

# Exercício Programa 2 - Parte 1 - Guia do Desenvolvedor

30 de junho de 2013

## 1 Introdução

### Integrantes:

- Victor Sanches Portella - N° USP: 7991152
- Mateus Barros Rodrigues - N° USP: 7991037
- Lucas Dário - N° USP: 7990940

## 2 Parte 1

Essa seção é referente ao EP2, onde fizemos a base do programa, a maior parte dos módulos usado para gerarmos o rio, para realizarmos o teste, e toda a estrutura em si.

### 2.1 Módulos e Tipos

Em nosso programa, criamos os seguintes tipos de primeira classe:

- **Terreno** Usado para representar cada "pixel" do jogo, contendo como informação seu tipo (Água ou Terra) e a velocidade.
- **linhaT**: Usado para representar cada linha do rio. Ela possui a sequência de terrenos que a formam, e indicação de suas margens e tamanho da linha.
- **Rio**: Representa o Rio em si, com seu fluxo de água, a sequência de linhas que formam o rio, o número de colunas de pixels e o tamanho mínimo que o rio deve ter de água.

- **List:** Representa uma lista duplamente encadeada circular com cabeça que opera com objetos do tipo Item<sup>1</sup>.

Módulos usados em nosso programa:

- **utils.h** Biblioteca que contém funções de uso geral.
- **terreno.h** Biblioteca que contém funções para manipulação de objetos do tipo Terreno.
- **testes.h** Biblioteca que contém as funções para execução de testes de certos parâmetros do rio em determinadas condições.
- **linhaT.h** Biblioteca para manipulação de objetos do tipo linhaT.
- **list.h** Biblioteca para manipulação de objetos do tipo List.

Agora, daremos uma explicação sobre o funcionamento dos tipos e das principais funções presentes nos módulos já mencionados.

### 2.1.1 Terreno

Terreno é o objeto que usamos para representar cada pixel. Ele é definido por um tipo e uma velocidade. Por definição, todo bloco de terra tem velocidade 0. Além disso, o tipo do bloco será um caractere, que é o mesmo que será impresso na tela. Por isso, temos os seguintes defines no nosso terreno.h:

AGUA	.
TERRA	#

### 2.1.2 linhaT

O tipo linhaT é usado para representar uma linha do rio, ou seja, uma sequência ordenada de objetos do tipo Terreno. Nela, temos as informações de tamanho da linha, as margens direita e esquerda e uma flag para indicar se a linha tem ou não um ostáculo.

As funções **setFluxo** e **getFluxo** funcionam de forma a executar o processo de normalização explanado no enunciado do exercício.

Já função **igualFluxo** recebe dois objetos do tipo linhaT. A primeira linha será usada como base para a segunda linha. Com isso, definiremos as velocidades de cada Terreno da linha 2 baseado nas velocidades da linha 1.

---

<sup>1</sup>Item foi definido como um vetor de objetos do tipo linhaT, mas pode ser facilmente mudado no cabeçalho desse módulo

Para isso, definimos temporariamente o fluxo da linha 1 como 1. Com isso, aplicamos a seguinte regra:

- Caso o terreno da linha 2 esteja adjacente a alguma terra, a velocidade desse terreno é automaticamente 0, independentemente das outras regras.
- Caso o terreno da linha 1 fosse terra e se transformou em água na linha 2, a velocidade dessa água será 0.1.
- Caso o terreno da linha 1 era água e continuou como água na linha 2, é sorteado um número inteiro  $x$ ,  $-1 < x < 1$ , e a velocidade da linha 2 será igual a  $V_1 + x$ , sendo  $V_1$  a velocidade do terreno da mesma coluna na linha 1.

Explicação de algumas funções importantes:

- **int geraObstaculo(linhaT, int):** Essa função cria um obstáculo na linha passada como argumento, com o tamanho passado. Caso não seja possível criar um obstáculo do tamanho pedido, a função retorna 0. Caso contrário, ela cria o obstáculo, coloca a velocidade das águas adjacentes ao obstáculo como 0, e retorna 1. É importante ressaltar que, caso o obstáculo fosse fazer com que o fluxo zerasse, a criação do obstáculo é cancelada e a função retorna 0.
- **linhaT geraLinha(linhaT, tamMin):** Essa função irá gerar uma nova linha baseada na linha passada como argumento. Para isso, usará dentro dela a função *igualFluxo* para manipular as velocidades, e definirá as margens da nova linha com base nas antigas, sendo que as margens tem igual probabilidade de aumentar, diminuir ou se manter, e as margens variam somente de 1 pixel de uma linha para outra, para evitar mudanças bruscas.

Após definidas as velocidades, os fluxos são normalizados com a função **setFluxo**, definindo o fluxo como o antigo fluxo da linha 1.

### 2.1.3 List

List é um tipo de primeira classe usado para representar uma lista duplamente encadeada circular com cabeça. A ideia do funcionamento dela é que ela é constituída de nós, cada um com uma conexão para o seu próximo e seu anterior. Toda list tem um *item Atual*, sendo esse um nó da lista o qual

o cliente tem acesso. E é através dele que navegaremos pela lista. A maioria das funções se baseia nele, como veremos.

A lista sempre tem um nó que define o fim/começo da lista, que chamaremos de **EOL** (*End Of List*). Nessa implementação de lista, definimos o tipo *Item* como um vetor de objetos do tipo *emphlinhaT*

Explicação geral das principais funções:

- **List listInit()**: Ao chamar a função, é retornada uma lista do tipo *List*, contendo um único nó (*EOL*).
- **mvEOL(List)**, **mvNext(List)**, **mvPrev(List)**: Essas são funções para navegação através da lista, ou seja, elas mudam o *item Atual* para o qual a lista aponta. As ações dessas funções são, respectivamente, apontar o *item Atual* para o *EOL*, para o próximo nó do *item Atual* e para o nó anterior do *item Atual*.
- **Item getItem(List)**: Essa função retorna o item apontado atualmente pela lista (o *item Atual*). Caso o *item Atual* seja o *EOL*, a função retorna **NULL**.
- **removeItem(List)**, **insertItem(List, Item)**: Essas são as funções de remoção e inserção de itens na lista, respectivamente. A primeira deleta da lista o nó do *item Atual*<sup>2</sup>, e define o novo *item Atual* como o nó anterior do nó que foi deletado. Já a segunda função insere o *Item* como um nó e o posiciona como o próximo do *item Atual* da lista.
- **int isEOL(List)**: Uma função que retorna 1 caso o nó para o qual a sua lista está apontando seja o *EOL*, e 0 caso contrário.
- **int emptyList(List)**: Essa função retorna 1 caso a lista contenha somente o *EOL*, e 0 caso contrário.

#### 2.1.4 Rio

Rio é um tipo de primeira classe usado para representar o Rio em si. Nele temos as informações do fluxo, tamanho (linhas e colunas), uma *List* de objetos *linhaT* que constituem o rio, além do tamanho mínimo que o rio precisa ter. Explicação das principais funções de manipulação do Rio:

- **Rio alocaRio(int, int, float, int)**: Essa função retorna um objeto do tipo *Rio*, sendo os parâmetros, respectivamente, para as seguintes informações:

---

<sup>2</sup>Mas caso o cliente tente deletar o *EOL*, a função não faz nada.

1. Número de linhas
  2. Número de colunas
  3. Fluxo
  4. Tamanho mínimo do rio
- **int atualizaRio(Rio):** Essa função atualiza o rio. Sendo as linhas numeradas de cima para baixo de  $1 \dots N$ , a função deleta a N-ésima linha, e gera uma nova linha baseada na linha 1, e a coloca no começo da lista. Ou seja, a linha nova vira a linha 1, a linha 1 vira a 2, etc. Além disso, o atualiza rio retorna um int dependendo se a geração da linha foi um sucesso, ou se deu algum erro. Tabela dos valores retornados:
    - + **SUCESSO\_ATUALIZA:** Sem erros, rio atualizado com sucesso.
    - + **FALHA\_ATUALIZA:** Erro ao tentar gerar uma nova linha. Não é recomendado que se use a função `desenhaRio()` nesse caso. Normalmente, rios sem linhas ou erros internos do sistema causam esse erro.
    - + **FALHA\_OBST:** Erro ao tentar criar um obstáculo na nova linha. Esse erro acontece quando o tamanho da barreira iria zerar o fluxo do rio, ou caso o tamanho da barreira fosse exceder o tamanho do rio.
  - **desenhaRio(Rio):** Essa função desenha o rio na stdout. Ou seja, imprime, em ordem, todos os terrenos das linhas da lista de objetos `linhaT`.
  - **LinhaT getLinha(Rio,int):** Retorna a i-ésima linha do Rio, sendo que as linhas são numeradas de  $1 \dots N$ , de cima para baixo, sendo que N é o número total de linhas. Meninos bolados

### 3 Parte 2

Essa seção é referente ao EP3, onde implementamos a representação do rio através de uma interface gráfica com a biblioteca *Allegro*. Criamos um módulo que contém todos os métodos referentes a essa interface, assim como para sua criação e manipulação.

### 3.1 Módulo visual

Nesse módulo estão localizadas as funções para que o rio seja desenhado através da biblioteca *Allegro*. Cada linha representa um vértice do rio, ao contrário da forma como é interpretado na parte 1, onde cada linha representa um conjunto de retângulos desenhado na tela.

A partir dessa parte, esse módulo também é usado como controle de atualização. Ou seja, fazemos a atualização do rio, assim como controle de atualização do Rio e velocidade do rio através desse módulo.

Para desenharmos o rio, consideramos o par de linhas, ao invés de cada linha separada. Assim, como mostrado na imagem que consta no enunciado do EP3, usamos o par de linhas para desenharmos as margens do rio.

Esse módulo tem um única função que é exportada para o usuário, que é a **visualInit(Rio,int,int)**. Ela cria uma janela com a biblioteca *Allegro*, de tamanho proporcional ao tamanho do rio. O inteiro passado como segundo argumento, que chamaremos de **D**, é com quantos pixels cada “quadrado” do rio será representado. O terceiro inteiro passado como argumento é o tempo, em milissegundos, entre uma atualização e outra do rio.

Principais funções privadas do módulo:

- **visualUpdate(Rio)**: Atualiza o rio 1 vez (gerando uma nova linha), e atualiza a janela do *Allegro*.
- **desenhaRioVisual(Rio)**: A principal função desse módulo. Ele desenha o rio na janela do *Allegro*, baseado nas informações da linha do Rio. Para desenharmos o rio, interpretamos cada linha como uma reta horizontal no mapa. Assim, usamos uma série de regras para desenharmos o rio. Denotando o tamanho da linha atual e da próxima linha na sequência como *linha* e *linhaProx*, respectivamente, temos<sup>3</sup>:
  - + **linha > linhaProx**: Ou seja, quando a margem diminui de tamanho de uma linha para outra. Nesse caso, desenhamos um retângulo de tamanho  $(linha - 1) * D$ , e completamos com um triângulo que liga os vértices das duas margens.
  - + **linha = linhaProx**: Desenhamos um retângulo de tamanho  $linha * D$ .
  - + **linha < linhaProx**: Desenhamos um retângulo de tamanho  $linha * D$ , e completamos com um triângulo que conecta os vértices das duas linhas subsequentes.

---

<sup>3</sup>Usamos o fato que as margens só podem varia de 1 quadrado de uma linha para a outra

- **inicializar()**: Inicializa todos os objetos necessários pelo *Allegro*. Retorna 0 caso haja algum problema.

As regras sobre a função **desenhaRioVisual** foram explicadas explicitamente pois, apesar de a ideia parecer simples, o código não ficou muito simples, então é de grande utilidade para o desenvolvedor entender a ideia antes de ler o código.

## 4 Parte 3

Nessa parte finalizamos o jogo, implementando todas as funcionalidades principais para a jogabilidade, como colisões, sistemas de vida e de recomeço de jogo, além de movimento do barco associado com o movimento do rio.

### 4.1 Módulo **vetor2D.c**

Esse módulo cria um tipo de primeira classe chamado **Vetor2D**, usado para representar um vetor bidimensional, com componentes X e Y. As funções fornecidas por esse módulo são bem simples e auto-explicativas, só é bom lembrar que os ângulos fornecidos pelas e/ou para as funções que mexem com o ângulo do vetor são dados em radianos, e mexem com o ângulo do vetor com relação à horizontal.

### 4.2 Módulo **barco.c**

Esse módulo cria o tipo **BarcoT**, que representa nosso barco como um tipo de primeira classe. O barco tem informações como *Tamanho*, *Posição*, *Velocidade* e *Quantidade de vidas*. Principais funções:

- **novoBarco(Vetor2D, Vetor2D, Vetor2D, int)**: Retorna um novo barco, sendo os parâmetros destinados para *Posição*, *Velocidade*, *Tamanho* e *Quantidade de Vidas*, respectivamente.
- **estaBatendo(BarcoT, Rio)**: Verifica se o barco está batendo em algum bloco de terra. Caso esteja colidindo, retorna 1, e retorna 0 caso contrário. Para a detecção de colisão, usamos um retângulo de lados definidos pela componente *Tamanho* do barco. Mesmo que o barco esteja rotacionado, a caixa de colisão continua na vertical. Apesar de parecer impreciso, funciona como uma boa aproximação, e torna a colisão menos complicada.

- **atualizaBarco(**BarcoT**,int, int, float, float)**: Essa função irá calcular a próxima posição do barco, assim como sua próxima velocidade, baseado nos argumentos passados que são, respectivamente: Número de remadas à esquerda, número de remadas à direita, velocidade do rio à esquerda e velocidade do rio à direita. O método para o cálculo da nova velocidade será explicado mais a frente.

### 4.3 Modificações na visual.c

Algumas modificações foram feitas na `visual.c`, visando implementar as funcionalidades de jogabilidade, e aqui discutiremos as principais modificações.

A primeira modificação foi na função **desenhaBarco**, que agora faz todas as chamadas de funções e cálculos necessários para a movimentação do barco, além de se responsabilizar por desenhar o barco na janela.

Além disso, foram implementadas algumas funções para melhorar a jogabilidade:

- **Invulnerabilidade temporária:** Ao bater em algum obstáculo ou pedaço de terra, o barco irá perder uma vida e ficará invulnerável por alguns segundos (definidos pela macro `TEMPO_INV`), e com uma velocidade fixa (definida pela macro `MS_INV`). Enquanto estiver nesse estado, o jogador pode atravessar blocos de terra livremente.
- **Score:** O Score, ou pontuação, é quase auto-explicativo. Serve como um objetivo no jogo, seja para competir com amigos ou para sua auto-superação.
- **Inclinação visual do barco:** Para deixar o jogo mais responsível para a visão do jogador, a inclinação da imagem do barco também varia de acordo com a inclinação de sua velocidade.