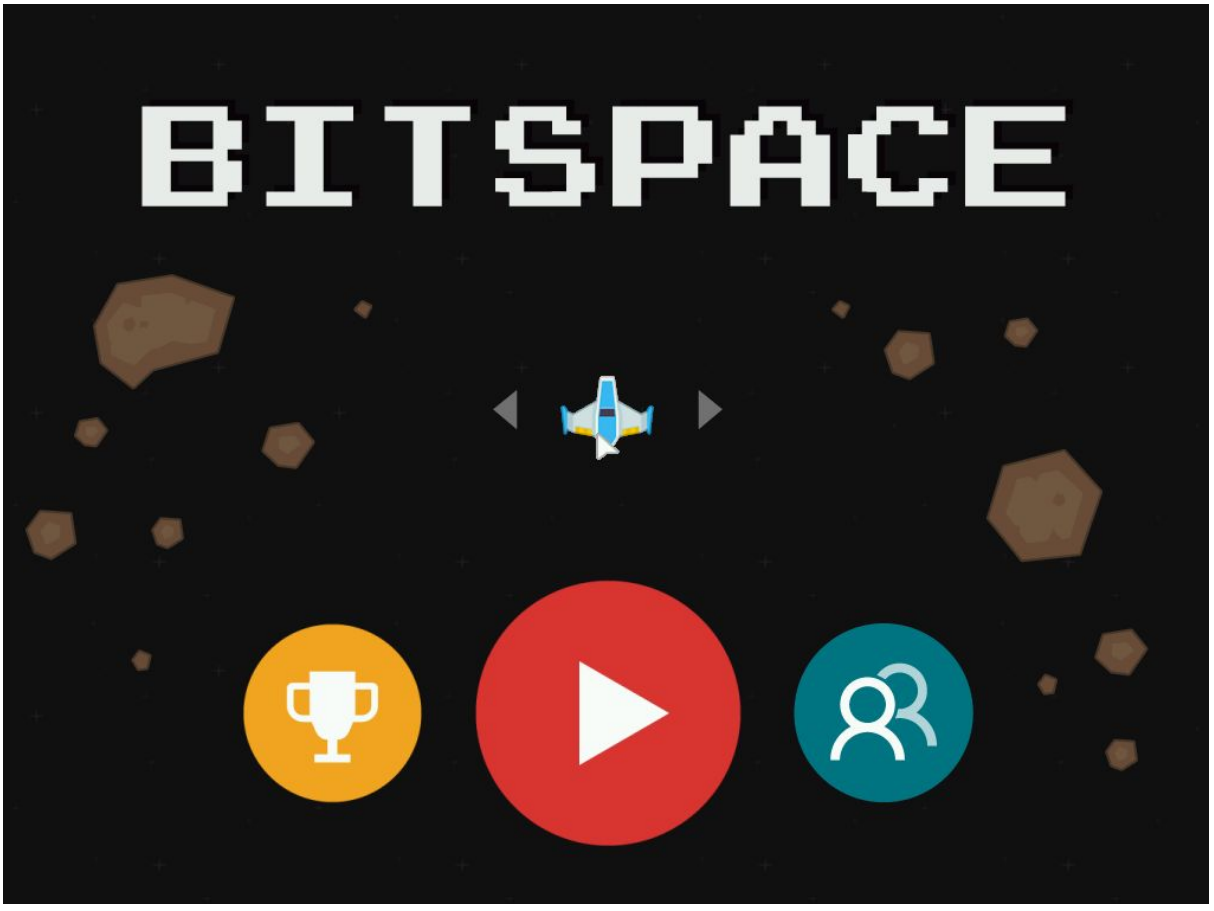
U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

BITSPACE

Relatório do projeto de Laboratório de Computadores



Turma 4 Grupo 3

César Medeiros - up201605344@fe.up.pt
Duarte Frazão - up201605658@fe.up.pt

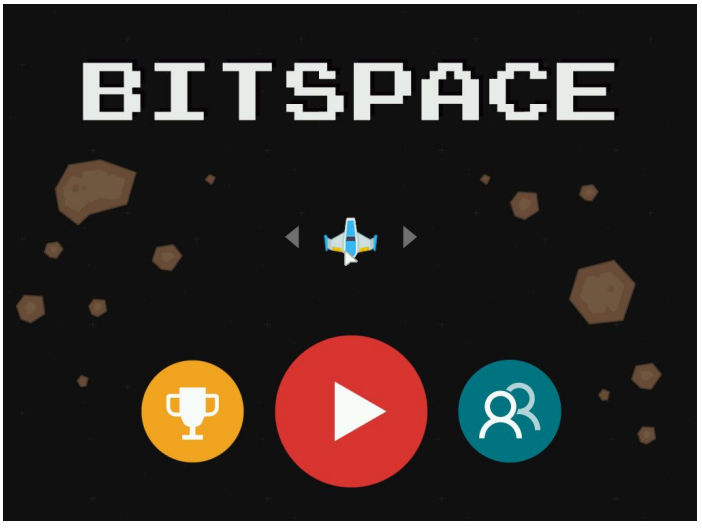
1. Instruções de utilização

Bitspace é um jogo passado no espaço em que os jogadores têm de tentar aguentar o tempo máximo que conseguirem sem serem atingidos por meteoritos, desviando-se ou destruindo-os com tiros laser, podendo adquirir mais naves para usar nas próximas rondas.

Menu inicial

No menu inicial há as seguintes opções:

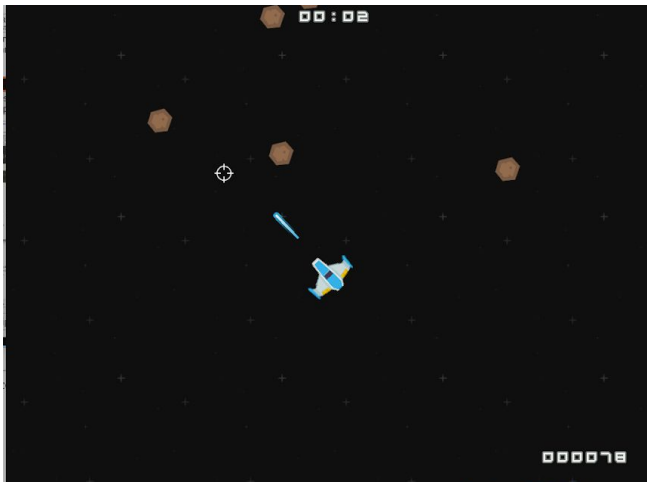
. Iniciar modo singleplayer ou multiplayer
. Mostrar LeaderBoards onde estão disponiveis os recordes anteriores
. Trocar de nave depois de naves novas terem sido desbloqueadas



SinglePlayer

Após a entrada inicial da nave começam a aparecer asteróides que o jogador tem de destruir ou dos quais tem de se desviar.

No canto inferior direito aparece a contagem do tempo, usada para a atribuição da pontuação.



Multiplayer

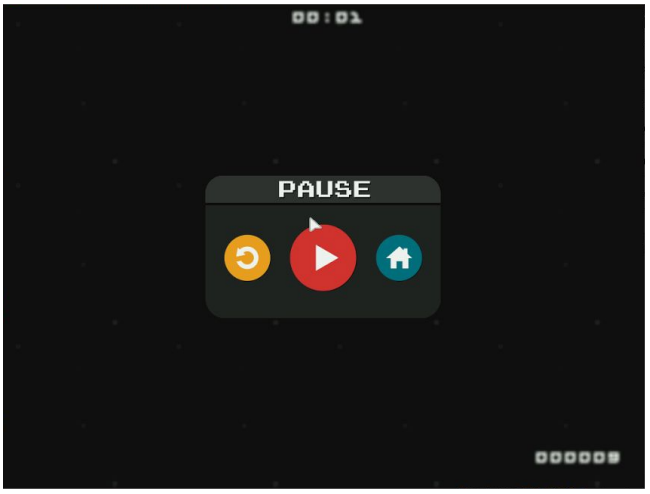
Este modo de jogo permite jogar em duas máquinas diferentes, em cooperação.

Para entrar neste modo ambos os jogadores selecionam a opção de multiplayer no menu que começará a sincronização entre as duas máquinas, quando os dois tiverem selecionado o jogo começa, se for necessário cancelar a sincronização basta clicar uma vez no esc, voltando para o menu inicial.

Pausa

Durante o jogo é possível pausar o jogo, deixando o ecrã ofuscado e entrado num menu de pausa onde podemos:

. Recomeçar o jogo
. Voltar para o menu inicial
. Continuar a jogar.



Game-Over

Quando uma nave atingir um asteroide dá-se a animação final, a nave explode, o ecrã ofusca e aparece o game over.

Controlo da nave

Movimento - Teclas ASDW
Rotação - Movimento do rato
Disparar tiros na direção da mira - Botão esquerdo do rato

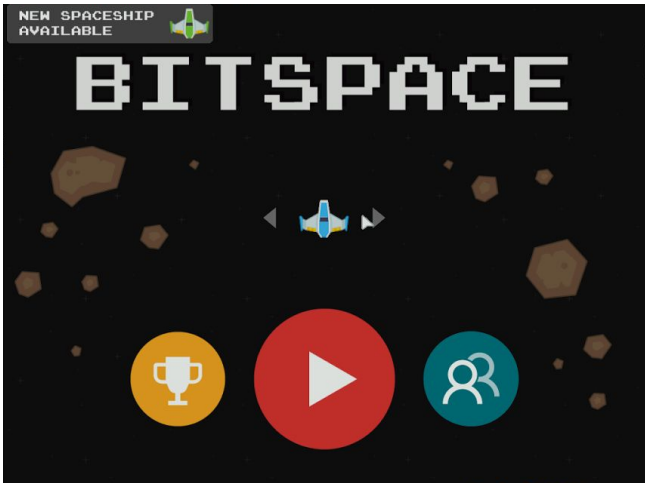
LeaderBoards

Nesta secção podemos ver anteriores resultados obtidos, com informação de data e pontuação obtida.

Podemos voltar para o menu inicial através do botão correspondente.

Notificação de nova nave disponível

Quando uma nova nave for desbloqueada, o jogador recebe uma notificação no canto superior esquerdo com a nova nave, podendo então escolhê-la para uma próxima ronda.



2. Estado do projeto

Perifério	Função	Interrupções
Timer 0	Atualização do jogo (frame-rate)	Sim
Teclado	Mudanças de estado e movimento da nave	Sim
Rato	Selecionar opções em menus, controlar rotação da nave, disparar tiros	Sim
Placa Gráfica	Parte gráfica do jogo	Não
RTC	Desbloquear naves e data para a leaderboard	Sim
Porta de série	Multiplayer	Sim

Timer

Responsável pela atualização do jogo, usando as interrupções geradas para dirigir o programa para a função que representa a máquina de estados responsável pelo jogo.

Foi criada a estrutura Timer para facilitar a interface com o timer.

Função principal onde é usado: Em BitSpace.c - void GameBrain(BitSpace *bitspace)

Teclado

Usado para através da leitura e identificação de teclas pressionadas para controlar o movimento da nave, entrar no menu de pausa e sair do jogo.

Foi criada a estrutura Keyboard para facilitar a interface com o teclado.

Funções principais onde é usado:

. Em SpaceShip.c - void updateSpaceShip(SpaceShip* spaceship)

. Em Keyboard.c - void keyboard_handler()

Rato

Usado para navegar pelos menus do jogo e selecionar opções.

Também serve para controlar a rotação da nave através da mira no jogo (manipulada com o movimento do rato) e disparar clicando no botão esquerdo do rato.

Foi usado uma fila para receber os packets, tornando o movimentos do rato (mira durante o jogo) mais fluido.

Foi criada a estrutura Mouse para facilitar a interface com o rato.

Funções principais onde é usado:

. Em Mouse.c - void updateMouse() e void packet_handler()

. Em SpaceShip.c - void updateSpaceShip(SpaceShip* spaceship)

Placa Gráfica

Responsável por toda a parte gráfica do jogo.

Escolhemos o modo 117h com resolução 1024 por 768 píxeis, permitindo-nos usar 64 mil cores no modo RGB(5:6:5).

Implementámos double buffering para obter um jogo mais fluido.

Desenvolvemos também um algoritmo de deteção de colisões com auxílio de um buffer auxiliar (cf. cap 4)

As fontes estão incluídas nas imagens onde é necessária a sua utilização.

São também usados sprites de dígitos que servem para apresentar informação mutável, como o tempo, score, entre outros.

O movimento dos sprites é definido pelos parâmetros velocidade e aceleração, no entanto, só a nave utiliza este último.

Para tornar mais evidente algumas imagens sobrepostas foi conveniente utilizar uma função que desfoca a imagem que estivesse a ser visualizada.

Funções principais onde é usada:

. Em BitSpace.c - ev_type_t Game(BitSpace *bitspace)

RTC

Foi criada a estrutura RTC para facilitar a interface com o rtc.

O RTC foi usado em duas situações distintas:

Poll da data e hora: No momento em que o jogador perde, os registos da data e hora são acedidos e o seu conteúdo é guardado numa string usada na leaderboard para apresentar juntamente com o score.

Interrupções do alarme: O alarme foi configurado para que às xx:xx:00 horas, ou seja, a cada minuto, seja lançada uma interrupção. Esta por sua vez indica que jogador se encontra a jogar há algum tempo e por isso deve ser recompensado desbloqueando novas funcionalidades, a saber, novas naves.

Funções principais onde é usado:

. Em RTC.c

Rtc* newRtc()
Rtc* enableRtc()
void rtc_asm_handler()
char * getDate()

. Em BitSpace.c - ev_type_t MainMenu(BitSpace *bitspace);

Porta de Série

A porta de série permite a comunicação entre duas máquinas para jogar em modo multiplayer, enviando essencialmente informação da posição, ângulo e tiros da nave.

É usada através de interrupções, com FIFOs ativos e recurso a duas filas, uma de receção e outra de transmissão, para conseguir uma implementação robusta e fiável.

Para começar o multiplayer temos um algoritmo de sincronização que garante que o jogo começa simultaneamente em ambas as máquinas (cf. cap 4).

A frequência de comunicação é extremamente alta, porque se tem de enviar obrigatoriamente em todos os frames a posição, ângulo e informação dos tiros, o que adicionou desafios à serial port. (cf. cap 4)

Foi usada a configuração para comunicação falado nas aulas teóricas: um header e um trailer para facilitar comunicação e o tratamento de erros (cf. cap 4).

A configuração usada foi:

COM Port: 1
Word Length: 8 bits
Number of stop bits: 1
Parity: Even
FIFO trigger level: 8 bytes
Interrupts subscritos: Transmitter Holding Register Empty, Received Data Available, Receiver Line Status
Bit-Rate: 115200 bps (máximo)

A configuração é feita na função newUart(), e a inicialização em enableUart() que trata de tudo o que é necessário para pôr a porta de série a funcionar num inicio de jogo pela ordem correta. (cf. cap 4)

Foi criada a estrutura Uart para facilitar a interface com a porta de série.

Funções principais onde é usado:

. Em Uart.c
int uart_position_handler()
Uart* newUart()
Uart* enableUart()
void uart_handler()
. Em SpaceShip.c - int sendPositionEvent(SpaceShip* spaceship)
.Em BitSpace.c - void synchronization(BitSpace* bitspace)

3. Estrutura do jogo e código

Módulos

BitSpace.c

Este módulo é o principal, gere todo o jogo através da state machine, ligando todos os periféricos.

Para ligar tudo desde os periféricos às naves e asteróides foi criada uma estrutura BitSpace, onde temos as estruturas relativas aos periféricos já referidas anteriormente, naves e asteróides, e outros membros que se foram demonstrando importantes, facilitando assim o desenvolvimento do jogo.

As principais funções são:

intGame - Funciona como interrupt handler geral

initGame - Inicializa o jogo, criando um objeto BitSpace e de todos os periféricos, subscrevendo-os

resetGame - Reset da informação que deve ser descartada dos objetos

endGame - Elimina objeto BitSpace, e também desubscreve interrupts e elimina outros objetos

GameBrain - Funciona como a state machine do jogo

Participação: César 50% / Duarte 50%

Peso: 15%

Bitmap.c

Este módulo trata da manipulação de bitmaps.
O código é da autoria de Henrique Ferrolho, que pode ser encontrado no seu blog
http://difusal.blogspot.pt/2014/09/minixtutorial-8-loading-bmp-images.html

Participação: : César 100%

Peso: 2%

Graphics.c

Contém funções que fazem interface com a placa gráfica, double buffering e ainda a função que ofusca o ecrã.

Participação: César 50% / Duarte 50%

Peso: 8%

Keyboard.c

Implementada a estrutura Keyboard.
Foram adaptadas as funções do lab3 para o projeto.

Contém funções que fazem interface com o objeto, atualizam os estados das teclas que interessam para o jogo, reset, atualizam, eliminam e criam o objeto teclado.

Participação: César 50% / Duarte 50%

Peso: 8%

Mouse.c

Implementada a estrutura Mouse.
Foram adaptadas as funções do lab4 para o projeto.
Foi usada uma fila de packets para tornar o movimento do rato fluido.
Contém funções que fazem interface com o objeto, nomeadamente fazendo reset à variação registada ou total, desenhando o rato consoante este esteja em jogo (mira) ou em menu, eliminando-o, criando-o e atualizando-o.

Participação: César 50% / Duarte 50%

Peso: 8%

Queue.c

Implementada a estrutura node e queue.
Fila de char, baseada numa lista simplesmente ligada, usada para o rato e serial port.
Contém funções úteis para trabalhar com queues, tamanho, primeiro e último valor, pop, push, limpar e eliminar.

Participação: Duarte 100%

Peso: 4%

RTC.c

Implementada a estrutura RTC.
Contém funções que fazem interface com o objeto, atualizam o estado das interrupções que interessam para o jogo, reset, atualizando, eliminando-o e criando o objeto RTC.

Participação: César 100%

Peso: 10%

Score.c

Implementada a estrutura Score que permite guardar o score atual da partida.

Contém funções que fazem interface com o objeto, tais como criar, atualizar, reset, e eliminar o objeto Score.

Participação: César 100%

Peso: 2%

SpaceShip.c

Implementada a estrutura SpaceShip, que contém uma flag de final de jogo (quando colide), ângulo, sprites usadas e tiros laser gerados pela nave.

Neste ficheiro temos as todas as funções relacionadas com a nave, tais como criar, atualizar, desenhar, eliminar, reset, sequências de início e fim de jogo, cálculo do ângulo e ainda o protocolo de envio de informação sobre a nave com recurso à serial port.

Participação: César 50% / Duarte 50%

Peso: 10%

Sprite.c

Implementada a estrutura sprite, que contém um pixmap, a sua posição x,y, a sua velocidade xspeed, yspeed e ainda a aceleração xace, yace.

Neste ficheiro também se encontram funções que desenham os sprites no ecrã e que testam a colisão entre dois sprites.

Participação: César 80% / Duarte 20%

Peso: 10%

Timer.c

Implementada a estrutura Timer, que contém o número de interrupções geradas pelo timer, uma flag de interrupt, hook id e IRQ line.

Foram adaptadas as funções do lab2 para o projeto.

Participação: César 50% / Duarte 50%

Peso: 4%

Uart.c

Implementada a estrutura Uart, que contem informação da configuração, filas usadas, IRQ line e hook id.

Aqui temos toda a interface do jogo com a porta de série, inicialização e configuração, transmissão e receção de informação através dos FIFOs e filas, manipulação das filas usadas, interrupt handler e um handler específico para comunicação recebida (esta trabalhando apenas na fila de receção), criar e eliminar o objeto.

Participação: Duarte 100%

Peso: 10%

Utilities.c

Contém a função int constrain() que permite restringir um valor a um certo intervalo. Usado para o cálculo da velocidade da nave.

Participação: César 100%

Peso: 1%

main.c

Responsável pela alocação de memória para a estrutura BitSpace.

Contém ainda o ciclo principal que chama a função que detecta as interrupções e a máquina de estados.

Participação: César 50% / Duarte 50%

Peso: 4%

Vbe.c

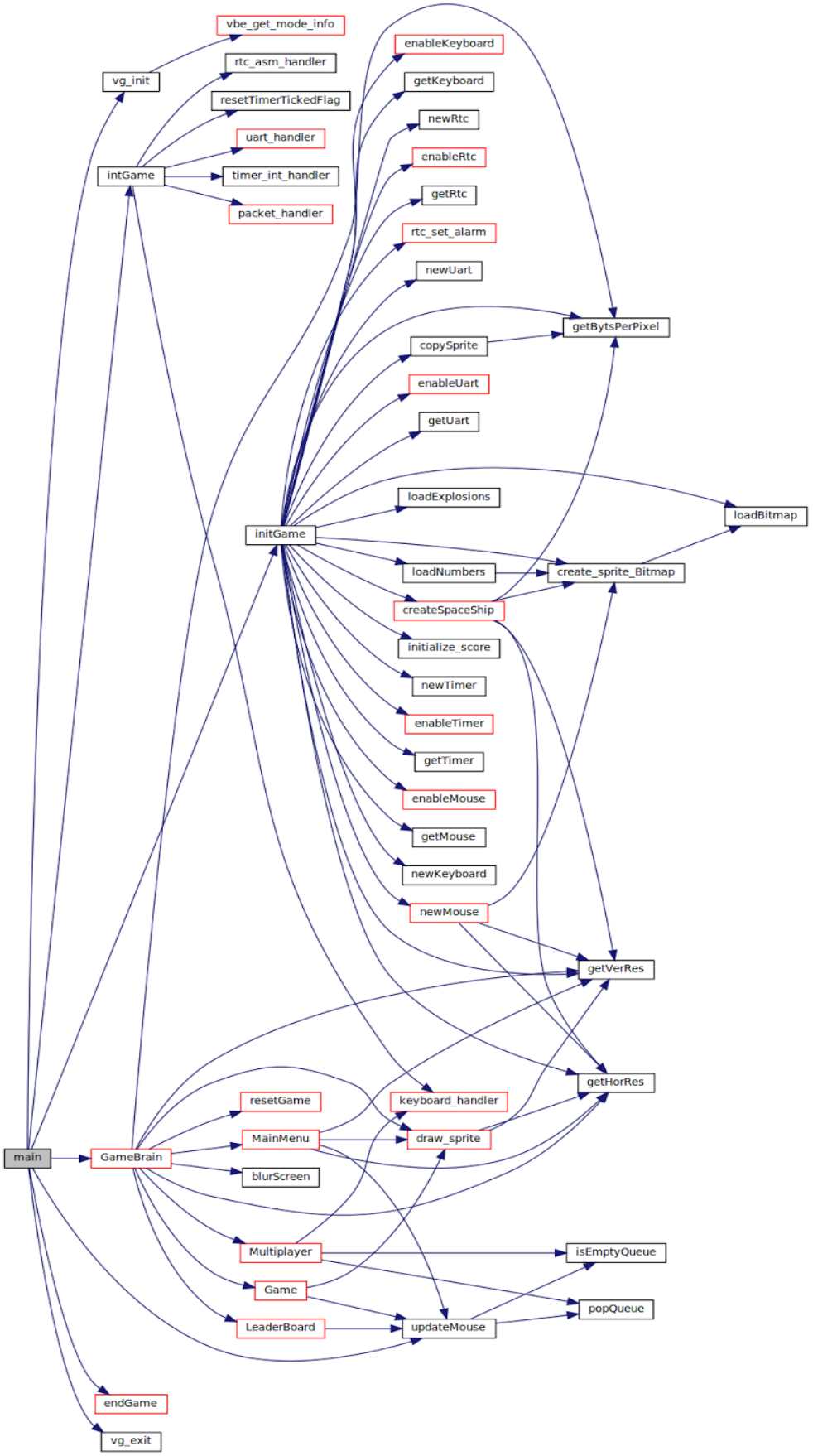
Implementada as estruturas vbe_mode_info_t e vbe_info_block_t que permitem obter informação sobre o modo gráfico. Destacam-se as informações fornecidas através dos membros XResolution, YResolution, BitsPerPixel, relativamente ao vbe_mode_info_t e VideoModePtr da estrutura vbe_info_block_t.

Participação: César 50% / Duarte 50%

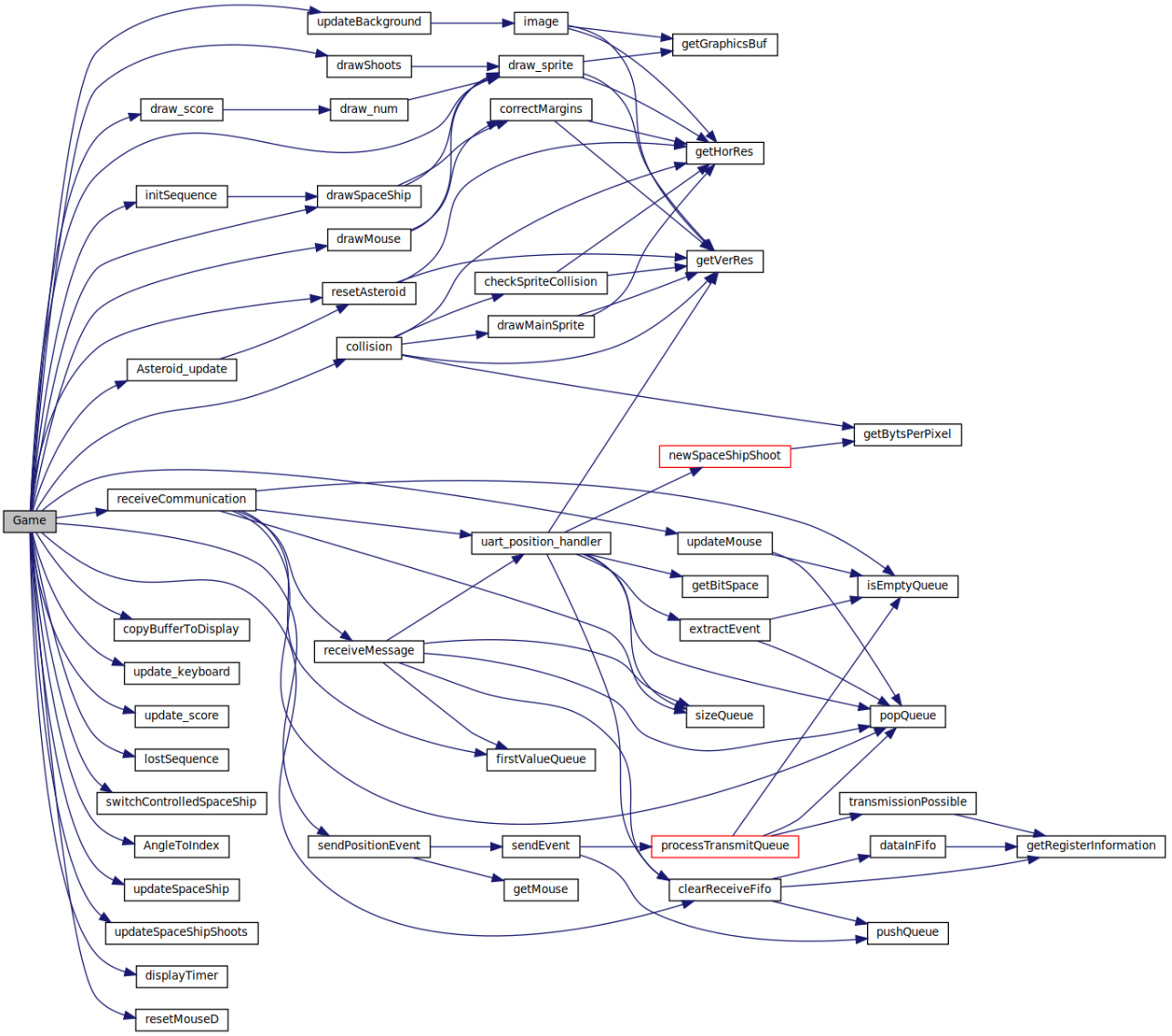
Peso: 4%

Function Call diagrams

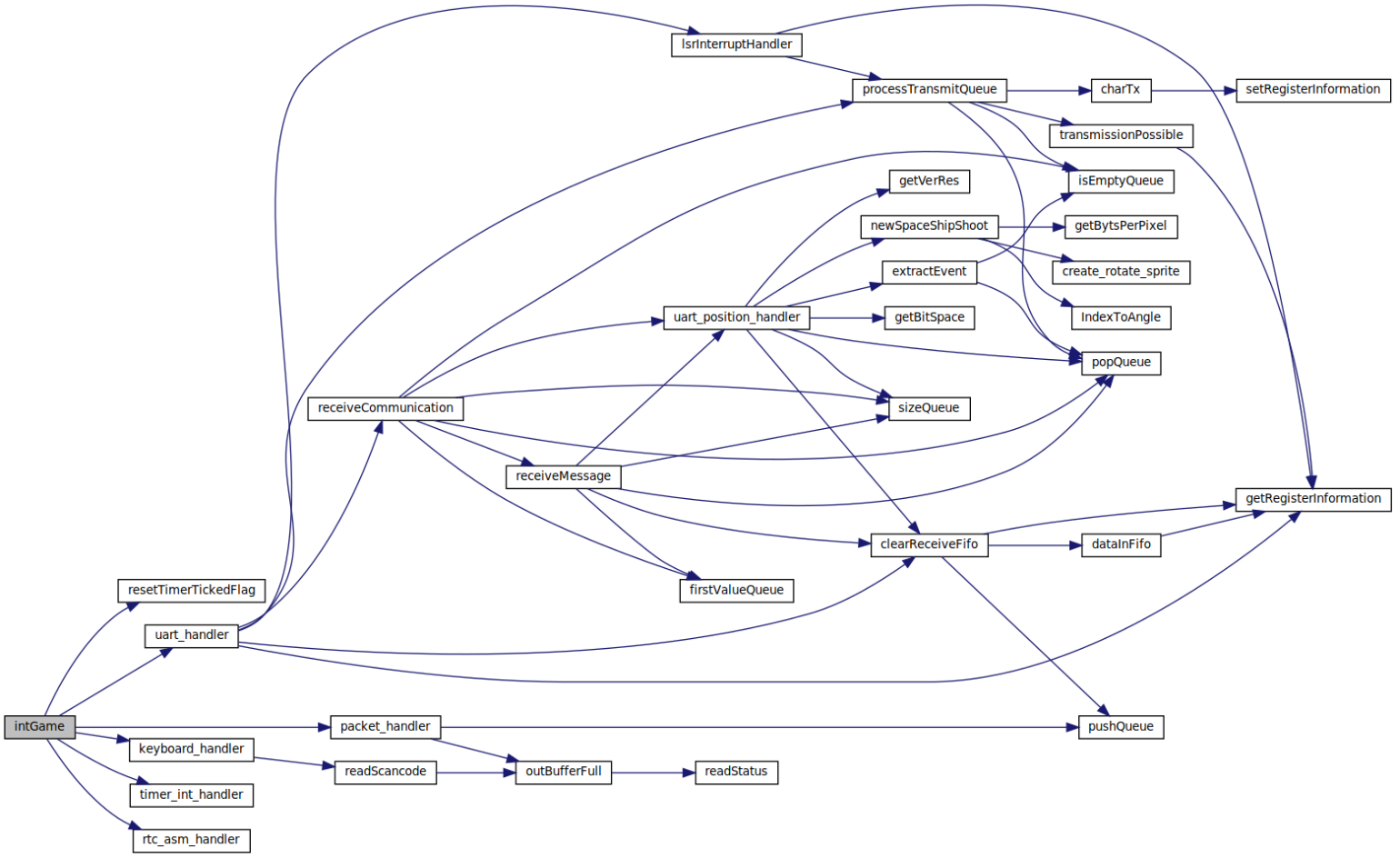
Inicialização do jogo (main)



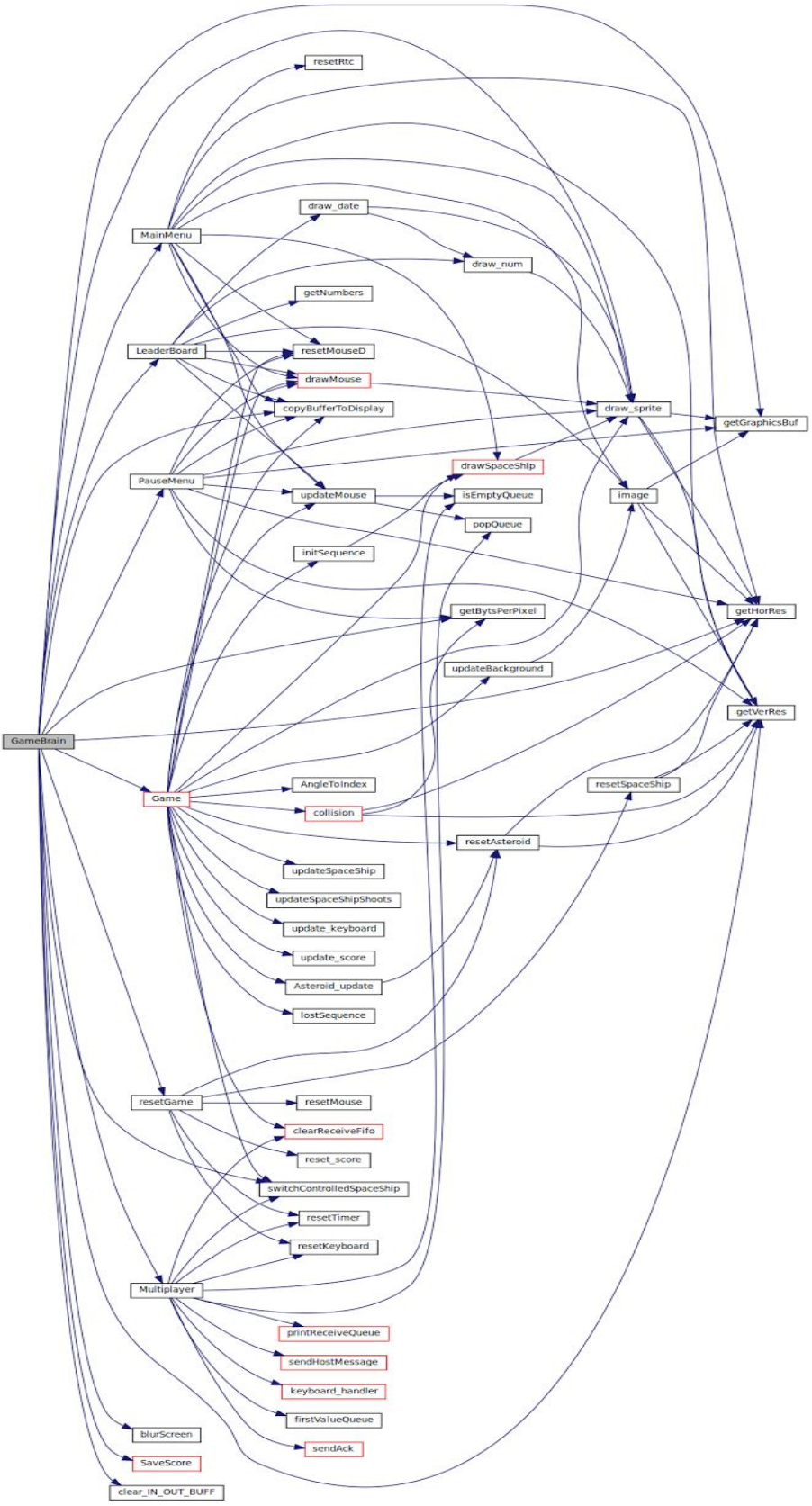
Mecânicas do jogo



Interrupções



Lógica do jogo



4. Desafios e detalhes de implementação

Rotação de uma imagem

Para conseguir que a nave pudesse disparar em todas as direções, foi necessário implementar um algoritmo para rodar imagens:

int create_rotate_sprite(Sprite *sp, float angle, unsigned short * picBack)

Para isso o algoritmo percorre cada pixel da imagem *picBack* e para cada pixel calcula a posição correspondente da imagem em *sp* usando o seguinte conceito:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

(https://en.wikipedia.org/wiki/Rotation_matrix)

Colisões

Para conseguir ter colisões precisas e eficientes criámos um algoritmo que detectasse colisões entre duas sprites:

int collision(Sprite *obj1, Sprite *obj2)

Primeiro são comparados os limites das duas sprites (os retângulos formados), tanto horizontalmente como verticalmente. Se estes não se intersectam não existe colisão, é feito este teste para otimizar o algoritmo, evitanto ter de passar pela segunda parte desnecessariamente, já que gasta mais recursos.

No caso de se intersetarem, entramos no algoritmo propriamente dito,
é criado um buffer auxiliar e pintado de preto onde é desenhado o primeiro sprite, depois vai-se fazer um processo semelhante ao de desenhar nesse buffer, sem realmente desenhar, mas apenas verificando se já havia alguma cor que não o preto ou a cor a ignorar (0x07E0) no lugar onde era suposto desenhar a segunda sprite; em caso afirmativo, significa que houve uma colisão entre esses dois objetos e termina o teste.

Orientação a objetos

Para uma interface simples e regular com o programa optámos por criar estruturas sempre que necessário:

BitSpace - Funciona como um objeto do jogo, reunindo todas os outros objetos do jogo.

Spaceship - Objeto da nave, onde guarda informação da posição e ângulo, tal como os tiros laser

Sprite - Contém a posição, velocidade e aceleração do bitmap

Rank - Para guardar a data correspondente às pontuações

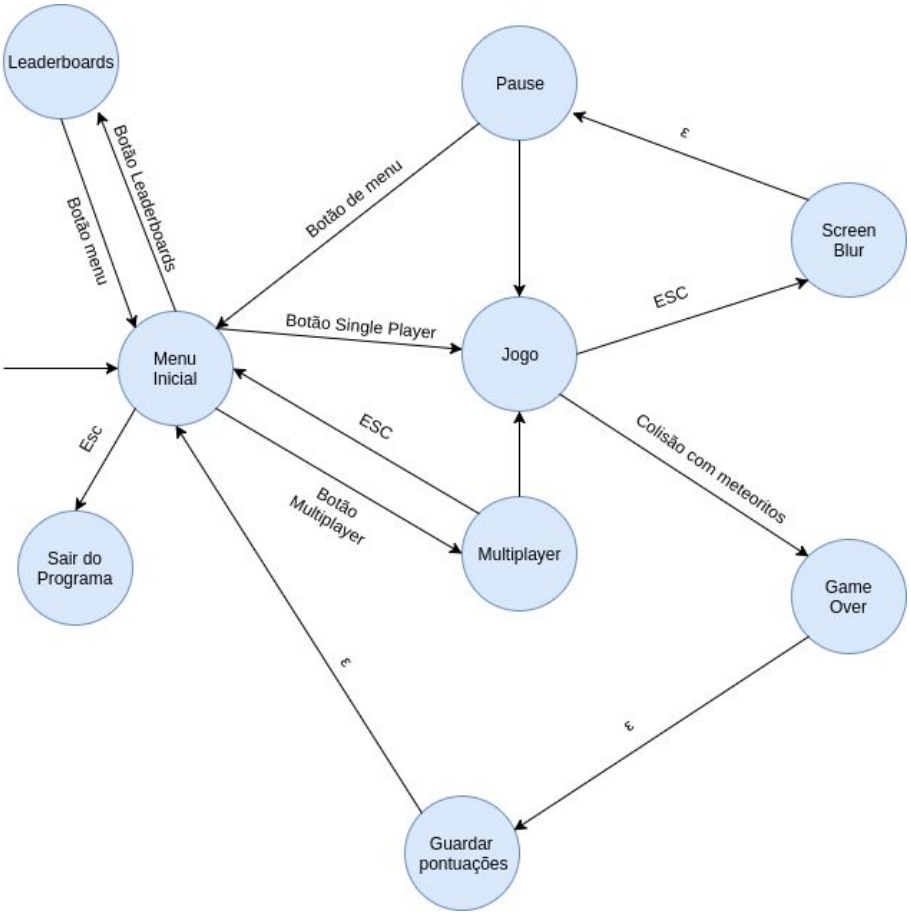
Timer, Keyboard, Mouse, Rtc, Uart para haver uma abstração dos periféricos

Máquina de estados e eventos

Para facilitar o desenvolvimento do jogo optámos por seguir o modelo de event driven design com uso de máquina de estados.

Estados e eventos declarados em BitSpace.h

GameBrain(BitSpace *bitspace) - Funciona como máquina de estados



RTC

O RTC apesar de ser um periférico simples de interagir apresenta algumas particularidades que se devem ter em conta para evitar o surgimento de alguns erros.

Neste projeto foi-nos útil usar poll da hora e da data e como os registos não são lidos em simultâneo era necessário garantir que a nenhum desses registos fosse modificado desde o ínicio da leitura. Para garantir isso os registos só são lidos se a flag UIP do registo A do RTC estiver low, ou seja, quando não está a ocorrer nenhuma atualização.
Ainda assim pode surgir outra situação. Se durante a leitura dos registos do rtc for atribuída outra tarefa ao cpu, quando o cpu retornar à tarefa de leitura os registos poderão ter valores diferentes do que teriam se a leitura fosse feita sem nenhuma interrupção.
Para isso desativam-se as todas as interrupções através da instrução “cli” e posteriormente reativam-se com a instrução ”sti”.

Código assembly

O código assembly foi usado na implementação do rtc. Na forma inline para desativar todas as interrupções “cli” e para reactiva-las “sti” e na forma “pura” para desenvolver o handler do rtc, que deteta se ocorreu alguma interrupção do alarme.

Porta de série

A porta de série foi o periférico onde de longe se gastou mais tempo, encontrando-se desafios e questões interessantes ao longo seu desenvolvimento e aplicação.
No modo multiplayer é enviada informação constante (todos os frames) sobre a posição, ângulo e tiros de cada nave.

Funcionalidades usadas

Como referido anteriormente estamos a usar FIFOs, é uma funcionalidade que dá duas grandes vantagens:

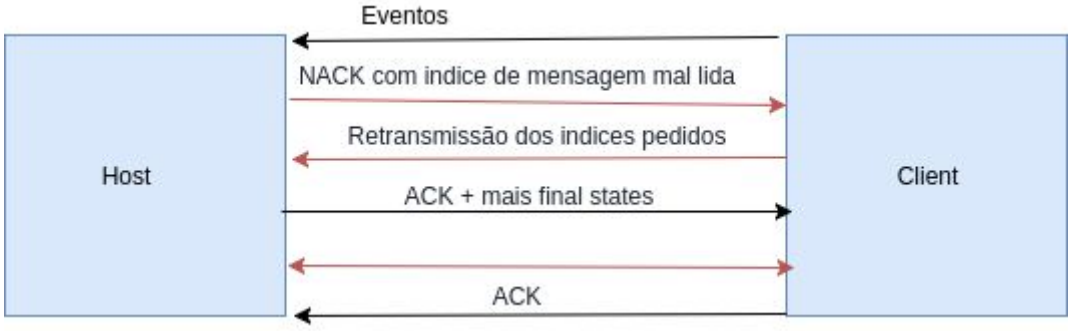
. Segurança - Por permitir ter mais espaço e tempo para receber e enviar informação, diminuindo também a probabilidade de acontecerem erros de overrun.

. Eficiência - Através do trigger level, permite gerar um menor overhead relativo ao envio de informação.

Além disso também optámos por implementar filas, para facilitar o envio e receção de informação, o que facilita em geral a comunicação com a porta de série.

Protocolo Inicial

A ideia inicial de protocolo de comunicação foi a seguinte:



As setas a vermelho só aconteceriam quando existissem erros na comunicação, enviando os índices das mensagens com erros (a terceira representa o mesmo que as duas primeiras mas em sentidos opostos)

Com a seguinte configuração de mensagem:

. Header: Indice de mensagem | Tamanho | Identificador de evento
. Body: Informação a enviar
. Trailer: '.'

Esta abordagem, apesar de teoricamente nos ter parecido uma boa ideia, ao aplicar no projeto não correu muito bem, principalmente porque implicava esperas de informação de ambos os lados e a nossa frequencia de comunicação era muito alta.

Protocolo Final

Com o desenvolvimento do projeto percebemos que demasiado controlo na informação estava a prejudicar a comunicação através da serial port, então optámos por simplificar o nosso protocolo.

Como o modo multiplayer que implementámos era de cooperação, passámos a comunicar apenas posição, ângulo e tiros nos dois sentidos e mais dois tipos de eventos, um de identificação de host e outro de terminar jogo. Deixamos de enviar acks e nacks.

Agrupámos a informação que é enviada todos os frames num tipo de mensagem que se mantém constante:

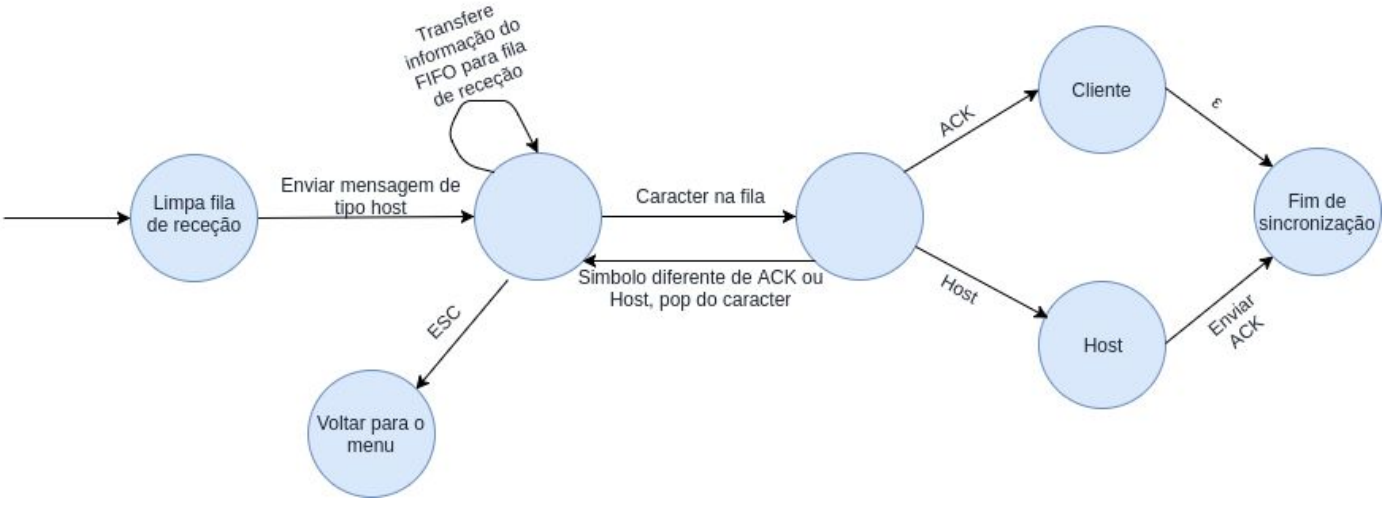
Header |	Posição x	Posição x	Posição y	Posição y	Angulo	Angulo	Flag de tiro	Trailer }

Desta forma conseguimos ter mais maneiras de testar erros de mensagem, otimizar o código para este tipo de mensagens de forma a conseguir obter um jogo fluido, apesar de estarmos a trabalhar com um bit-rate grande e com alta frequência de comunicação.

Sincronização

Para esta nova abordagem resultar é necessário que ambos os jogadores comecem o jogo ao mesmo tempo, já que enviar mais informações como posição de asteroides seria impensável, para isso criámos um algoritmo que realizasse a sincronização e determinasse quem seria o jogador 1 e 2, mantendo o mesmo código para as duas máquinas.

void synchronization(BitSpace* bitspace)



O algoritmo resume-se neste diagrama de estados.

Começa por limpar a fila toda e envia uma mensagem do tipo host. Neste momento pode parar a sincronização clicando no ESC. Enquanto a fila de receção estiver vazia vai atualizando-a até receber um caracter. Se este for o header de um ack (‘+’), essa máquina é um cliente, se for mensagem do tipo host (‘H’), é um host, se não for nenhum dos dois é um caracter que não importa e nesse caso descarta-o.

O objetivo da função é sincronizar e ao mesmo tempo conseguir ter o mesmo código nas duas máquinas. Como um lado vai clicar primeiro para sincronizar, este vai enviar uma mensagem de host que vai ser posteriormente eliminada, já que a segunda máquina começa também por limpar a fila de receção. Logo, quem vai receber efetivamente uma mensagem de host é aquele que clicar primeiro, o qual depois envia uma mensagem de ack, que sinaliza que quem o receber é o cliente (foi usado mensagem de ack e não criada uma própria por já haver código feito para o ack).

Ultimos pontos sobre porta de série

Apesar da porta de série ser o periférico mais exigente em relação aos detalhes, a principal dificuldade encontra-se em conseguir aplicá-la num jogo em que a frequência de comunicação é elevada, onde pequenos detalhes, muitas vezes extremamente difíceis de detetar, são essenciais para conseguir um jogo fluido.

Conclusões

Pontos positivos

. Muito desafiante
. Ter uma metodologia completamente diferente das outras cadeiras (Labs)
. A quantidade de conhecimento que obtemos e somos propostos a aprender

Pontos Negativos

. O início da cadeira é um grande choque, apesar de começar com o periférico mais fácil não se tem bem noção do que se está a fazer por não se ter tido muito contacto com a matéria. O facto de se poder ser avaliado apenas em 3 labs já ajuda para quem tiver dificuldades, no entanto seria benéfico para os alunos se fosse dado mais algum apoio extra para o primeiro lab.

. As avaliações dos labs saírem demasiado tarde, o que nos impede de aprender com os erros. Como reconhecemos que é difícil avaliar rapidamente os labs, uma alternativa sugerida seria rever na teórica o que devia ter sido feito no último lab, com aspetos e detalhes do que vamos ser avaliados, posteriormente a todas as turmas o terem realizado. Se isto se realizasse na aula teórica, evitar-se-ia que informação escrita passasse para anos seguintes, dando apenas uma ideia aos alunos do que deveria ter sido feito, o que seria muito benéfico porque os alunos já tiveram contacto prático com o periférico e podem sedimentar o conhecimento com pequenos detalhes teóricos que possam ter ficado mal aprendidos.

. Os créditos da cadeira não refletem o trabalho necessário, 6 créditos é muito pouco se comparados com os créditos atribuídos a outras cadeiras onde a quantidade de trabalho é, comparativamente, bastante mais reduzida.

. Na avaliação do lab3 pedimos reavaliação de duas questões e apesar de já termos reenviado mail, ainda não obtivemos resposta.

. Seria também benéfico para os alunos uma introdução a mecanismos essenciais de um projeto, algo que fomos porém capazes de suprir através do blog do Henrique Ferrolho.

Instruções de instalação

Para instalar o jogo primeiro deve entrar na pasta scripts, dentro de proj.

Dentro desta encontrará três scripts, que servem para evitar usar caminhos absolutos nas imagens e simplificar a instalação do jogo, assim deve executar as seguintes instruções dentro da pasta scripts.

. sh install.sh
. sh compile.sh
. sh run.sh

Isto permite uma instalação do jogo independente da pasta onde se encontra o projeto.