Documentação - Space Wars

Lucas Paiolla — Caíque Corrêa — Eduardo Brancher

11221911 — 11276281 — 8587409

Contents

1 Disposição dos diretórios e arquivos

Os arquivos do projeto estão dispostos em quatro pastas e mais outros arquivos no diretório que contém as quatro pastas. Os quatro diretórios são:

1. `base` - Contém bibliotecas que são completamente independentes do resto do projeto. Em teoria, estas bibliotecas podem ser utilizadas em qualquer outro projeto. (Mas todos os arquivos do projeto foram feitos inteiramente por nós.)

2. `fisica` - Este diretório é um dos mais importantes e foi o foco da primeira parte do projeto. Ela utiliza bastante o arquivo `vetores.h` da `base` para fazer os cálculos. (Entraremos em detalhes sobre o diretório mais para frente.)

3. `graficos` - Este diretório é responsável pela parte visual do jogo. Ela foi foco da segunda parte do projeto e utiliza a biblioteca `xwc`, disponibilizada pelo professor.

4. `IO` - Este diretório contém arquivos responsáveis por leitura e escrita de arquivos. (Atualmente fazemos apenas leitura de configurações.)

Já na raiz, existem arquivos de configurações, além do `Makefile` e o arquivo de entrada (que contém o método `main`).

2 Diretório base

2.1 Arquivo auxiliar

A biblioteca auxiliar possui uma série de utilidades para o projeto.

Entre elas, as declarações do tipo `string` e `Bool`, além de um gerenciador de erros (`exception handling`). Para isso, há no `.h` um `enum` com uma série de códigos de erros.

Este `enum` é usado como um tipo `erroCode`. Há nesse arquivo uma função chamada `throwException` que recebe o nome da função onde o erro aconteceu, uma mensagem de erro e o `erroCode`.

Dessa forma, depurar o jogo fica muito mais fácil e isto se mostrou absurdamente útil para nós ao longo do desenvolvimento.

Além disso, ele arquivo conta com macros para máximo e mínimo de dois números e funções `mallocSafe`, `freeSafe`, `pause` e `geraRandomicoEntre`.

A documentação detalhada de cada coisa pode ser encontrada no arquivo `auxiliar.h` e as implementações em `auxiliar.c`.

2.2 Arquivo leitor

Esta biblioteca contém um leitor de arquivos muito simples. Ele ele ser inicializado pelo método `initLeitor` passando um nome de arquivo a partir do executável do jogo.

Após isso, o método `proxLeitura` faz um leitura de um token do arquivo (definido como uma string) e retorna-o. Já o método `getLeitura` apenas retorna o último token lido sem avançar a leitura. A função `imprimeAtual` imprime o `getLeitura` no *stdout*. E a função `strigual` compara uma string com a última leitura.

Por fim, `disposeLeitor` deve ser usado quando o leitor termina seu trabalho, liberando o arquivo.

2.3 Arquivo vetores

O foco dessa biblioteca é em um struct definido por `vet2D` e tem dois `double` x e y. Podemos fazer basicamente tudo o que se espera de um biblioteca para vetores. Além disso, adicionamos um struct para matrizes chamado `mat2D`. Ainda não adicionamos muitos métodos para lidar com essa struct, apeas os dois métodos `multiplicaPorMatriz` que multiplica um vetor por uma matriz e o `rotaciona`, que rotaciona um vetor por um ângulo utilizando uma matriz de rotação.

Talvez futuramente adicionemos operações básicas com matrizes.

3 Diretório fisica

Chegamos ao coração do jogo. Inicialmente tínhamos três tipos de "corpos" que produziam ou recepiam algum efeito físico: as duas naves, os projéteis e um planeta. O grupo pensou bastante em uma maneira de generalizar o conceito de corpo e também de generalizar o número de corpos que podem aparecer de cada tipo (como, por exemplo, ter mais de um planeta em jogo).

Após pensar bastante, surgiu o conceito de **objeto**, que se baseia em Programação Orientada a Objetos. Abstraímos da seguinte maneira:

Todos os "corpos" físicos que aparecem na tela são chamados de objetos e serão structs com uma outra struct mais fundamental dentro delas chamada `Objeto`. Detalhes da implementação serão detalhadas em 3.1.1.

O importante é que muitos métodos não precisam saber qual é o tipo de objeto, passamos para estes métodos apenas um ponteiro para o objeto e os métodos fazem seu trabalho.

Para pegar os objetos de uma nave, projétil, planeta ou booster (estes serão falados com mais detalhes em 3.2) utilizamos o método `GetObjeto`, que, dados o tipo que queremos (um desses quatro acima) e o seu índice (pois há arrays para cada tipo de objeto).

3.1 Arquivo fisica

3.1.1 Implementação dos objetos

A struct de um objeto possui os seguintes campos: m, r, p, v e s. Que significam respectivamente: massa, raio, posição, velocidade e sprite.

Após isso, temos que as structs `Nave`, `Projetil`, `Planeta` e `Booster` possuem um campo `Objeto o`. O nome `o` deve ser padronizado pois há os seguintes macros:

1. `vel` para `o.v`
2. `mass` para `o.m`
3. `pos` para `o.p`
4. `radius` para `o.r`
5. `spr` para `o.s`

Dessa forma, se `p1` é um planeta, não precisamos escrever `p1.o.r`, mas simplesmente `p1.radius`, abstraíndo o struct Objeto.

Além disso, há um tipo defindo por um enum chamado `TipoObj` que contém todos os tipos de objeto em uma sequência especificada e o último elemento do enum é utilizado caso queiramos sabe o número de objetos diferentes.

Depois temos quatro arrays diferentes, um para cada objeto e seus máximos são definidos por constantes. Os arrays das naves e dos planetas possui um tamanho constante, pois os elementos desses arrays são os mesmos do começo ao fim do jogo. Já os arrays de projéteis e boosters conterão elementos diferentes ao longo do jogo, por isso seu total de objetos varia.

Por isso, existe um array que contém o total de cada array. Este array se chama `tot_obj` e tem tamanho `NUM_TIPO_OBJ`. A ordem dos tamanhos é a mesma que a do enum `TipoObj`, assim podemos fazer `tot_obj[NAVE]` para saber a quantidade de naves. Perceba que assim podemos alterar o máximo de naves e todo o resto do código funciona normalmente.

3.1.2 Uso dos objetos

A biblioteca contém os seguintes métodos que usam objetos apenas, sem ter que saber que tipo de objeto é aquele:

1. `Forca` - dá a força gravitacional entre dois objetos
2. `incVel` - incrementa a velcidade de um objeto baseado em uma força
3. `IncPos` - incrementa a posição de um objeto baseado em sua velocidade
4. `DistanciaEntre` - dá a distância entre dois objetos
5. `GetObjeto` - dado um tipo e um índice, dá o objeto do vetor respectivo
6. `SetObjeto` - dados um tipo, um índice e um objeto, transforma o índice do vetor respectivo no objeto passado
7. `ObjetoDuplicado` - diz se dois objetos distintos estão na mesma memória
8. `ObjetoIgual` - diz se todas as propriedades de dois objetos são iguais
9. `CalculaForcaSobre` - dado um objeto, acha a resultante gravitacional sobre ele
10. `AtualizaObjeto` - atualiza a posição, velocidade e sprite de um objeto
11. `AtualizaObjetos` - atualiza todos os objetos
12. `giraObjetoVel` - gira o sprite de um objeto para que aponte para sua velocidade

3.1.3 Outros usos da biblioteca

Outros usos da biblioteca são:

1. Devolver padrões de cada objeto
2. Checagem de colisão entre objetos
3. Fazer o manejo da vida das naves
4. Fazer o manejo dos projéteis da tela
5. Atualizar o jogo como um todo e dizer se ele deve continuar

Para um melhor entendimento, veja o arquivo `fisica.h`.

3.2 Arquivo gerenciadorBoosters

Os boosters são algo extra que o professor não pediu, mas achamos legal implementar. Eles são objetos que ficam voando pela tela, sendo atraídos gravitacionalmente pelos outros corpos (se quiser desativar isso, basta trocar a massa dos boosters para 0). Eles são lidos de um outros arquivo de configuração que deve ser nomeado como booster.cfg (ao contrário do outro arquivo de input, o arquivo de boosters não é requisitado ao usuário, ele é único).

Para isso, apesar de os boosters estarem definidos em `fisica.h`, o arquivo `gerenciadorBoosters.h` faz exatamente o que o nome diz.

Os boosters são definidos num arquivo e ficam em um array chamado `boostersPreCriados`. Quando eles vão aparecer na tela, para que os jogadores possam pegá-los, é escolhido um booster aleatoriamente.

Além disso, há um booster padrão definido via código (pela função `defineBoosterPadrao`). Ele é o booster que as naves possuem se não estão com nenhum booster atualmente. E depois de pegar um (colidindo com ele), o booster ficará na nave por um tempo. Esta e outras propriedades estão definidas no arquivo de leitura.

Entre as funcionalidades dessa biblioteca, temos:

1. `defineBoosterPadrao` - que deve ser chamada para colocar o booster padrão na posição 0 do vetor de boosters pré criados.
2. `criaNovoBooster` - que coloca um novo booster na tela.
3. `removeBoosterDaTela` - que remove um booster cujo tempo na tela se esgotou
4. `capturaBooster` - chamada quando uma nave colide com um booster.
5. `resetaBooster` - que reseta uma nave para o booster padrão.
6. `boosterVaiSpawnar` - que retorna se, naquele frame, algum booster deve aparecer, baseado na probabilidade de um booster aparecer.
7. Checagem de colisões de naves com booster
8. Atualização dos boosters em tela e dentro das naves

4 Diretório gráficos

Dentro deste diretório se encontram todas as sprites do jogo, a biblioteca gráfica do professor, uma abstração dessa biblioteca construída por nós (o arquivo `graficos.h`) e uma biblioteca que une esta com a parte física do jogo, chamada `display.h` (os nomes, infelizmente não são muito bons, mas não conseguimos achar melhores - aceitamos sujestões).

4.1 Arquivo graficos

Uma das primeiras coisas que encontramos no arquivo é o tamanho da tela. Para se ter uma tela menor, basta ajustar a constante `SIZE_X_WIN`. E para ter uma proporção de tela diferente, basta ajudar o `SIZE_RATIO`. Pretendemos futuramente jogar essas configurações para um arquivo de configurações dos gráficos.

4.1.1 Funcionamento das sprites

O coração da segunda parte do projeto são as chamadas sprites: structs com dois campos: `double angle` e `NOME_SPR img`. O ângulo é simplesmente o ângulo que a sprite está fazendo com o eixo x. Ele define qual rotação da imagem será exibida e, atualmente, se acha o ângulo do **vetor velocidade** com o eixo x.

Na próxima etapa, vamos mudar isso para o vetor aceleração, pois é o que se espera de um jogo desse tipo.

Já o `NOME_SPR` é um tipo enum que deve conter todos os sprites existentes no jogo em uma certa ordem (e o último enum dá o número de sprites que há). É importante que a ordem do enum seja a mesma com que as pastas estão dispostas no diretório `pics`.

Observe que ainda há muitos sprites não implementados (como todos os dos projéteis - pois as naves não atiram ainda).

Por fim, é importante observar que cada sprite possui várias pics de rotação. Em cada pasta de uma sprite há essas rotações e um arquivo size que contém três números: as dimensões da sprite e o número de pics.

4.1.2 Outras funcionalidades

De resto, esta biblioteca possui métodos para pegar PIC e Mask de uma sprite, com base no ângulo atual dela, métodos para desenhar sprits em um ponto da tela, métodos para mexer no ângulo e outras funcionalidades.

É importante também dizer que existem duas janelas principais, uma visível pelo usuário (`showingWindow`)e outra que está escondida e é usada entre uma atualização e outra do jogo (`workbench`).

Entre uma atualização e outra, imprimimos tudo o que tem para imprimir no `workbench` e, quando tudo foi imprimido, colamos o `workbench` por cima da `showingWindow`.

4.2 Arquivo display

O arquivo display possui responsabilidade de integrar a parte gráfica com a parte física. Uma funcionalidade importante dela é converter as coordenadas usadas pela física em coordenadas da tela (em pixels).

Fora isso, ela possui funções para imprimir cada tipo de objeto.

5 Diretóro IO

Este diretório é dedicado a arquivos de entrada e saída. Atualmente, entretanto, só estamos lendo arquivos de configuração e não escrevemos nada.

5.1 Arquivo IO

O arquivo IO está separado em duas partes: ler os planetas e naves de um arquivo de input digitado pelo usuário e a parte de ler as configurações dos booster de um arquivo `booster.cfg`.

Para entender como os inputs são feitos, vamos explicar como é a sintaxe de um arquivo de configurações.

5.1.1 Arquivos .cfg

Os arquivos .cfg possuem a seguinte sintaxe:

`nome_variavel = valor`

Podemos definir algumas variáveis que servem para todo o jogo, (como `dt`). Ou podemos criar um novo objeto fazendo:

`tipo_obj = [`

Declarar as variáveis desse objeto (as não declaradas ficam com um valor padrão definido `hardcoded`) E

`]`

Para o arquivo dos boosters, não é preciso dizer qual o tipo do objeto, basta abrir uma região de objeto.

Olhe os arquivos .cfg para ter uma noção melhor.

6 Outros arquivos

6.1 Arquivo debug

Por fim, o arquivo `debug` foi feito para testar o projeto enquanto vamos criando-o. Ele possui também a função `main`, que é a abertura do programa.

6.2 Makefile

O `Makefile` deve ser executado pelo terminal do `bash` para compilar o projeto. Se você quiser recompilar tudo, pode digitar

`make clean_all`

E assim todos os códigos objeto serão deletados.