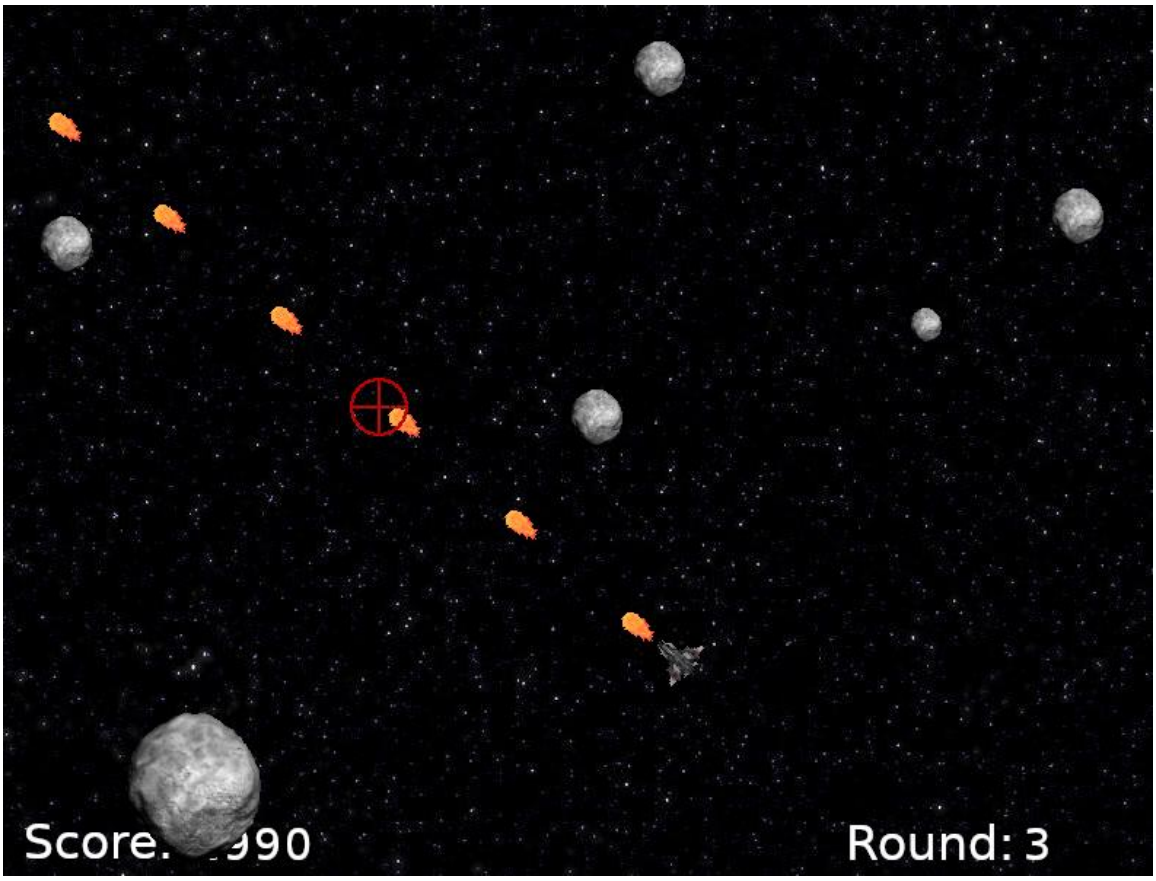
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Asteroids Survival



Laboratório de Computadores 2018/2019

2º ano – MIEIC

Turma 6 – Grupo 1

Gaspar Santos Pinheiro: up201704700@fe.up.pt

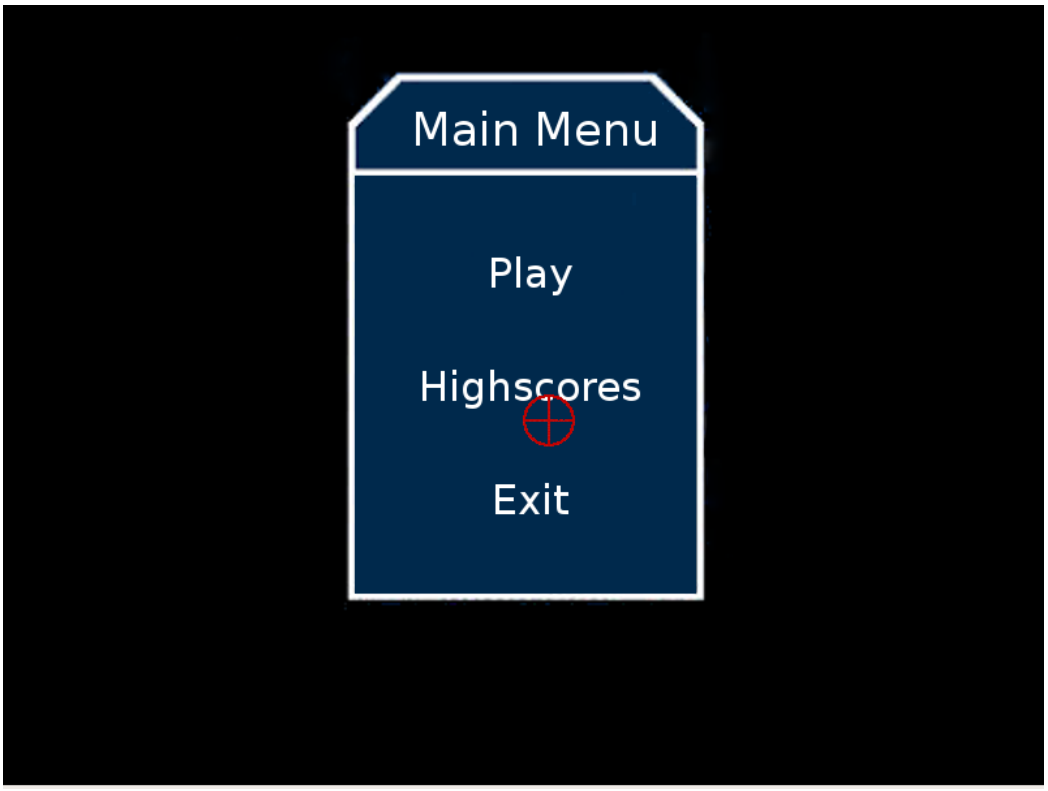
João Filipe Carvalho de Araújo: up201705577@fe.up.pt

Índice

1. Instruções de Uitilização

1.1) Main Menu

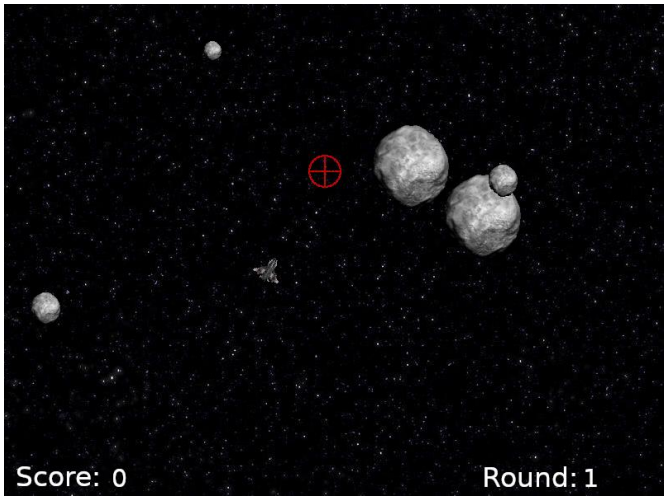
Quando o programa é iniciado, é mostrado ao utilizador o seguinte menu principal:



O utilizador pode usar tanto as teclas W, A, D, S como o rato para navegar este menu, podendo confirmar a tua opção com a tecla ENTER ou com o botão esquerdo do rato. O corportamento de cada opção é o seguinte:

- **Play:** Inicia um novo jogo;
- **HighScores:** Mostra uma tabela com os melhores resultados da sessão atual;
- **Exit:** Fecha o programa.

1.2) Play



Quando o jogo se inicia, o jogador passa a assumir o controlo de uma nave, podendo usar as teclas W, A, S, D para fazer a nave mover-se para cima, para a esquerda, para baixo e para a direita, respetivamente. A nave também pode ser rodada usando o rato, sendo que esta aponta sempre na direção atual do rato. A qualquer momento durante o jogo, o utilizador pode premir a tecla ESC para pausar jogo, sendo enviado para o main menu. Se voltar a escolher a opção play, o jogo começa exatamente onde o jogador o deixou.

Clicando o botão esquerdo do rato, é criada uma bola de fogo que se move na direção da posição em que o rato se econtra no momento em que o disparo é iniciado. A uma taxa fixa dependente da ronda atual, começam a aparecer asteroides no ecrã que se movem aleatóriamente, a uma velocidade também característica da ronda atual. O jogador pode destruir os asteroides causando colisões entre estes e as bolas de fogo, aumentando assim o seu score. Caso o asteróide seja de dimensão pequena é destruído. Se for de dimensão média ou grande, o asteróide é dividido em dois mais pequenos. Após todos os asteróides terem sido destruídos, a ronda avança, aumentado a dificuldade através de fatores como o número de asteróides por ronda, a sua velocidade, ou a sua frequência de geração.

1.3) Novo HighScore

Quando o jogador perde, isto é, ocorre uma colisão entre a nave e um dos asteroides, este pode ser levado para dois ecrãs diferentes. Caso o score atingido seja inferior ou igual ao 3 melhores registados, o jogador é levado para o menu de Game Over. Por outro lado, atingindo um novo HighScore, uma mensagem do sucedido aparece no ecrã, juntamente com uma outra pedindo o nome do jogador. Este pode então inserir o seu nome usando o teclado, podendo apagar a última letra escrita usando a tecla BACKSPACE. O máximo de caracteres para o nome do jogador é 12 e o nome do jogador pode incluir espaços. Quando estiver satisfeito com o nome inserido, pode então largar a tecla ENTER e será levado para o menu dos HighScores. Se desejar não ser inserido nos HighScores, pode pressionar a tecla ESC e será levado para o main menu.

1.4) Game Over

Sempre que o jogador perder e não atingir um novo HighScore, será enviado para este menu onde lhe é mostrado o score atingido. Pode pressionar a tecla ESC para voltar ao main menu ou ENTER para ir vêr s High Scores.

1.5) HighScores

Nesta opção, é mostrada uma lista com os 3 melhores resultados no jogo, mostrando para cada um, o nome do jogador, a ronda atingida, o score final e ainda a data e hora em que cada resultado foi atingido. Pode largar a pressionar ESC para voltar ao main menu ou ENTER para ir diretamente para o jogo.

2. Estado do Projeto

2.1) Dispositivos Usados

Dispositivo	Funcionalidade	Interrupções
Timer	Atualizar o estado do jogo (Atualizar o ecrã, criar asteróides, frequência de disparo).	Sim
Teclado	Navegar os menus, controlar a nave, escrita do nome, tecla ESC para pausar o jogo.	Sim
Rato	Navegar os menus, rodar a nave, disparar bolas de fogo e indicar a direção das suas trajétorias.	Sim
Placa Gráfica	Visualizar menus e jogo.	Não
RTC	Determinar a data e hora em que um novo highscore foi atingido.	Não

2.2) Timer

Usado para atualizar o jogo, ou seja, a cada interrupção do timer são chamadas as funções game_handler() e showFrame() (proj.c). A primeira é responsável pela alteração do estado do programa, se se encontra nos menus ou se o jogo está a decorrer. A segunda função, de forma resumida, realiza double buffering e, conforme o estado atual do jogo, mostra no ecrã o esperado. Por exemplo, se o jogo está a decorrer, então esta função será responsável por mostrar no ecrâ a nave e o rato, mover asteróides, bolas de fogo e criar novas, se o utilizador o indicar. Ou seja, o timer acaba por ser o responsável pela correta chamada das funções que fazem o jogo funcionar.

2.3) Teclado

O teclado é usado para navegar os menus, movendo para a opção de cima com a tecla W, para a de baixo com a tecla S e ENTER para confimar, sendo que a cada interrupção do rato é registada a tecla clicada (keyboard_manager() → devicesManager.c). Esta informação é usada pelo game_handler()→game.c para atualizar o estado do jogo. É também responsável pela navegação da nave, sendo que a informação da tecla utilizada é usada para mover a nave na direção correta (print_ship() → visuals.c). Serve também para registar o nome do utilizador, para guardar o seu highscore quando este é atingido.

2.4) Rato

Usado para navegar os menus. A cada interrupção do rato, a função mouse_manager() →devicesManager.c atualiza a posição do rato. Sendo que depois, em update_frame() → visuals.c, é mostrado no ecrã na posição calculada. No game_handler() (game.c), caso o jogo não esteja a decorrer, utiliza-se a posição atual do rato para saber em que opção do menu, este se encontra (which_menu()→game.c). Durante o decorrer do jogo, é utilizado para fazer rodar a nave (rotate_sprite()→rotation.c). Em print_ship() (visuals.c) é calculado o ângulo entre a posição atual da nave e do rato, usando depois um método de rotação de sprites, para obter a nave rodada de forma correta. Quando o botão esquerdo do rato é pressionado, forma-se uma bola de fogo que se move na direção do rato, sendo que a rotação correta da bola de fogo é calculada da mesma forma que a rotação da nave.

2.5) Placa Gráfica

Utilizada para visualizar o jogo. Foi usado o modo 0x115, cuja resolução é 800x600, tendo um máximo de 16 777 216 cores possíveis. Foi usada a técnica de “double buffering”, isto é, a cada interrupção do timer, é chamada uma função (showFrame() → visuals.c) que copia para a VRAM todo o conteudo do second buffer, que por sua vez, é novamente calculado (update_frame() → visuals.c) para o próximo frame. Foram usadas XPMs para as imagens. Sempre que o novo frame é calculado, as posições dos objetos no ecrã são atualizadas, dando a ideia de movimento. Temos duas funções de deteção de colisão, uma para calcular a mesma entre a nave e os asteroides e outra para calcular entre as bolas de fogo e os asteroides. Esta última (checkFireballCollision() → gamePlay.c), simplesmente

identifica a ocorrência de uma colisão caso os retângulos ocupados pelas sprites respetivas estejam sobrepostos. A outra (checkShipCollision() – gamePlay.c) representa os asteróides como círculos e a nave como um triângulo, obtendo-se uma colisão mais correta. A única função da VBE que utilizamos foi a vbe_get_mode_info() → (graphics.c), para inicializar o modo gráfico de forma correta.

2.6) Real Time Clock

Usado para calcular a data e hora, no momento em que o jogador colide com um asteróide. Em checkShipCollision()→gamePlay.c, quando se verifica a colisão, é chamada a função getCurrentDate()→rtc.c), que retorna a data e hora atual. Esta informação é guardada, sendo que no caso de o jogador atingir um novo highscore, esta informação é apresentada na opção HighScores do main menu.

3. Organização e Estrutura do Código

3.1) Proj

É a base do projeto, sendo responsável pela chamada da função que inicializa o jogo (startGame()), pelo “handling” das interrupções dos dispositivos e pela chamada da função que termina o jogo (exitGame()).

Peso: 4% Autor: Gaspar Pinheiro (100%)

3.2) Game

Módulo responsável pela inicialização e encerramento do jogo e pelo “handling” do estado atual do jogo. A struct Game contém toda a informação do jogo e está dividida em outras structs, que estão declaradas em game.h. Funções:

- **start_game():** Alloca toda a memória necessária para o jogo (incluindo o segundo buffer), inicializa o modo gráfico, faz a subscrição dos dispositivos nos quais são usadas interrupções, inicializa variáveis da struct game, como por exemplo, a posição inicial do rato e da nave, carrega as xpms de todas as imagens usadas no projeto.
- **reset_stats():** Função auxiliar de start_game() que também é chamada quando o jogador perde. Serve para inicializar as variáveis relacionadas com a dificuldade do jogo, assim, sempre que o jogador perder e decidir, de seguida, jogar novamente, a dificuldade inicial será sempre a mesma.
- **game_handler():** Esta função é chamada após a análise das interrupções dos dispositivos, de forma a manter o estado do jogo sempre atualizado. É uma state machine que permite a navegação dos menus usando o rato e o teclado.
- **which_menu():** Função auxiliar de game_handler(), que é chamada caso o utilizador se encontre a navegar os menus. Serve para identificar a posição atual do rato relativamente às diferentes opções, permitindo, por exemplo, fazer “highlight” da opção HighScores, quando o rato se encontra sobreposto nessa opção.
- **Exit_game():** Faz o unsubscribe dos dispositivos, liberta a memória alocada no inicio do jogo, e sai do modo gráfico.

Peso: 12% Autor: Gaspar Pinheiro (60%)

3.3) Timer

Este módulo foi importado do lab 2, sendo que as interrupções do timer são usadas para atualizar o ecrã (realizar animações).

Peso: 4% Autor: Gaspar Pinheiro (60%)

3.4) Devices Manager

Módulo responsável por lidar com o input do teclado e do rato. Funções:

- **keyboard_manager():** Analisa o scancode recebido do teclado (lido pelo "interrupt handler" do kbc - kbc_ih()), e guarda a informação obtida de forma a que seja mais facilmente interpretada por outras funções do projeto.
- **mouse_manager():** Chama o "interrupt handler" do rato, que lê os packets recebidos. Atualiza a posição do rato consoante a informação recebida e verifica se o botão esquerdo está a ser precionado, passando essa informação para a struct game.

Peso: 5% Autor: Gaspar Pinheiro (40%)

3.5) Mouse

Importado das aulas do lab4.

Peso: 5% Autor: 50%-50%

3.6) Keyboard

Importado das aulas do lab3.

Peso: 5% Autor: 50%-50%

3.7) Visuals

Módulo responsável pela atualização do ecrã consoante o estado atual do jogo e pela gestão do jogo enquanto este está a decorrer. Funções:

- **showFrame():** Copia para a VRAM o conteudo que está no segundo buffer e atualiza este ultimo para o próximo frame, chamando a função updateFrame();
- **updateFrame():** Trabalha no sefundo buffer. Identifica o estado atual do jogo e, conforme este, copia para o segundo buffer o necessário. Por exemplo, se o jogo se econtra no main menu, esta função mostra no ecrã a sprite correspondente ao main menu. No caso de o jogo estar a decorrer, esta função é responsável pela criação de asteroides, disparos, chamada de funções de colisões, etc.
- **Print_ship():** Função auxiliar de update_Frame() que calcula a nova posição da nave, roda a sua sprite conforme o angulo entre esta e o rato e mostra a nave no ecrã.
- **Print_mouse():** Função auxiliar de update_Frame() que verifica a posição do rato calculado previamente para que este não exceda os limites do ecrã. Mostra o rato no ecrã.

Peso: 10% Autor: Gaspar Pinheiro (70%)

3.8) Graphics

Importado das aulas do lab5, com algumas alterações:

- Todas as funções de desenho deixaram de copiar os pixeis diretamente para a VRAM e passram a copiar para o segundo buffer;
- A função print_sprite, antes de copiar o pixel, verifica se a cor deste é a cor que pretendemos ignorar, sendo que cada sprite é um retângulo com o conteudo essencial no seu interior rodeado por pixeis da cor que pretendemos ignorar (0xffc0cb);
- Nova função checkCoordinates(): Responsável por verificar as coordenadas dos asteroides e da nave, fazendo com que sempre que as posições recebidas estejam nos limites máximos do ecrã, estas são alteradas para o outro lado do ecrã. Assim, por exemplo, se um asteróide se mover para cima, ao chegar ao limite superior do ecrã,

passa diretamente para o limite inferior do ecrã, continuando a mover-se para cima.

Peso: 8% Autor: Gaspar Pinheiro (60%)

3.9) Rotation

Módulo responsável pela rotação de uma sprite. Baseado no seguinte video: https://www.youtube.com/watch?v=EnC1WKnN-H4&t=1595s, com algumas alterações, uma vez que estamos a usar o modo 0x115, com 24 bits por pixel. Esta função é usada na rotação da nave e das bolas de fogo, uma vez que estas não são circulares.

Peso: 4% Autor: Gaspar Pinheiro (100%)

3.10) Fireball

Módulo responsável pela criação, atualização das posições, "print" e destruição das bolas de fogo. Funções:

- **create_fireball():** Cria e adiciona uma nova fireball ao array de fireballs da struct game. Cada fireball é caracterizada pela sua posição e velocidades, nos eixos dos Xs e dos Ys.
- **move_fireballs():** Percorre o array de fireballs atualizando as suas posições, calculando uma nova sprite rodada na direção do rato, e fazendo o "print" dessa sprite rodada. A fireball é destruida assim que excede os limites do ecrã.
- **destroy_fireball():** Percorre o array de fireballs a partir daquela que se pretende eliminar, movendo cada fireball para a sua posição anterior e decrementa o número de fireballs que se econtram no ecrã.

Peso: 8% Autor: Gaspar Pinheiro (100%)

3.11) Asteroid

Módulo responsável pela criação, atualização das posições, “print”, separação e destruição de asteróides. Funções:

- **create_asteroid():** Cria um asteróide de tamanho aleatório, calcula a sua posição inicial, sendo esta também aleatória. O movimento do asteroide também é aleatório, sendo que é calculado um ângulo aleatório e a partir destes são geradas as velocidas nos eixos dos Xs e dos Ys.
- **move_asteroids():** Percorre o array de asteróides, verificando qual o tamanho de cada um e fazendo “print” da sprite correta. Atualiza as suas posições e chama a função checkCoordinates() para essas novas posições.
- **split_asteroid():** Função para separar um asteróide em dois mais pequenos. Caso o asteróide tenha o tamanho mais pequeno é destruído. Por outro lado, se o seu tamanho for médio ou grande, o asteróide original é atualizado e cria-se um novo asteróide. Os dois ficam com um tamanho mais pequeno que o orginal e as suas velocidades são perpendiculares à velocidade do original.
- **destroy_asteroid():** Percorre o array de asteróides a partir daquele que se pretende eliminar, movendo cada asteroide para a sua posição anterior e decrementa o número de asteróides que se encontram no ecrã.

Peso: 9% Autor: Gaspar Pinheiro (100%)

3.12) GamePlay

Módulo responsável pela deteção de colisões entre bolas de fogo e asteróides e entre a nave e asteróides. Verifica também a passagem de rondas, aumento a dificuldade a cada ronda. Funções:

- **detectCollision():** Faz a deteção de colisões entre duas sprites, usando apenas os seus retângulos. Ou seja, se as sprites que se econtram nas posições recebidas, com as alturas e larguras indicadas, se econtram sobrepostas, então a função devolve true.
- **checkFireballCollision():** Percorre o array de fireballs e, para cada uma, percorre o array de asteróides, verificando se há colisões entre eles. Em caso afirmativo, o score é aumentado, a fireball é destruída e o asteroide é dividido em dois (ou destruído, caso já esteja no tamanho pequeno).
- **checkShipCollision():** Percorre o array de asteróides e verifica se algum está a colidir com a nave. A deteção de colisão é feita considerando a nave como um triângulo (como 3 pontos) e o asteróide como um círculo (raio e centro). Calcula a distância entre cada um dos pontos da nave e o centro do círculo e caso alguma dessas distâncias seja menor que o raio (do asteróide), considera que houve colisão. Caso haja colisão, verifica se o score do jogador está entre os 3 melhores registados, sendo que o jogo vai para diferentes estados consoante o score do jogador esteja ente os 3 melhores registados. Em caso afirmativo, o estado passa a ser NEW_HIGHSCORE. Em caso negativo, o estado passa a ser GAME_OVER.
- **next_round():** Aumenta algumas estatísticas do jogo, como a ronda atual, o score e o número de asteróides destruídos, calcula a dificuldade para a nova ronda, sendo que a cada 5 rondas a velocidade com que os asteroides se movem no ecrã aumenta e para as outras mudanças de ronda, aumenta-se o numero de asteroides por ronda e a frequência com que estes aparecem no ecrâ.
- **end_round():** Reinicia a posição da nave, e destrói os asteroides que se possam encontrar no ecrã, tal como todas as bolas de fogo.
- **get_numer_length():** Função auxiliar para obter o tamanho de um dado número. Usado maioritamente quando é feita a impressão de um número no ecrã.

- **print_HUD():** Chamada em cada frame, para mostrar, durante o jogo, o score atual e a ronda atual.

Peso: 14% Autor: João Filipe (60%)

3.13) HighScores

Módulo responsável pela gestão dos highscores, como guardá-los num ficheiro, carregá-los do mesmo, verificar se o jogador atual atingiu um novo high score e, em caso afirmativo e guardá-lo no array de HighScores. É ainda responsável por imprimir no ecrã o array de HighScores e possui, ainda, uma função mais genérica para o resto do programa, que imprime uma string nas coordenadas dadas.

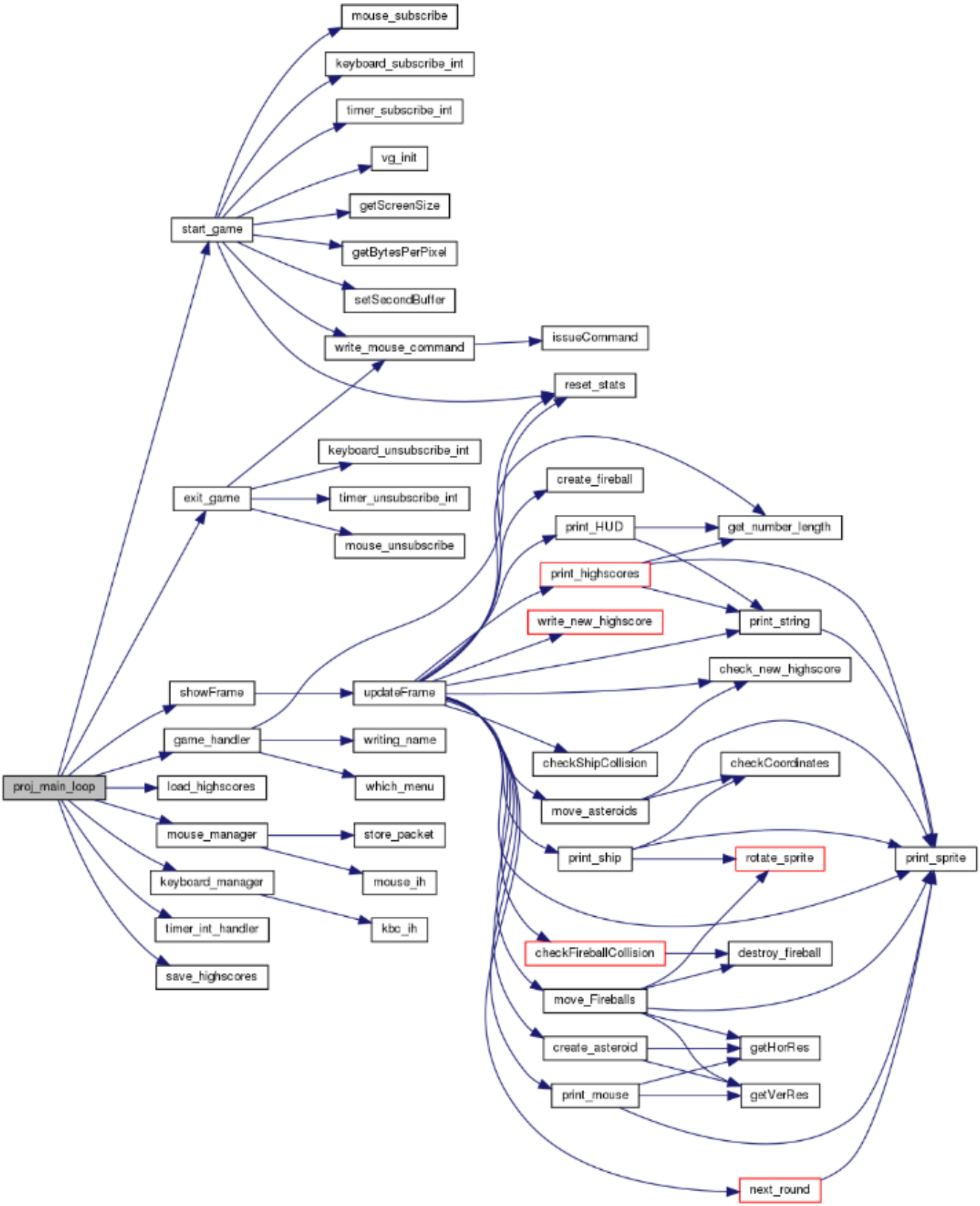
Peso: 8% Autor: 50%-50%

3.14) RTC

Este módulo é responsável pela implementação do RTC. É usado para obter a hora e data atual, que é guardada numa struct do tipo Date. A função getCurrentDate() é chamada sempre que o jogador perde o jogo e atinge um novo high score.

Peso: 4% Autor: Gaspar Pinheiro (100%)

3.15) Call Graph



4. Detalhes de Implementação

Pensamos que seja relevante mencionar a forma como organizamos toda a informação necessária numa única struct - Game. Esta informação é passada para as funções do projeto usando apenas como um dos seus parâmetros o apontador para a struct Game. Dentro desta struct existem outras, que armazenam a informação necessária de uma forma que nos faciltou a implementação do projeto, dada a facilidade de interpretação da informação e a organização do código.

Outra ideia fundamental da organização do código é o funcionamento paralelo de uma máquina de estados (game_handler() – game.c) juntamente com a função que atualiza o ecrã (update_frame() – visuals.c). A máquina de estados, que é chamada após a análise do input recebido pelo rato e pelo teclado, atualiza o estado do jogo consoante esse input e no ciclo seguinte do main loop, a função update_frame() responsabiliza-se por mostrar no ecrã apenas aquilo que é suposto para o dado estado. Por outro lado, esta função (update_frame()), acaba por fazer a gestão do jogo enquanto este está a decorrer. Não só chama as funções que realizam o "print" da nave, do rato, dos asteróides e das bolas de fogo, como também é responsável por criar novas bolas de fogo quando o botão esquerdo do rato está a ser pressionado, criar novos asteróides a uma taxa fixa para cada ronda, verificar todo o tipo de colisões e ainda verificar se a ronda acabou, fazendo as alterações necessárias à difculdade da ronda.

Uma das maiores dificuldades encontradas foi a de rodar uma sprite, o que fazemos não só na nave, como também em cada fireball, de forma a que estes objetos fiquem sempre direcionados para o rato. Foi necessária alguma pesquisa e acabos por usar o algoritmo (com algumas adaptações) explicado no seguinte video:
https://www.youtube.com/watch?v=EnC1WKnN-H4&t=1608s .

Outra parte do projeto que necessitou de estudo autónomo foi a parte de guardar em ficheiros de texto os High Scores, tal como carregá-los.

5. Conlusões

Esta unidade curricular mostrou-se, efetivamente, um desafio, exigindo de nós mais horas semanais do que qualquer outra cadeira.

Gostaríamos também de salientar a nossa dificuldade inicial na disciplina, onde, por exemplo, na primeira aula prática, fomos apenas capazes de escrever quatro linhas de código. Pensamos que tal se tenha sucedido devido à grandiosidade de texto que nos é apresentada em cada handout. Embora tivessemos sido capazes de nos adaptar, penso que toda a informação de cada periférico podia ser dividida, dado que alguma da informação dos handouts não era totalmente necessária à realização das funções do lab.

Em relação ao projeto, foi interessante a forma como nos tornou mais atentos para a organização do código e para a sua eficiência, dado que era importante não ter um programa pesado para que o jogo pudesse funcionar com fluidez.