Choose Your Destiny

Projeto de Laboratório de Computadores

Da autoria de:

- *João Gil Marinho Mesquita, up201906682*
- *Rui Filipe Teixeira Alves, up201905853*
- *Turma 7, Grupo 3*

03/01/2021

ÍNDICE

INTRODUÇÃO

Este projeto foi realizado no âmbito da Unidade Curricular Laboratório de Computador do 2º Ano do MIEIC, com os seguintes objetivos:

- O uso da interface de hardware dos periféricos mais comuns do PC
  - Timer, Video Card, Keyboard, Mouse e Real Time Clock, UART
- Desenvolvimento software de baixo nível e software integrado
- Programar estruturalmente na linguagem C
  - Utilização de várias estruturas de dados, Documentação,
- Utilização de várias ferramentas de desenvolvimento de software
  - Minix 3
  - Doxygen
  - Visual Studio Code
  - GIMP
  - Oracle VM VirtualBox
  - Photoshop

O projeto consiste num story-telling onde o jogador decide o seu destino (Branching story). Para tal, o jogador tem de tomar certas decisões, utilizando o teclado/rato e consoante essas decisões a história irá tomar rumos diferentes.

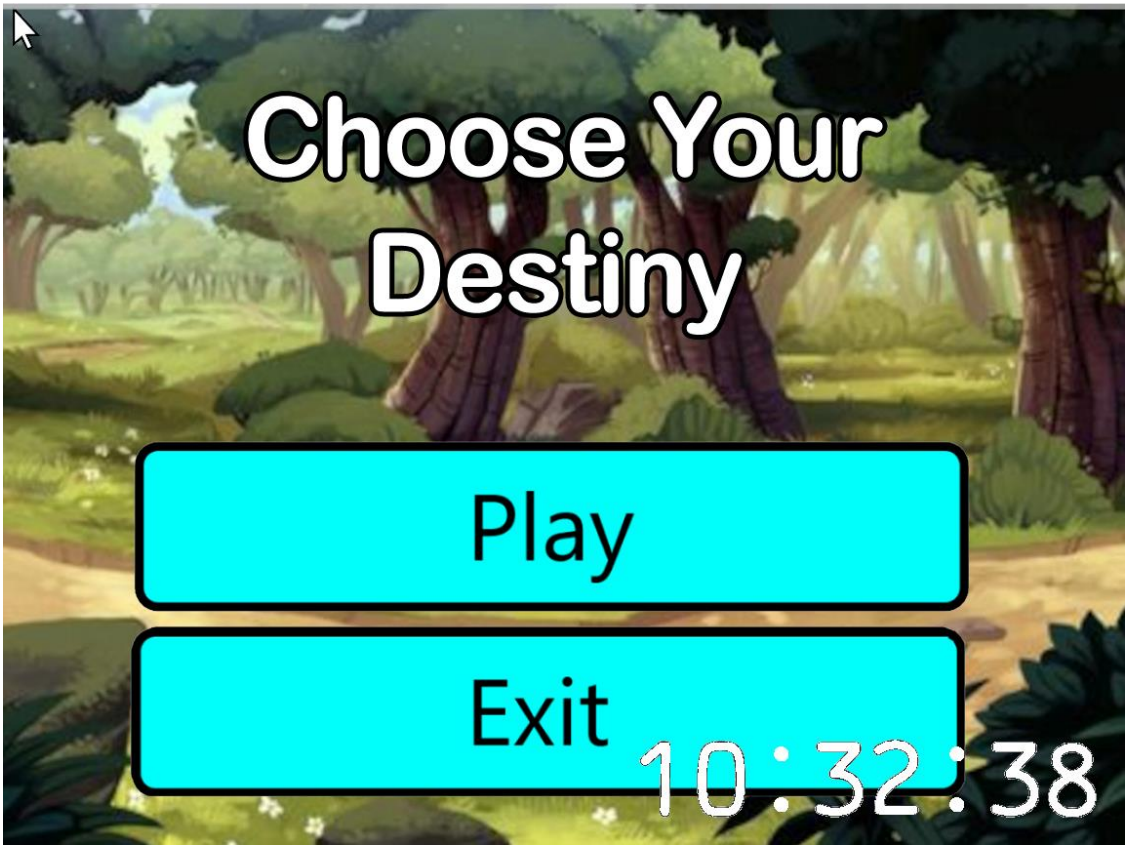
O principal do objetivo do jogo é que o jogador introduza os dados que lhe são pedidos através dos periféricos do computador que tem disponíveis (neste caso o rato e o teclado), de forma a ajudar a personagem a ultrapassar os obstáculos que vão aparecendo.

Instruções

Menu principal

No Menu Inicial temos duas opções: Play e Exit. Caso queira jogar, deve deslocar o rato para a zona onde diz "Play" e de seguida carregar com o rato esquerdo. Caso queira sair do jogo, deve deslocar o rato para a zona onde diz "Exit" e depois carregar com o rato esquerdo.

Caso clique na opção Play, o jogo irá para o estado Game Mode. Caso carregue na opção Exit, o jogo/programa termina.



Game Mode

Este estado destina-se a perguntar ao utilizador que tipo de modo quer jogar: *Single Player* ou *Race Mode*. Dividindo o ecrã em 2, carregando na parte da esquerda do mesmo irá começar o jogo no modo Single Player. Se carregar na parte da direita do ecrã irá para um estado onde estará à espera do outro jogador com o qual vai jogar.

Após esse estado de espera, ambos os modos irão para o Nível 1 onde, consoante o modo que escolheu, irá ter algumas alterações.

RACE MODE

O Race Mode consiste num modo *multiplayer* onde o objetivo é terminar o jogo o mais rapidamente possível.

Este modo difere do modo single Player uma vez que sempre que o oponente passa para o nível seguinte, é mostrada uma mensagem a indicar o nível que o oponente passou ou se este já perdeu. Outra diferença é que no final do jogo ambos os jogadores vão para um modo de *Highscores* onde podem verificar quem foi o jogador que conseguiu mais pontos.

Os jogadores começam o jogo ao mesmo tempo. Caso um jogador selecione a opção de "Race Mode" mais cedo, irá para um estado "*Waitting for Player*".

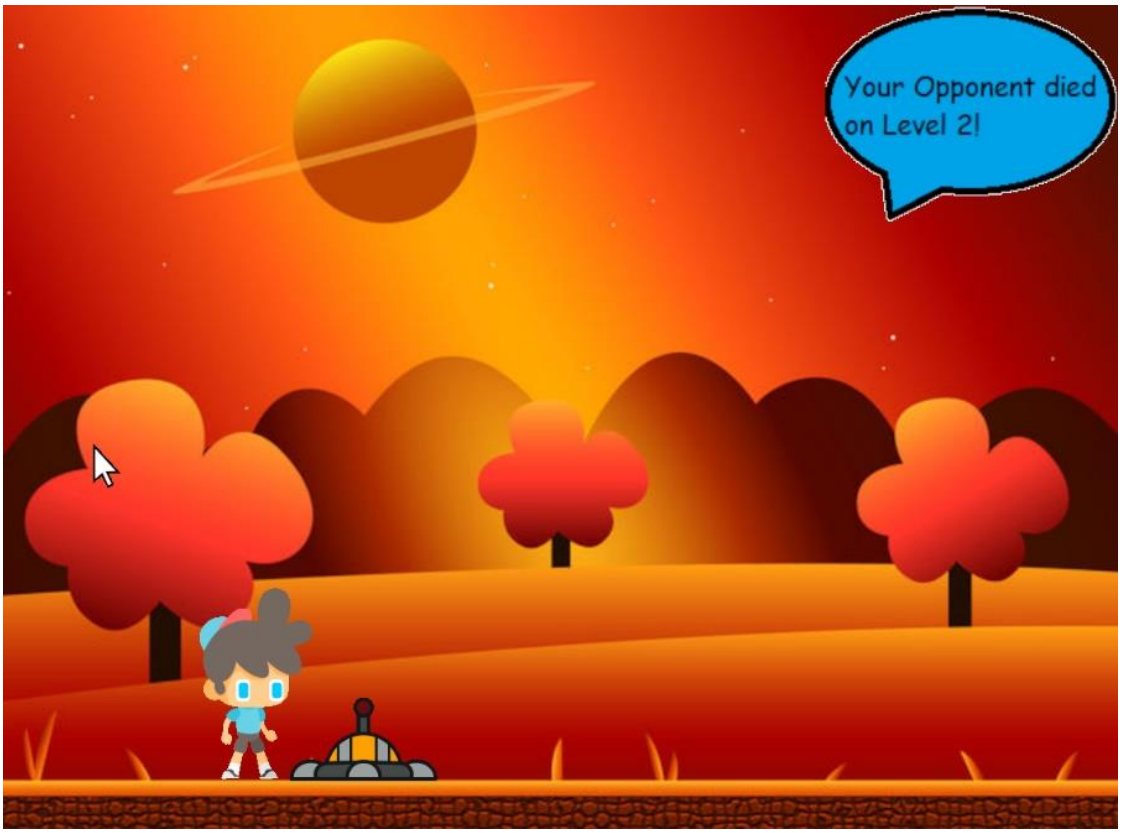


Figura 1- Mensagem do outro utilizador

WAITTING FOR PLAYER

Este estado destina-se à obtenção da confirmação, por parte de cada jogador, de que o outro jogador também já está pronto para jogar.

Enquanto o oponente não estiver disponível, é mostrada no ecrã a seguinte imagem:

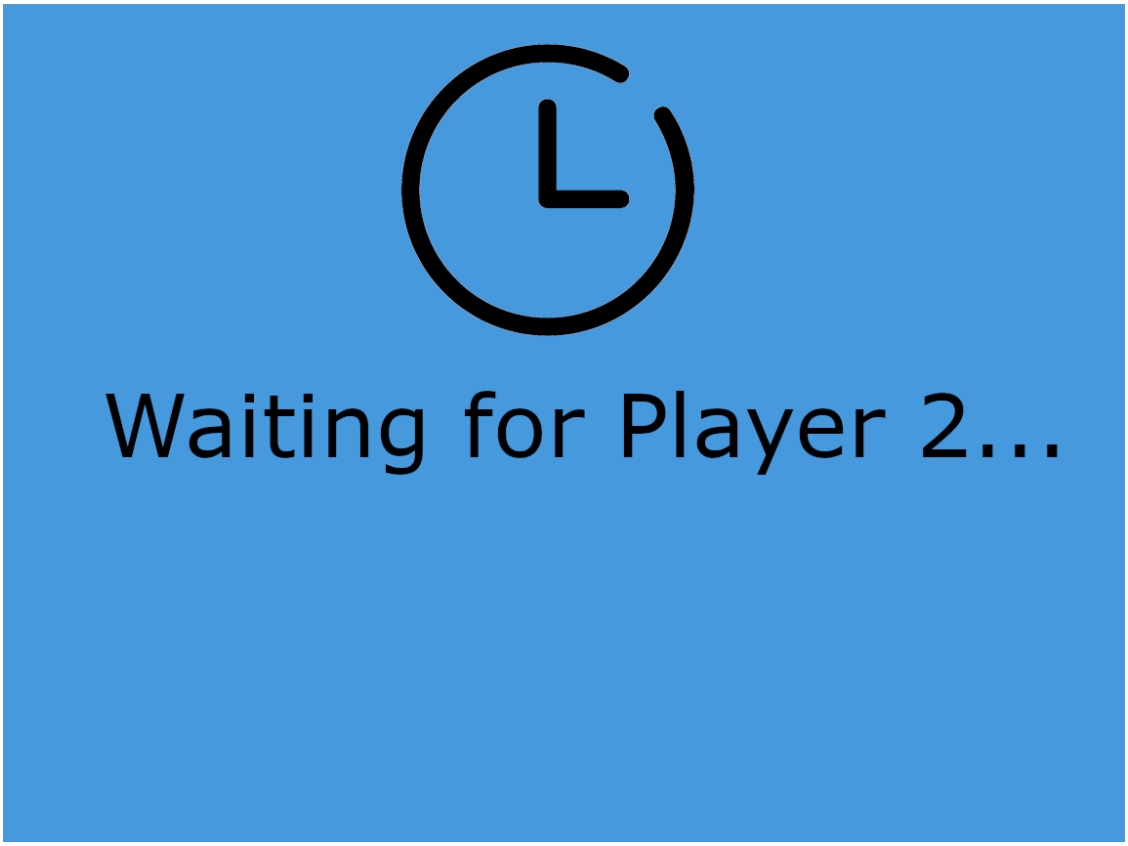


Figura 2- Ecrã de Espera pelo outro utilizador

Nível 1

No nível 1 deparamo-nos com a personagem do jogo. Este vai mover-se até ao centro do mapa e de seguida irá aparecer uma sucessão de letras no ecrã. O objetivo deste nível é selecionar essas letras, desta vez não com o rato mas sim com o teclado, carregando na tecla respetiva. Após selecionar todas as letras, a personagem vai deslocar-se até ao final do mapa, passando para o Nível 2.

Caso a tecla pressionada não seja a correta, o jogador perde o jogo. Quando isto acontece, o jogador morre e o jogo vai para o estado *Game Over*.

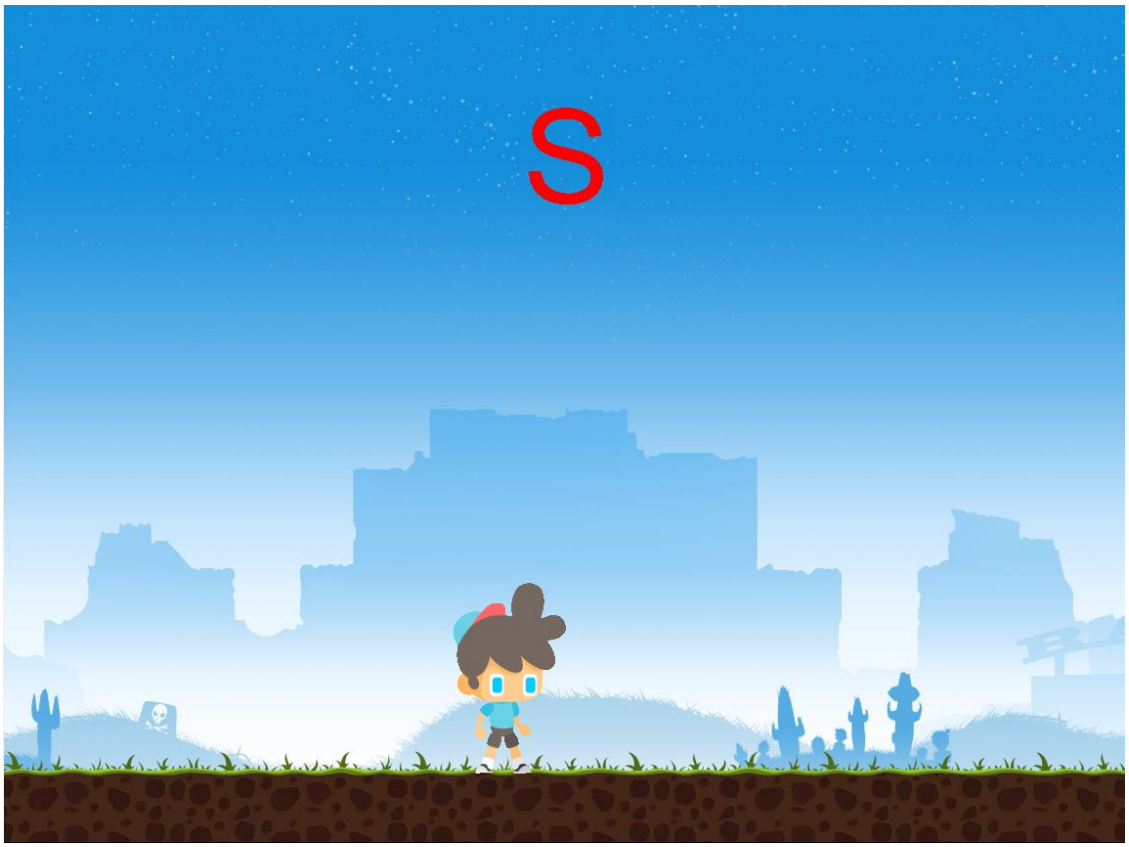


Figura 3- Nível 1

Nível 2

Neste nível irá aparecer um obstáculo que a nossa personagem tem de ultrapassar: uma mina. Para ultrapassar este obstáculo, a personagem precisa da ajuda do utilizador. Essa ajuda será uma indicação do que ele tem de fazer, com a ajuda do rato. O boneco tem de saltar a minha e, para isso, o utilizador terá de desenhar uma risca para cima. Para isso tem de, carregando ao mesmo tempo com o botão esquerdo, arrastar o rato para cima. A risca será desenhada no ecrã à medida que arrasta o rato.

A risca branca visualizada na imagem que se segue é a risca desenhada pelo utilizador. Como a risca foi de baixo para cima e na vertical, então a personagem irá saltar a mina.

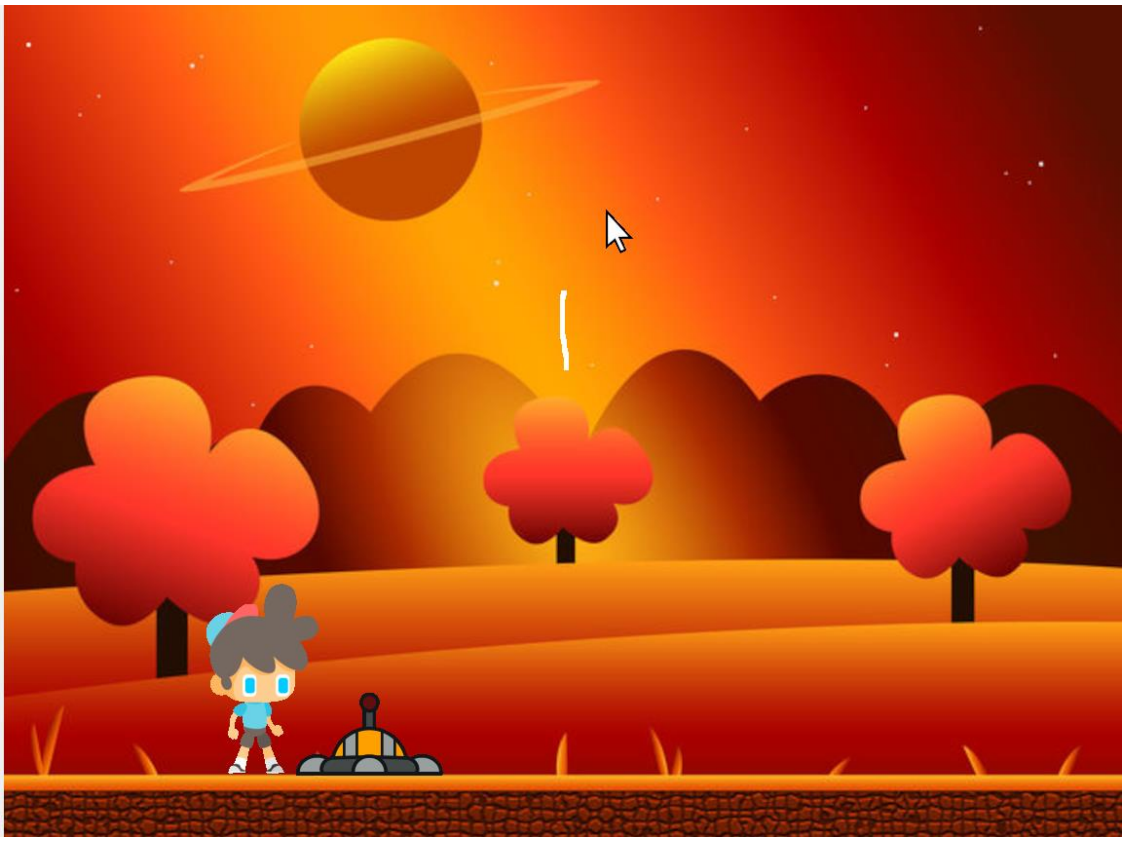


Figura 4- Nível 2 com o desenho introduzido pelo utilizador

Caso o que desenhe seja uma risca vertical, de baixo para cima, o boneco irá saltar por cima da mina e de seguida deslocar-se até ao final do mapa, entrando depois no nível 3. Caso contrário o boneco irá correr para a frente e pisar na mina e, consequentemente, a mina irá explodir e o boneco irá morrer e o utilizador perder o jogo.

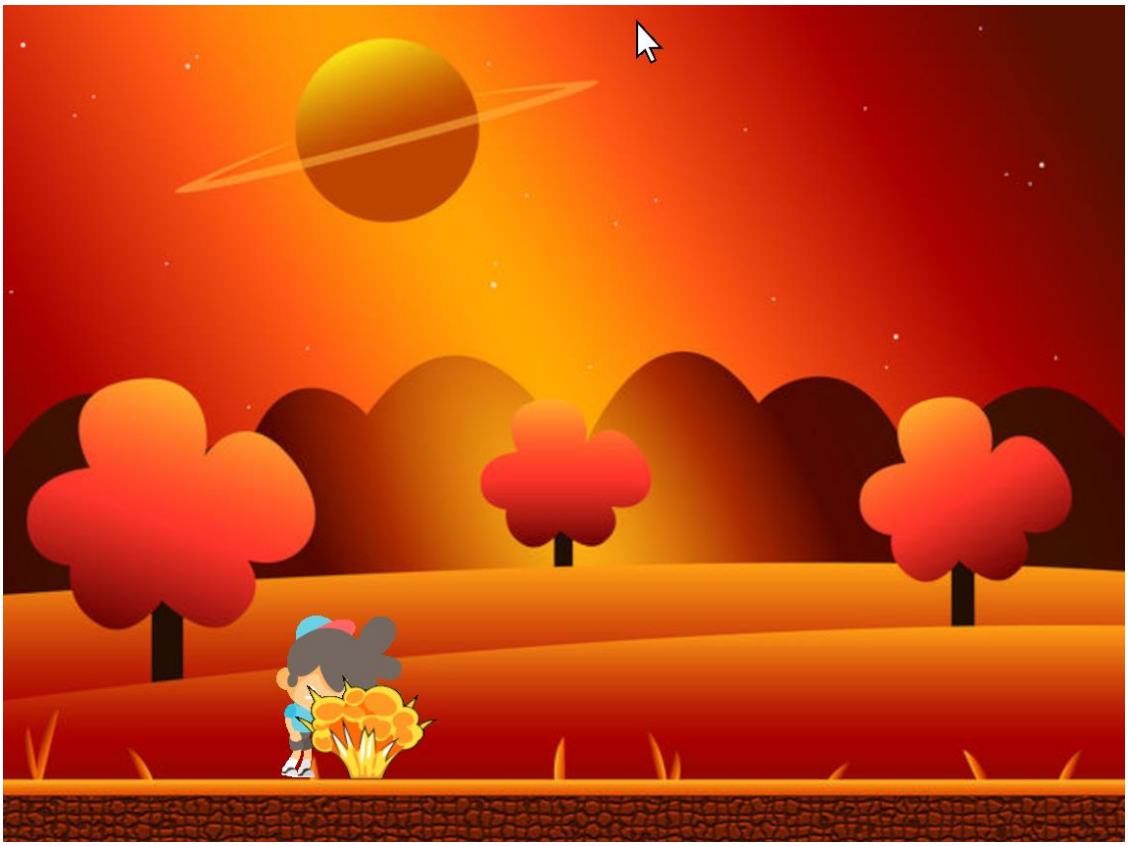


Figura 5- Bomba a explodir pois o utilizador perdeu o Nível 2

Nível 3

No nível 3 a personagem tem um novo desafio: derrotar o Boss Naruto. Este boss desloca-se em direção à personagem e, consequentemente, irá utilizar uma técnica onde cria clones seus, de forma a confundir o boneco e a defender-se.

Para ultrapassar este obstáculo, o boneco precisa que o utilizador o ajude a descobrir qual é o verdadeiro. Para isso, deve selecionar com o rato esquerdo do rato o clone que deseja escolher como verdadeiro.

Caso o clone selecionado seja o correto, o boneco irá atirar uma shuriken, matando assim o inimigo. Após isto, o jogo irá para o estado *Win*.

Em caso contrário, o inimigo irá atirar uma shuriken e matar o boneco, perdendo assim o jogo.

Após uma série de animações, chegamos à parte de escolher qual é o verdadeiro *Naruto*, como se mostra na imagem a seguir representada.

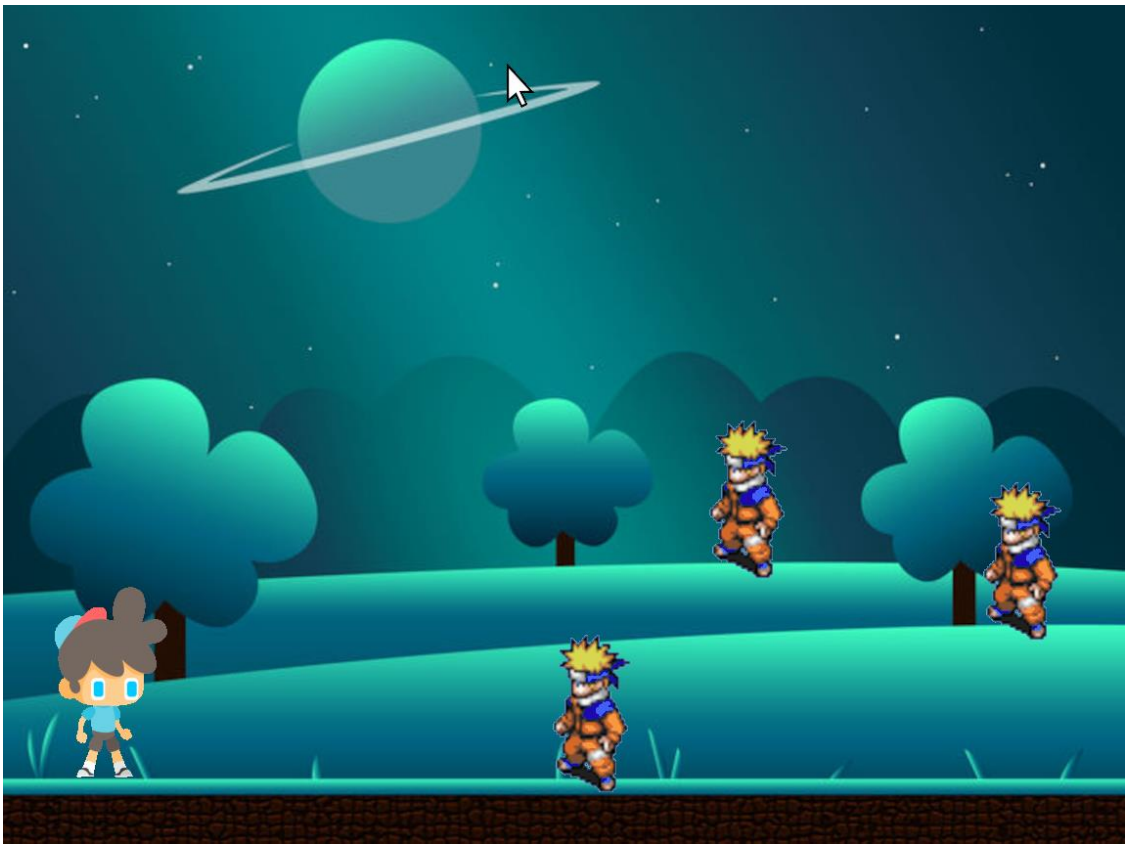


Figura 6- Escolha do Clone Verdadeiro no Nível 3

Game Over

Quando o utilizador perde o jogo, irá aparecer "Game Over" no ecrã e após alguns segundos o jogo irá para o Menu Inicial, para o utilizador voltar a jogar ou sair do jogo.

Figura 7- Ecrã exibido quando o utilizador perde o jogo.

Win

Caso o modo seja *Race Mode* o jogo irá para o estado *Show Scores*.
Caso contrário, estes estado apenas serve para indicar ao utilizador que o jogo acabou e que ganhou.

Figura 8- Ecrã Exibido quando o utilizador vence o jogo

SCORE

Caso o modo de jogo seja *Race Mode*, quando ambos os jogadores terminam o jogo, são exibidos os scores de cada um (scores esses que dependem do nível em que cada jogador ficou e do tempo que demorou), como se verifica na próxima imagem:

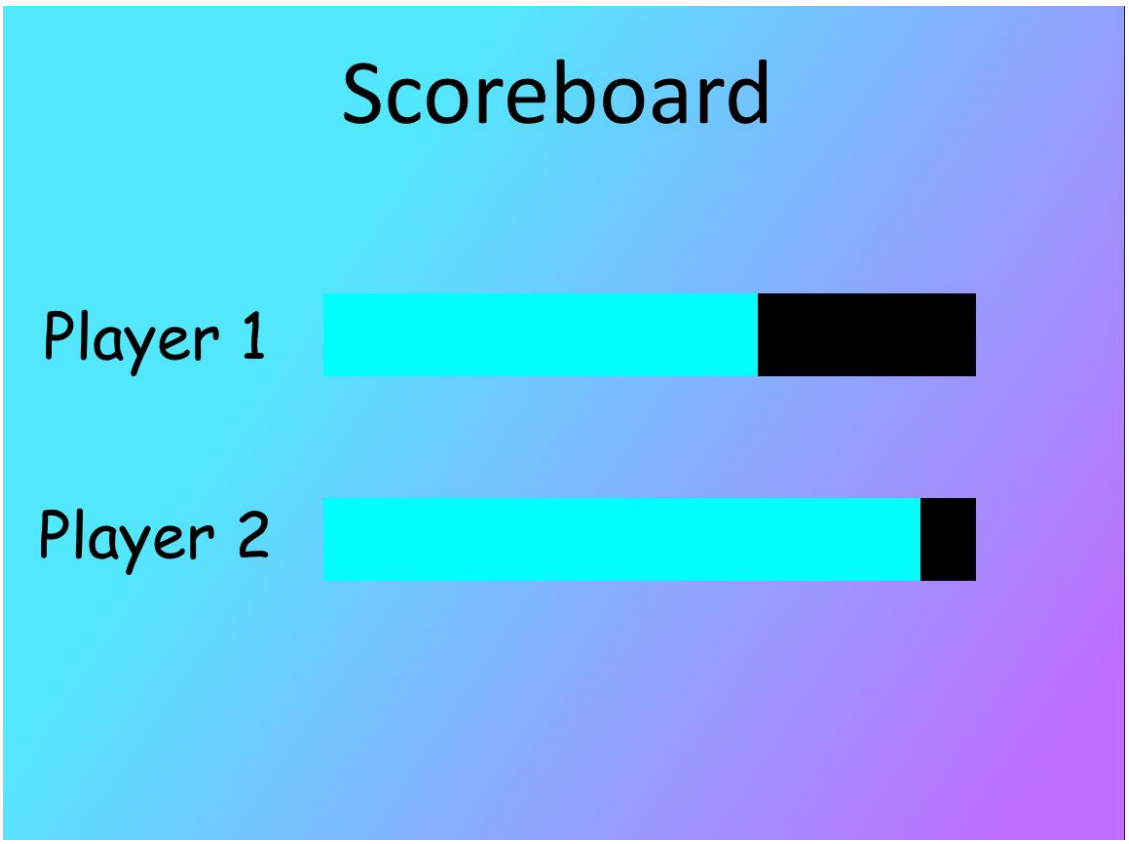


Figura 9- Scores de ambos os jogadores após o jogo.

STATUS DO PROJETO

Funcionalidades Implementadas:

- Timer, Keyboard, Mouse, Video Card, Real Time Clock, UART.

Dispositivos	Utilidades	Interrupções
Timer	Movimento e animação das Sprites e atualização do jogo	Sim
Keyboard	Input do Utilizador	Sim
Graphic Card	Apesentar a interface do jogo	N/A
Mouse	Selecionar opções e desenhar.	Sim
RTC	Obter a data real.	Sim
UART	Comunicação entre dois computadores, enviando informação entre eles.	Sim

TIMER

Funcionalidade

O *timer* é um dispositivo implementado no nosso computador e que gera interrupções a uma dada frequência. Desta forma, aproveitamos o timer para controlar todas as estruturas do jogo, quer seja para atualizar o estado do jogo, atualizar a *vídeo RAM*, animar as Sprites do jogo, fazer a troca entre os buffer que estamos a utilizar (utilização de *double buffering*).

Por predefinição, a frequência do timer é de 60Hz, ou seja, gera 60 interrupções por segundo. Para fazer as animações das Sprites e atualizar a *vídeo RAM*, decidimos utilizar apenas uma taxa de 30 frames por segundo. Esta taxa foi escolhida para que o desenvolvimento do jogo fosse feito sem perturbações, ou seja,

de forma a que não seja notória a atualização da *vídeo RAM* e uma vez que não prejudica em nada a jogabilidade e o utilizador.

Código

Tudo isto não seria possível sem a utilização de uma variável *timer_counter* e destas interrupções do *timer*, pois é através desta que conseguimos fazer esta gestão de cada etapa do jogo e das restantes variáveis. A utilização desta variável é possível pois sempre que existe uma interrupção gerada pelo *timer* esta é incrementada e, com o seu valor, conseguimos controlar a taxa de frames por segundo. Para a utilização de 30fps, colocámos uma condição (timer_counter % 2 == 0), de forma a só verificar atualizações a cada 2 frames.

Esta variável é não só essencial para os estados do jogo como também para fazermos a troca entre cada um dos buffers.

Para utilização do timer, foram implementadas as funções *timer_subscribe_int(), timer_unsubscribe_int()* no ficheiro timer.c .

KEYBOARD

Funcionalidade

O *keyboard* é utilizado em todos os modos como forma de terminar o programa, quando se carrega na tecla *Esc*, e no nível 1, uma vez que neste nível o objetivo do utilizador é carregar nas teclas que são mostradas no ecrã.

Código

Para fazer esta verificação das teclas pressionadas, utilizámos a função *kbc_ih(),* que é chamada sempre que existe uma interrupção do keyboard. Esta função é usada para processar a tecla que foi carregada.

As funções implementadas para o keyboard encontram-se declaradas no ficheiro kbc.h .

GRAPHIC CARD

Funcionalidade

A *Graphic Card* é a interface de vídeo que vai mostrar todo o nosso jogo. Sem a utilização da *Graphic Card* não seria possível criar uma imagem e mostrá-la ao utilizador e, portanto, é essencial no contexto deste projeto.

Para tal, a *Graphic Card* manipula cada pixel do nosso ecrã de forma a que a junção de todos os pixéis formem uma imagem. Num contexto "humano" esta ideia parece simples, contudo o nosso ecrã tem milhares de pixéis e é necessário haver uma concordância entre todos para que a imagem exibida seja a desejada.

De forma a que esta "pintura" de pixéis não fosse notória ao utilizador, tirámos partido da utilização de double buffering e page flipping.

As imagens exibidas na plataforma foram obtidas de websites sem direitos de autor e criadas por nós no *Photoshop*. Todas as imagens foram editadas e convertidas para .xpm (formato que especifica a cor de cada pixel) no *GIMP* de forma a que pudéssemos usá-las corretamente.

Websites:

- http://mirrors.pdp-11.ru/_sprites/spritedatabase.net/file/1278.html
- http://www.picturetopeople.org/text_generator/others/transparent/transparent-text-generator.html
- https://www.freepik.com/free-photos-vectors/game-background

Código

Neste Projeto foi utilizado o modo gráfico 0x14C, com resolução de 1152x864 e Direct Color Mode.
Para utilização apropriada da *Graphic Card* criámos as seguintes funções no ficheiro vbe.h :
set_video_mode(), init_graphic_mode(), vbe_get_mode_info2(), swap_buffers(), set_display_start(), drawCursorPositions(), drawRtcTime().

Estas funções estão explicadas no tópico "Detalhes de Implementação"

Mouse

Funcionalidade

O Mouse é utilizado no Menu Inicial para selecionar a opção que deseja (Play/Exit), no Nível 2 para desenhar (indicar o movimento que desejamos do boneco) e no Nível 3 para selecionar o clone.

Code

A Informação do mouse é enviada através de 3 bytes, sendo enviado um de cada vez e gerando uma interrupção. Como tal, utilizamos a função *mouse_ih()* e a função *mouse_packets_parsing()* para manipularmos estes bytes.

Estes bytes contêm a informação de qual botão do mouse está a ser premido tal como do seu deslocamento. Utilizando isso e atualizando a Sprite do Cursor, é-nos possível recriar o movimento que o utilizador introduz. Esta Sprite apenas é visualizada nos níveis/estados no qual este é necessário, ou seja no Menu Inicial, no Nível 2 e no Nível 3, através da função *drawCursorPositions()*.

RTC

Funcionalidade

O *Real Time Clock* é um circuito integrado que contém a data e o dia atual. Como tal, utilizamo-lo de forma a podermos mostrar ao utilizador a hora atual e de forma a diferenciar o score de cada utilizador (quanto mais tempo demorar, menos score adquire).

Code

Para utilização do *RTC* implementamos funções para lerem e escreverem nos registos do mesmo, de forma a obtermos a data atual, declaradas no ficheiro rtc.h : *writeRtcData(), readRtcData(), readRTCdate(), writeRTC(), rtc_subscribe_int(), rtc_unsubscribe_int(), setupUpdateInterrupts(), rtc_ih(), convert_bcd_to_int()*.

Serial Port

Funcionalidade

A Serial Port é utilizada para enviar mensagens entre 2 utilizadores. Neste projeto, utilizamos a UART para comunicar em modo (FULL-)Duplex, ou seja, que permite a comunicação em ambas as direções. Para além disso, implementámos FIFOs (First In First Out), que reduzem o número de interrupções e permitem uma comunicação mais rápida.

Código

Para a utilização da UART, implementámos várias funções que simplificam o processo e tornam-no mais modular, tal como *uart_subscribe_int(), uart_unsubscribe_int(), uart_write_reg(), uart_read_reg(), uart_setup_lcr(), uart_en_interrupts(), uart_set_bitrate(), uart_en_fifo(), uart_send_message(), entre outras.*

Organização do Código/Estruturação

Os módulos cujas funções eram necessárias para a realização dos labs não estão aqui especificados, uma vez que a sua explicação está nos handouts da unidade curricular. Os módulos do Timer e do Keyboard têm um peso de 5%, enquanto que o módulo do Mouse tem um peso de 7.5%.

De forma a simplificar e separar o projeto nas suas partes, decidimos implementar os seguintes módulos:

sprite.c / sprite.h

Este módulo destina-se à criação de Sprites (imagens) que não têm a sensação de animação, isto é, que são apenas uma imagem. É aqui que estão implementadas todas as funções que irão manipular a *struct* Sprite, criada por nós, de forma a que possamos alterar a sua posição atual no ecrã. Para isso, criamos as seguintes funções: create_sprite(), destroy_sprite(), animate_sprite(), draw_sprite_on(), draw_background(), clears_last_movement_on(), copyPosition().

Este modulo tem um peso de aproximadamente 10%.

```
typedef struct {
  int x, y;            // current position
  int width, height;   // dimensions
  int xspeed, yspeed; // current speed
  uint8_t *map;           // the pixmap
  enum xpm_image_type type;
} Sprite;
```

animSprite.c / animSprite.h

Este módulo destina-se a criar Sprites animadas no nosso jogo. Para isso, decidimos criar uma struct que recebe tudo o que a Sprite necessita para que seja

possível criar a sensação de animação durante o jogo. Esta struct baseia-se num conjunto de Sprites que, em conjunto, vão criar esta sensação.

De forma a criarmos este tipo de estruturas e as manipularmos, implementamos as seguintes funções: create_animSprite(), animate_animSprite(), draw_animated_sprite_on(), destroy_animSprite().

Tem aproximadamente um peso de 10%.

```
typedef struct {
    Sprite* sp;     // standard sprite
    int aspeed;     // no. frames per pixmap (animation speed)
    int cur_aspeed; // no. frames left to next change   (current animation speed)
    int num_fig;    // number of pixmaps
    int cur_fig;    // current pixmap
    uint8_t** map;  // array of pointers to pixmaps
} AnimSprite;
```

Game.c / Game.h

Nestes ficheiros encontramos a estrutura principal do projeto, Game, onde é guardada a informação de todos os dados necessários para o desenvolvimento do jogo. É também aqui que está implementada a função principal do projeto: *choose_your_destiny()*. Esta função dá subscribe e unsubscribe a todos os periféricos que utilizamos, manipula todos os dados e estados do jogo, com a utilização de um único *drive_receive()* e *while loop*.

É neste módulo que está implementada a função que carrega todas as imagens (setupSprites()) e todas as funções necessárias para o desenvolvimento do jogo, quer seja para validar inputs ou atualizar dados.

Este módulo é, indiscutivelmente, a parte mais importante do projeto, com um peso de aproximadamente 37.5%.

rtc.c / rtc.h

No ficheiro rtc.h temos as macros do *real time clock* tal como uma struct que criamos com o intuito de guardar a informação da data.

Neste módulo implementámos as funções necessárias para comunicar com o periférico. Para tal, criámos as funções *writeRtcData(), readRtcData(), readRTCdate(), writeRTC(), rtc_subscribe_int(), rtc_unsubscribe_int(), setupUpdateInterrupts(), rtc_ih(), convert_bcd_to_int()* para que conseguíssemos ler/escrever dos/nos registos do rtc de forma a obtermos os dados que necessitamos, criar interrupções do mesmo e tratá-las da devida forma e converter

os dados dos seus registos para inteiros, uma vez que estes estão na forma bcd (binary-coded decimal).

Este módulo tem um peso de aproximadamente 5%.

uart.c / uart.h

No ficheiro uart.h temos definidas as macros e funções da uart tal como uma struct que criamos com o intuito de guardar a informação da data.

Neste módulo implementámos as funções necessárias para comunicar com o periférico. Para tal, criámos as funções *uart_subscribe_int(), uart_unsubscribe_int(), uart_write_reg(), uart_read_reg(), uart_setup_lcr(), uart_en_interrupts(), uart_set_bitrate(), uart_en_fifo(), uart_send_message()*, com o intuito de configurar a uart e enviar/ler as mensagens.

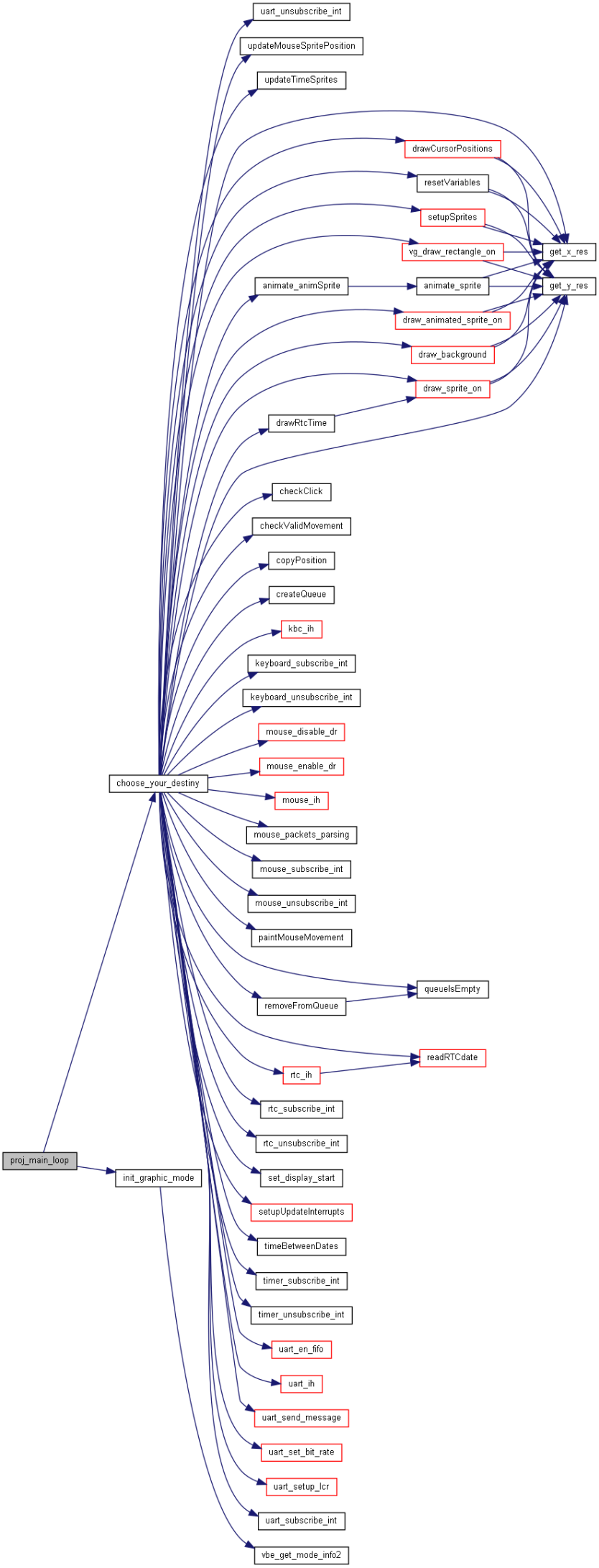
Este módulo tem um peso de aproximadamente 10%.

vbe.c / vbe.h

Este módulo contém não só as funções necessárias para o laboratório 5 (vbe_get_mode_info2(), vg_draw_rectangle(), init_graphic_mode(), set_video_mode()) mas também funções extras que necessitámos de implementar para que o processo de double buffering e page flipping fosse possível, tal como: swap_buffers(), clear_buffer() e set_display_start().

O peso deste módulo é de aproximadamente 10%.

Gráfico de Chamada de Funções


uart_unsubscribe_int
updateMouseSpritePosition
updateTimeSprites
drawCursorPositions
resetVariables
setupSprites
vg_draw_rectangle_on
get_x_res
animate_animSprite
animate_sprite
get_y_res
draw_animated_sprite_on
draw_background
draw_sprite_on
drawRtcTime
checkClick
checkValidMovement
copyPosition
createQueue
kbc_ih
keyboard_subscribe_int
keyboard_unsubscribe_int
mouse_disable_dr
mouse_enable_dr
choose_your_destiny
mouse_ih
mouse_packets_parsing
mouse_subscribe_int
mouse_unsubscribe_int
paintMouseMovement
queueIsEmpty
removeFromQueue
readRTCdate
rtc_ih
rtc_subscribe_int
rtc_unsubscribe_int
proj_main_loop
init_graphic_mode
set_display_start
setupUpdateInterrupts
timeBetweenDates
timer_subscribe_int
timer_unsubscribe_int
uart_en_fifo
uart_ih
uart_send_message
uart_set_bit_rate
uart_setup_lcr
uart_subscribe_int
vbe_get_mode_info2

Detalhes de Implementação

Função Principal

O código que implementamos tem como base principal a utilização de apenas um *while loop* e a chamada de apenas um *drive_receive,* onde são recebidas todas as interrupções causadas pelos periféricos utilizados. Para que tal fosse possível, tivemos de estruturar o nosso código e, como tal, criámos *state machines*, de forma, também, a controlar o fluxo do jogo e analisar de forma diferente cada uma das interrupções consoante a nossa necessidade, tornando, assim, o nosso programa o mais eficaz e eficiente possível. Esta abordagem obrigou-nos a várias simplificações e a que tivéssemos uma atenção redobrada na escrita do nosso código, o que acabou por torná-lo mais intuitivo e fácil de entender.

State Machines

As *state machines* criadas podem-se dividir em 2 tipos: principal (state machine do ficheiro game.h) e secundárias, para cada nível do jogo (level1_state, level2_state, level3_state). Assim, tornou-se mais simples a implementação de cada nível e a utilização de cada periférico consoante o estado do jogo em que íamos.

```
typedef enum{
    MENU,
    GAME_MODE,
    WAITING_FOR_PLAYER,
    LEVEL1,
    LEVEL2,
    LEVEL3,
    WIN,
    GAMEOVER,
    SHOW_SCORES,
} state_machine;

typedef enum{
    MOVING_TO_MIDDLE1,
    STOP_AT_MIDDLE1,
    MOVING_TO_END1,
    STOP_AT_END1,
    GAMEOVER_LV1,
} level1_state;
```

```
typedef enum{
    MOVING_TO_MINE2,
    STOP_TO_DRAW2,
    JUMPING_MINE2,
    MOVING_TO_END2,
    STOP_AT_END2,
    GAMEOVER_LV2,
} level2_state;

typedef enum{
    MOVING_TO_BUNSHIN,
    STANDING_BEF_JUTSU,
    KAGE_BUNSHIN_NO_JUTSU,
    SPAWN_BUNSHINS,
    CHOOSE_REAL,
    WIN_LVL3,
    GAMEOVER_LVL3,
} level3_state;
```

Para implementação destas *states machines* usámos *switch case*. Desta forma, foi-nos possível aproveitar cada uma das funcionalidades dos periféricos apenas nos

estados em que necessitávamos disso e relacionar cada um dos estados, aumentando assim a eficiência do nosso programa.

Sprites / Sprites animadas

De forma a que o utilizador pudesse escolher cada uma das opções disponíveis no Menu Inicial, o que decidimos fazer foi criar uma Sprite. Uma Sprite é uma *struct* implementada no ficheiro sprite.h, com o objetivo de simplificar a implementação de objetos no nosso jogo, usando a metodologia de programação orientada a objetos em C. Para dar a sensação de movimento de um objeto, decidimos implementar mais uma *struct* AnimSprite, que consiste na junção de várias Sprites e, assim, conseguimos alternar entre várias Sprites, movendo não só o objeto entre várias posições como também dando a sensação de animação, com a utilização de Sprites sequenciais (por exemplo, a AnimSprite boy (definida na função *setupSprites(Game *game),* que nos permite visualizar a corrida da personagem do jogo). Para isso implementámos a função *create_animSprite(int x, int y, int xspeed, int yspeed, int aspeed, uint8_t no_pic, enum xpm_image_type type, xpm_map_t pic1, ...),* que recebe um número variável de argumentos, definido no parâmetro *no_pic.*
Utilizando esta metodologia conseguimos recriar a sensação de animação de um dado objeto e, recebendo o deslocamento que o utilizador pede no rato, recriar o rato, que não temos disponível no modo de vídeo do minix, criando uma Sprite com a imagem de um cursor e movendo-a consoante esse deslocamento, que é recebemos nas interrupções do mouse, usando as funções *mouse_packets_parsing(struct packet *packet, uint8_t byte, int byte_no)*, declarada no ficheiro mouse.h, e *updateMouseSpritePosition(Game *game, struct packet *packet*) declarado no ficheiro game.h.
Analisando a posição desta Sprite do Mouse, verificamos se a mesma colide com as opções de "Play" ou de "Exit" e, caso isso aconteça e o botão do rato esquerdo seja selecionado (verificação feita na função *checkClick(Sprite *sp, struct packet *packet, int x0, int x1, int y0, int y1)),* o jogo muda para o estado correspondente.

Desenhar no nível 2

O objetivo no nível 2 é que quando a personagem pare em frente à mina o utilizador desenhe uma "seta" de baixo para cima, na vertical. Após o utilizador desenhar, temos de verificar se o desenho é uma linha e se corresponde ao que queremos (vertical, de baixo para cima). Para isso, usamos a função *checkValidMovement(Game *game)*, onde é imposta a condição de que o deslocamento no eixo dos x não pode ser superior a um valor, e que o seu tamanho tem de ser considerável.

Eficiência a desenhar / Double Buffering

Para desenhar o background e todos os objetos que quiséssemos no nosso projeto, utilizamos as funções *draw_background(Sprite *sp, uint8_t *buffer), draw_animated_sprite_on(AnimSprite *asp, uint8_t *buffer), draw_sprite_on(Sprite *sp, uint8_t *buffer), drawRtcTime(Game *game, uint8_t *buffer).* Para sobrepor os objetos ao background e o mouse a esses objetos, o que fizemos foi desenhar primeiro o background e, posteriormente, desenhar as restantes Sprites. Uma vez que pintar pixel a pixel todas as imagens seria muito dispendioso em termos de eficiência, decidimos criar funções para casos distintos de atualização da VRAM, como por exemplo para desenhar o background (utilizando a função *draw_background*) aproveitando a existência da função *memcpy*, que nos permite copiar o *pixmap* da figura toda para o buffer.

Caso esta metodologia fosse feita apenas com um buffer iríamos ter problemas de eficiência quando pretendemos mexer, por exemplo, o rato, pelo que optámos pela implementação de double buffering. O double buffering consiste na criação de dois buffers, sendo um o buffer atual e o outro o buffer para a próxima imagem, alternando entre eles, pelo que, assim, conseguimos ultrapassar esse problema. Para tal, passamos por parâmetro o buffer atual a todas as funções cujo fim seja desenhar.

PAGE FLIPPING

De facto, a utilização de double buffering não é suficiente para contornar os problemas de eficiência do programa, já que obriga a copiar o buffer atual para a variável que guarda o endereço virtual da VRAM. Logo, foi necessário implementar Page Flipping, que se resume a alterar o endereço para o qual aponta o início da memória gráfica. Para este propósito, criamos a função set_display_start. Assim, foi eliminada a necessidade de copiar o buffer inteiro para outra variável.

Serial Port

A uart é um periférico que permite enviar mensagens entre computadores a uma determinada frequência que pode ser alterada. Por sua vez, esta requer a criação de um protocolo de comunicação de forma a identificar as mensagens recebidas. Assim, criámos o seguinte protocolo:

$$\begin{cases} mensagem \,\%\, 3 == 0 \; e \; mensagem \,\%\, 10 \neq 0, & Atualização\ de\ nível\ do\ outro\ utilizador \\ mensagem \,\%\, 10 == 0, & Score\ no\ final\ do\ jogo \\ Número\ primo, significado\ específico \end{cases}$$

Após a criação do protocolo, procedemos à criação de várias funções para inicializar a uart. Obviamente, criámos funções para a subscrição de interrupções da uart (*uart_subscribe_int()* e *uart_unsubscribe_int()*).
Após isso, definimos funções para escrever e ler de um registo específico da uart (0-7) e de dar *setup* ao *line control register* que define o número de bits por *char*, o número de *stop bits* e a paridade (*uart_setup_lcr()*). Ainda no *line control register*, é possível mudar a *bit rate*, para a qual utilizámos a função *uart_set_dlb()*, que torna os registos da *DLB* disponíveis e a função *uart_set_bit_rate()*, que escreve nesses registos o *byte* menos significativo e o mais significativo.
Com o objetivo de aperfeiçoar o envio de mensagens e diminuir o número de interrupções, utilizámos *fifos*. Estas funcionam como um buffer que pode guardar um número de carateres especificado ao chamar a função *uart_en_fifo()*, que ativa as mesmas e atualiza o *trigger level*. Ao utilizar este método, um utilizador só recebe interrupção quando o número de mensagens recebidas ultrapassa o *trigger level*, ou quando existe algum carater no buffer e não foi gerada interrupção durante um certo intervalo de tempo definido.
Por fim, também utilizámos funções de leitura e escrita de mensagens. No caso da leitura, é lido o *line status register* para verificar se ainda possui algum carater no *buffer* e, enquanto tiver, lê-se o *receiver buffer register,* de forma a obter o próximo *byte*.
No caso da escrita, basta enviar o byte para o *transmitter holding register*, após verificar a possibilidade de escrita, que pode ser confirmada através da função *uart_check_read()* e *uart_check_send()*.

Conclusões

Após o desenvolvimento deste projeto, chegamos à conclusão de que esta unidade curricular representa bastante bem a realidade que é o nosso curso, preparando-nos para esta área da Engenharia Informática.

Com alguma dispersão na informação disponível, obriga-nos a organizar melhor o nosso estudo e a sermos autodidatas, desenvolvendo bastante a nossa capacidade de estudo. Apesar deste desenvolvimento, consideramos isto um ponto negativo da unidade curricular, uma vez que obriga a um aumento da carga horária, levando-nos a um grande desgaste e stress, pois inicialmente torna-se bastante complicado de entender o que devemos fazer, onde devemos procurar as coisas e como aplicar os conceitos dados nas aulas teóricas. Assim, consideramos que a unidade curricular e os alunos iriam beneficiar caso tivessem a informação mais bem estruturada. Por outro lado, de notar a falta de aulas práticas para a aplicação dos conceitos do RTC e da Serial Port, que, principalmente a Serial Port, ainda têm alguma complexidade e pormenores.

Acabamos este projeto com uma capacidade de programação superior àquela com que entramos, uma vez que nos leva à modularização e simplificação de código tal como a melhorarmos as nossas capacidades de resolver problemas e erros. A aplicação destes conceitos de programação com a utilização de periféricos é, sem dúvidas, um ponto muito importante e fulcral da unidade curricular, levando-nos a um aumento da nossa motivação perante a área que estamos, uma vez que, apesar de com grande esforço, conseguimos acabar este projeto e verificar que nos foi bastante útil e interessante.

Desta forma, acabamos este projeto com a ideia de que esta unidade curricular cumpriu bastante bem os objetivos pretendidos, quer seja a nível de aprendizagem de manipulação de vários periféricos, quer seja através do desenvolvimento das nossas capacidades de programação na linguagem C e do conceito de programação orientada a objetos ou no desenvolvimento de software de baixo nível e de várias ferramentas de desenvolvimento de software.

Apesar de um ano atípico, consideramos muito importante todo a disponibilidade apresentada pelos docentes para o esclarecimento de dúvidas, que foi muito positiva e essencial para a nossa aprendizagem.