Universidade Federal da Grande Dourados

Um Guia para o Desenvolvimento do Projeto Pacman

Autor:
André C. O. Sanches

Orientador:
Adailton José A. da Cruz

22 de Janeiro de 2017

Conteúdo

1 Introdução **2**

2 Estrutura do Jogo **2**

3 Loop principal **2**

4 Inicialização do Sistema **3**
4.1 Carregando mapas em arquivos 4

5 Entrada do Jogador **5**

6 Atualização e Lógica do Jogo **6**
6.1 Detectando colisões 8
6.2 Perseguindo o Pacman 9

7 Desenhando no console **10**

1 Introdução

Este guia é destinado ao alunos de Sistemas de Informações que estão com dificuldades em concluir o projeto final da discplina de Laboratório de Programação II. Alunos da disciplina de Laborátorio de Programação I também poderão utilizar o guia para entender melhor o funcionamento geral do projeto, embora neste guia sejam utilizados conceitos ainda não abordados na disciplina de Laboratório de Programação I.

A proposta deste guia é abordar as etapas mais complicadas no processo de desenvolvimento do jogo Pacman, evitando exibir soluções prontas, mas orientando os alunos sobre quais algoritmos ou estruturas podem ser implementadas. Entre as etapas de maior dificuldade que serão abordadas neste guia, podemos citar: a entrada do usuário que exige a não-interrupção da execução, movimentação dos personagens, algoritmo de busca, mapa, colisões, entre outros.

Como forma de aprimorar o aprendizado da disciplina, grande parte do código desenvolvido neste guia utiliza conceitos dos conteúdos vistos em sala de aula, como arquivos, ponteiros, recursão e os conceitos de fila, pilha e lista.

2 Estrutura do Jogo

Antes de iniciar a criação do código em si, é de extrema importância entedermos como estruturar o nosso jogo. A etapa de estruturação é importante pois nos dá uma visão geral sobre como o jogo irá funcionar e como as entidades irão interagir. Modularize seu projeto com funções genéricas, defina constantes e idente seu código - essas práticas ajudarão o seu trabalho a ter melhor qualidade tanto em funcionalidade quanto na documentação. As Seções 3, 4, 5, 6 e 7 são partes de uma estrutura genérica de um jogo.

3 Loop principal

Existem jogos que não exigem interação em tempo real com o jogador. Podemos citar, por exemplo, o jogo da velha, onde cada rodada necessita aguardar até que um jogador decida onde marcar no tabuleiro. Nestes casos, uma simples função de entrada, como *scanf*, pode ser utilizada para coordenadar a interação do jogo.

Em jogos como Pacman, a execução do jogo não pode parar para aguadar a entrada do usuário. Se utilizarmos a função *scanf* para ler o movimento escolhido pelo jogador, o pacman, os fantasmas e qualquer outra entidade contida no jogo permaneceria congelada até que a entrada fosse feita. A Figura 1 mostra um exemplo do fluxo que podemos utilizar no loop principal do jogo.

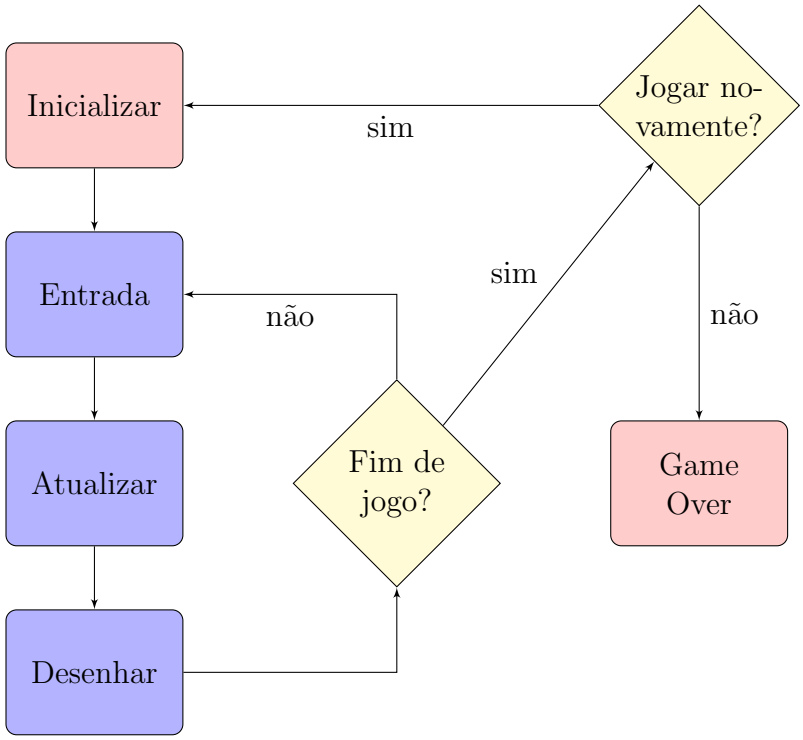

Figura 1: Exemplo de fluxo de execução que pode ser utilizado no projeto

4 Inicialização do Sistema

Criar uma função de inicialização do jogo pode ser importante para a organização do código. Existem inúmeras variáveis que devem ser inicializadas no jogo, como vida do jogador, pontuação, posições das entidades e inclusive o mapa. Como o jogo pode ser jogado novamente mesmo após o *Game Over*, podemos usar a função de inicialização para resetar o estado do jogo atual para um estado de novo jogo.

4.1 Carregando mapas em arquivos

As funções de inicialização também podem carregar o mapa a partir de arquivos. Essa parte é interessante do ponto de vista que podemos retirar a declaração do mapa de dentro do código e utilizar arquivos para carregá-lo. Dessa forma, separamos mapas (ou fases) de códigos, podendo carregar diferentes mapas no mesmo programa sem necessitar de uma recompilação. O Código 1 mostra um exemplo de código capaz de carregar um arquivo de mapa em uma matriz. Note que é necessário que o arquivo contenha a quantidade de linhas e colunas da matriz.

Código 1: Carregando o mapa em arquivo

```
#include <stdio.h>

int m_lin, m_col;

void carregarMapa() {
    FILE *arquivo_mapa;
    int i, j;

    arquivo_mapa = fopen("mapa", "r");

    /* A primeira linha do arquivo deve conter
    a quantidade de linhas e colunas do mapa */

    fscanf(arquivo_mapa, "%d%d", &m_lin, &m_col);
    mapa = (int**) malloc(sizeof(int*) * m_lin);

    for (i = 0; i < mapa_lin; i++)
    {
        mapa[i] = (int*) malloc(sizeof(int) * m_col);

        for (j = 0; j < m_col; j++)
        {
            fscanf(arquivo_mapa, "%d", &mapa[i][j]);
        }
    }
}
```

5 Entrada do Jogador

Para resolver o problema da interrupção de processamento para aguardar a entrada do jogador, podemos usar a função *kbhit* da biblioteca *conio.h*. A função *kbhit* retorna o valor 1 caso existam dados no buffer de entrada, entretanto ela não faz uso deles, apenas verifica a existência dos dados. Caso existam, significa que o jogador pressionou alguma tecla no teclado, e portanto devemos coletá-lo. Essa etapa não consiste em verificar se a tecla é válida ou não, mas apenas apanhar o valor do buffer. O tratamento sobre quais dados foram inseridos pode ser feito posteriormente. O Código 2 demonstra como pode ser feito esse módulo de entrada.

Código 2: Sistema de input com kbhit

```
#include <conio.h>

char entrada(char ch)
{
    char input;

    // Se houver dados no buffer
    if (kbhit()) {
        input = getch();
        return input;
    }

    // Senao, retorna o valor antigo
    else {
        return ch;
    }
}
```

Note que a função recebe como parâmetro uma variável do tipo *char*, e caso não existam dados no buffer, essa variável é retornada novamente. Isso foi feito para garantir que mesmo que o usuário não entre com nenhum dado, a entrada mais antiga não se perca. O Código 3 mostra como a função de entrada pode ser utilizada em conjunto com o restante do sistema. Observe que a variável *controleEntrada* irá receber um novo valor caso o usuário pressione alguma tecla. Entretanto, caso o usuário não pressione nada, a

execução do jogo não é interrompida e a variável *controleEntrada* continua com o mesmo valor.

Código 3: Estrutura geral do sistema com entrada do jogador

```
#include <conio.h>

int main() {

    char controleEntrada;

    // LOOP PRINCIPAL DO JOGO
    while (1)
    {
        controleEntrada = entrada(controleEntrada);

        processaJogo();

        desenhaJogo();
    }

    return 0;
}
```

6 Atualização e Lógica do Jogo

A etapa de atualização e lógica do jogo é onde se concentrará a maior parte do seu código. Como grande parte dessa etapa é puramente criativa, esta seção conterá apenas dicas e soluções para as partes mais difíceis no processo de desenvolvimento do jogo.

Uma dica importante para o desenvolvimento do jogo é criar *structs* genéricas para as entidades. Por exemplo, sabemos que o pacman e os fantasmas possuem uma posição na coordenada (x, y) do mapa. Também sabemos que cada entidade possui um movimento que não se altera até que outro caminho esteja disponível - um fantasma nunca deve voltar para trás, assim como o pacman não deve parar seu movimento atual caso o jogador tente ir em alguma direção com muro (exceto o movimento contrário). Sabemos

também que cada entidade no jogo possui uma cor exclusiva. Com isso, podemos criar uma nova *struct* para armazenar todos esses dados.

Criada uma *struct* genérica, podemos criar funções para a *struct* genérica, como por exemplo:

- moverEntidade(struct Entidade *e): altera a posição da entidade de acordo com seu movimento. O Código 4 exibe um exemplo dessa função;
- moverRandomico(struct Entidade *e): pode ser utilizada por entidades fantasmas para decidir um movimento randômico quando houver mais de um caminho disponível;
- moverPerseguindo(struct Entidade *origem, struct Entidade *destino): pode ser utilizada por entidades fantasmas para decidir qual caminho é mais curto para alcançar o pacman. A Seção 6.2 falará mais sobre como fazer uma busca utilizando o algoritmo BFS;
- trocarMovimento(struct* Entidade e, int movimento): altera o movimento de uma entidade, verificando se o novo movimento é valido ou não.

Código 4: Exemplo da função moverEntidade

```
int moverEntidade(struct Entidade *e) {
    int var_y = 0;
    int var_x = 0;
    switch (e->movimento)
    {
        case CIMA:
        var_y = -1;
        break;
        case BAIXO:
        var_y = 1;
        break;
        case ESQUERDA:
        var_x = -1;
        break;
        case DIREITA:
        var_x = 1;
```

```
            break;
        }

        if (mapa[e->y + var_y][e->x + var_x] != MURO) {
            e->y = e->y + var_y;
            e->x = e->x + var_x;
        }
}
```

6.1 Detectando colisões

Caso você esteja usando uma *struct* para armazenar os dados das entidades, você pode alterar o valor na matriz, na posição em que se encontra certa entidade, para um valor único que identifica se existe ou não uma entidade naquela posição. Você pode, por exemplo, atribuir o valor -1 àquela posição na matriz (desde que o valor -1 não represente nenhuma outra característica do seu mapa). Dessa forma, você pode criar uma função para verificar se na posição que a entidade se encontra já existe outra entidade. O Código 5 mostra como isso é possível.

Caso exista uma entidade naquela posição, você pode usar a função *trocarMovimento* para alterar o movimento da entidade que colidiu. Como sabemos o movimento atual, podemos fazer ela se movimentar na direção oposta à colisão.

Código 5: Função para detectar colisões

```
int detectaColisao(struct Entidade *e) {
    if (mapa[e->y][e->x] == -1) {
        switch (e->movimento) {
            case CIMA:
                trocaMovimento(e, BAIXO);
                break;
            case BAIXO:
                trocaMovimento(e, CIMA);
                break;
            case ESQUERDA:
                trocaMovimento(e, DIREITA);
                break;
```

```
            case DIREITA:
                trocaMovimento(e, ESQUERDA);
        }
        return 0;
    }
    else {
        return 1;
    }
}
```

6.2 Perseguindo o Pacman

No jogo, alguns fantasmas perseguem o pacman buscando o menor caminho até ele. Nessa situação, podemos utilizar diversos algoritmos de busca. Entre os mais utilizados, podemos citar o algoritmo A*, Dijkstra e Busca em Largura (BFS - Breadth First Search). Nesta seção, abordaremos o uso do algoritmo de busca em largura, por ser mais fácil de implementar e compreender.

A busca em largura consiste em, dado um elemento inicial, fazer a busca sempre visitando os elementos vizinhos, e ir adicionando esses vizinhos em uma fila. Assim que todos os vizinhos do elemento são explorados, a busca passa para o vizinho mais recentemente adicionado à fila e repete os passos, explorando novos vizinhos e adicionando-os à lista, desde que ainda não tenham sido adicionados à ela.

Para implementar uma busca em largura, é essencial entendermos o conceito de filas. As filas serão necessárias para armazenar os vizinhos recentemente visitados e os vizinhos que já foram explorados. O Código 6 mostra um pseudocódigo para o algoritmo de busca em largura.

Para adaptar o pseudocódigo, podemos pensar nos elementos como se fossem posições na matriz. Uma posição na matriz é representada por uma coordenada (x, y). Dessa forma, podemos criar uma *struct*, que chamaremos de **Vértice**, para representar cada posição na matriz, contendo a variável x e y. Recorrendo às aulas, você deverá implementar o conceito de filas para variáveis do tipo *struct* Vértice implementado.

Como estamos interessados apenas em descobrir o próximo movimento que o fantasma deverá fazer para se aproximar do pacman, precisamos de uma variável pai em cada vértice, para sabermos qual foi o vértice pai que

gerou o vértice filho. Dessa forma, assim que a busca se encerrar, poderemos retornar do vértice destino até o vértice origem apenas caminhando pelos vértices pais.

Código 6: Pseudocódigo da busca em largura

```
elementoDestino = pacman

filaElementos = Fila()
filaVisitados = Fila()

filaElementos.inserir(elemento1)
filaVisitados.inserir(elemento1)

// Enquanto a fila de elementos nao estiver vazia
enquanto filaElementos.vazia() == falso:

    // elementoAtual recebe o primeiro elemento na fila
    elementoAtual = filaElementos.remover()

    para i iterando entre vizinhos de elementoAtual:
      se elementoAtual == elementoDestino:
          break

      // Se o elemento vizinho ainda nao foi visitado
      se filaVisitados.contem(i) == falso entao:
          filaElementos.inserir(i)
          filaVisitados.inserir(i)
      fim_se
    fim_para
fim_enquanto
```

7 Desenhando no console

A etapa de desenho faz parte da criatividade do desenvolvedor do jogo, portanto esta Seção apenas irá apresentar dicas de como desenhar de forma mais organizada no console.

As funções de desenho devem vir após todo o processamento do jogo, pois caso imprima os componentes do jogo antes do processamento, o jogador poderá, por exemplo, colidir com um fantasma próximo sem ver que realmente iria colidir. Você pode separar uma função de desenho para cada entidade no jogo. Por exemplo, você pode criar as funções *desenharMapa*, *desenharEntidade* e *desenharHud* para imprimir todos os elementos da tela de jogo.

Para dar mais dinamismo ao jogo, evite limpar a tela e imprimir tudo novamente. Operações de saída tendem a ser muito lentas, e isso poderá fazer a tela do jogador ficar piscando. Para resolver esse problema, você pode usar a função gotoxy da biblioteca conio, redesenhando apenas nas novas posições das entidades. A lista a seguir mostra algumas das funções que ajudam a desenhar no console nativo do windows:

- **gotoxy**: função da biblioteca *conio.h* para posicionar o cursor em uma posição específica do console.
- **textcolor**: função da biblioteca *conio.h* para alterar a cor de impressão do console. Assim que imprimir o caractere com a cor desejada, você deve voltar à cor anterior para que os próximos caracteres não continuem sendo impressos na mesma cor.
- **_setcursortype**: função da biblioteca *conio.h* para alterar o tipo de cursor impresso na tela. Utilize *_setcursostype(_NOCURSOR)* para que nenhum cursor seja impresso na tela, melhorando a estética do seu jogo.

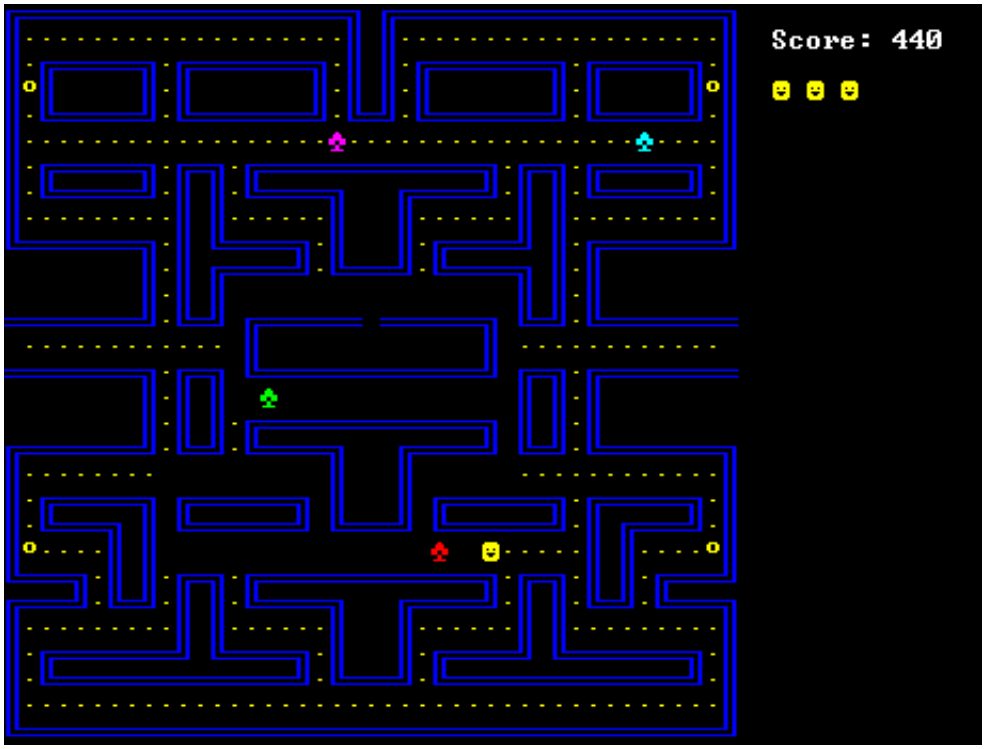

Figura 2: Jogo desenvolvido durante a produção deste guia