PROJETO DE LCOM
Robinix

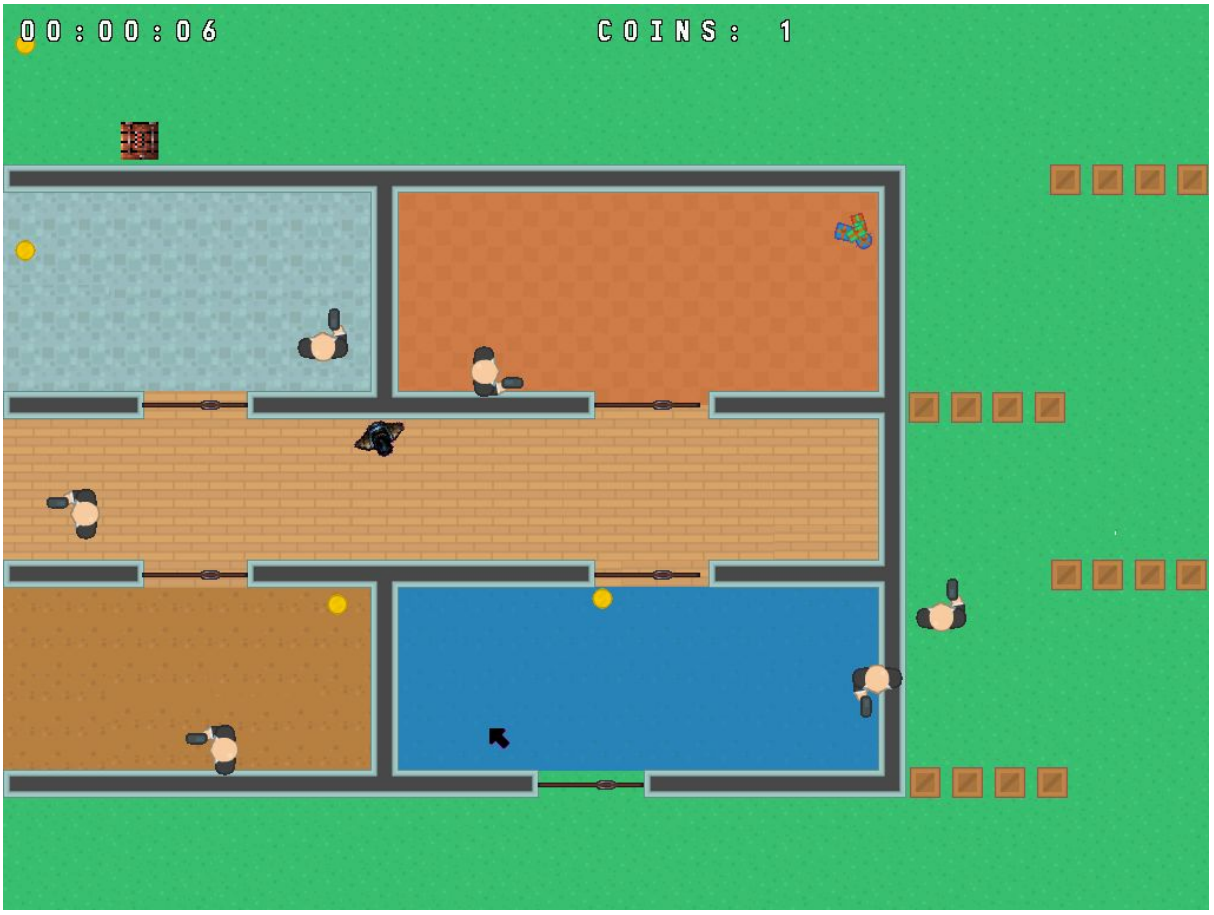


Figura 1 - Durante o jogo, no nível 2 (single player) de Robinix

João Malheiro de Sousa - up201605926@fe.up.pt

Miguel Pereira Duarte - up201606298@fe.up.pt

Faculdade de Engenharia da Universidade do Porto - LCOM, MIEIC 2º ano T5G05

Índice

Instruções de Utilização

Main Menu

Ao iniciar o jogo, é mostrado o Main Menu representado de seguida (figura 2) que contém várias opções. Para navegar por todos os menus do jogo deve ser **pressionado o botão de lado esquerdo do rato.**
O Main Menu tem 3 botões :

- Play
- High Scores
- Exit :(- Sai do jogo

Figura 2

É também de notar que no menu inicial está a data e a hora actual como é possível ver na figura 2, estando essa informação presente em vários menus.

Game Type Menu

Se o user optar por Play no menu inicial, é-lhe agora mostrado mais um menu com 3 opções (figura 3):

- Solo - user “contra o PC”
- Multi - modo multijogador cooperativo (para dois jogadores) através de Serial Port (Descrito abaixo)
- Back - volta para o Main Menu

Figura 3

Aquando da seleção de Solo, o user vai poder escolher entre 2 níveis (1 ou 2) ou botão back para voltar ao Menu anterior (figura 4).

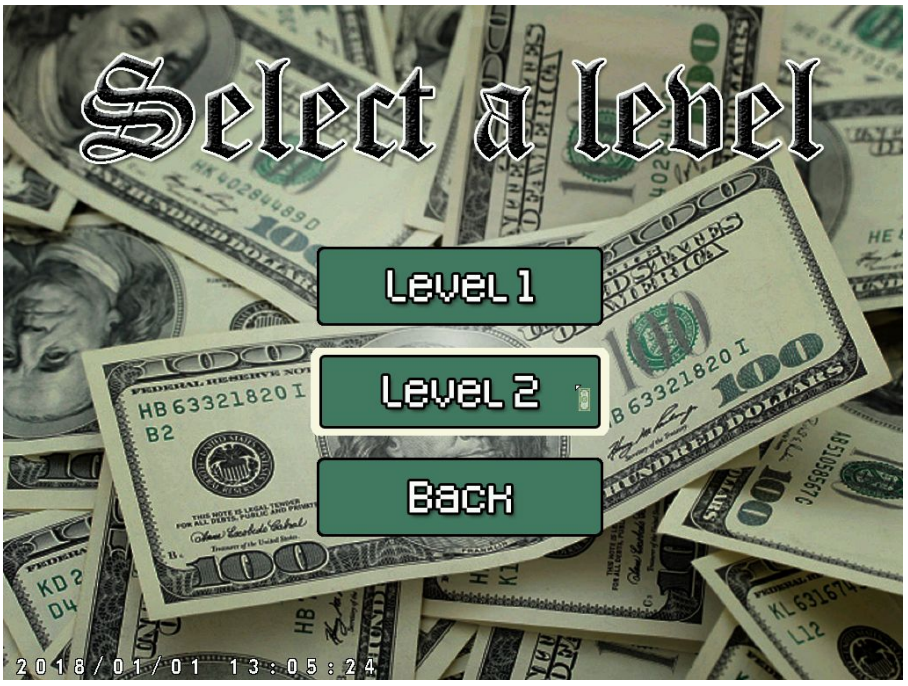


Figura 4

High Scores

Neste menu (figura 5), o user tem acesso à consulta dos 5 (ou menos, se não existirem 5) melhores resultados obtidos a jogar Robinix, ordenado por quantidade de pontos obtidos. Existe também a opção back que regressa ao menu anterior.

Pause Menu

Menu apresentado após premir a tecla ‘ESC’ enquanto se está a jogar. O utilizador pode escolher retomar o jogo (ao pressionar ‘ESC’ novamente) ou voltar ao menu principal (pressionando a tecla ‘m’). (Figura 6)

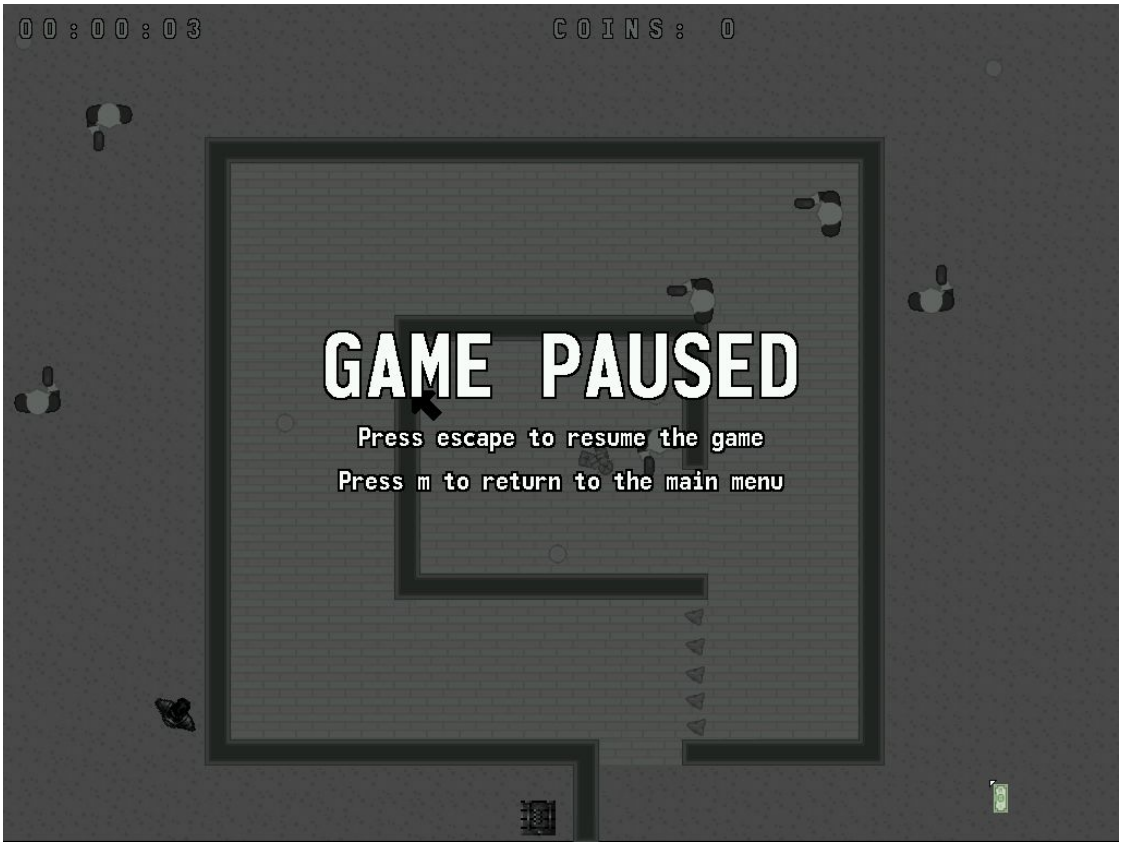


Figura 6

Score Submit Screen

Informa o utilizador, após o término com sucesso de um nível (quer em modo multijogador ou em “utilizador vs computador”), da sua pontuação e do tempo que demorou a completar o nível, informando-o também se foi o novo high score registado. Quer tenha sido ou não, é neste menu que o utilizador pode inserir o nome que deseja que fique associado à pontuação conseguida, que será posteriormente gravada e eventualmente adicionada ao ficheiro de pontuações. (Figura 7)

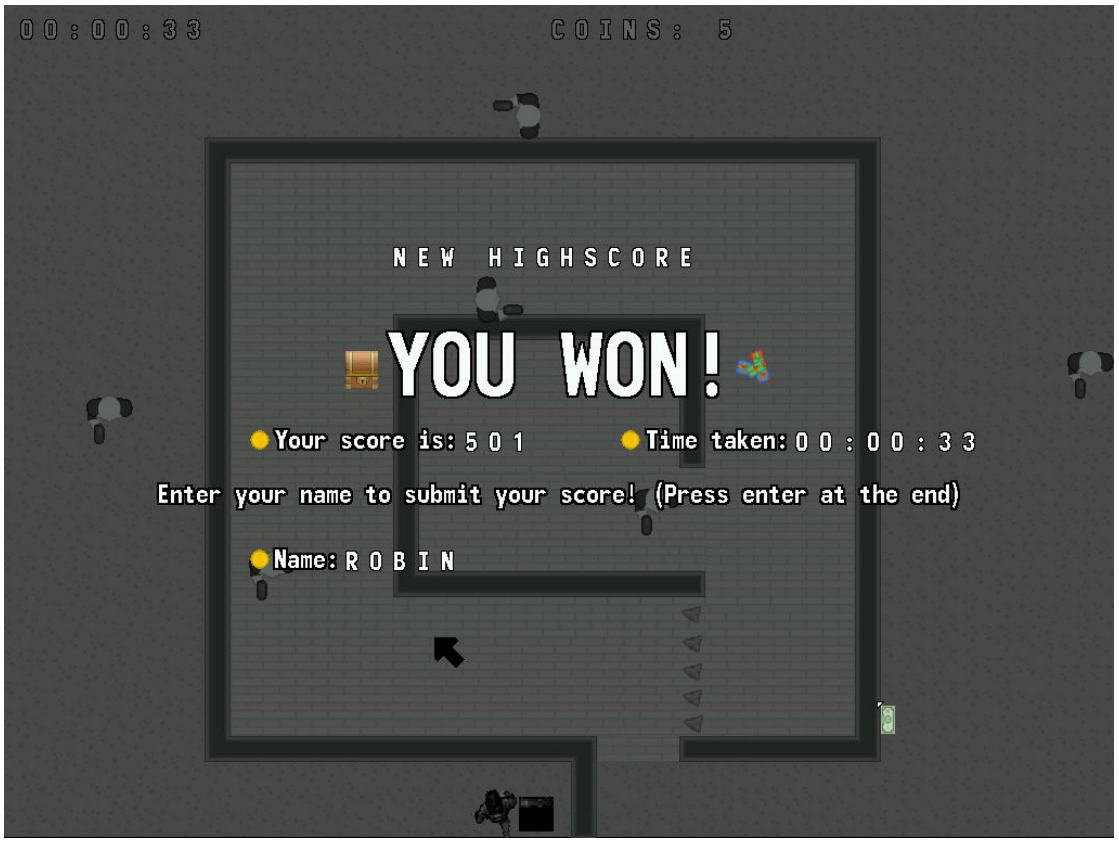


Figura 7

Losing Screen

Após colidir com algum guarda, quer em modo multijogador ou não, o utilizador é informado que perdeu, sendo apresentado este ecrã animado, no qual se pode pressionar a tecla “Escape” para regressar ao menu principal. (Figura 8)

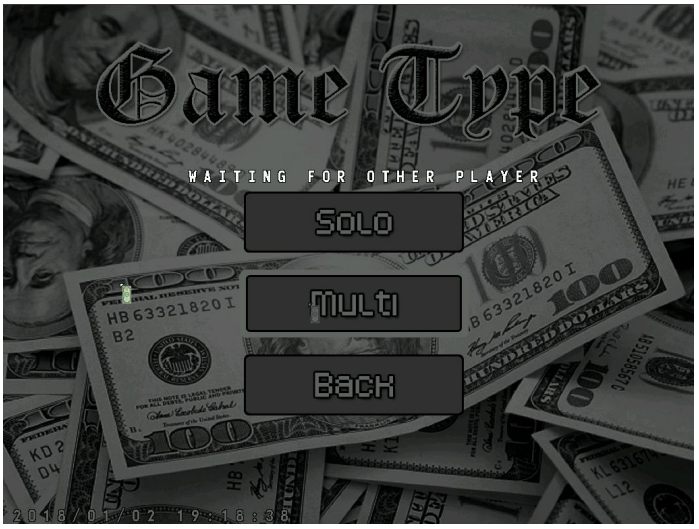
Figura 8

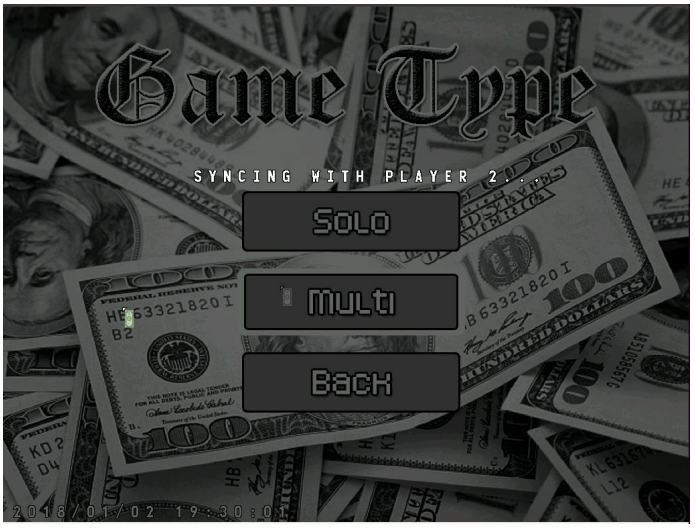
Menu Multiplayer

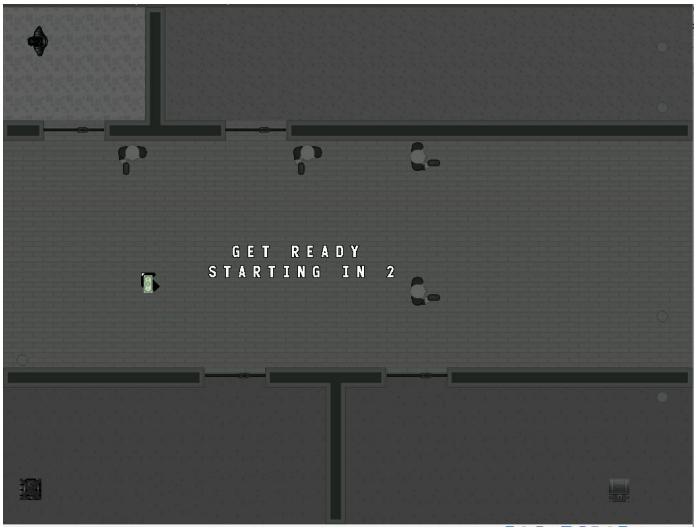
Abaixo apresentamos vários ecrãs relativos ao modo multijogador, no qual se entra após selecionar a opção “Multi” no menu “Game Type”.

Do canto superior esquerdo para a direita e depois para baixo, as imagens representam o ecrã de espera pelo outro jogador, o ecrã de sincronização com o outro jogador, o ecrã de “countdown” até entrar no jogo e o ecrã de jogo multiplayer, respetivamente.

Foram omitidos os ecrãs de derrota e vitória pois são iguais aos de singleplayer, apresentados acima.







Periféricos Implementados

Como tinha sido referido na proposta de projecto, o objetivo era implementar todos os periféricos dados, tendo sido esse objetivo concretizado.

Na tabela abaixo explicitamos de forma breve a sua utilização:

Periférico	Utilização	Interrupções
Timer	Sincronização entre computadores no modo multijogador, atualização do estado do jogo, atualização de animações, movimento dos guardas (movimento em caminho predefinido), verificação de colisões	Sim
Teclado	Interação com jogo e submissão de scores	Sim
Rato	Navegar entre menus e interação com objectos durante o jogo	Sim
Placa Gráfica	Visualização do jogo	Não se aplica
RTC	Consulta da Data em vários pontos do jogo, mostrando a data atual no menu principal, e associar às pontuações a data em que foram efetuadas	Sim, de update para manter um registo da data atual sempre atualizada
Serial Port	Modo de jogo multijogador cooperativo	Sim, também usando FIFO

Timer

O timer (interrupções) é utilizado para:

- Atualizar o estado de jogo
- Sincronização entre computadores no modo multi-jogador
- Atualização de movimentos dos guardas (através dos checkpoints)
- Animações
- Chamada a funções de deteção de colisões durante o jogo
- Chamada a funções de processamento de eventos baseado no estado atual

Implementação principal no módulo timer.c

Teclado

O teclado (interrupções) é utilizado para :

- movimento do player (‘W’ para cima, ‘A’ para a esquerda, ‘S’ para baixo e ‘D’ para a direita)
- interação com o menu de Pause (‘ESC’ para voltar ao jogo e ‘M’ para sair do jogo atual e voltar ao Main Menu)
- input do nome do high score após término de um nível com sucesso

Implementação principal no módulo keyboard.c

Rato

O rato (interrupções) é utilizado para :

- Navegar entre os vários menus presentes no jogo através da colisão da posição do rato com os Botões dos menus.
- Interação com objectos nos níveis (portas)
- Rotação da personagem de jogo (está sempre virada em direção ao rato)

Implementação principal no módulo mouse.c

Placa Gráfica

Modo: Optamos pelo modo 0x117 (com resolução 1024 x 768. Para as cores é utilizado o modo RGB 5:6:5 (65,536 cores)
Buffering: Foi implementado double buffering (para melhorar a fluidez gráfica no jogo)
Animações: Temos vários tipos de “animações”. Nos guardas, o bitmap desenhado altera consoante a direção do seu movimento. Quanto às animações na personagem, são alternadas entre 6 bitmaps de 8 em 8 ticks. De notar também a rotação do bitmap da personagem virada para onde o utilizador tem o rato não utilizando vários bitmaps para vários ângulos mas apenas um.
Colisões: Foram implementados dois tipos de colisões neste projecto. Colisão rato com um retângulo que verifica se as coordenadas do rato (x,y) estão dentro da área do botão(utilizado com os botões do menu). O outro tipo é colisão entre bitmaps que é pixel-perfect. Numa primeira fase, foi utilizado Axis-Alligned Bounding Box para detecção de colisões (AABB) e numa segunda fase, pixel perfect collision através da sobreposição de imagens pixel a pixel (utilizado para interações do rato com objectos nos níveis e colisões jogador-parede, jogador-guarda e jogador-limite do ecrã).
Escrita de texto no ecrã: Para escrevermos mensagens no ecrã usamos tanto fonts(algumas com números, ‘/’ e ‘:’ e outras com todas as letras do alfabeto), como imagens com texto para display em menus, etc.

Implementação principal no módulo video_gr.c, com utilização de funções do módulo vbe.c, e usado para o módulo bitmap.c

RTC

O Real Time Clock (RTC), através de interrupções de update, é utilizado para :

- Mostrar nos menus “pré-jogo” a data e hora atual (yyyy/mm/dd hh:mm:ss) (ver figura 9)
- Guardar numa pontuação a data da realização desta (informação mostrada nos highscores, se a pontuação for alta o suficiente, mas sempre guardada internamente e, quando fechado o jogo, num ficheiro de scores)
- Verificar se é Período Natalício. Se sim, a chave para abertura da porta de final de nível muda de um clássico baú para alguns presentes de natal. (ver figuras 10a e 10b)

Implementação principal no módulo rtc.c

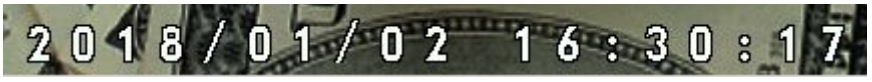


Figura 9

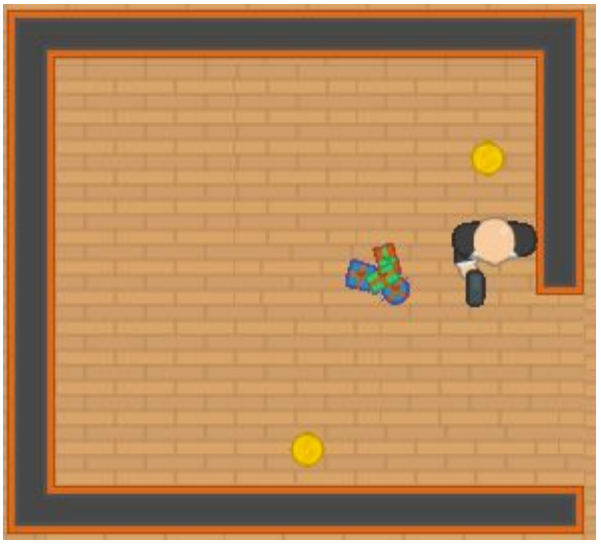


Figura 10a

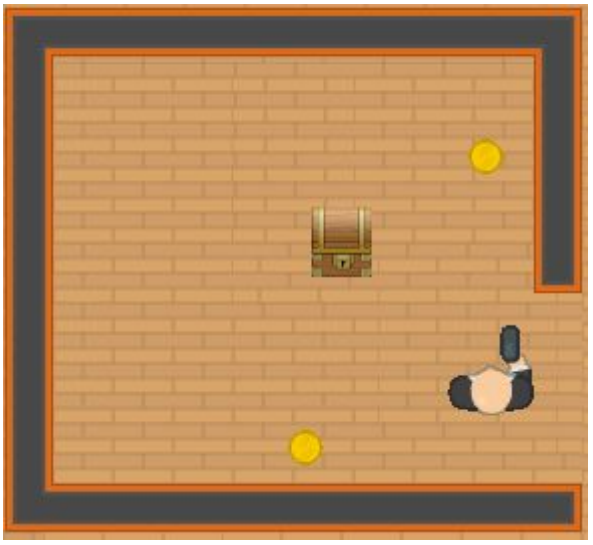


Figura 10b

Serial Port

A porta de Série é usada para o modo de multijogador cooperativo, no qual os dois jogadores têm de trabalhar em conjunto para completar o nível.
Neste projeto utilizamos porta de série com interrupções e FIFO ativado.
Ao selecionar a opção “Multi” no menu “Play”, podemos começar a pesquisar pelo outro jogador, que quando selecionar a mesma opção irá começar a sincronizar-se connosco.
Após esta fase de sincronização (que é feita sincronizando os ticks atuais dos relógios de cada máquina) com troca de informação, é combinado um tick entre as máquinas dos dois utilizadores para começar o jogo, dando-se um breve delay com uma contagem decrescente, de seguida iniciando-se o jogo.
O jogo em modo multijogador é semelhante ao jogo para apenas um jogador, mas desta vez para abrir a saída ambos os jogadores têm de apanhar o tesouro, e só conseguem ganhar após os dois jogadores terem chegado à saída. As moedas que cada jogador apanha são também contabilizadas para o outro jogador, pois os jogadores estão a jogar de forma cooperativa. Se qualquer um dos jogadores perde durante o jogo, ambos perdem.

Implementação principal no módulo uart.c, que apresenta também a implementação de uma “software queue”, para utilização conjunta com o FIFO de hardware da Porta de Série. Utilização principal com estabelecimento do protocolo de comunicação no módulo robinix.c

Estrutura e Organização do código

- bitmap.c

Neste módulo foi utilizado o código do aluno Henrique Ferrolho (obtido em http://difusal.blogspot.pt/2014/09/minixtutorial-8-loading-bmp-images.html, com permissão para utilização dada pelo Professor Souto). É com este módulo que desenhamos e "movemos" as imagens, atualizando a sua posição. Utilizamos tanto Gimp como Adobe PhotoShop para criar as imagens utilizadas.
Efetuamos várias alterações ao código de desenho de bitmaps referido acima, nomeadamente a criação de uma função que é capaz de ignorar uma certa cor, de modo a permitir transparência nos bitmaps. (No nosso caso foi usado o rosa mais vivo, #ff00ff ou 0xf81f em RGB 565).
Também foram criadas outras funções desenvolvidas por nós, como funções que permitem a rotação de Bitmaps consoante um ângulo especificado (utilizados no sprite do jogador, por exemplo), ou funções de colisão entre bitmaps.

Importância do módulo: 7%
Contribuição de cada membro: João: 20% Miguel: 80%

- checkpoint.c

Neste módulo foi implementada a struct Checkpoint, que tem como finalidade permitir aos guardas moverem-se pelo mapa do nível entre checkpoints. Inclui construtor e destrutor de "objetos" Checkpoint.

Importância do módulo: 2%
Contribuição de cada membro: João: 100% Miguel: 0%

- font.c

Neste módulo foi implementada uma função que, mediante a string de texto para mostrar no ecrã, string font respectiva (identificador do tipo de letra a utilizar) e coordenadas, mostra no ecrã a string pretendida.

Importância do módulo: 4%
Contribuição de cada membro: João: 70% Miguel: 30%

- game.c

Este módulo foi maioritariamente usado para efetuar testes às diferentes funções do jogo (daí até o nome anterior “tests.c”), tomando partido de passagem de argumentos à chamada “service run” para selecionar a funcionalidade desenvolvida. Entretanto, a maior parte das funções de teste foram removidas pois, quando a funcionalidade que estava a ser testada foi completamente desenvolvida e integrada no “projeto principal”, tornaram-se obsoletas. Apesar disto, teve um papel fundamental na fase de desenvolvimento.
É, de momento, usado para efetuar o início do jogo através da criação do objeto gestor de jogo Robinix, e entrar no ciclo de interrupções, chamando o handler respeitvo para cada interrupção, até que o estado do jogo seja o de saída, quando efetua chamadas aos destrutores necessários e às funções de “unsubscribe”.

Importância do módulo: 4%
Contribuição de cada membro: João: 50% Miguel: 50%

- gamestats.c

Módulo utilizado para guardar informação sobre o jogo atual para que, no final deste, quando o jogador tiver sucesso, seja calculada a pontuação e sejam mostradas essas estatísticas no ecrã.

Importância do módulo: 4%
Contribuição de cada membro: João: 80% Miguel: 20%

- guard.c

Módulo criado para conter a struct Guard e funções que a utilizam. Existem dois tipos de guardas, cíclicos e não cíclicos (Os cíclicos percorrem os checkpoints aos quais estão associados sempre no mesmo sentido, os não cíclicos invertem o sentido após chegar ao fim do seu percurso). Está neste módulo também implementado um enum constituinte da struct Guard denominado guard_direction_enum que aponta para onde o guarda está virado, o que é importante para escolher o bitmap de guarda a desenhar. Resumindo, este módulo contém construtor, destrutor, função para desenhar guarda e 2 funções auxiliares para ver se o guarda vai atingir um checkpoint no próximo tick (interrupt do timer) e tratar dessa situação se se proceder.

Importância do módulo: 5%
Contribuição de cada membro: João: 70% Miguel: 30%

- keyboard.c

Módulo importado das aulas práticas correspondentes ao lab3, com alterações para envio de eventos para o objeto gestor de jogo, Robinix.

Importância do módulo: 4%
Contribuição de cada membro: João: 50% Miguel: 50%

- level.c

Módulo criado para a mais fácil, eficiente e correta organização e estruturação de um nível no jogo. É aqui que se encontram funções de criação, destruição e de desenho no ecrã tanto de um nível como dos componentes de um nível (moedas, saída, portas).
Contém funções de criação de níveis (dois de singleplayer e dois de multiplayer) que criam níveis baseado em definições internas à função. Possui também uma função que faz a ligação entre a seleção de nível e estas funções, criando o nível desejado.
Durante o jogo, é responsável por efetuar o desenho, update (movimento dos guardas e deteção de colisões, por exemplo) e envio de alguns eventos relativos ao jogo para o objeto gestor de jogo Robinix.

Importância do módulo: 4%
Contribuição de cada membro: João: 50% Miguel: 50%

- main.c

Ponto de entrada no programa. Responsável por efetuar o processamento dos argumentos passados ao programa e chamar as funções respetivas. Tal como referido no módulo game.c, esta funcionalidade é menos relevante na iteração final do projeto mas foi fundamental durante o desenvolvimento deste.

Importância do módulo: 1%
Contribuição de cada membro: João: 50% Miguel: 50%

- menu.c

Módulo que contém as structs Menu e Button e funções relativas a este. Permite a geração de um menu interativo através do seu construtor, sendo possível dar como argumento um identificador inteiro do menu desejado. Possui funções de desenho e

atualização do menu (funções para verificar cliques, atualizar “mouse over”). Para uso exclusivo como membro do módulo menumanager.c
Importância do módulo: 3%
Contribuição de cada membro: João: 20% Miguel: 80%

- menumanager.c

Módulo onde se encontra implementado o objeto MenuManager que é responsável por gerir todos os menus do jogo (Objetos Menu referidos acima). Possui funções que gerem diferentes atividades do rato como carregar com o botão de lado esquerdo e dar “hover” num botão.

Importância do módulo: 4%
Contribuição de cada membro: João: 20% Miguel: 80%

- mouse.c

Módulo importado das aulas práticas correspondentes ao lab4 ao qual foi adicionado interface com o objeto gestor de jogo, Robinix, para envio de eventos relacionados com o rato (movimento e cliques).

Importância do módulo: 4%
Contribuição de cada membro: João: 49% Miguel: 51%

- player.c

Módulo que contém funções e estruturas relativas à personagem do jogo controlada pelo jogador. (Nomeadamente desenho, movimento e criação do jogador).

Importância do módulo: 5%
Contribuição de cada membro: João: 20% Miguel: 80%

- robinix.c

Módulo responsável pela principal execução do programa, agregando todos os módulos. Possui a principal implementação da máquina de estados e dos eventos que para esta são usados. Também é aqui feita a maior parte do processo de comunicação através da porta de série, definindo-se o “protocolo de comunicação”.

Importância do módulo: 18%
Contribuição de cada membro: João: 30% Miguel: 70%

- rtc.c

Módulo implementado com vista a assegurar os 3 usos que planeamos para o Real Time Clock (alteração de bitmaps aquando de período natalício, data e hora nos menus do jogo e a gravação de um score com a data respetiva num ficheiro de texto). Para isso utilizamos interrupts de update e uma variável privada (usando a keyword “static”) neste módulo denominada curr_date. Neste implentação, criamos várias funções : enable_interrupts(), disable_interrupts(), subscribe_rtc(), unsubscribe_rtc(), funções que transformam datas e retornam as strings correspondentes, função que mediante duas datas, retorna o número de segundos entre elas (extremamente útil para o cálculo do score final de um nível) e uma função getDate() para o retorno do membro privado.
Para além destas funcionalidades, este módulo contém também funções que analisam a configuração do RTC e transformam a informação lida deste (data_bcd_to_binary(), por exemplo).

Importância do módulo: 4%
Contribuição de cada membro: João: 80% Miguel: 20%

- score.c

Aquando da conclusão do nível com sucesso, é criado um objecto Score com a data respectiva a essa score e o número total de pontos obtidos pelo utilizador. É útil para posteriormente guardar informação de scores no programa e no ficheiro de texto.

Importância do módulo: 4%
Contribuição de cada membro: João: 60% Miguel: 40%

- scoremanager.c

Objeto que gere os Scores de um jogo. No início do jogo, são lidos do ficheiro todos os scores lá contidos e posteriormente organizados (através de bubble sort, ordenados por valor de pontuação). É neste módulo em que se encontram as funções de escrita e leitura do ficheiro. Quando uma pontuação é adicionada, o array de Scores é reordenado, mantendo-se se sempre ordenado.

Importância do módulo: 5%
Contribuição de cada membro: João: 60% Miguel: 40%

- sprite.c

Módulo responsável pela animação de imagens, utilizando para isso vários bitmaps diferentes, entre os quais alterna. O efeito do uso deste módulo é visível na animação da saída, assim que aberta, e também no jogador, quando este se movimenta, por exemplo.

Importância do módulo: 3%
Contribuição de cada membro: João: 50% Miguel: 50%

- timer.c

Módulo importado das aulas práticas correspondentes ao lab2. É responsável por efetuar chamadas periódicas a funções de update do estado do jogo, desenho e processamento de eventos, por exemplo.

Importância do módulo: 3%
Contribuição de cada membro: João: 50% Miguel: 50%

- uart.c

Implementação base do driver para a porta de série. A nossa implementação utiliza interrupts e FIFO, tendo também 2 queues implementadas em software para lidar com o I/O da porta de série (receive_queue e send_queue). Estas queues foram implementadas tendo como base um array de chars pois é o tipo de informação que enviamos e recebemos por cada interação com a porta de série.

Importância do módulo: 5%
Contribuição de cada membro: João: 20% Miguel: 80%

- utilities.c

Funções auxiliares genericamente úteis para o projecto.

Importância do módulo: 1%
Contribuição de cada membro: João: 50% Miguel: 50%

- vbe.c

Módulo importado das aulas práticas correspondentes ao lab5, possuindo chamadas a funções VBE utilizadas no nosso projeto.

Importância do módulo: 1%

- video_gr.c

Módulo importado das aulas práticas correspondentes ao lab5, com algumas adições, tais como a utilização de um “back buffer” (técnica de double buffering) ou técnicas de modificação de imagem como transformar uma imagem em preto e branco e alterar a sua luminosidade (funções usadas para o menu de pausa, por exemplo, para melhor efeito visual).

Importância do módulo: 4%
Contribuição de cada membro: João: 45% Miguel: 55%

- video_utils.c

Funções e constantes úteis para a programação do módulo gráfico (video_gr.c).

Importância do módulo: 1%
Contribuição de cada membro: João: 40% Miguel: 60%

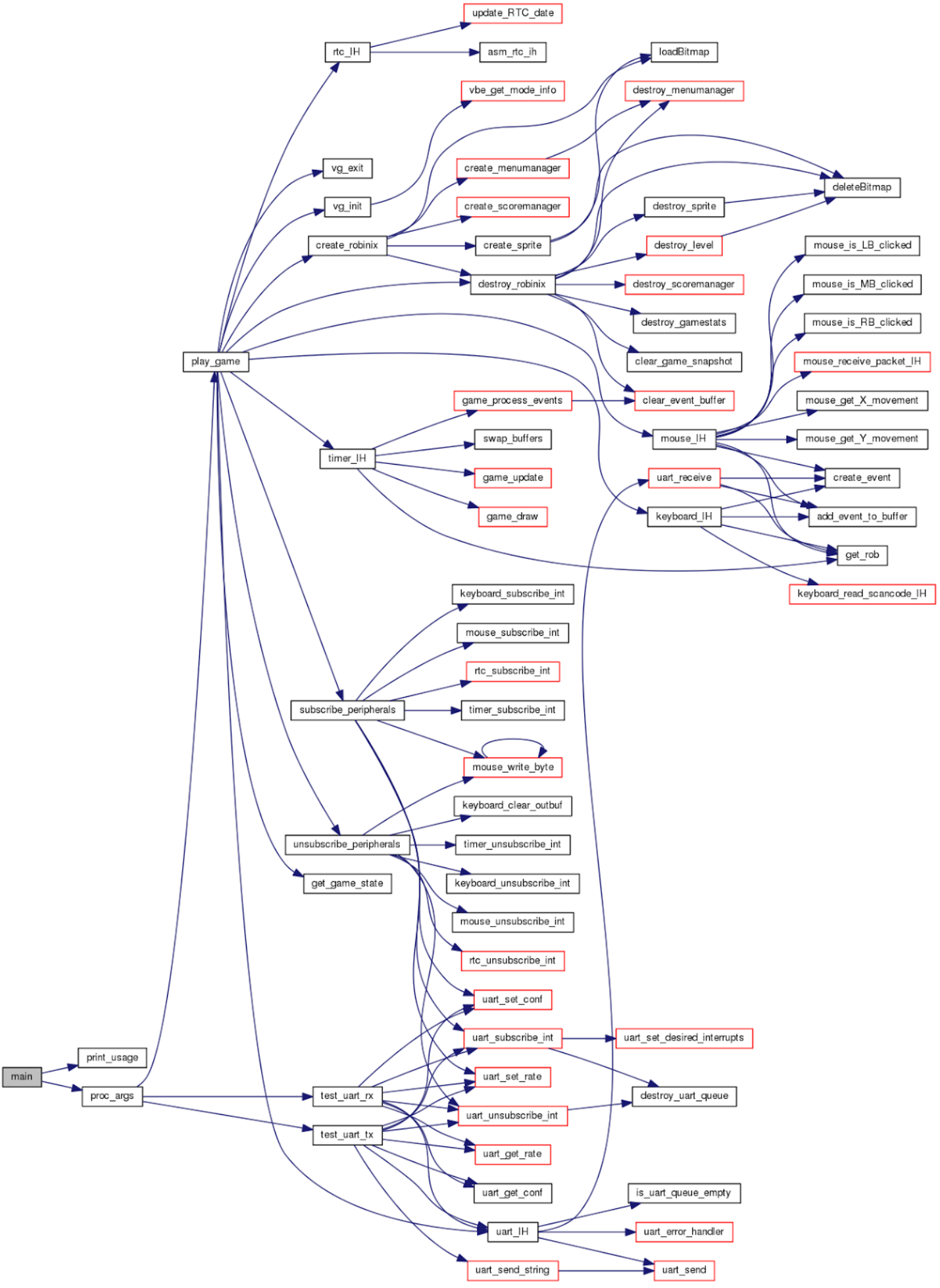
- Módulos Assembly

Foram implementados os interrupt handlers de vários periféricos em Assembly. Como estes módulos não são 100% independentes (são chamados a partir do módulo “pai” respetivo) não achamos relevante incluir cotações percentuais relativas a eles, por estarem assim já incorporados noutros módulos.
Os módulos são os seguintes (também existe o módulo para o teclado que não é referido aqui porque já foi avaliado no lab respetivo):

- mouse_assembly.S
- rtc_assembly.S

Function Call Diagram

O diagrama apresentado é o de chamadas de funções a partir do main().
Apresenta-se truncado, tendo as funções a vermelho chamadas a funções não representadas aqui. Porém, essas funções não chamam praticamente nenhumas outras e não achamos, portanto (também devido à já grande extensão do diagrama) relevante incluir também as suas chamadas.
Nota: A razão pela qual os construtores de objetos (funções create_X) aparecem por várias vezes como chamando o seu próprio destrutor (funções destroy_X) é porque, em caso de erro na criação de um dos membros do objeto, é chamado o destrutor para desalocar os restantes membros, não deixando assim “memory leaks” no programa, e apenas posteriormente retornando com estado de erro.


update_RTC_date
rtc_IH
asm_rtc_ih
loadBitmap
vbe_get_mode_info
destroy_menumanager
vg_exit
create_menumanager
deleteBitmap
vg_init
create_scoremanager
destroy_sprite
create_robinix
create_sprite
destroy_level
mouse_is_LB_clicked
destroy_robinix
destroy_scoremanager
mouse_is_MB_clicked
destroy_gamestats
mouse_is_RB_clicked
play_game
clear_game_snapshot
mouse_receive_packet_IH
game_process_events
clear_event_buffer
mouse_get_X_movement
mouse_IH
swap_buffers
mouse_get_Y_movement
timer_IH
uart_receive
create_event
game_update
keyboard_IH
add_event_to_buffer
game_draw
get_rob
keyboard_read_scancode_IH
keyboard_subscribe_int
mouse_subscribe_int
rtc_subscribe_int
subscribe_peripherals
timer_subscribe_int
mouse_write_byte
keyboard_clear_outbuf
unsubscribe_peripherals
timer_unsubscribe_int
get_game_state
keyboard_unsubscribe_int
mouse_unsubscribe_int
rtc_unsubscribe_int
uart_set_conf
uart_subscribe_int
uart_set_desired_interrupts
print_usage
main
uart_set_rate
proc_args
test_uart_rx
destroy_uart_queue
uart_unsubscribe_int
test_uart_tx
uart_get_rate
uart_get_conf
is_uart_queue_empty
uart_IH
uart_error_handler
uart_send_string
uart_send

Detalhes de implementação

Para o nosso projeto, consideramos bastante útil por motivos de organização efetuar uma implementação de uma máquina de estados, tendo formas de desenho, update e processamento de eventos diferente para cada estado. Achamos que esta capacidade de abstrair o nosso projeto em estados e eventos foi fulcral para o seu desenvolvimento de forma organizada, juntamente com a elevada modularidade que procuramos implementar.

No decorrer deste projeto, utilizamos programação orientada a objetos recorrendo a structs e pointers para estas, incluindo construtores e destrutores para facilitar gestão da memória do programa e para evitar “memory leaks”. Também foi uma boa abordagem a tomar pois aproximou-nos de algo com o qual já estamos mais habituados a lidar, nomeadamente em C++, permitindo mais fácil raciocínio e maior organização.

Também foram implementados alguns detalhes bastante interessantes como efeitos visuais aplicados a uma cópia da memória gráfica para usar como menu de pausa - efeitos como tornar a imagem em preto e branco e escurecê-la, através de alteração das componentes de cor de cada pixel por um fator especificado.
Outros detalhes visuais bastante interessantes na nossa implementação foram a criação de abstrações que permitiram a utilização de animações usando várias frames consecutivas (ver Lose Screen, animação da porta de saída, animações do jogador); e também a capacidade de rotação de um bitmap sem recurso a imagens previamente rodadas, utilizando cálculo matricial para esta rotação (rotação implementada no módulo bitmap.c).

As colisões usadas, na sua maior parte têm 2 fases, para permitir maior eficiência mas ainda preservando bastante precisão: Uma primeira fase aplica a técnica AABB (Axis-Aligned Bounding Box) que interseta os retângulos externos das imagens. Porém, como para o nosso jogo - devido às imagens usadas por causa da rotação, nomeadamente - a precisão com esse modo não era satisfatória, desenvolvemos um outro modo mais preciso que efetua colisões com precisão de pixel. Este método desenha o maior dos bitmaps (por ser mais eficiente esta parte do que a seguinte) num buffer temporário e procede a tentar “desenhar” o outro bitmap na posição onde seria colocado, efetuando comparações pixel a pixel para saber se é possível o desenho. Se algum pixel se sobrepuser com uma parte não “transparente” (ver próxima parte) da imagem (ocupada pelo outro bitmap), sabemos que houve colisão. Como esta técnica é bastante dispendiosa, apesar de extremamente precisa, apenas a aplicamos após a primeira fase de colisão, usando

o método AABB, como previamente descrito, obtendo assim precisão extremamente boa sem grande impacto na “performance” do projeto.

Como referido em parte anteriormente, foi também bastante importante implementar “transparência” no desenho dos nossos bitmaps - Definimos uma cor a ignorar e usamo-la como fundo para as nossas imagens, permitindo assim ter um efeito de transparência no nosso projeto. Isto também foi bastante importante para a nossa implementação de colisão “pixel-perfect” descrita acima pois, de outra forma, o método supostamente mais preciso iria ter a mesma precisão que o método AABB também previamente descrito, pois interessaria apenas a parte mais externa das imagens.

Também é relevante referir o bastante robusto protocolo de comunicação implementado (usando Serial Port), que envolve conseguir sincronismo ao nível do segundo entre as duas máquinas aquando do início do jogo. Também é mantida comunicação durante o jogo para trocar mensagens como entrar no “Lose State” se um dos utilizadores perder, abortar se um dos jogadores der “force quit” ou ocorrer algum erro, incrementar a contagem de moedas, ou conseguir sucesso apenas quando ambos os jogadores chegam à saída.

Instruções de instalação

No repositório, dentro da pasta do projeto são disponibilizados alguns scripts para ajudar na instalação, compilação e execução do projeto.
Para tal, de modo a conseguir correr o projeto corretamente, devido à utilização de alguns ficheiros, deve ser executado o script “install.sh” como superuser, de modo a copiar os “resources” do projeto para uma localização “neutra”, de modo a que o programa os saiba usar.
De seguida, deve ser executado o script “compile.sh” que compila o source code presente na pasta src usando o makefile lá disponibilizado e move o executável um nível acima para poder ser usado com o próximo script: “run.sh”.
Este último script serve para executar o programa, permitindo a passagem de argumentos para este (semelhante ao efetuado durante os labs). Correndo o script sem passagem de argumentos mostra-nos as chamadas possíveis ao executável.
Para a execução principal do projeto deve ser efetuada a chamada ao script “run.sh” com o argumento “play”. (Ou seja, “sh run.sh play” ou “./run.sh play” se o script estiver marcado como executável).

Conclusão

Em geral, pensamos que este projeto tenha sido um culminar satisfatório desta unidade curricular que, apesar de trabalhosa, transmite conceitos essenciais para compreensão do funcionamento interno de um computador, dando boas bases de programação em baixo nível.

Por vezes, os conceitos eram confusos, principalmente no início, mas após compreender os conceitos base (as chamadas ao sistema sys_inb e sys_outb são fundamentais) é uma questão de compreender a organização do periférico em si porque de resto são apenas chamadas ao sistema usadas várias vezes.

Porém, achamos que alguns aspetos da cadeira poderiam ser melhorados. Compreendemos a razão da utilização do sistema operativo específico numa versão específica. No entanto, isto gera dificuldades na pesquisa de soluções para problemas que tenhamos, que por vezes podem não estar especificados na documentação e são, de outra forma, difíceis de encontrar devido à sua especificidade e, por vezes, idade (utilização de chamadas “deprecated” em C, por exemplo, por não haver outras alternativas, presentes em versões mais recentes).

Por último, achamos que a exigência da cadeira não corresponde aos créditos que a unidade curricular tem. Isto visto que existe uma correlação directa entre os créditos de uma cadeira e o tempo dispendido a estudar/trabalhar nela, sendo então que a cadeira deveria ter uma maior valorização em termos de créditos.