snake’

Neste módulo encontra-se a implementação da classe Snake, que representa a cobra que o jogador é responsável por movimentar. Esta classe utiliza um classe ‘auxiliar’ também implementada neste módulo, classe SnakeNode, que representa um ‘nó’ da cobra, que é implementada com recurso a uma lista duplamente ligada. Esta implementação está explicada mais a fundo no capítulo “Detalhes de Implementação” do presente relatório. Esta classe foi implementada pelo Rui Alves.

Módulo ‘StateMachine’

Neste módulo encontra-se a implementação da classe StateMachine, que representa uma máquina de estados utilizada no nosso jogo. O funcionamento da máquina de estados está explicado mais a fundo no capítulo “Detalhes de Implementação” do presente relatório. Esta classe foi implementada pelo Rui Alves, tendo sido projetada por ambos os membros.

Outros Módulos

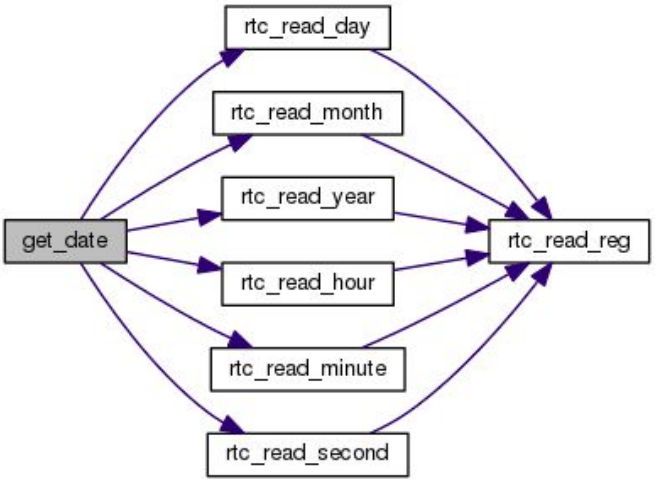
Os módulos a seguir apresentados contêm funções extra que possuem utilidade para o projeto / que não se enquadram nos grupos de módulos mencionados anteriormente.

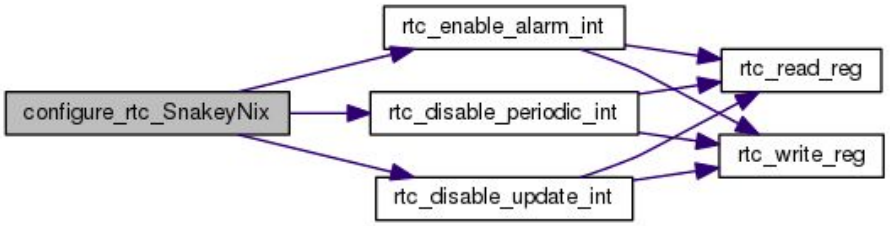
Módulo ‘SnakeyNix’

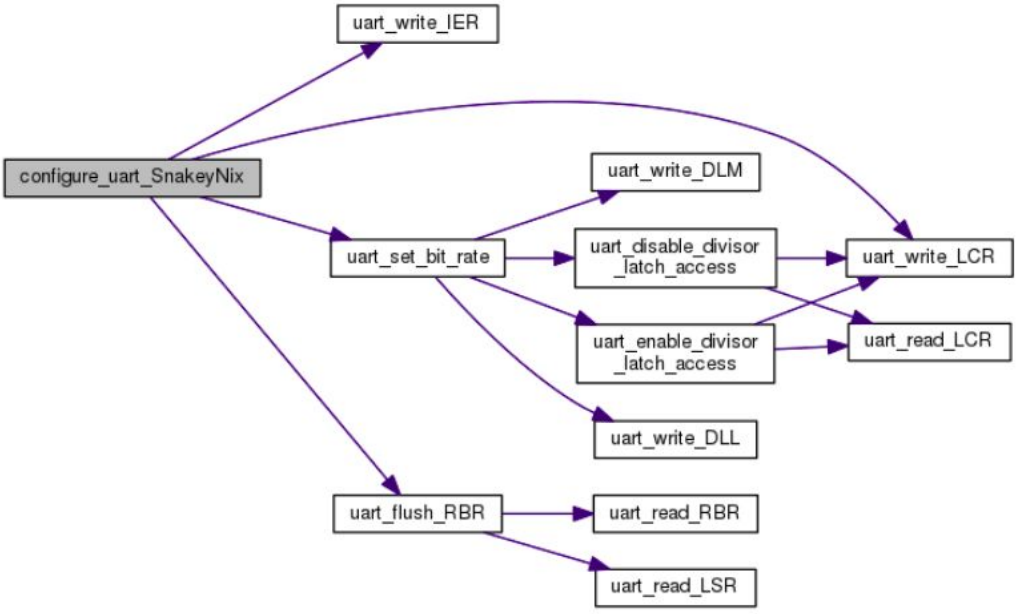
Neste módulo encontra-se a função main(), onde é criado o objeto de jogo e inicializado o modo de vídeo. Este módulo foi implementado pelo Rui Alves.

Módulo ‘utilities’

Neste módulo encontram-se funções diversas com utilidades distintas que foram implementadas ao longo dos laboratórios e do projeto. Contém *typedefs* utilizados ao longo de todo o projeto, macros úteis, funções de arredondamento e valor absoluto, funções de configuração e de impressão de data e hora. Este módulo foi implementado por ambos os membros do grupo.







Módulo ‘font’

Neste módulo encontram-se funções para manipulação de texto no modo de vídeo. No entanto, o único texto que precisamos de imprimir “dinamicamente” consiste apenas em números e alguns carateres especiais, pelo que apenas desenhamos sprites para os carateres que necessitamos, pelo que não manipulamos todo o tipo de carateres. Os *fonts* estão ainda disponíveis em tamanhos diferentes. Este módulo foi implementado por ambos os membros do grupo.

Gráficos do Jogo

Todos os gráficos do jogo (bitmaps dos objetos de jogo, botões, fundos e *font*) foram realizados pela Filipa Durão.

Os fonts são imagens separadas para cada número, realizadas num programa de desenho digital para Linux (Pinta). Foram escritos alguns carateres especiais e todos os números em dois tamanhos diferentes (grandes e pequenos) para impressão no ecrã em dois momentos diferentes, grandes no ecrã final (pontuação final) e pequenos durante o jogo (com a data, hora e pontuação). Estas imagens foram **convertidas para bitmaps R5 G6 B5 (16 bits)** e foram integradas no módulo **font**.

Todo o resto do aspeto visual foi criado a partir de simples fotografias de texturas de tecido, o fundo, a cobra, as maçãs, entre outros. Os botões usam as ditas imagens de tecido e texto inserido por nós (mais uma vez no Pinta).

Documentação do Projeto

Todos os módulos do projeto foram devidamente documentados utilizando *Doxygen*. No *Doxyfile* foi selecionada a opção de gerar diagramas de chamada de função. A maior parte da documentação foi efetuada pela Filipa Durão.

Para além do *Doxyfile em /proj/doc* , adicionamos também uma pasta comprimida com toda a documentação ao módulo *Documents* no redmine.

Detalhes de Implementação

No nosso projeto, tal como no decorrer dos laboratórios, desenhamos as **funções** de forma a estas ficarem **distribuídas por várias camadas**, de modo ao código ficar mais bem estruturado e organizado e também a facilitar a construção do projeto (uma camada de funções que interage mais com os periféricos, maioritariamente input e output de registos, uma camada um pouco mais alta que utiliza estas funções para fazer verificações, entre outros e camadas de níveis superiores que utilizam estas funções como abstrações. Desta forma, problemas complicados tornam-se mais ou menos triviais).

Utilizamos também no nosso jogo uma **máquina de estados**, sendo todo o nosso código **orientado a eventos**. Desta forma, aquando uma interrupção (originada por qualquer periférico) , a causa da interrupção é analisada e são gerados eventos (por exemplo, tecla 'W' premida, movimento do rato, *tick* do timer, botão premido, …) que são colocados num ***buffer* de eventos** na máquina de estados, sendo os eventos tratados e, consequentemente, o *buffer* esvaziado a cada ciclo após a verificação de interrupções de todas as origens, sendo os *interruption handlers* independentes da aplicação. Os **event handlers estão contidos num array**, e são **indexados pelo tipo de evento**, o que torna o tratamento de eventos bastante eficiente. Esta implementação é um pouco mais trabalhosa de início, mas torna o desenvolvimento muito mais fácil e o código muito mais legível.

O nosso jogo é também fortemente **orientado a objetos**, sendo todos os elementos instâncias de uma classe. Todo o jogo é em si um objeto da classe *Game*. Todos os objetos têm construtor, destrutor, métodos get e set para os seus atributos e funções para desenho dos mesmos. A definição da struct é feita no ficheiro 'nome_classe.c' ao invés de ser definida no header file, de forma a tornar os atributos visíveis apenas no próprio módulo, de forma a encapsular a informação (na imagem abaixo apresentada, ficheiro .c à esquerda e ficheiro .h à direita). Alguns exemplos de classes usadas são as classes Game, Snake, Apple, Button, …

```c
struct Apple_Class{
    // Apple's Position
    short x_pos;
    short y_pos;

    // Apple's Graphics
    Bitmap* bitmap;

    // Apple's status
    int is_active;
};
```

```c
//////////////////////
//  Apple "Class"  //
//////////////////////

struct Apple_Class;
typedef struct Apple_Class Apple;
```

Quanto à **geração de *frames*** , os eventos são gerados aquando a ocorrência de interrupções (provenientes do teclado, timer, rato, RTC e porta śerie) , sendo posteriormente “tratados”. A cada *tick* do timer (60 vezes por segundo) , todos os objetos do jogo são desenhados num buffer auxiliar (o que permite que o rato tenha um *frame-rate* de 60 FPS, embora os restantes objetos do jogo só sejam atualizados a cada 2 *ticks* do timer, tendo na realidade um *frame-rate* de 30 FPS). Após todo o frame ser gerado, 100% do conteúdo deste buffer auxiliar é copiado diretamente para o buffer da memória de vídeo (utilizando a função *memcpy* da stdlib.h), estratégia de *double buffering.*

Utilizamos também algum **código assembly**, quase exclusivamente nos *device drivers* de alguns periféricos (timer , mouse e teclado), tendo usado sintaxe AT&T.

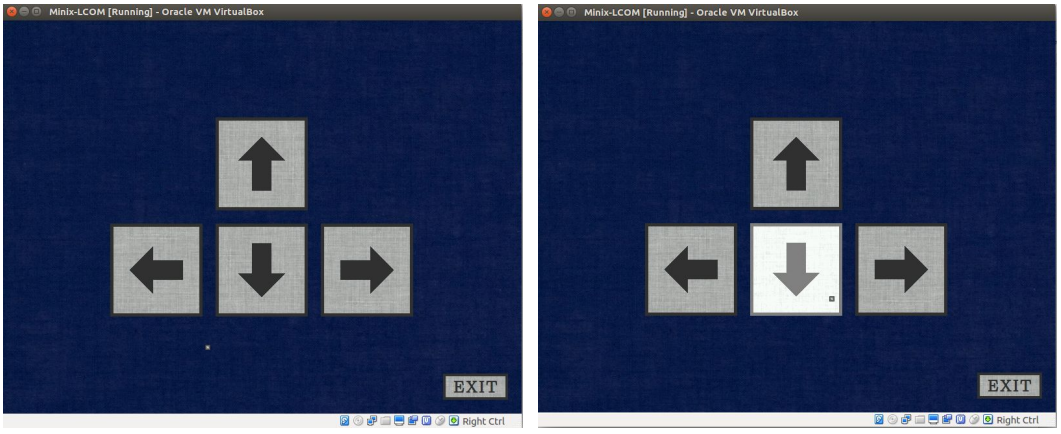
Para estabelecer comunicação entre duas máquinas, fizemos uso da **porta série**. Para tornar esta comunicação **robusta** e o mais à prova de falhas possível, elaboramos um **protocolo de comunicação**, em que cada mensagem é composta por um **header byte**, pelo conteúdo da mensagem em si e, por fim, por um **trailer byte**. Esta forma de comunicação permite identificar mensagens corrompidas/inválidas, tendo desta forma a comunicação robustez para detetar erros para além da verificação da paridade da mensagem e dos *stop-bits*. Para tornar este protocolo ainda mais robusto, implementamos também um sistema de ***acknowledgement bytes***, idêntico ao usado pelo “*ps2 mouse controller*”. Desta forma, quando uma das máquinas recebe uma mensagem, envia para a outra máquina um *acknowledgement byte* (ACK) para notificar que a mensagem foi recebida com sucesso. Caso haja alguma falha na receção (header byte inválido, trailer byte inválido, …) é enviado um *not-acknowledged byte* (NACK) para solicitar um **re-envio de toda a mensagem**. No caso de ocorrer uma destas falhas de comunicação, são efetuadas mais duas tentativas (perfazendo um total de 3 tentativas), com um *delay* de 10 milisegundos entre o envio das mensagens. Se nestas tentativas não houver sucesso de comunicação, é enviado um *error byte* (ERROR). Este protocolo de comunicação revelou-se bastante robusto na fase de testes.

Para **sincronizar os dois jogadores**, cada um na sua máquina, o programa força a que o Jogador 2 só possa jogar se o Jogador 1 estiver também a jogar, visto que o Jogador 2 interage diretamente com o jogo do Jogador 1 (no entanto, ambas as máquinas podem jogar como Jogador 1 sem qualquer problema). Para esta comunicação ser efetuada, quando o Jogador 2 se junta ao jogo, tenta estabelecer comunicação com o Jogador 1. Caso não obtenha resposta, fica num estado de espera (tal como mostrado anteriormente no capítulo “Instruções de Utilização”). Quando o Jogador 1 se junta, notifica o Jogador 2 que se juntou, e ambos começam o jogo em simultâneo. Caso o Jogador 1 perca o jogo ou saia por vontade própria, é enviada uma mensagem para o Jogador 2 a notificar que o Jogador 1 saiu, sendo o Jogador 2 notificado e o seu jogo terminado.

Utilizamos alguns conceitos de **estruturas de dados**, que se revelaram bastante úteis na implementação da classe **Snake**. Para implementar a cobra, utilizamos uma **lista duplamente ligada**, que permite facilmente representar a cobra. Desta forma, mantendo um apontador para a cabeça da cobra e para o último ‘nó’ da cobra, é bastante eficiente adicionar nós quer no fim da cobra quer no início. Com a implementação da cobra em lista

duplamente ligada, é também bastante eficiente fazer o movimento da cobra: basta adicionar um novo nó na cabeça da cobra na direção em que a cobra se está a mover (este nó passa a ser a cabeça da cobra) e remover o último nó da cobra. Os outros nós mantêm a sua posição. Desta forma, o movimento da cobra é igualmente eficiente para qualquer tamanho que a cobra tome

No nosso jogo foi também necessário ter atenção há **ocorrência de colisões,** quer no movimento da cobra (colisão com a maçã e com ela própria, sendo esta última a condição de derrota), quer no clique do rato (com os botões no modo 'Player 2' e no 'Menu' e com a cobra quando o clique é efetuado para colocar a maçã, sendo que a maçã não pode ser colocada no interior da cobra). Quanto às colisões da cobra com a maçã, apenas é feita a verificação da sobreposição da posição da cabeça da cobra e da posição da maçã. Esta verificação apenas precisa de ser feita cada vez que a cobra se move. Quanto à verificação da colisão da cobra com ela própria, é necessário verificar se a posição da cabeça da cobra se sobrepõe à posição de qualquer outro dos nós que constituem a cobra. Esta verificação também é apenas feita aquando o movimento da cobra. As colisões do clique do rato também são apenas verificadas aquando o evento de clique de rato. No entanto, nos menus e no modo 'Player 2', os botões acendem quando o rato se sobrepõe aos mesmos (como demonstrado na imagem abaixo). Esta verificação de colisões é verificada sempre que ocorre um evento de deslocação do rato.



Conclusões

A cadeira de LCOM foi, muito provavelmente, a cadeira mais difícil que tivemos no nosso percurso académico. No entanto, foi também a mais desafiante e a mais divertida, visto que foi a primeira cadeira em que realmente precisamos pesquisar e de descobrir coisas por nós próprios. É de louvar também a participação dos docentes na cadeira, visto que todas as dúvidas colocadas no moodle são respondidas quase de imediato e de forma muito completa.

No entanto, há alguns tópicos que gostariamos de ver lecionados com mais profundidade:

- Como no projeto a maior parte dos alunos implementa um jogo, era interessante e muito útil que pelo menos uma aula teórica fosse dedicada a explicar o funcionamento de uma máquina de jogo: Como é que é estruturado? Quais são as funções chave a implementar num jogo? Onde gerar e atender a eventos? Estas foram algumas das questões que nos surgiram no início do desenvolvimento no projeto, não tendo sido trivial encontrar uma resposta para estas questões. Uma explicação prévia, mesmo que breve, do tipo de estruturação necessária poderia ter dado um auxílio no desenho inicial do projeto.
- A porta série é sem dúvida o periférico mais desafiante. Apesar de não ser objeto obrigatório de avaliação, o seu uso é indispensável para quem quiser atingir notas altas e para quem quer aprender mais sobre comunicação / protocolos de comunicação. Mesmo sendo estes tópicos abordados posteriormente no decorrer do curso, pensamos que seria muito útil ter pelo menos mais uma ou duas aulas dedicadas à porta série, devido à dificuldade acrescida deste periférico.

Por outro lado, houve tópicos cruciais abordados que serão úteis no desenvolvimento de qualquer aplicação futura que achamos importante salientar:

- Máquinas de Estados são um tópico "difícil de digerir" inicialmente, mas o tópico foi abordado em mais do que uma aula teórica, o que ajudou bastante a perceber as subtilezas da programação orientada a eventos.
- As aulas iniciais em que se abordou a estrutura do sistema operativo foi bastante útil para perceber as diferenças entre os vários níveis de privilégios e a estrutura das várias camadas do sistema operativo. Vamos ter no curso a cadeira de Sistemas Operativos, mas achamos importante esta introdução mais prematura, que ajuda a compreender melhor a estrutura (ainda que simplificada) de um sistema operativo.

Apêndice - Instruções de Instalação

Para instalar e correr o jogo, incluímos 3 *shell scripts* dentro da pasta 'proj', que podem ser utilizados para uma instalação rápida e fácil. Estes scripts foram feitos com a ajuda do blog online de um ex-aluno da FEUP, Henrique Ferrolho, que se encontra na página http://difusal.blogspot.pt/2014/08/minixtutorial-3-setting-up-project-with.html (foram feitas algumas alterações, não tendo os scripts sido copiados). Os scripts "install.sh" e "play.sh" devem ser executados como **super user**, pois necessitam das devidas permissões.

Para mover os ficheiros de imagem para uma pasta neutra (/home/lcom/SnakeyNix) e colocar o ficheiro de configuração no seu local correto (etc/system.conf.d), o utilizador deve executar o script **installSnakeyNix**, com o comando "sh installSnakeyNix.sh".

De seguida, deve proceder à compilação do código ao executar o script **compile**, com o comando "sh compile.sh" (que move também o executável, após a compilação, para a pasta atual, pasta proj).

Por fim, executar o script **play** para inicializar o jogo, com o comando "sh play.sh". Todos os scripts se encontram devidamente comentados para mais fácil compreensão do seu funcionamento. As imagens a seguir exemplificam o processo anteriormente explicado:

```
# sh installSnakeyNix.sh
cp resources/apple.bmp ..
cp resources/background.bmp ..
cp resources/bar.bmp ..
cp resources/bigEight.bmp ..
cp resources/bigFive.bmp ..
cp resources/bigFour.bmp ..
cp resources/bigNine.bmp ..
cp resources/bigOne.bmp ..
cp resources/bigSeven.bmp ..
cp resources/bigSix.bmp ..
cp resources/bigThree.bmp ..
cp resources/bigTwo.bmp ..
cp resources/bigZero.bmp ..
cp resources/downDefaultButton.bmp ..
cp resources/downHoveredButton.bmp ..
cp resources/eight.bmp ..
cp resources/exitDefaultButton.bmp ..
cp resources/exitHoveredButton.bmp ..
cp resources/five.bmp ..
cp resources/four.bmp ..
cp resources/leftDefaultButton.bmp ..
cp resources/leftHoveredButton.bmp ..
cp resources/losingScreen.bmp ..
cp resources/mouse.bmp ..
cp resources/nine.bmp ..
cp resources/one.bmp ..
cp resources/p1leftScreen.bmp ..
cp resources/p1waitingScreen.bmp ..
cp resources/player1DefaultButton.bmp ..
cp resources/player1HoveredButton.bmp ..
cp resources/player2DefaultButton.bmp ..
cp resources/player2HoveredButton.bmp ..
cp resources/rightDefaultButton.bmp ..
cp resources/rightHoveredButton.bmp ..
cp resources/seven.bmp ..
cp resources/six.bmp ..
cp resources/smallExitDefaultButton.bmp ..
cp resources/smallExitHoveredButton.bmp ..
cp resources/snake.bmp ..
cp resources/space.bmp ..
cp resources/three.bmp ..
cp resources/two.bmp ..
cp resources/twoDots.bmp ..
cp resources/upDefaultButton.bmp ..
cp resources/upHoveredButton.bmp ..
cp resources/zero.bmp ..

 Installation finished.
 Write 'sh compile.sh' to compile the code and 'sh play.sh' to run the game.
```

```
# sh compile.sh
     create  src/Apple.d
     create  src/Bitmap.d
     create  src/Button.d
     create  src/Game.d
     create  src/Mouse.d
     create  src/Snake.d
     create  src/SnakeyNix.d
     create  src/StateMachine.d
     create  src/assembly_kbd.d
     create  src/assembly_ps2.d
     create  src/assembly_timer.d
     create  src/font.d
     create  src/kbd.d
     create  src/portaserie.d
     create  src/ps2.d
     create  src/rtc.d
     create  src/timer.d
     create  src/utilities.d
     create  src/vbe.d
     create  src/video_gr.d
     create  src/.depend
    compile  src/SnakeyNix.o
    compile  src/utilities.o
    compile  src/Bitmap.o
    compile  src/video_gr.o
    compile  src/vbe.o
    compile  src/Game.o
    compile  src/Snake.o
    compile  src/Apple.o
    compile  src/Mouse.o
    compile  src/timer.o
    compile  src/ps2.o
    compile  src/rtc.o
    compile  src/font.o
    compile  src/StateMachine.o
    compile  src/kbd.o
    compile  src/Button.o
    compile  src/portaserie.o
    compile  src/assembly_kbd.o
    compile  src/assembly_timer.o
    compile  src/assembly_ps2.o
       link  src/SnakeyNix

 Compilation finished.

# sh play.sh
```