U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MINIXIONARY

PICTIONARY A DOIS JOGADORES PARA MINIX

Mestrado Integrado em Engenharia Informática e Computação

Laboratório de Computadores 2020/2021

Turma 9 — Grupo 3

Pedro Gonçalo Correia up201905348@fe.up.pt
Diogo Costa up201906731@fe.up.pt

Índice

1. Instruções de Utilização do Programa

1.1. Menu Inicial

Ao iniciar o programa, é apresentado um menu inicial. No canto superior direito é apresentada a data e hora atual, que será sempre visível durante toda a execução do programa, bem como uma mensagem de saudação que dependerá da hora atual:

- "*GOOD MORNING*" — entre as 4 e as 12 horas;
- "*GOOD AFTERNOON*" — entre as 12 e as 19 horas;
- "*GOOD EVENING*" — entre as 19 e as 22 horas;
- "*GOOD NIGHT*" — entre as 22 e as 4 horas.

Ecrã do menu inicial com a mensagem "GOOD EVENING". Permite iniciar um novo jogo ou sair do programa.

Este menu possui ainda dois botões que o utilizador poderá selecionar com o rato.

NEW GAME — Para iniciar um novo jogo. Redireciona para o menu de aguardar outro jogador *(secção 1.2)*.

EXIT GAME — Para encerrar o programa.

1.2. Menu de Aguardar outro jogador

Este jogo é jogado apenas a dois jogadores. Assim, antes de iniciar um novo jogo o utilizador é redirecionado para este menu, onde aguarda que outro jogador de junte e assim iniciem o jogo. Mal os dois jogadores estejam prontos a começar, isto é, tenham entrado neste menu, o jogo inicia, arbitrando um papel (*Adivinho* ou *Pintor*) para cada jogador na primeira ronda.

Ecrã do menu de aguardar por outro jogador. O jogador pode abortar a espera regressando ao menu inicial.

As reticências do texto (“*WAITING FOR OTHER PLAYER…*”) são animadas, aparecendo e desaparecendo ao longo do tempo para dar ao utilizador a noção de espera.

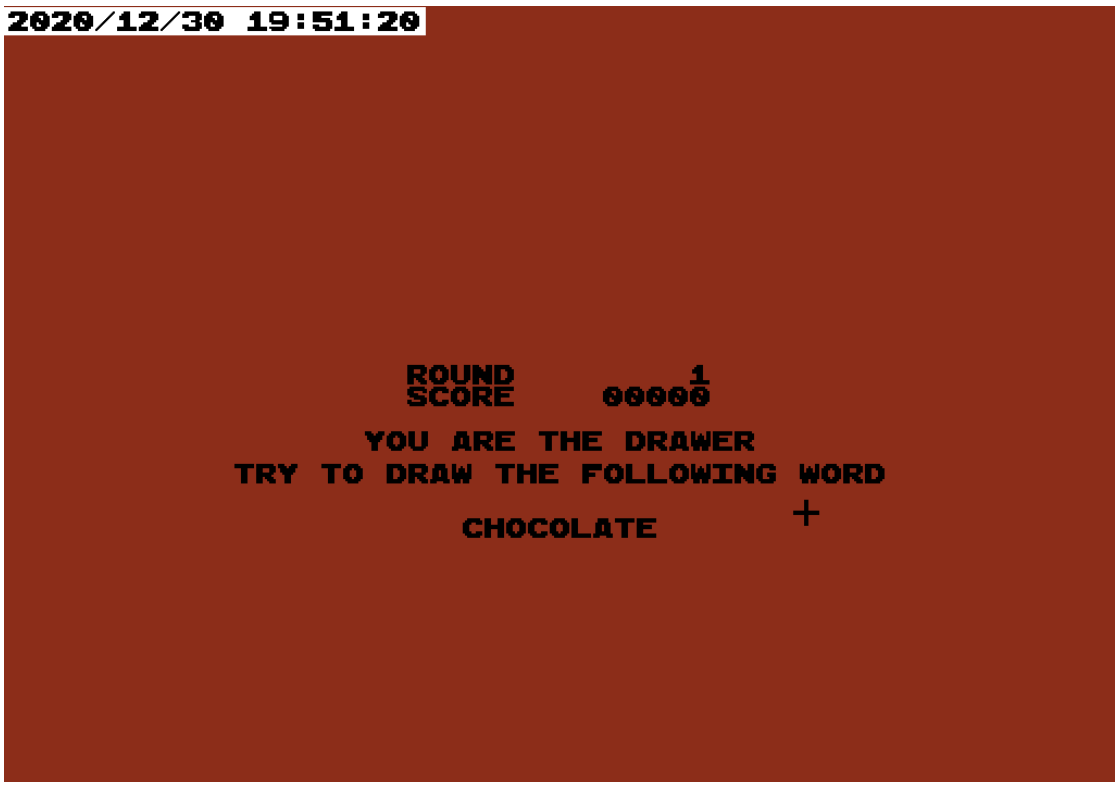
MAIN MENU — Para regressar ao menu inicial *(secção 1.1)*. Cancela a espera pelo outro jogador, deixando de estar disponível para um novo jogo.

1.3. Ecrã de Nova Ronda

Quando uma nova ronda está prestes a iniciar (após começar o jogo ou após ganhar uma ronda anterior), o utilizador é redirecionado para um ecrã de nova ronda, onde pode ver o score e número da ronda atual no jogo, bem como o seu papel (*Pintor* ou *Adivinho*) na ronda que se vai seguir. O ecrã que lhe é apresentado depende do seu papel.

1.3.1. Papel de *Pintor*

O objetivo do *Pintor* é desenhar uma palavra para o *Adivinho* tentar adivinhar. Além da informação descrita no parágrafo anterior, neste ecrã é ainda apresentada a palavra que ele deverá adivinhar. Esta palavra é escolhida aleatóriamente, a partir de uma lista de palavras interna ao jogo.



Ecrã de início de ronda para o Pintor. Inclui o número da ronda e score, bem como a palavra a ser desenhada.

Após três segundos, a ronda começa, dando lugar ao ecrã de jogo *(secção 1.4)*.

1.3.2. Papel de *Adivinho*

O objetivo do *Adivinho* é tentar adivinhar a palavra desenhada pelo *Pintor*. Como é natural, ele desconhece a palavra e terá de a ter descobrir pelo desenho do outro jogador.

Ecrã de início de ronda para o Adivinho. Inclui o número de ronda e o score.

Para uma melhor sincronização do início de jogo, o computador do *Adivinho* apenas inicia a ronda ao receber a informação de que a ronda iniciou para o *Pintor*. Nessa altura passa para o ecrã de jogo *(secção 1.4)*.

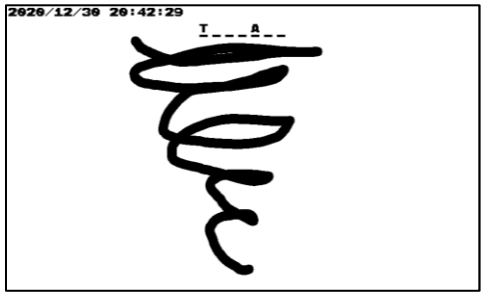
1.4. Ecrã de Jogo

Embora os papéis de ambos os jogadores sejam bastante diferentes, o ecrã de jogo tem vários aspetos em comum para ambos.

Exemplo do ecrã de jogo. Neste caso, o jogador é o Adivinho. Está a escrever na caixa de texto a palavra TORNADO, após ter feito quatro tentativas falhadas e o tempo para o fim da ronda estar a 2 segundos do fim. A palavra tem 7 letras, começando com T e sendo a antepenúltima um A. O jogo vai na oitava ronda, com um score de 2536 pontos.

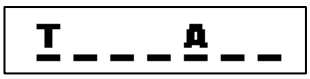
Tela de desenho

A tela ocupa a maior parte do ecrã do jogo. É onde é mostrado o desenho do *Pintor*. No topo está visível a ambos jogadores a *Pista*.



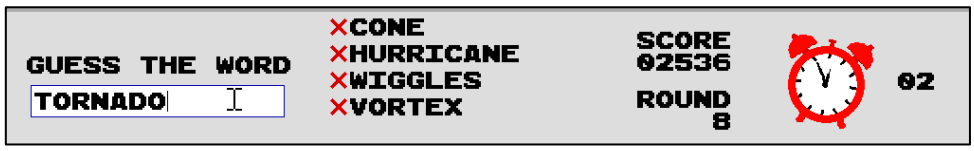
Pista

Uma ajuda que permite saber o número de letras na palavra correta, e que a cada 16 segundos revela uma nova letra na posição correta da palavra, de modo a auxiliar o *Adivinho* se este tiver muita dificuldade a adivinhar. A pista nunca revela todas as letras da palavra durante a ronda, deixando sempre pelo menos uma em branco.



Barra de Jogo

A barra de jogo, situada na parte inferior do ecrã, inclui detalhes sobre o jogo e a ronda atual.



Temporizador de fim de ronda

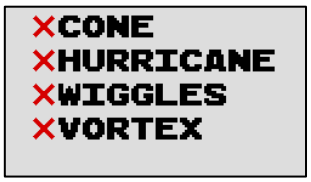
O temporizador indica quantos segundos faltam para o final da ronda. Cada ronda tem 60 segundos. Quando o temporizador chega a zero para ambos os jogadores, estes perdem o jogo. O relógio vermelho abana ligeiramente para a esquerda e para a direita, alertando aos jogadores que o tempo é limitado e está sempre a decrescer.

Score e Ronda

Indicam o score e o número da ronda atual. Quando o *Adivinho* descobre a palavra correta, o score aumenta, tanto mais quantos mais segundos ainda restarem no temporizador de fim de ronda. O jogador é, portanto, recompensado pela rapidez.

Palavras adivinhadas

A lista de palavras que o *Adivinho* já tentou esta ronda (mostra no máximo as cinco mais recentes). Uma cruz vermelha indica que a tentativa foi errada. Um 'visto' a verde significa que a palavra introduzida era de facto a palavra correta. Adivinhar a palavra correta leva a que a ronda seja completada com sucesso para ambos jogadores. Uma tentativa errada, por outro lado, desconta 5 segundos no temporizador de fim de ronda.



Se a palavra for demasiado grande, parte dela será omitida com reticências.

Cinco tentativas: as quatro primeiras erradas e a última correta. Uma das tentativas foi muito longa, pelo que parte do texto foi omitido por reticências.

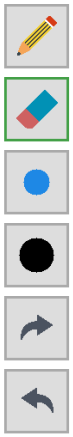
1.4.1. Papel de Pintor

O *Pintor*, além do já referido anteriormente, pode ainda ver no ecrã elementos específicos do seu papel.

Exemplo do ecrã de jogo para um Pintor.

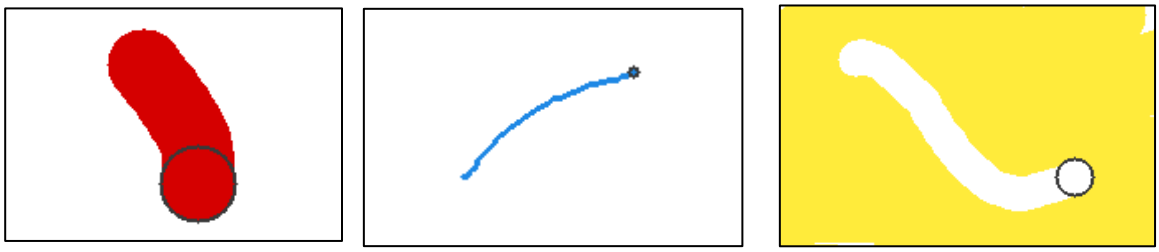
Tela de desenho

O *Pintor* pode desenhar no canvas clicando com o botão esquerdo do rato e arrastando ao longo da tela de desenho. Para o auxiliar, tem também ao seu dispor 6 botões relativos a funcionalidades da tela. Se preferir, em vez de premir botões poderá recorrer a atalhos do teclado:



1. **Lápis (P)** — Seleciona o lápis como ferramenta principal de desenho. O lápis faz traços da cor selecionada.
2. **Borracha (E)** — Seleciona a borracha como ferramenta principal de desenho. A borracha faz traços brancos.
3. **Cor do traço (C)** — Altera a cor do traço.
4. **Tamanho do traço (T)** — Altera o tamanho do traço.
5. **Desfazer (Ctrl+Z)** — Desfaz o último traço desenhado.
6. **Refazer (Ctrl+Y)** — Refaz o último traço desfeito (se nenhum outro traço foi feito entretanto).

O utilizador pode pintar também com o botão direito do rato, sendo o traço então executado com a ferramenta não selecionada (lápis ou borracha), permitindo rapidamente trocar entre elas. O cursor toma a forma da cor e tamanho do traço a desenhar.



Exemplos de cursores de desenho a pintar. À esquerda um traço vermelho de grossura máxima, no meio um traço azul de grossura mínima, e à direita um traço de borracha (sobre fundo amarelo) com grossura média.

Informação de palavra

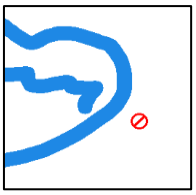
O *Pintor* pode a qualquer momento ver a palavra que deve desenhar no canto inferior esquerdo.

1.4.2. Papel de Adivinho

Aplica-se exclusivamente ao *Adivinho* o seguinte:

Tela de desenho

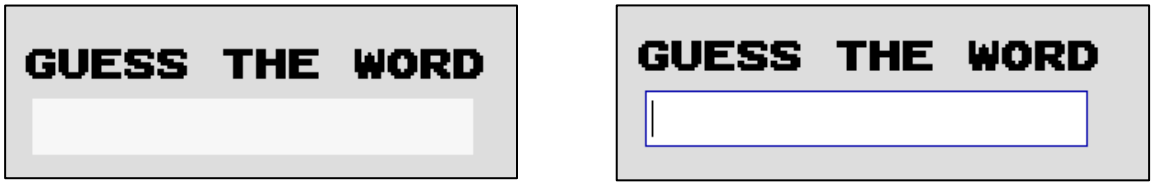
O *Adivinho* não pode desenhar nem afetar de qualquer outro modo o conteúdo na tela, apenas ver o que está a ser desenhado pelo *Pintor*. Como tal, se o cursor estiver a pairar sobre ela adquirirá a forma de "bloqueado".



Cursor do Adivinho aparece como bloqueado ao pairar sobre a tela.

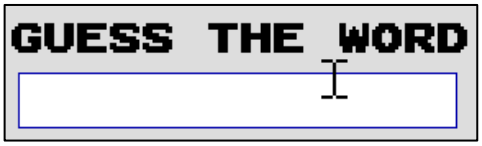
Caixa de Texto

Para o *Adivinho* tentar descobrir a palavra, deve escrevê-la na caixa de texto. Para escrever na caixa de texto deve primeiro selecioná-la com o rato, e de seguida escrever a palavra com o teclado. Quando a caixa de texto está selecionada, um brilho azul aparece em torno desta e um cursor de texto pisca para o utilizador saber que pode começar a escrever.



À esquerda, uma caixa de texto não selecionada. À direita, uma caixa de texto selecionada.

O cursor do rato também reage quando está a pairar sobre a caixa de texto.



Cursor do rato a pairar sobre a caixa de texto.

O utilizador poderá introduzir na caixa de texto letras, números e espaços, apagar texto com o backspace (apaga o texto à esquerda) ou delete (apaga o texto à direita), mudar a posição do cursor de texto com as setas para a esquerda e para a direita (se o Ctrl estiver premido, este texto ficará selecionado) ou o rato (se arrastar, o texto ficará selecionado). A caixa de texto suporta tantos caracteres quanto o utilizador quiser escrever, mesmo que sejam mais do que o espaço visível. Nesse caso a parte visível dependerá da posição atual do cursor de texto.

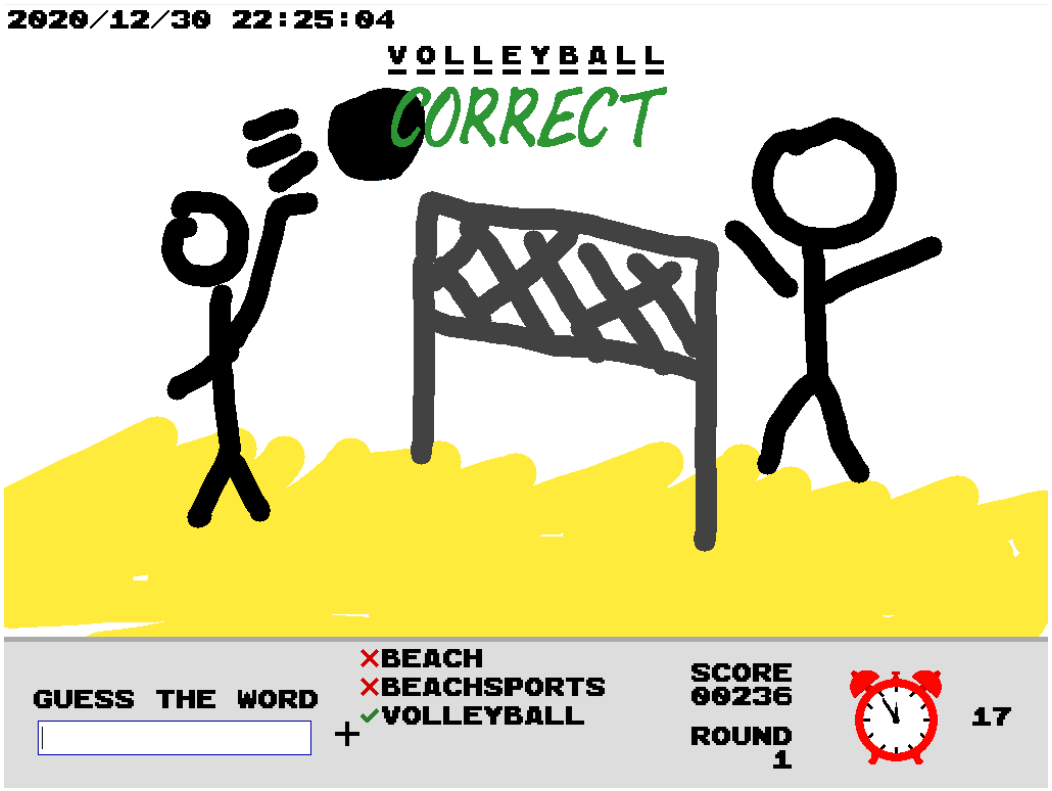
O texto selecionado poderá ser eliminado com backspace, copiado com Ctrl+C ou cortado com Ctrl+X (depois poderá ser colado com Ctrl+V), ou substituído ao digitar outros caracteres ou colar texto por cima.

Depois de escrita, a palavra pode ser submetida clicando enter.

Utilizador a selecionar parte do texto "VERY LONG TEXT" com o rato.

1.4.3. Palavra Correta (Ronda ganha)

Quando o *Adivinho* introduz a palavra correta, a ronda é vencida. O score é atualizado, o temporizador para, e a palavra "*CORRECT*" flutua (tremendo ligeiramente para a esquerda e direita de modo a dar ênfase) abaixo da *Pista*, agora com a palavra completa.

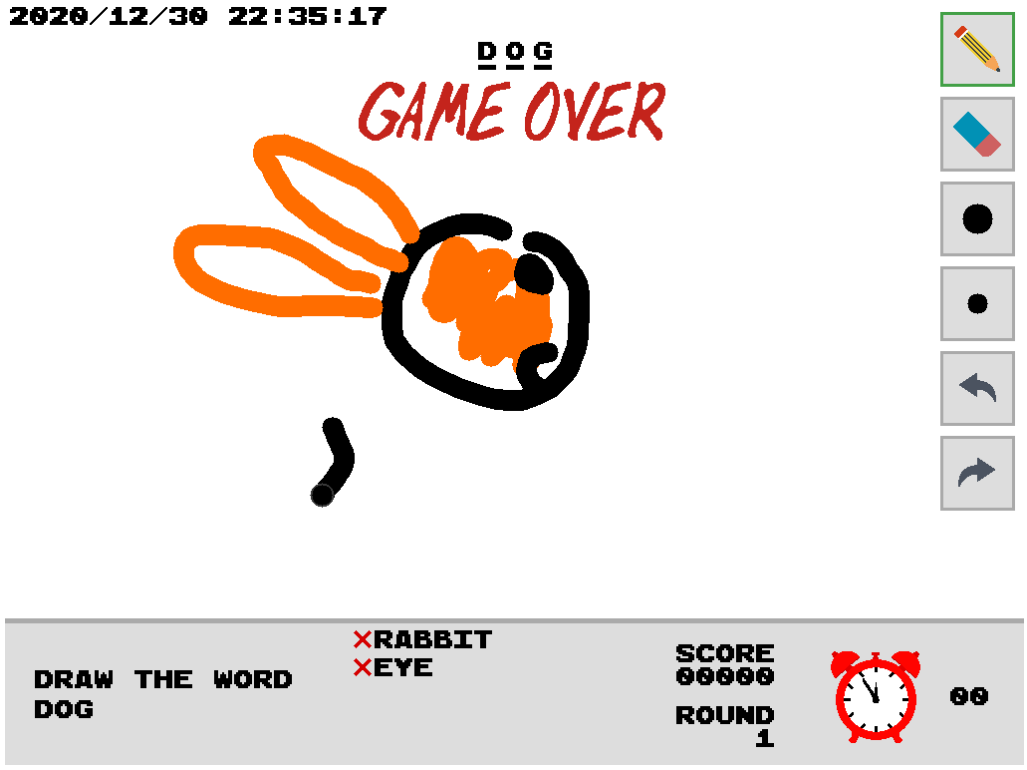


Notificação de ronda vencida.

Após três segundos, uma nova ronda prepara-se para começar, regressando ambos os jogadores ao ecrã de nova ronda *(secção 1.3)*, desta vez com os papéis invertidos.

1.4.4. Fim do tempo (Ronda perdida)

Caso o temporizador chega a zero para ambos os jogadores, a ronda termina e eles perderam o jogo. A palavra "*GAMEOVER*" flutua (tremendo ligeiramente para a esquerda e direita de modo a dar ênfase) abaixo da *Pista*, agora com a palavra completa.

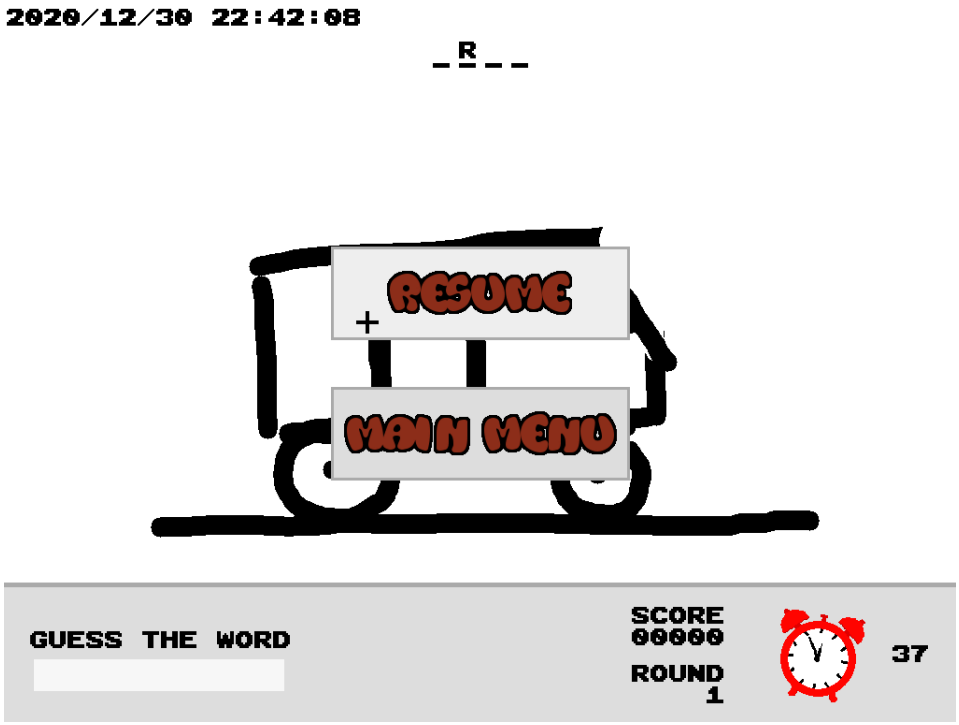


Notificação de fim do jogo.

Após três segundos, ambos os jogadores são redirecionados para o ecrã de fim de jogo *(secção 1.6.1)*.

1.5. Menu de Pausa

O utilizador pode, independentemente do seu papel, abrir o menu de pausa ao clicar na tecla Esc quando está no ecrã de jogo *(secção 1.4)*. Note-se que, apesar do nome, este ecrã não coloca verdadeiramente o jogo em pausa: o temporizador continua a contar o tempo e o outro jogador pode continuar a jogar normalmente.



Jogador com o menu de pausa aberto.

Ao abrir o Menu de Pausa, são apresentadas duas opções de menu por cima do jogo. Apesar de o jogo continuar visível e em andamento, enquanto se está no menu de pausa não se pode interagir com nenhum elemento do jogo (botões da tela de desenho, caixas de texto, tela de desenho).

RESUME — Sai do menu de pausa, voltando ao jogo *(secção 1.4)*. O mesmo efeito pode ser obtido voltando a premir a tecla Esc no menu de pausa.

MAIN MENU — Para sair do jogo. Termina o jogo para ambos os jogadores, regressando diretamente ao menu principal o jogador que premiu *(secção 1.1)*, e ao ecrã de fim de jogo o outro jogador *(secção 1.6.2)*.

1.6. Menu de Fim de jogo

1.6.1. Fim de tempo

Três segundos após a notificação de fim de tempo *(secção 1.4.4)*, o jogador é remetido para o menu de fim de jogo, com a mensagem "*TIME HAS RUN OUT*". Neste ecrã pode ver o score final e a ronda alcançada. Quando quiser regressar ao menu inicial *(secção 1.1)*, poderá premir o botão "*MAIN MENU*".

1.6.2. O outro jogador abandonou o jogo

Quando é perdida a ligação ao outro jogador, ou quando este abandona o jogo através do menu de pausa *(secção 1.5)*, o jogador é remetido para o menu de fim de jogo, com a mensagem "*THE OTHER PLAYER HAS LEFT*". Neste ecrã pode ver o score final e a ronda alcançada. Quando quiser regressar ao menu inicial *(secção 1.1)*, poderá premir o botão "*MAIN MENU*".

2. Estado do Projeto

Todas as funcionalidades pretendidas e descritas na *secção 1* foram implementadas.

Tabela de periféricos

Periférico	Usado para	Interrupções
Timer	Controlar *frame rate*; controlar *timeout* do protocolo de comunicação; *ticks* de animação do temporizador de fim de ronda e das mensagens de "*GAMEOVER*" e "*CORRECT*".	S
KBD	Pausar e retomar o jogo (tecla *Esc*); usar a caixa de texto; atalhos da caixa de texto e da tela de desenho.	S
Mouse	Movimentar o cursor; clicar em botões (do menu e da tela de desenho); selecionar caixa de texto e selecionar texto dentro dela; desenhar na tela de desenho.	S
RTC	Ler data/hora; interrupções periódicas para o cursor da caixa de texto piscar e para decrescer o temporizador de fim de ronda; alarmes para fornecer nova pista ao *Adivinho* e para transitar para outro ecrã ao fim de alguns segundos (início de ronda e fim de ronda).	S
Serial port	Sincronização e comunicação entre computadores (receber e enviar mensagens do protocolo de comunicação, por exemplo notificar início de ronda ou enviar tentativa de adivinhar palavra);	S
Video card	Desenho dos ecrãs de menus e de jogo.	N

2.1. Timer

O *Timer* está configurado para gerar interrupções a cada 60 segundos. A sua função principal é **controlar o *frame rate* do programa**, de modo a que este seja de 60 FPS (***draw_frame()***). Para além disso, é usado para medir os tempos de ***timeout* do protocolo de comunicação** (***protocol_tick()***) (*secção 4.8*) e para **atualizar os *ticks* de jogo** para a animação do temporizador de fim de ronda e das mensagens de "*GAMEOVER*" e "*CORRECT GUESS*" (***game_timer_tick()***).

Foram implementadas e usadas funções para a **subscrição e cancelamento da subscrição de interrupções** (***timer_subscribe_int()*** e ***timer_unsubscribe_int()***), bem como o seu ***interrupt handler***

(***timer_int_handler()***) que **gera um evento** do *Timer* para ser processado **posteriormente** pelo código dependente da aplicação.

Os eventos do *Timer* são processados na função ***dispatch_timer_tick()***, onde são chamadas as funções para os usos referidos anteriormente.

2.2. Keyboard

O teclado é utilizado para **controlo do jogo** (entrar e sair do menu de pausa (***menu_react_kbd()***), bem como atalhos da tela de desenho (***canvas_react_kbd()***) e das caixas de texto (***text_box_react_kbd()***)). Para além disso, é também usado para **obter *input* de texto** a escrever nas caixas de texto (***text_box_react_kbd()***).

Foram implementadas e usadas funções para **subscrever e cancelar subscrição de interrupções** (***kbd_subscribe_int()*** e ***kbd_unsubscribe_int()***). O seu ***interrupt handler*** (***kbc_ih()***) faz o processamento dos bytes recebidos pelo *Keyboard Controller (KBC)* de forma genérica e independente da aplicação, **convertendo-os em *scancodes***. A cada *scancode* gerado, este handler **gera um evento**, que será **posteriormente** tratado pelo código dependente da aplicação (permitindo assim a separação entre o processamento independente da aplicação do dependente da aplicação). A leitura dos bytes do *Keyboard Controller (KBC)* dentro do *interrupt handler* é feita através da função ***kbc_read_data()***.

Os eventos do teclado são processados em ***dispatch_keyboard_event()***, onde o *scancode* é lido (***kbd_handle_scancode()***) e convertido numa atualização de estado do teclado (***kbd_event_t***) que é comunicado ao menu, e a caixas de texto ou tela de desenho que estejam a escutar eventos.

2.3. Mouse

O **movimento do rato** é usado para atualizar a **posição do cursor** (***cursor_move()***). O **botão esquerdo do rato** é usado para clicar em botões, selecionar a caixa de texto e selecionar o texto dentro dela. Tanto **o botão esquerdo como o direito do rato** poderão ser usados para desenhar na tela de desenho, arrastando o cursor enquanto se mantém o botão premido.

A **subscrição e cancelamento de subscrição de interrupções** é feita nas funções ***mouse_subscribe_int()*** e ***mouse_unsubscribe_int()***. No início e no final do programa, é respetivamente **ativado e desativado o *data reporting***, a partir das funções ***mouse_enable_dr()*** e ***mouse_disable_dr()***.

A chamada de comandos do *KBC* está implementada por camadas de **diferentes graus de abstração** (outros exemplos onde isso se verifica na *secção 4.1*). A função ***kbc_write_reg()*** escreve um byte num registo do KBC, sendo usado pelas funções ***kbc_issue_command()*** e ***kbc_issue_argument()*** que escrevem, respetivamente, um comando e um argumento (usando, para isso, o registo adequado). Estas funções são utilizadas em ***write_byte_to_mouse()***, uma função que generaliza a tarefa de enviar um qualquer byte para o rato, nomeadamente para ativar ou desativar *data reporting*. As duas funções já referidas que fazem essa ativação e desativação chamam, por fim, essa função, com o byte correspondente como argumento.

O ***interrupt handler*** das interrupções do rato (***mouse_ih()***) recebe os bytes do *KBC* (***kbc_read_data()***) processando-os de modo genérico e independente da aplicação, de forma a poder produzir *packets* do rato. Quando um *packet* do rato está pronto a ser processado, é **gerado um evento** do rato, que será

posteriormente tratado pelo código dependente da aplicação (permitindo assim a separação entre o processamento independente da aplicação do dependente da aplicação).

Os eventos do rato são tratados na função ***dispatch_mouse_packet()***, que obtém o *packet* do rato (***mouse_retrieve_packet()***) e atualiza a posição do cursor (***cursor_move()***) e dos botões clicados (***cursor_update_buttons()***), e comunica a atualização desse estado aos botões, caixas de texto e tela de desenho que estejam a escutar eventos (***handle_update_cursor_state()***).

No final da execução do programa, após desativar as subscrições tanto do rato como do teclado, é realizado o ***flush* do *output buffer* do *KBC***, de modo a que, se um byte tivesse ficado neste registo, este não impedisse o Minix de receber interrupções.

2.4. RTC

As ***Update Interrupts*** do *RTC* são usadas para **ler a data/hora** (***rtc_read_date()***), de modo a poder mostrá-la ao utilizador (***rtc_get_current_date()*** para obter a data e ***date_draw()*** para a mostrar no ecrã), bem decidir a mensagem de saudação a mostrar no menu inicial (*secção 1.1*), e ainda para **gerar uma seed** (***rtc_get_seed()***) no início do programa para o *random number generator*.

As **interrupções de alarme** do *RTC* são usadas para **gerar alarmes**. Dependendo do estado do jogo, o alarme pode ser usado para fornecer uma nova pista ao *Adivinho* (***game_give_clue()***), passar do ecrã de nova ronda *(secção 1.3)* para o ecrã de jogo *(secção 1.4)* (***handle_start_round()***), passar das mensagens de fim de ronda para a ronda seguinte ou para o menu de fim de jogo *(secção 1.6)* (***game_rtc_alarm()***).

As **interrupções periódicas** do *RTC* são usadas para para a **animação do cursor da caixa de texto** a piscar quando esta está selecionada (***text_box_cursor_tick()***), e para **reduzir o tempo do temporizador de fim de ronda** (***game_rtc_pi_tick()***).

A **subscrição às interrupções** do *RTC* é feita com a função ***rtc_subscribe_int()*** e **anulada** com ***rtc_unsubscribe_int()***. O seu interrupt handler (***rtc_ih()***) é genérico e independente da aplicação, tratando dos três tipos de interrupções referidos. No caso das *Update Interrupts*, é lida a data hora como já foi indicado. Para as interrupções de alarme e interrupções periódicas é gerado um evento respetivo para ser **posteriormente** tratado pelo código dependente da aplicação.

Os eventos de interrupções de alarme e interrupções periódicas são tratados pelas funções ***dispatch_rtc_alarm_int()*** e ***dispatch_rtc_periodic_int()***, respetivamente.

No início do programa é lido o *REGISTER_C* (***rtc_flush()***) de modo a limpar alguma interrupção que tivesse ficado pendente de uma execução anterior do código. É também ainda lida desde logo a data de modo a tê-la disponível para consulta no módulo ***rtc*** desde o primeiro frame (***rtc_read_date()***).

Os três tipos de interrupções podem ser ativados e desativados com as funções genéricas ***rtc_enable_int()*** e ***rtc_disable_int()***. No caso das *Update Interrupts* e interrupções de alarme, podem ainda ser ser usadas funções mais específicas para as ativar, que chamam a função ***rtc_enable_int()*** com os devidos argumentos. As *Update Interrupts* podem ser ativadas com ***rtc_enable_update_int()***, como é feito no início do programa. Já as interrupções de alarme podem ser ativadas com ***rtc_set_alarm_in()*** em que em argumento é passado o intervalo de tempo pretendido até que a interrupção de alarme ocorra. Caso se pretenda especificar o tempo em que se quer a interrupção de alarme (e não o tempo até ela ocorrer), pode usar-se simplesmente ***rtc_enable_int()*** com os devidos argumentos.

As interrupções periódicas são ativadas no início do programa e configuradas para gerar uma interrupção a cada meio segundo. As interrupções de alarme apenas são ativadas durante a execução do programa, nas alturas em que são necessárias.

Tanto as interrupções periódicas como as *Update Interrupts* são desativadas no final do programa (como estavam antes de o programa correr), seguindo-se a chamada a ***rtc_unsubscribe_int()***.

2.5. Serial port

Foi usada a porta série para efetuar a **comunicação entre os computadores** dos dois jogadores, **segundo o protocolo estabelecido** (e descrito na *secção 4.8*). Tanto a **transmissão**, como **receção** de bytes, como **deteção de erros** é realizada por via de **interrupções**, cuja subscrição é feita com a função ***com1_subscribe_int()*** e desativada no final do programa com ***com1_unsubscribe_int()***.

A **configuração** da porta série é feita no início do programa com recurso à função ***protocol_config_uart()***, que usa os seguintes parâmetros: **8 bits por palavra**, **paridade ímpar**, **dois bits de paragem** e um **bit-rate de 9600 bits por segundo** (**uart_config_params()**, que por sua vez chama **uart_set_bit_rate()** para definir o bit-rate). Foram **ativadas as FIFOs** de hardware embebidas na porta série (***uart_enable_fifo()***) com um **Interrupt Trigger Level** que dispara uma interrupção quando a FIFO tem **8 bytes**. Foram também **usadas filas** (*software FIFO*s, implementadas no módulo ***queue***) para armazenar os bytes recebidos e por transmitir. A função de configuração ainda ativa os três tipos de interrupções (***uart_config_int()***), inicializa as filas (***uart_init_sw_queues()***) e limpa as FIFOs de harware (***uart_clear_hw_fifos()***).

No início do programa, é realizado o ***flush*** **dos bytes recebidos** (***uart_flush_received_bytes()***), de modo a não correr o risco de estar a receber uma mensagem a meio. O funcionamento desta função é descrito mais pormenorizadamente na *secção 4.8*.

O *interrupt handler* da porta série (***com1_ih()***) é genérico. Para cada interrupção pendente, identifica-a (***uart_identify_interrupt()***).

No caso da interrupção *Transmitter Empty*, realiza a **transmissão** de bytes (da fila de *software*) a serem enviados para a porta série (***uart_send_bytes()***). Os bytes são previamente enviados para uma fila de software por código dependente da aplicação, através da função (***uart_send_byte()***). É o protocolo de comunicação que se responsabiliza de enviar os bytes pare essa fila, conforme as mensagens que se pretendam enviar. Mais detalhes do protocolo na *secção 4.8*.

Para as interrupções *Character Timeout Indication* e *Received Data Available*, **recebe** os bytes da porta série para a fila de bytes a serem processados (***uart_receive_bytes()***). De seguida, **gera um evento** de bytes da porta série recebidos, de modo a que **posteriormentes** sejam tratados pelo código dependente da aplicação. Estes eventos são tratados na função ***dispatch_uart_received_data()***, que chama a função ***protocol_handle_received_bytes()***, que os processa de acordo com o protocolo implementado (ver mais informações na *secção 4.8*).

Se a interrupção for causada por *Line Status*, verifica-se a **ocorrência de um erro** (***uart_handle_error()***, que chama ***uart_check_error()***), sendo **gerado um evento** de erro da porta série caso tenha ocorrido, de modo a que seja tratado **posteriormente** pelo código dependente da aplicação.

Estes eventos são tratados na função ***dispatch_uart_error()***, que delega à função ***protocol_handle_error()*** o tratamento do erro, feito de acordo com o protocolo implementado (ver mais informações na *secção 4.8*).

Pela porta série são enviadas **mensagens** (***message_t***), bem como **bytes de *acknowledgment*/*non-acknowledgment*** para garantir robustez na transmissão das mensagens. Cada mensagem tem um **tipo** (***message_type_t***) (como por exemplo jogador saiu do jogo, ronda a começar, novo traço na tela de desenho) e, se aplicável a esse tipo, o seu **conteúdo** (no caso do traço da tela de desenho, a cor e grossura). A frequência com que as mensagens são enviadas depende das ações dos jogadores (porém é enviada uma mensagem de *ping* a cada 5 segundos, para garantir que o outro computador ainda está a correr o processo), mas apenas é enviada uma mensagem de cada vez, aguardando-se o seu *acknowledgment* antes de enviar a próxima (as mensagens por enviar ficam guardadas numa fila). Mais informações acerca do protocolo na *secção 4.8*.

Os recursos alocados à porta série são libertados no final do programa com ***protocol_exit()***.

2.6. Video card

O jogo corre em **modo gráfico**, no modo **0x118**. Como tal, a **resolução é de 1024x768** e as cores estão codificadas em ***direct color mode***, com **24 bits por pixel** (8 bits para cada cor, vermelho, verde e azul), permitindo um total de 2^24 = **16,777,216 cores** (cerca de 16.8M). O modo é inicializado na função ***vg_init()***, que chama as funções *VBE 0x01 RETURN VBE MODE INFORMATION* (ver função ***vbe_get_mode_inf()***) e *VBE 0x02 SET VBE MODE* (ver função ***vbe_change_mode()***) para inicializar o modo gráfico pretendido.

Foi implementado **double buffering** com **page flipping**. A alocação dos dois *buffers* (*back buffer*, onde é desenhada a próxima *frame*, e *front buffer*, onde está presente a *frame* atual) é feita em ***vg_init()***, em espaços de memória física consecutivas. No final de **desenhar cada frame** (***draw_frame()***) no ***back buffer***, é usada a função ***vg_flip_page()*** para **realizar a troca entre os buffers**. Esta função chama a função *VBE 0x07 SET DISPLAY START* (ver ***vbe_set_display_start()***) de modo a que o pixel mostrado no topo esquerdo do ecrã corresponda ao começo do *buffer* que se pretende visível, trocando de seguida os papéis de ambos os *buffers* no programa.

As imagens usadas no projeto estão guardadas como ***XPM***s, na **pasta xpm** do código fonte. São carregadas com recurso à função ***xpm_load()*** fornecida pela LCF, com o tipo *XPM_8_8_8*, correspondente ao modo do nosso programa.

É utilizada uma **fonte de texto**, cujo código relevante está implementado no módulo ***font*** (***font_load()*** e ***font_unload()*** para alocar e dealocar recursos; ***font_draw_char()*** para desenhar um único caracter; ***font_draw_string()***, ***font_draw_string_limited()*** e ***font_draw_string_centered()*** para desenhar uma string). É usada para **apresentar texto no ecrã**, como a data atual, o *score* e número da ronda, o tempo para o fim da ronda, a palavra a adivinhar, entre outros. O texto dos botões e das mensagens de "*GAMEOVER*" e "*CORRECT*", sendo estáticos e estilizados, foram desenhados por imagens independentes em vez da fonte.

Há também uma implementação de objetos animados (***xpm_animation_t***, funções relevantes são ***xpm_load_animation()***, ***xpm_unload_animation()*** e ***vb_draw_animation_frame()*** implementados no módulo graphics) usado para animar o temporizador de fim de ronda.

Para além do *front* e *back buffer*, há ainda um dois ***buffers* correspondente à tela de desenho**. O primeiro é onde é armazenado o desenho atual da tela. A cada *frame*, o conteúdo deste *buffer* é copiado para o *back buffer* (***canvas_draw_frame()***), de modo para que o desenho possa ser mostrado de modo eficiente e sem ser corrompido por objetos que se movam por cima dele (como o cursor ou as mensagens de "*GAMEOVER*" e "*CORRECT*"). O segundo *buffer* serve para tornar mais eficiente a funcionalidade de *undo* da tela: inicialmente está pintado a branco; sempre que o desenho se torna suficientemente grande, a parte mais antiga é desenhada neste *buffer* (deixando de ser permitido fazer *undo* dessa parte) e sempre que seja necessário redesenhar o *buffer* principal da tela (por causa da ação de *undo*), é copiado o conteúdo deste segundo *buffer* para o principal, apenas tendo de ser redesenhados os traços mais recentes.

É ainda realizada uma **deteção simples de "colisões"** entre um ponto (cursor do rato) e vários elementos gráficos (botões, caixas de texto, tela de desenho) para determinar se este está a pairar sobre eles (***button_is_hovering()***, ***text_box_is_hovering()***, ***canvas_is_hovering()***).

No final do programa é chamada a função ***vg_exit()***, fornecida pela *LCF*, para retornar ao modo de texto *default* do Minix.

3. Organização e estruturação do código

3.1. Módulos Desenvolvidos

3.1.1. button (3%)

Módulo dedicado à estrutura e manipulação de botões. Os botões são usados nos menus e nas e nas ferramentas e ações da tela de desenho. Este módulo segue uma estrutra orientada a objetos, com uma função para inicializar um botão e funções para manipulá-lo, desenhá-lo e atualizar o seu estado.

A estrutura principal deste módulo é ***button_t***, com a posição, tamanho, estado e ícone do botão, bem como um apontador de função para a ação que é realizada quando este é clicado.

Existe também a estrutura auxiliar ***button_icon_t***, com informação do ícone que será desenhado no botão, e a enumeração ***button_state_t***, para representar o estado atual de um botão em relação aos eventos do cursor (premido, cursor a pairar sobre o botão, etc.).

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.2. canvas (7%)

Módulo dedicado à inicialização, manipulação e eliminação da tela de desenho. Como há apenas uma única tela inicializada no programa num dado momento, o seu estado é guardado por meio de variáveis estáticas, com funções que permitem a outros módulos manipular a tela. A tela deve sempre ser inicializada antes do seu uso, e eliminada quando já não será mais usada.

A tela possui uma instância estática de ***frame_buffer_t*** (ver módulo ***graphics***) para armazenar o desenho e poder copiá-lo eficientemente para o *frame buffer* quando estiver a ser desenhada a próxima frame (***canvas_buf***). Possui ainda outra instância estática de ***frame_buffer_t*** (***canvas_base_buf***) para tornar mais eficiente a funcionalidade de *undo*. Sempre que o desenho se torna suficientemente grande, a parte mais antiga do desenho é desenhada neste segundo *buffer* (deixando de ser permitido fazer *undo* dessa parte, visto que são apagadas as respetivas instâncias ***stroke_t***, estrutura que é explicada no parágrafo seguinte) e sempre que é necessário redesenhar o *buffer* principal (devido a um *undo*), é copiado o conteúdo deste segundo *buffer*, apenas tendo de ser redesenhados os traços mais recentes.

Este módulo possui a estrutura ***stroke_t***, que representa um traço (desde que o botão do rato foi premido até ser largado) na tela, com determinada cor e grossura, bem como apontadores para os traços anterior e seguinte. O atributo mais relevante no traço é uma array de "átomos de traço" (***stroke_atom_t***) que são as várias posições pelas quais o cursor passou a desenhar, as quais devem ser unidas por linhas. Os traços são guardados para permitir as ações "desfazer" e "refazer" (Ctrl+Z, Ctrl+Y). As principais funções de desenho são ***canvas_new_stroke()*** (inicia um novo traço), ***canvas_new_stroke_atom()*** (adiciona um "átomo" ao traço atual), ***canvas_undo_stroke()*** (desfaz o último traço, adicionando-o aos traços desfeitos) e ***canvas_redo_stroke()*** (refaz o último traço desfeito, recuperando-o dos traços desfeitos).

A tela possui ainda uma enumeração ***canvas_state_t***, para representar o seu estado atual em relação aos eventos do cursor (premido, cursor a pairar sobre a tela, etc.).

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.3. clue (2%)

Módulo dedicado à estrutura e manipulação de *Pistas*. As *Pistas* permitem ao *Adivinho* saber o tamanho da palavra e algumas das suas letras. Este módulo segue uma estrutura orientada a objetos, com funções para inicializar, limpar e manipular e desenhar uma instância da estrutura ***word_clue_t***. Pode ser dada uma pista numa posição aleatória, numa posição específica ou revelada a palavra completa.

A única estrutura deste módulo é ***word_clue_t***, que representa a *Pista*. Os seus atributos mais relevantes são a string correspondente à palavra completa e outra a representar as letras já desvendadas.

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.4. cursor (2%)

Módulo dedicado ao cursor do rato. Permite inicializar o rato, bem como eliminar os seus recursos (no final do programa). Possui informação relativamente à posição do cursor e se os botões esquerdo e direito do rato estão premidos, bem como se o rato deve ser desenhado como uma seta, um cursor de texto, um cursor de desenho ou um cursor desativado.

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.5. date (3%)

Módulo dedicado à estrutura e manipulação de datas. As datas são usadas para representar e armazenar os valores lidos pelo *RTC* de modo a poderem ser mostradas aos jogadores durante o jogo.

A estrutura principal deste módulo é ***date_t*** com campos para ano, mês, dia do mês, hora, minuto e segundo. Para além do já referido, é possível escrever datas no ecrã, comparar duas datas com um operador menor na forma de uma função (***date_operator_less_than()***) e ainda somar uma data com outra estrutura temporal, ***rtc_alarm_time_t*** (***date_plus_alarm_time()***), de modo a calcular a data na qual deve ser gerada a próxima interrupção de alarme.

O código deste módulo foi implementado por Diogo Costa.

3.1.6. dispatcher (15%)

Módulo dedicado ao despacho e tratamento de eventos específico à aplicação. Botões, caixas de texto e a tela de desenho podem ser associados por funções de *bind* (***dispatcher_bind_buttons()***, ***dispatcher_bind_text_boxes()*** e ***dispatcher_bind_canvas()***), de modo a escutarem e poderem reagir aos eventos.

A função principal deste módulo é ***dispatcher_dispatch_events()***, que lê todos os eventos em fila de espera, delegando-os à função que processa o evento respetivo.

A enumeração principal deste módulo é ***event_t***, que representa um evento lançado por um *interrupt handler* de um dos vários dispositivos (através da função ***dispatcher_queue_event()***), de modo a que se possa dar o seu despacho e processamento por parte do código específico da aplicação.

O módulo possui ainda *handlers* para serem chamados após o processamento dos eventos dos periféricos, como por exemplo funções com a ação que deve ser realizada após receber determinada mensagem da porta série, ou após clicar num botão.

O código deste módulo foi implementado principalmente por Pedro Gonçalo Correia. O despacho de interrupções do RTC foi implementado por Diogo Costa.

3.1.7. font (2%)

Módulo dedicado à fonte do jogo. Permite carregar e eliminar os recursos associados à fonte, bem como desenhar um determinado caracter ou string numa posição do ecrã. A fonte inclui as 26 letras do alfabeto em maiúsculas, os dígitos de 0 a 9, bem como os símbolos apresentados a seguir entre aspas: “: - . /”.

O código deste módulo foi implementado por Diogo Costa.

3.1.8. game (17%)

Módulo dedicado à lógica do jogo e às suas rondas. Permite carregar e eliminar os recursos associados a um jogo, bem como começar e eliminar um novo jogo ou ronda do jogo, ou atualizar o seu estado.

Possui diversas estruturas não visíveis para os restantes módulos, referentes aos dados atuais do jogo. Destaca-se ***game_t***, com o estado do jogo, o *score*, o número da ronda atual e o estado da ronda atual (***round_t***); e ***round_t***, com informação do temporizador da ronda, das tentativas de adivinhar a palavra, da *Pista* (ver módulo ***clue***) e do papel do jogador (*Pintor* ou *Adivinho*). As estruturas ***drawer_t*** e ***guesser_t*** possuem informação apenas aplicável ao role de *Pintor* ou *Adivinho*, respetivamente.

O código deste módulo foi principalmente implementado por Pedro Gonçalo Correia, com as funções que fazem uso do RTC implementadas por Diogo Costa.

3.1.9. graphics (3%)

Módulo dedicado ao desenho de gráficos num *frame buffer*. Permite desenhar píxeis, linhas verticais, horizontais ou oblíquas, retângulos, círculos, imagens ou animações.

A estrutura principal do módulo é o ***frame_buffer_t***, que representa o conteúdo e informação relativa a um *frame buffer*. No programa existem dois *frame buffers* (em ***video_gr.c***) para o desenho da frame (que alternam com *page flipping*) bem como dois *frame_buffers* da tela de desenho, para permitir desenhar eficientemente o desenho no ecrã (mais acerca destes últimos nas *secções 2.6, 3.1.2 e 4.6*).

Existe também uma estrutura ***xpm_animation_t***, que representa uma animação (conjunto de frames do mesmo tamanho que devem ser mostradas em sucessão).

Grande parte do código foi adaptado do nosso trabalho no lab5. O módulo foi desenvolvido por ambos os elementos do grupo.

3.1.10. i8042 (0.1%)

Módulo dedicado a constantes do *Keyboard Controller* (*KBC*).

O código deste módulo foi reutilizado da nossa implementação dos *labs 3 e 4*, tendo sido desenvolvido por ambos os elementos do grupo em conjunto.

3.1.11. i8254 (0.1%)

Módulo dedicado a constantes do *i8254 Timer*.

O código deste módulo foi fornecido pelo professor durante a realização do *lab2*.

3.1.12. kbc (1.4%)

Módulo dedicado à interface genérica com o *Keyboard Controler* (*KBC)*, servindo como base à implementação dos módulos ***keyboard*** e ***mouse***. A funcionalidade principal do módulo é enviar comandos e argumentos para o *KBC* e receber dados do *KBC*.

O código deste módulo foi reutilizado da nossa implementação dos *labs 3 e 4*, tendo sido desenvolvido por ambos os elementos do grupo em conjunto.

3.1.13. keyboard (1.9%)

Módulo dedicado à interface com o teclado. Permite subscrever e cancelar a subscrição das interrupções do teclado. Possui a implementação do interrupt handler independente da aplicação, que lê os scancodes do teclado e gera um evento para posteriormente ser tratado pelo código dependente da aplicação. Está ainda implementada uma função que processa o scancode, convertendo-o num evento de atualização do estado do teclado dependente da aplicação, para ser usada pelo módulo ***dispatcher***.

O módulo possui a estrutura ***kbd_event_t*** que representa um evento do teclado dependente da aplicação. Este evento consiste na informação de qual tecla foi pressionada e se o *Ctrl* estava premido quando ela foi pressionada.

O código deste módulo foi em grande parte reutilizado da implementação do *lab3*, tendo essa parte sido desenvolvida de forma igual por ambos os elementos do grupo. A parte referente aos eventos do teclado específicos da aplicação foi implementada por Diogo Costa.

3.1.14. menu (5%)

Módulo dedicado ao menu do jogo. Possui informação do estado do menu, isto é, o ecrã que deve ser mostrado (menu inicial, ecrã de jogo, menu de pausa, entre outros), bem como os botões que são utilizados nos vários menus. As funções do módulo permitem inicializar e eliminar os recursos do menu, mudar de ecrã (associando diferentes objetos (como botões) ao despacho de eventos do módulo ***dispatcher***), e desenhar o ecrã atual.

Módulo implementado principalmente por Diogo Costa. Ecrãs relativos ao início da ronda e ao fim do jogo implementados por Pedro Gonçalo Correia.

3.1.15. mouse (2%)

Módulo dedicado à interface com o rato. Permite subscrever e cancelar a subscrição das interrupções do rato, bem como ativar e desativar data reporting. Possui um interrupt handler independente da aplicação, que lê os bytes recebidos e gera um evento para posteriormente ser tratado pelo código dependente da aplicação. Esses bytes são posteriormente convertidos em packets (***mouse_retrieve_packet()***) e processados no módulo ***dispatcher***.

O código deste módulo foi reutilizado da implementação do *lab2*, tendo sido desenvolvido de forma igual por ambos os elementos do grupo.

3.1.16. proj (1%)

Módulo que contém a função ***proj_main_loop()***, responsável pela inicialização e término do programa, bem como do loop principal que chama a função ***driver_receive()*** (ver secção 3.3).

O código deste módulo foi desenvolvido de igual modo por ambos os elementos do grupo.

3.1.17. protocol (10%)

Módulo dedicado ao protocolo de comunicação entre computadores. Partindo da implementação independente da aplicação do módulo ***uart***, este módulo introduz um protocolo de transmissão de mensagens específico para o programa. Permite configurar a porta série com os parâmetros deste protocolo, limpar os recursos alocados para a transmissão, enviar mensagens e processar mensagens recebidas, lidar com erros e timeouts na transmissão. As mensagens por enviar são armazenadas numa fila. As mensagens recebidas são processadas em função do seu tipo, com uma array de apontadores para funções indexada pelo tipo de mensagem que elas tratam.

O módulo possui a estrutura ***message_t***, que inclui o tipo de uma mensagem, o tamanho do seu conteúdo e o seu conteúdo. A mensagem pode ser de vários tipos, definidos pela enumeração ***message_type_t***. O conteúdo da mensagem depende do seu tipo, sendo representado na estrutura como um array de bytes.

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.18. queue (2%)

Módulo dedicado à implementação genérica de filas (*FIFO*s). As filas são usadas para armazenar os bytes recebidos e por enviar da porta série (módulo ***uart***), bem como as mensagens do protocolo por enviar (módulo ***protocol***), assim como para os eventos a serem processados pelo códido dependente da aplicação (módulo ***dispatcher***). Este módulo segue uma estrutra orientada a objetos, com uma função para inicializar e eliminar uma queue e funções para manipulá-la.

Este módulo foca-se na manipulação da estrutura ***queue_t***, implementada internamente como uma array de dados genéricos, com informação do tamanho de cada elemento, número de elementos e capacidade, bem como índices da cabeça e cauda da lista. As operações permitidas são remover o elemento da cabeça (***queue_pop()***), adicionar um elemento à cauda (***queue_push()***), ver o elemento no

topo da cabeça (***queue_top()***), esvaziar a fila (***queue_empty()***), ver se ela está cheia (***queue_is_full()***) ou vazia (***queue_is_empty()***).

Se a capacidade não for suficiente para adicionar um elemento, a capacidade é duplicada internamente de forma automática.

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.19. rtc (5%)

Módulo dedicado à interface com o *Real Time Clock* (*RTC*). Permite subscrever e anular a subscrição de interrupções do *RTC*, bem como ativar e desativar os 3 tipos de interrupções (*Update Interrupts*, interrupções de alarme e interrupções periódicas) com as configurações pretendidas.

Quanto às interrupções periódicas, é possível escolher a configuraçãos dos bits *RS3-RS0* do *REGISTER_A* para obter o período pretendido. Quanto às interrupções de alarme, é possível configurar um alarme para um tempo específico, como também para daqui a um tempo específico (usando a soma ***date_plus_alarm_time()*** de ***data_t*** com ***rtc_alarm_time_t*** do módulo date).

O módulo mantém uma estrutura instância de ***date_t*** que vai sendo atualizada a cada *Update Interrupt* e que pode ser consultada usando a função ***rtc_get_current_date()***. É ainda possível ler a configuração do *RTC* e também obter uma *seed* usada para gerar números pseudo-aleatórios (i.e. número dado como argumento à função ***srand()***). O *interrupt handler* do *RTC* é genérico e independente da aplicação detetando os 3 tipos de interrupções (*Update Interrupts*, interrupções de alarme e interrupções periódicas), gerando eventos das duas últimas para serem posteriormente tratadas pelo código dependente da aplicação.

O código deste módulo foi implementado por Diogo Costa.

3.1.20. scan_codes (0.1%)

Módulo dedicado à definição de constantes para os scancodes de várias teclas do teclado usadas no projeto.

O código deste módulo foi implementado por Diogo Costa.

3.1.21. text_box (8%)

Módulo dedicado à estrutura e manipulação de caixas de texto. Uma caixa de texto é utilizada pelo *Adivinho* para adivinhar a palavra desenhada em cada ronda. Este módulo segue uma estrutura orientada a objetos, com uma função para inicializar e eliminar uma caixa de texto e funções para desenhá-la, manipulá-la e atualizar o seu estado.

A estrutura principal deste módulo é ***text_box_t***, com uma string para o texto escrito e número de caracteres, a posição e tamanho da caixa de texto, a posição do cursor de texto e a porção de texto selecionada, a parte do texto visível na caixa de texto (quando a palavra é demasiado grande para o tamanho da caixa de texto), e um apontador de função para a ação que é executada quando o texto é submetido.

Existe também a enumeração auxiliar ***text_box_state_t***, para representar o estado atual de uma caixa de texto em relação aos eventos do cursor (selecionada, cursor a pairar sobre o a caixa de texto, etc.).

O código deste módulo foi implementado por Diogo Costa.

3.1.22. timer (0.3%)

Módulo dedicado à interface com o *i8254 Timer*. Permite alterar a frequência das interrupções, ler a configuração, subscrever e anular subscrição das interrupções. O *interrupt handler* gera um evento do timer, para ser posteriormente processado no módulo ***dispatcher***.

O código deste módulo foi reutilizado do *lab2*, tendo sido implementado em conjunto por ambos os elementos do grupo.

3.1.23. uart (7%)

Módulo dedicado à interface com a porta série da *COM1*. Permite subscrever e cancelar subscrições da *COM1*, inicializar e libertar recursos (filas de bytes transmitidos e recebidos, ver módulo ***queue***), enviar e receber bytes, detetar erros, limpar todos os bytes recebidos e as *FIFO*s de hardware, configurar parâmetros de comunicação, ativar e desativar as *FIFO*s de hardware, e alterar o *bit-rate* da comunicação. O interrupt handler da *COM1* é genérico e independente da aplicação, detetando e processando as seguintes interrupções:

- *Transmitter Empty*
- *Character Timeout Indication*
- *Received Data Available*
- *Line Status*

Os bytes a enviar são armazenados numa fila de bytes implementada em *software* (***queue_t***) até ser possível enviá-los pela porta série. Os bytes recebidos são também armazenados numa fila de bytes implementada em *software*, até serem posteriormente processados pelo protocolo específico da aplicação.

O código deste módulo foi implementado por Pedro Gonçalo Correia.

3.1.24. utils (0.1%)

Funções utilitárias para obter o byte mais significativo e o byte menos significativo de uma half-word, bem como chamar a função ***sys_inb()***, guardando o resultado numa variável de 8 bits.

O código deste módulo foi reutilizado do *lab2*, tendo sido implementado em conjunto por ambos os elementos do grupo.

3.1.25. vbe (0.5%)

Funções utilitárias para interface com a VBE. Permite chamar as seguintes funções da VBE:

- ***0x01*** *RETURN VBE MODE INFORMATION*
- ***0x02*** *SET VBE MODE*
- ***0x07*** *SET DISPLAY START*

Estas funções são usadas no módulo ***video_gr***: as primeiras duas para obter informações sobre e configurar o modo da placa gráfica no início do programa, e a terceira para implementar *page flipping*. O *page flipping* é realizado durante o *vertical retrace*.

O código deste módulo foi adaptado do *lab5*, tendo sido implementado em conjunto por ambos os elementos do grupo.

3.1.26. video_gr (1.5%)

Módulo dedicado à interface com a placa gráfica. Permite inicializar a placa gráfica no modo pretendido, obter informação quanto à resolução vertical, horizontal e número de bytes por pixel, obter o back buffer para desenhar nele e realizar o *page flipping*.

Internamente, possui dois *frame buffers* (***frame_buffer_t***, ver módulo ***graphics***) e informação de qual deles é o *front buffer*.

O código deste módulo foi adaptado do *lab5*, tendo sido implementado em conjunto por ambos os elementos do grupo.

3.2. Código Utilizado da Internet

3.2.1. Desenho de Linhas Oblíquas (vb_draw_line())

Este código encontra-se na função ***vb_draw_line()*** do módulo ***graphics***. A fonte a partir da qual nos baseámos foi o seguinte excerto, da autoria de Alois Zingl:

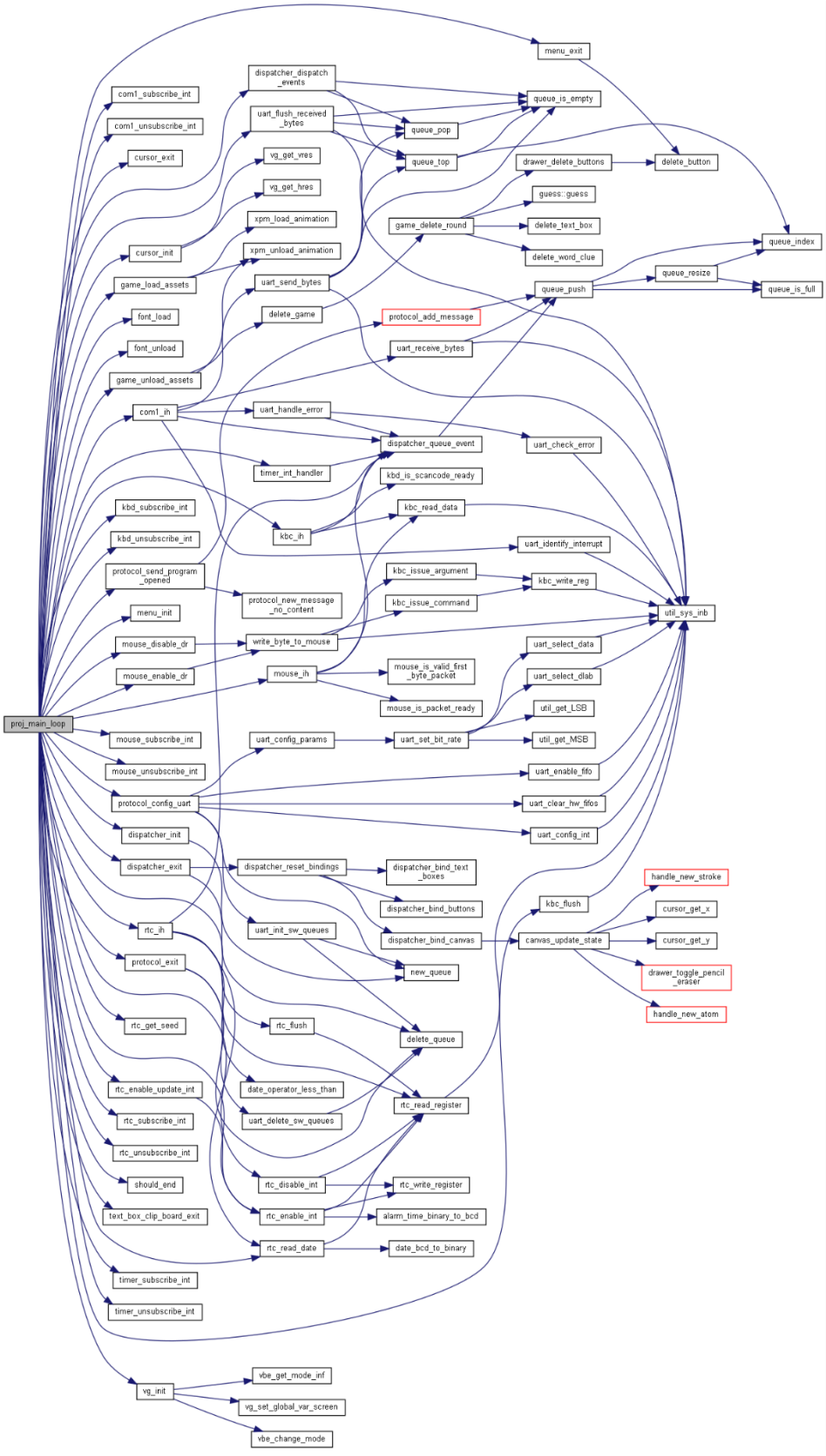
```
void plotLine(int x0, int y0, int x1, int y1)
{
   int dx =  abs(x1-x0), sx = x0<x1 ? 1 : -1;
   int dy = -abs(y1-y0), sy = y0<y1 ? 1 : -1;
   int err = dx+dy, e2; /* error value e_xy */

   for(;;){  /* loop */
      setPixel(x0,y0);
      if (x0==x1 && y0==y1) break;
      e2 = 2*err;
      if (e2 >= dy) { err += dy; x0 += sx; } /* e_xy+e_x > 0 */
      if (e2 <= dx) { err += dx; y0 += sy; } /* e_xy+e_y < 0 */
   }
}
```

Este código pode ser encontrado em: http://members.chello.at/~easyfilter/bresenham.html

O código permite desenhar, recorrendo ao Algoritmo de Bresenham, uma linha oblíqua entre dois pontos. Foi utilizada no projeto para poder desenhar traços na tela de desenho, a partir de várias posições do rato. Foram realizadas alterações ao código, de modo a adaptá-lo ao estilo do código do projeto e às

suas necessidades. A alteração mais significativa foi a substituição da função *setPixel()* por ***vb_draw_circle()***, de modo a desenhar uma linha com grossura e aspeto arredondado.

3.3. Gráfico de Chamada de funções



proj_main_loop()

No início do programa, a função ***proj_main_loop()*** realiza a inicialização e subscrição de interrupções dos vários dispositivos, a alocação de recursos de jogo (por exemplo *XPM*s) e a inicialização do cursor (***cursor_init()***), do menu (***menu_init()***) e da fila de eventos (***dispatcher_init()***).

Possui também o *loop* principal do programa, onde é chamada a função ***driver_receive()***. No *loop*, conforme as interrupções geradas, são chamados os respetivos *interrupt handlers* (***mouse_ih()***, ***kbc_ih()***, ***rtc_ih()***, ***com1_ih()***, ***timer_int_handler()***) e, no final de cada iteração, a função ***dispatcher_dispatch_events()*** para tratar dos eventos gerados.

No final do programa, após a saída do loop principal, a função ainda faz a libertação dos recursos alocados (*XPM*s, filas, entre outros) e a finalização dos vários dispositivos (como anular subscrição das interrupções, operações de *flush*, desativar *data reporting* do rato, voltar ao modo de texto inicial do Minix).

dispatcher_dispatch_events()

Esta função, chamada a cada iteração do *loop* principal do programa, lê todos os eventos que foram gerados pelos *interrupt handlers*, delegando cada um para o *dispatcher* do respetivo evento (de modo a fazer o seu tratamento e despacho).

Nota: As funções para as quais os eventos são delegados não são incluídos no gráfico de chamadas de funções do *Doxygen*, visto que a sua chamada é feita por meio de uma array de apontadores para funções indexada pelo tipo de evento (e como tal a chamada não é detetada pelo *Doxygen*). As funções chamadas são ***dispatch_mouse_packet()***, ***dispatch_keyboard_event()***, ***dispatch_rtc_periodic_int()***, ***dispatch_rtc_alarm_int()***, ***dispatch_uart_received_data()***, ***dispatch_uart_error()*** e ***dispatch_timer_tick()***. Cada uma processa o evento respetivo de modo a atualizar o estado do programa.

4. Detalhes de Implementação

4.1. Estruturação do Código por Camadas

As funções do código do nosso projeto estão organizadas por camadas com diferentes graus de abstração, desde um nível mais baixo, que interage diretamente com os periféricos de forma genérica e independente da aplicação, passando por *dispatchers* de eventos específicos da aplicação mas que atuam de forma genérica (por exemplo, sobre uma lista de botões e caixas de texto no despacho de eventos; as camadas de mais alto nível é que associam objetos específicos a estas listas) até chegar por fim à lógica do jogo, mais concreta.

A estruturação por camadas está presente nas várias componentes do projeto, sendo as camadas de abstração superior são construídas incrementalmente a partir das de mais baixo nível. Um exemplo desta hierarquia já havia sido referido na *secção 2.3*, no que diz respeito à ativação do *data reporting* no rato. Outro exemplo é a mostrar texto no ecrã: começando por uma função para desenhar um único pixel (***vb_draw_pixel()***), passando por uma que desenha uma imagem (***vb_draw_img()***), uma que desenha uma única letra (***font_draw_char()***) e por fim uma função que desenha a string pretendida (***font_draw_string()***).

Esta estruturação permite um desenvolvimento mais modular e torna o código mais versátil, sendo muito mais fácil acomodar alterações ou novas funcionalidades.

4.2. Interrupt handlers genéricos

Na estruturação do projeto, separámos o tratamento de interrupções, independente da aplicação, do tratamento de eventos, dependente da aplicação. De facto, os *interrupt handlers* chamados a partir do driver receive loop interagem com os dispositivos de forma genérica, efetuando algum processamento básico (se necessário para esse dispositivo, como ler os bytes do teclado e do rato, ou transferir bytes de ou para a porta série, ou ler a data no *RTC*) e gerando um novo evento (uma simples notificação genérica de que a interrupção ocorreu, independente da aplicação). Apenas após tratar as interrupções de todos os dispositivos geradas numa iteração do ciclo da função ***driver_receive()*** é que se dá por fim lugar ao despacho de todos os eventos lançados para o tratamento dependente da aplicação (com a função ***dispatcher_dispatch_events()***).

4.3. Código orientado a eventos

Como já ficou implícito nas secções anteriores, para uma maior modularização e versatilidade do código, optámos por um estilo orientado a eventos. Quando um evento é gerado num *interrupt handler*, este é adicionado a uma fila de eventos que devem ser processados. No final de cada iteração do ciclo da função ***driver_receive()***, é chamada a função ***dispatcher_dispatch_events()***, onde todos os eventos presentes na fila são processados pela aplicação até a fila estar vazia. Há 7 tipos de eventos:

- **MOUSE_EVENT** — Novo *packet* do rato disponível para ser processado.
- **KEYBOARD_EVENT** — Novo *scancode* do teclado disponível para ser processado.
- **RTC_PERIODIC_INTERRUPT_EVENT** — Interrupção periódica do RTC.
- **RTC_ALARM_EVENT** — Interrupção de alarme do RTC.
- **UART_RECEIVED_DATA_EVENT** — Novos bytes lidos da porta série.
- **UART_ERROR_EVENT** — Erro detetado na porta série.
- **TIMER_TICK_EVENT** — Interrupção do Timer.

Para cada um existe uma função que o trata e chama os vários handlers que atualizam o estado de jogo conforme o evento. Os vários *dispatchers* estão contidos num array, indexados pelo tipo de evento, de modo a que a chamada é feita chamando o apontador de função no índice do array correspondente ao tipo do evento.

4.4. Máquinas de Estados

As funções de ***button_t***, ***textbox_t*** e da tela de desenho (módulo ***canvas***) que reagem a eventos do rato tomam a forma de máquinas de estado (***button_update_state()***, ***canvas_update_state()*** e ***text_box_update_state()***). O alfabeto de input é o conjunto dos valores que podem ser tomados por um tripleto de booleanos: (cursor a pairar, botão esquerdo premido, botão direito premido); os estados possíveis são os estados relevantes para o respetivo objeto em relação ao cursor (***button_state_t***, ***text_box_state_t*** e ***canvas_state_t***, respetivamente); e a output são os efeitos que cada transição de estado pode produzir no objeto, bem como a chamada de determinadas ações (por exemplo, a ação de quando um botão é clicado, ou desenhar na tela de desenho).

Optámos por utilizar um tripleto como input e não um evento de cada vez visto que, caso dois eventos sejam detetados ao mesmo tempo (por exemplo, o utilizador começar a premir os botões esquerdo e direito do rato ao mesmo tempo), a transição entre estados pode ter de ser diferente de se cada evento ocorresse sequencialmente (primeiro o utilizador premir o botão esquerdo e logo a seguir o botão direito). Assim, esta foi a solução que nos conferiu maior flexibilidade, simplicidade e legibilidade, por não ter de definir explicitamente um evento diferente para cada combinação de valores do tripleto.

4.5. Programação Orientada a Objetos

Na implementação de vários módulos recorreu-se a um estilo de código orientado a objetos. De facto, os tipos de dados ***button_t***, ***clue_t***, ***queue_t*** e ***text_box_t*** estão definidos apenas nos respetivos ficheiros de extensão "*.c*", e não no *header* (que só tem a sua declaração e *typedef*), de modo a que os restantes módulos não tenham acesso aos seus atributos, encapsulando assim os detalhes de implementação.

Objetos destas "classes" são usados nos outros módulos enquanto apontadores, visto que, estando no *header* apenas declarado o tipo de dados (e não definido), o tipo é incompleto durante a compilação dos outros módulos e o seu tamanho desconhecido. São criados com um "construtor" que inicializa o objeto e retorna um apontador, e são sempre destruídos quando já não são necessários através um "destrutor", que recebe um apontador para o objeto e liberta os seus recursos. Os objetos são manipulados por meio de "métodos", funções que recebem como primeiro argumento um apontador para o objeto.

4.6. Geração de Frames

O programa gera *frames* a um *frame-rate* de 60 FPS, sendo cada *frame* desenhada como resposta a um evento do *Timer* (que está configurado para gerar 60 interrupções por segundo).

Para tornar o jogo mais fluído, evitando artefactos visuais causados por a frame ser mostrada enquanto está a ser desenhada, a geração de *frames* é feita com recurso à técnica de *double-buffering*. De facto, no módulo ***video_gr*** estão declarados dois *buffers* (ocupando espaços de memória física consecutivos), que a cada *frame* alternam entre os papéis de *front buffer* (que contém a frame a ser mostrada no ecrã) e *back buffer* (onde está a ser desenhada a próxima frame). Quando uma *frame* termina de ser completamente desenhada num *buffer*, dá-se a troca de papéis e a nova *frame* será desenhada no outro *buffer*. Para garantir eficiência no uso desta técnica, a troca entre buffers é realizada por *page-flipping*, executando uma chamada à função *VBE 0x07 SET DISPLAY START* para alterar a posição de memória correspondente ao pixel do topo esquerdo do ecrã (para a posição de início do buffer que se pretende mostrar).

Esta função *VBE* é chamada com a opção *SET_DISPLAY_START_DURING_VERTICAL_RETRACE*, de modo a forçar a ocorrência de sincronização vertical e impedindo a ocorrência de artefactos visuais.

Para garantir a eficiência da geração de frames com a tela de desenho, e para que o desenho desta não seja corrompido por objetos desenhados por cima dela (cursor do rato, botões, mensagens de "*GAMEOVER*" e "*CORRECT*"), recorremos a um outro *buffer*, com resolução correspondente ao tamanho da tela de desenho, de modo que o desenho da tela é primeiro armazenado nesse *buffer*, e a cada frame que é gerada, é copiado para o *back buffer*. Para além disso, para evitar que a funcionalidade de *undo* fosse muito ineficiente (por ter de redesenhar todo o desenho de raíz no *buffer* principal da tela), existe ainda um *buffer* secundário da tela de desenho. Assim, existe um limite máximo nos traços (e seu comprimento) desenhados, a partir do qual deixa de ser possível fazer *undo* das partes mais antigas. Essas partes são desenhadas no *buffer* secundário da tela, podendo este ser copiado para o *buffer* principal da tela sempre que haja necessidade de redesenhar (devido a um *undo*). Assim, apenas são redesenhadas as partes mais recentes do desenho, evitando-se assim a potencial lentidão associada a desenhos grandes.

4.7. Detalhes do RTC

A leitura da data atual é feita com recurso às *Update Interrupts*. Através do seu uso, é garantido que o próximo ciclo de atualização dos registos do tempo do *RTC* ocorrerá pelo menos 999 ms depois de cada uma das referidas interrupções, evitando assim que os registos do tempo sejam atualizados durante a sua leitura. É ainda feita uma leitura no início do programa (antes do ciclo da função ***driver_receive()***) de modo a ter a data real disponível a ser consultada no módulo ***rtc*** desde o primeiro *frame*. Devido a esta leitura, teve-se o cuidado extra de verificar a *UIP flag* do *REGISTER_A* e, caso ela esteja ativa, esperar 244 microssegundos antes de efetuar a leitura dos registos temporais. A leitura do tempo é feita na configuração *default* e, como tal, os valores são lidos em *BCD*, sendo feita pelo programa a conversão entre *BCD* e binário, para poder configurar alarmes.

Quanto às interrupções de alarme, por vezes, pode acontecer que, estando já um alarme configurado, queiramos configurar um novo e ignorar o antigo (como acontece quando uma ronda termina, o alarme de nova pista deixa de fazer sentido, e o programa fica à espera de um alarme para mudar de ecrã). O problema que se coloca é que, o alarme antigo pode dar set à *flag AF* imediatamente antes de configurar o novo alarme e essa *AF* ser interpretada como o sendo relativa ao alarme configurado mais recentemente. A solução encontrada para este problema foi guardar no módulo ***rtc*** a data para a qual o último alarme foi configurado e fazer com que o interrupt handler apenas considere válidas as interrupções de alarme quando a data atual não é anterior à data para a qual o último alarme foi configurado.

4.8. Detalhes do Protocolo de Comunicação

Para realizar a comunicação dos entre os dois computadores, recorremos à porta série. A implementação da interface com a porta série (módulo ***uart***), independente da aplicação, e do protocolo de comunicação (módulo ***protocol***), dependente da aplicação, são feitas em módulos diferentes.

Para uma comunicação mais eficiente, a interface com a porta série é feita através de interrupções, quer na transmissão, como na receção, como na deteção de erros. A comunicação é feita com verificação de paridade e de dois stop-bits, conferindo maior robustez. Para reduzir a frequência das interrupções sem correr o risco de perda de dados, estão ativas as *FIFO*s de 16-bytes embebidas na porta série (de receção e transmissão), sendo o *trigger level* para disparar uma interrupção de 8 bytes. Para além disso, a leitura e escrita dos bytes de e para a porta série é feita com recurso a duas queues (*software FIFO*s) de modo a permitir a separação entre o código dependente da aplicação e independente da aplicação.

A comunicação é feita através de mensagens e de *acknowledgement*/*non-acknowledgment* bytes. A estrutura básica de uma mensagem é um ***header***, seguido de um ***body***, seguido de um ***trailer***. O *header* da mensagem é constituído por um byte que marca o início da mensagem, seguido de um byte que indica o tamanho do corpo da mensagem. O corpo da mensagem é constituído por um byte que indica o tipo da mensagem, seguido de bytes para o conteúdo desta, se aplicável (diferentes tipos de mensagem podem ter diferentes conteúdos, ou mesmo nenhum). Já o *trailer* é constituído por um único byte a marcar o fim da mensagem. Após o envio de uma mensagem, é esperado que se receba um *acknowledgment byte* antes de enviar a próxima (entretanto, as novas mensagens pendentes são adicionadas a uma fila). Assim, se a mensagem não for recebida com sucesso (ou for inválida) e seja enviado um *non-acknowledgment*, a

mensagem é enviada novamente. Se após um período de tempo não for recebido um *acknowledgement* ou *non-acknowledgment*, assume-se que foi perdida a ligação (*timeout*) ao outro computador, ou seja, que este saiu do jogo.

À medida que são recebidos os bytes de uma mensagem, o protocolo atualiza o seu estado, que influencia a forma como serão processados os bytes seguintes. Por exemplo, se estiver no estado ***NOT_RECEIVING_MESSAGE*** (não está a receber uma mensagem) e receber um byte que marca o início de uma nova mensagem, passará ao estado ***MESSAGE_START_BYTE_DETECTED***, ou seja, quando ler o próximo byte, irá interpretá-lo como o tamanho do corpo da mensagem. Nesta perspetiva, poderá considerar-se que a leitura de bytes por parte do protocolo está implementada como uma máquina de estados.

Quando uma mensagem está a ser recebida, há também um limite de tempo sem receber bytes a partir do qual se desiste de receber o resto da mensagem (*timeout*), sendo enviado um non-acknowledgment. O controlo do tempo limite de espera por receber o próximo byte de uma mensagem ou um acknowledgment é feito pela função ***protocol_tick()***, que é chamada a cada evento do *Timer*. Esta função também é usada para enviar uma mensagem de ping, a cada 5 segundos, para o outro computador. A mensagem de *ping* não contém informação, mas o seu acknowledgment permite garantir que o outro computador ainda está ativo (evitando-se assim situações em que o processo de um computador é encerrado forçosamente, enquanto o outro fica preso num ecrã a aguardar mensagem para prosseguir, ficando indefinidamente à espera).

Sempre que há um erro a receber a mensagem, é chamada a função ***protocol_handle_error()***, onde se esvazia o *buffer* da mensagem a ser recebida e, para garantir que já não estão a ser enviados bytes da mesma, é chamada a função ***uart_flush_received_bytes()*** onde se lêem os bytes da fila de bytes recebidos da porta série e se executam três tentativas de leitura do *Receiver Buffer Register* com um ligeiro delay entre elas (se após as três tentativas não for recebido nenhum byte, é extremamente provável que o outro computador terminou de enviar a mensagem e está a aguardar *acknowledgment)*. Ao fim desta leitura, faz-se a confirmação de que não foi recebido um *acknowledgment*/*non-acknowledgment* byte antes ou após a mensagem (são guardados o primeiro e último bytes quando se faz o *flush*, precisamente para esta verificação), e, por fim, é emitido um *non-acknowledgment*.

Para garantir a sincronização dos computadores, é aproveitado o facto de que cada computador tem um papel diferente na ronda. No início do jogo, ambos os computadores geram um número aleatório e enviam-no ao outro. Aquele que tiver gerado o maior número é considerado o Pintor, e o outro o Adivinho. Assim, é o Pintor que define o início de uma nova ronda e a palavra a adivinhar, bem como se a ronda foi ganha. A ronda apenas é perdida depois de ambos os jogadores enviarem ao outro uma mensagem de fim do tempo, de modo a garantir que, mesmo que haja algum atraso na comunicação, as tentativas do adivinho ainda sejam contabilizadas se o tempo não tivesse terminado na altura em que foram submetidas.

5. Ficheiros *XPM*

Os ficheiros *XPM* usados no programa encontram-se na pasta ***xpm*** do código fonte do projeto no *Gitlab*.

O ficheiro *XPM* correspondente à fonte (***font.xpm***) foi adaptado de:

https://opengameart.org/content/make-scores-pixel-font

Já os ficheiros usados para o texto estático dos ícones dos botões do menu (***menu_exit_game.xpm***, ***menu_main_menu.xpm***, ***menu_new_game.xpm*** e ***menu_resume.xpm***) foram gerados com recurso ao website:

https://cooltext.com

As fontes correspondentes aos textos estáticos de mensagens de fim de ronda (***correct.xpm*** e ***gameover.xpm***) foram obtidas em:

https://www.1001fonts.com/

Os restantes *XPM*s são da autoria de Pedro Gonçalo Correia.

6. Conclusões

Apesar de por vezes bastante frustante, a unidade curricular de Laboratório de Computadores mostrou-se desafiante e promotora do raciocínio e aprendizagem autónomos. Dito isto, temos duas sugestões a fazer:

- Ainda que os *lab*s não sejam contabilizados para a nota final, gostariamos que fosse efetuada pelo menos uma avaliação qualitativa (alguns comentários breves seria suficiente) do que foi bem implementado e do que poderia ser melhorado, de forma a poder aprender com os erros e melhorar nos próximos *labs*/projeto.
- Tendo em conta que a porta série é um periférico desafiante e interessante de aprender, mas como é muito mais difícil do que os restantes periféricos, achamos que seria muito proveitoso a existência de pelo menos uma aula prática (*lab*) dedicada a esta, para poder ter o apoio dos docentes (é muito provável surgirem dificuldades na sua utilização, comparativamente aos restantes dispositivos). Outra vantagem seria que talvez desta forma mais grupos fossem cativados a tentar incluir a porta série no seu projeto. Este *lab* poderia ser opcional (como os *labs* não contam para avaliação, quem quisesse podia trabalhar no projeto durante a aula) e curto, requerindo apenas a implementação de funções mais básicas.