UNIVERSIDADE FEDERAL DO PARÁ
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Marcos Romero Gonzaga de Almeida

PROGRAMAÇÃO DE JOGOS 2D USANDO O MRDX E FUNDAMENTOS DE JOGOS 3D

Belém – PA
2004

UNIVERSIDADE FEDERAL DO PARÁ
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Marcos Romero Gonzaga de Almeida

PROGRAMAÇÃO DE JOGOS 2D USANDO O MRDX E FUNDAMENTOS DE JOGOS 3D

Trabalho de Conclusão de Curso apresentado para obtenção do grau de Bacharel em Ciência da Computação.
Orientador:
Prof. Dr. Eloi Favero.

Belém – PA
2004

PROGRAMAÇÃO DE JOGOS 2D USANDO O MRDX E FUNDAMENTOS DE JOGOS 3D

Este trabalho foi julgado em _____/_____/_____ pela banca examinadora, que atribuiu o conceito _____ .

_____

Prof. Dr. Eloi Favero
ORIENTADOR

_____

MEMBRO DA BANCA EXAMINADORA

_____

MEMBRO DA BANCA EXAMINADORA

"Agradeço aos meus pais por todo o apoio que eles têm me dado para a realização de meus sonhos".

S U M Á R I O

RESUMO

Neste trabalho apresentam-se técnicas de programação de jogos 2D e 3D. As técnicas 2D são apresentadas com base no MRDX, que é uma biblioteca para programação de jogos proposta por este trabalho. Mostra-se a estrutura de um jogo e os fundamentos envolvidos, como sprites, interação com o cenário e controladores; também são apresentados os tópicos de animação e de edição de mapas. O MRDX, desenvolvido pelo autor, em C/C++ para DirectX, é um conjunto de funções de alto nível que simplificam a criação de jogos, permitindo o desenvolvimento de jogos de forma rápida e econômica; mostrando-se bem adequado para ensino de programação de jogos. As técnicas 3D são apresentadas com base na biblioteca OpenGL, apresentando-se um estudo de caso, com um exemplo didático.

Palavras-chaves : programação de jogos, jogos 2D, DirectX, animação, editor de mapas, motor para jogos, jogos 3D, OpenGL.

ABSTRACT

This work presents techniques for 2D and 3D game programming. The 2D techniques are based on the MRDX, which is a library for game programming, propose by this work. It shows the structure of a game and the fundamentals related, such as sprites, interaction with the scenario and controllers; the topics of animation and maps edition are shown too. The MRDX, developed by the author, in C/C++ for DirectX, is a set of high level functions which simplifies the task of game creation, allowing an economic and fast development of a game, being valuable for teaching game programming. The 3D techniques are present based on the OpenGL library, showing a case study, with a didactic example.

Keywords: game programming, 2D games, DirectX, animation, maps editor, game engine, 3D games, OpenGL.

1 INTRODUÇÃO

Atualmente, em 2004, a indústria de jogos tem um movimento financeiro maior do que a indústria cinematográfica. Ela cresceu de tal forma que para o desenvolvimento de um jogo profissional são necessários centenas de participantes, milhares de dólares e cerca de três anos para a conclusão do projeto. O mercado de jogos concentra-se principalmente nos jogos para PC e para os videogames como o Playstation 2 da Sony, Gamecube da Nintendo e o Xbox da Microsoft. Há também as plataformas alternativas como a Internet e os dispositivos móveis. O livro de Saltzman [SAL2000] é uma boa fonte de consulta para se conhecer as principais empresas atuantes no mercado de jogos.

O Brasil tem se organizado para correr atrás deste disputadíssimo mercado. Existem alguns casos de sucesso comercial como o da empresa Continuum [CON2004] que lançou o jogo Outlive. Alguns eventos de grande porte voltado especificamente para jogos já foram realizados como o Wjogos – Workshop Brasileiro de Jogos e Entretenimento Digital [WJO2004] e o IN2Games – Congresso Internacional de Inovação em Jogos para Computadores [IN22004]. Até cursos universitários têm sido criados para atender esta demanda. Os principais são: Pós-Graduação em Desenvolvimento de jogos para Computadores – Unicenp - PR; Graduação em Design e Planejamento de Games – Anhembi – Morumbi; Curso de Extensão em Desenvolvimento e Design de Jogos 3D – PUC-RJ. O anexo 1 deste trabalho possui mais informações sobre estes cursos.

O autor possui uma página de jogos [ROM2004] onde publica os seus trabalhos. O trabalho principal é o MRDX que é uma biblioteca para desenvolvimento de jogos 2D em C/C++ para DirectX. Ele foi apresentado no 1º Workshop Brasileiro de Jogos e Entretenimento Digital [WJO2004] que ocorreu em Fortaleza-CE no dia 10 de outubro de 2003. O MRDX será a base para o desenvolvimento deste TCC. Todos os programas citados no decorrer deste trabalho estão disponíveis na página de jogos [ROM2004].

A linguagem utilizada nos programas deste TCC é a C/C++. Segundo Hattan [HAT2004], esta ainda é a linguagem padrão usada no desenvolvimento de jogos profissionais para o PC e videogames. O leitor interessado em uma revisão da linguagem, pode consultar uma apostila sobre programação em C/C++ que está disponível na seção de material didático da página de jogos do autor [ROM2004]. Se necessário, o livro de Deitel [DEI2000] possui mais informações sobre a linguagem C/C++.

Os tópicos que compõe este TCC estão descritos a seguir:

- **Conhecendo o MRDX** : Esta seção apresenta todas as funcionalidades do MRDX e como deve ser usado.
- **A Estrutura de um Jogo** : É apresentado um simples jogo desenvolvido com o MRDX para que seja explicado os principais elementos que compõe quase todos os jogos.
- **Técnicas de Jogos 2D** : Comenta-se sobre tópicos comuns em jogos 2D como sprites, interação com o cenário e controladores.
- **Animação** : Expõe fundamentos necessárias para a programação da animação e mostra uma classe que encapsula estas implementações.
- **Editor de Mapas** : Noções de *Tile Based Games* são apresentadas aqui e um simples editor de mapas é desenvolvido.
- **Fundamentos de Jogos 3D** : Visão geral de jogos 3D e comentários sobre OpenGL e movimentação 3D.

2 A HISTÓRIA DOS VIDEOGAMES

A indústria de jogos eletrônicos é bem nova mas evoluiu bastante em pouco tempo. Nesta seção apresentam-se os principais sistemas e gerações de sistemas que marcaram a história dos videogames (está disponível uma versão ilustrada desta história na página de jogos do autor [ROM2004]):

- **Atari 2600 :** O videogame da Atari foi lançado em 1977, mas se popularizou em 1980 quando começou a trazer os jogos de fliperamas como o “Space Invaders” para seu sistema.
- **Microcomputadores :** A revolução dos microcomputadores pessoais foi iniciada pelo Apple e seguido por diversos outros como o ZX-Spectrum, Commodore 64, TRS-80 e MSX. O principal tipo de aplicação destes sistemas eram os jogos.
- **A geração de 8 bits :** A Nintendo entrou no mercado de videogames com o NES (Nintendo Entertainment System) que era um excelente produto a um preço bem acessível. No Brasil havia os compatíveis como o Phantom System e o Dynavision. A Sega veio em seguida com o Master System para concorrer com o NES. No Brasil a Sega é representada pela Tec Toy.
- **A geração de 16 bits :** A Sega saiu na frente na corrida dos 16 bits com o lançamento do Mega Drive. A Nintendo surgiu depois com o Super Nintendo, e fez muito sucesso com o jogo “Street Fighter II” da Capcom. Havia também o Neo Geo da SNK que era bem superior, porém tinha um custo muito elevado.
- **32 bits e o CD-ROM :** A Panasonic lançou o 3DO, mas não obteve sucesso. A Sega investiu no “Saturn”, mas também não teve sucesso. Um dos motivos diz respeito à dificuldade de desenvolvimento. A Sony entrou no mercado com o “Playstation” que teve uma grande aceitação e se tornou um dos videogames mais vendidos da história.
- **A geração de 64 bits :** A Atari pretendia retornar ao mercado de jogos de uma forma triunfal, por isso lançou o “Jaguar”, um videogame de 64 bits, mas que não teve uma boa aceitação no mercado. A Nintendo fez uma aliança com a Silicon Graphics para o desenvolvimento do Nintendo 64, mesmo assim ela perdeu boa parte de seu mercado para o Playstation.

- **Dreamcast :** Por volta de 1999 a Sega iniciou a geração atual dos videogames de 128 bits com o lançamento do “Dreamcast”. Vinha equipado com um modem de 56k e usava o Windows CE para facilitar o desenvolvimento e a conversão de jogos do PC. As vendas no Japão não foram como esperado levando a Sega a anunciar em abril de 2001 o fim do Dreamcast.
- **Playstation 2 :** Em 2000 a Sony lançou o Playstation 2 integrando jogos, filmes em DVD e som digital. Tem como grande vantagem poder executar todos os jogos do Playstation 1. O Playstation 2 conta com o apoio de quase todas as grande produtoras, gerando assim um extenso catálogo de jogos.
- **Gamecube :** O Gamecube é o videogame de 128 bits da Nintendo. Foi feito apenas para jogos e utiliza um minidisco com um padrão DVD próprio para dificultar a pirataria. Atualmente, a Nintendo não está investindo em redes online e não tem um apoio muito grande de outros produtores de jogos.
- **Xbox :** A Microsoft resolveu entrar no mercado e desenvolveu o Xbox. Usa DVD e possui um HD de 10 GB para armazenamento das informações dos jogos. Várias produtoras de peso têm lançado jogos para o Xbox e a Microsoft tem investido bastante em sua rede online chamada “Xbox Live”.

3 CONHECENDO O MRDX

Neste capítulo será demonstrada a forma de utilização do MRDX. O entendimento deste capítulo é importante para o acompanhamento deste TCC, pois os programas desenvolvidos nos capítulos seguintes são baseados no MRDX.

3.1 - Noções preliminares

O desenvolvimento de um jogo envolve vários desafios de programação, entre eles:

- conhecimento profundo de tecnologia de ponta sobre bibliotecas gráficas, de sons, entre outras;
- domínio de técnicas de programação para abordar diversas dimensões ortogonais de um jogo: por exemplo, linha de tempo, ciclo de eventos, manipulação de imagem e som, interação com o usuário;
- complexidade no desenvolvimento de programas grandes.

Com o objetivo de enfrentar de forma sistemática estes desafios de programação é prática corrente, o desenvolvimento de bibliotecas como *motores de jogos*. O uso de uma biblioteca cria um conjunto básico de serviços comuns, permitindo ao programador se concentrar nos aspectos criativos da programação de jogos, dando uma maior qualidade e maior produtividade no seu desenvolvimento.

Neste trabalho apresenta-se uma biblioteca para programação de jogos chamado MRDX, para a plataforma Windows com DirectX. Este motor de jogos tem dois objetivos: (1) facilitar o desenvolvimento de jogos; (2) ser adequado para o ensino de programação de jogos.

O MRDX é definido como um conjunto de primitivas, implementadas em C/C++. O MRDX é implementado como uma biblioteca de funções em dois arquivos principais: o arquivo “mrdx.h” que define os protótipos das funções e as estruturas de dados; e o arquivo “mrdx.cpp” que contém a definição das funções em si. O MRDX foi desenvolvido com o compilador Microsoft Visual C++ 5.0.

O MRDX é mantido com o código fonte aberto para que possa ser estudado e estendido com novas funcionalidades pelos usuários. Informações sobre a implementação do MRDX encontram-se no anexo 2.

O MRDX fornece os seguintes serviços básicos para programação de jogos:

- Suporta arquivos de imagem do tipo BITMAP;
- Suporta arquivos de sons no formato WAV e MIDI;
- Suporta entrada de dados através do teclado, mouse e joystick;
- Automaticamente inicializa o DirectX e configura o modo de vídeo para a resolução de 640x480, tela cheia e 256 cores;
- Oferece uma estrutura de dados chamada IMAGEM que é utilizada para controlar os sprites dos jogos;
- Implementa primitivas para detecção de colisão entre objetos do tipo IMAGEM;

Para usá-lo, deve-se criar um projeto do tipo “Win32 Application”, acrescentar o arquivo “mrdx.cpp” ao projeto e criar um arquivo novo onde estará o código do jogo. Neste novo arquivo devem existir estas declarações:

```
#include “mrdx.h”
void Jogo_Main(void)
{
}
```

Na função “Jogo_Main( )” é que será inserido o código de seu jogo, de uma forma semelhante à função “main( )” de um programa padrão em linguagem C. Uma das diferenças é que a função “Jogo_Main( )” é reexecutada constantemente.

É importante lembrar que para a execução dos programas é preciso ter o DirectX (RunTime) instalado na máquina do usuário. A instalação do DirectX é dividida em duas partes: a) RunTime - É a parte que deve estar presente na máquina do usuário para a execução dos programas; b) SDK - É o kit de desenvolvimento para a criação dos programas usando o DirectX.

Além disso, é preciso configurar o arquivo de projeto para indicar onde devem ser encontrados os arquivos do DirectX (SDK), conforme o anexo 3.

3.2 - Lidando com imagens

Todos os desenhos no MRDX são feitos em uma memória secundária e ao final de cada ciclo do jogo deve-se passar os dados desta memória para a memória de vídeo onde é mantida a imagem que está na tela. Esta operação é feita através do uso da função "Mostrar_Tela( )" que está descrita seguir:

Protótipo: **int Mostrar_Tela(void);**

Comentários: Usada para mostrar na tela o conteúdo da memória secundária onde são feitos os desenhos. Deve ser chamada ao final de cada ciclo.

Exemplo:

```
//faça todos os seus desenhos...
Mostrar_Tela( );
```

No início de cada ciclo do jogo é preciso limpar a tela atual para que possa ser feito o desenho do próximo quadro. A função "Limpar_Tela( )" tem este objetivo.

Protótipo: **int Limpar_Tela(int cor =0);**

Comentários: Limpa a memória secundária. Pode-se passar uma cor como parâmetro para preencher a tela com esta cor.

Exemplo:

```
//Construir o próximo quadro do jogo...
Limpar_Tela( );

//faça todos os seus desenhos...
Mostrar_Tela( );
```

No MRDX foi definida uma estrutura chamada IMAGEM que será usada para controlar seus objetos no jogo. Ela possui a seguinte forma:

```
struct IMAGEM  {
        int x,y;
        int largura;
        int altura;
        int num_quadros;
        int quadro_atual;
        int estado;
        int dados[MAX_DADOS];
        LPDIRECTDRAWSURFACE4 quadros[MAX_QUADROS];
        };
```

Descrição dos campos de IMAGEM:

- **x, y** : Posição da IMAGEM. Esta coordenada é da parte superior esquerda.
- **largura, altura** : Dimensões da IMAGEM.
- **num_quadros** : Quantidade de quadros que esta IMAGEM possui.
- **quadro_atual** : Indica a posição do quadro que está sendo utilizado.
- **estado** : Pode ser ATIVO ou INATIVO. Usado para controlar quais são as imagens que estão ativas no jogo.
- **dados[MAX_DADOS]** : Espaço extra para guardar informações específicas de seu jogo. A constante MAX_DADOS está definida em 32 por ser uma quantidade suficiente para a maioria dos objetos de um jogo.
- **quadros[MAX_QUADROS]** : Local de armazenamento dos quadros de IMAGEM. A constante MAX_QUADROS está definida em 128.

Um programador declara uma variável do tipo IMAGEM desta forma:
IMAGEM jogador;

O MRDX trabalha no modo de vídeo de 640x480 com palheta de 256 cores. Ele reserva as primeiras oito posições (0 a 7) para algumas cores básicas que são : PRETO, BRANCO, VERDE, AZUL, VERMELHO, AMARELO, CINZA, ROXO. Então, não utilize estas posições quando fizer os seus bitmaps. Para utilizar a transparência basta utilizar nos desenhos a cor de índice zero nas partes que não devem ser desenhadas. Todos os bitmaps de um jogo feito no MRDX devem estar usando a mesma palheta de cores, pois apenas uma palheta fica ativa no jogo.

As imagens são obtidas a partir de arquivos bitmaps. É comum que em um único arquivo bitmap haja diversas sub-imagens de mesmo tamanho, como na ilustração a seguir :

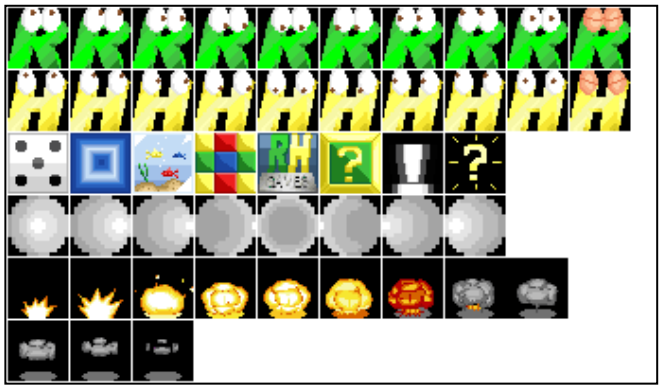

Figura 3.1 – Sub-imagens em um bitmap.

As funções que lidam com os arquivos bitmaps e a estrutura IMAGEM estão descritos a seguir:

Protótipo: **int Carregar_Bitmap(char * nome_arq);**

Comentários: Lê um arquivo do tipo bitmap (sem compressão) de 256 cores e carrega na memória. A cada chamada ele substituirá o bitmap anterior pelo que está sendo carregado no momento.

Exemplo:

Carregar_Bitmap("arte.bmp");

Protótipo: **int Criar_Imagem(IMAGEM &imagem, int largura, int altura, int num_quadros);**

Comentários: Aloca memória para uma variável do tipo IMAGEM. O parâmetro "num_quadros" especifica o número de quadros de imagens que serão utilizados.

Exemplo:

IMAGEM jogador;
Criar_Imagem(jogador, 32, 32, 4);
// tamanho = 32x32 e num_quadros = 4

Protótipo: **int Destruir_Imagem(IMAGEM &imagem);**

Comentários: Libera a memória alocada por uma variável do tipo IMAGEM. Faça isso quando estiver finalizando seu programa.

Exemplo:

Destruir_Imagem(jogador);

Protótipo: **int Carregar_Quadro(IMAGEM &imagem, int quadro, int cx, int cy);**

Comentários: Carrega um quadro em uma variável IMAGEM a partir de um bitmap previamente carregado na memória. O parâmetro "quadro" indica a posição em que deve ser carregado o quadro na variável IMAGEM. O bitmap deve estar com imagens do mesmo tamanho. Os parâmetros "cx" e "cy" indica as coordenadas no bitmap do local onde está o quadro desejado. Por exemplo, se tivermos um bitmap com dois quadros de imagem, um ao lado do outro, as coordenadas desses quadros serão: (0,0) e (1,0).

Exemplo:

Carregar_Bitmap("arte.bmp");
Criar_Imagem(jogador, 32, 32, 2); //2 quadros
Carregar_Quadro(jogador, 0, 0, 0); //quadro 0. posição (0,0)
Carregar_Quadro(jogador, 1, 1, 0); //quadro 1. posição (1,0)

Protótipo: **int Desenhar_Imagem(IMAGEM &imagem);**

Comentários: Desenha uma IMAGEM na tela. A função irá usar a posição (x, y) e o quadro_atual da IMAGEM para o desenho.

Exemplo:

Desenhar_Imagem(jogador);

Protótipo: **int Colisao_Imagens(IMAGEM &imagem1, IMAGEM &imagem2);**

Comentários: Retorna '1' se as duas IMAGENS estiverem se sobrepondo e '0' se não.

Exemplo:

if(Colisao_Imagens(jogador, bomba)) //fim de jogo...

3.3 - Entrada de dados

A entrada de dados em um jogo é feita através de três dispositivos : teclado, mouse e joystick. A leitura do teclado tem como objetivo simplesmente o de saber se uma determinada está sendo pressionada ou não. Para cada tecla há um código associado.

Protótipo: **int Testar_Tecla(int dik_cod);**

Comentários: Retorna '1' se a tecla passada como parâmetro estiver sendo pressionada. Os códigos das teclas são:

Códigos	Descrição
DIK_UP	Seta para cima
DIK_DOWN	Seta para baixo
DIK_LEFT	Seta para esquerda
DIK_RIGHT	Seta para direita
DIK_ESCAPE	Esc
DIK_0 - 9	0 a 9 do teclado principal
DIK_TAB	Tab
DIK_A - Z	Letras de A a Z
DIK_RETURN	Enter
DIK_LCONTROL	Control esquerdo
DIK_RCONTROL	Control direito
DIK_LSHIFT	Shift esquerdo
DIK_RSHIFT	Shift direito
DIK_LMENU	Alt esquerdo
DIK_RMENU	Alt direito
DIK_SPACE	Barra de espaço
DIK_F1 - F12	Teclas de funções F1 a F12
DIK_NUMPAD0 - 9	0 a 9 no teclado numérico
DIK_NUMPADENTER	Enter no teclado numérico

Exemplo:

if(Testar_Tecla(DIK_RIGHT)) jogador.x += 4; / /mover para direita

O mouse funciona de uma forma diferente. Para poder monitorar seu movimento não se trabalha com coordenadas absolutas, pois não há um ponto de referência que irá servir como origem, mas sim com coordenadas relativas, ou seja, será recebida como entrada de dados a variação da posição do mouse em relação a sua última posição.

Protótipo: **int Mouse_Xvar(void);**

Comentários: Retorna a variação da posição do mouse no eixo X desde o último ciclo do jogo.

Exemplo:
jogador.x += Mouse_Xvar( ); // move jogador no eixo de X

Protótipo: **int Mouse_Yvar(void);**

Comentários: Retorna a variação da posição do mouse no eixo Y desde o último ciclo do jogo.

Exemplo:
jogador.y += Mouse_Yvar( ); // move jogador no eixo de Y

Protótipo: **int Mouse_Bot_Esquerdo(void);**

Comentários: Retorna '1' se o botão esquerdo do mouse estiver pressionado.

Exemplo:
if(Mouse_Bot_Esquerdo( )) ...

Protótipo: **int Mouse_Bot_Direito(void);**

Comentários: Retorna '1' se o botão direito do Mouse estiver pressionado.

Exemplo:
if(Mouse_Bot_Direito( )) ...

O MRDX permite que seja verificado se há algum joystick disponível no computador. Na leitura dos movimentos no joystick é assumido um valor booleano para indicar se o jogador está pressionando em alguma das direções. As seguintes funções gerenciam a entrada do joystick:

Protótipo: **int Joy_Cima(void);**

Comentários: Retorna '1' se o joystick estiver sendo pressionado para cima.

Exemplo:

if(Joy_Cima( )) jogador.y -= 4; //move jogador para cima

Protótipo: **int Joy_Baixo(void);**

Comentários: Retorna '1' se o joystick estiver sendo pressionado para baixo.

Exemplo:

if(Joy_Baixo( )) jogador.y += 4; //move jogador para baixo

Protótipo: **int Joy_Esquerda(void);**

Comentários: Retorna '1' se o joystick estiver sendo pressionado para esquerda.

Exemplo:

if(Joy_Esquerda( )) jogador.x -= 4; //move jogador para esquerda

Protótipo: **int Joy_Direita(void);**

Comentários: Retorna '1' se o joystick estiver sendo pressionado para direita.

Exemplo:

if(Joy_Direita( )) jogador.x += 4; //move jogador para direita

Protótipo: **int Joy_Bot(int bot);**

Comentários: Retorna '1' se o botão do joystick que foi passado como parâmetro estiver sendo pressionado. O primeiro botão tem o valor de 0.

Exemplo:

if(Joy_Bot(0)) ...

Protótipo: **int Joy_Existe(void);**

Comentários: Retorna '1' se existir um joystick conectado ao computador.

Exemplo:

if(Joy_Existe( )) ...

3.4 - Efeitos sonoros e músicas

Os arquivos de som do tipo *wave* são muito utilizados para a produção de efeitos sonoros em jogos. Eles também podem ser utilizados para músicas, porém o formato *wave* ocupa muita memória, sendo mais recomendado o formato *midi* para as músicas.

Para trabalhar com os arquivos *wave*, o MRDX mantém uma lista interna de todos os sons que foram carregados. Para acessá-los, é preciso que seja guardado um identificador de cada som, que é gerado no momento que o som é armazenado.

Protótipo: **int Carregar_Wav(char *nome_arq);**

Comentários: Carrega um arquivo de som no formato wave e retorna um identificador que será usado para manuseá-lo. O som wave deve ter o formato de 11 KHz, 8-bit, mono.

Exemplo:

```
int som_wav;
som_wav = Carregar_Wav("bip.wav");
```

Protótipo: **int Tocar_Wav(int id, int modo = 0);**

Comentários: Toca um som wave que foi previamente carregado. Para que o som fique repetindo passe como segundo parâmetro a palavra REPETIR.

Exemplo:

```
Tocar_Wav(som_wav);
Tocar_Wav(som_wav, REPETIR);
```

Protótipo: **int Parar_Wav(int id);**

Comentários: Pára um som wave.

Exemplo:

```
Parar_Wav(som_wav);
```

Protótipo: **int Parar_Todos_Wav(void);**

Comentários: Pára todos os sons wave.

Exemplo:

```
Parar_Todos_Wav( );
```

Protótipo: **int Wav_Tocando(int id);**

Comentários: Retorna '1' se o som wave estiver tocando.

Exemplo:
if(Wav_Tocando(som_wav)) ...

Os arquivos de sons no formato midi são uma excelente opção para o uso de músicas em jogos, pois consegue armazenar músicas com vários minutos e dificilmente chega a ocupar 100Kb de memória. O MRDX trata os arquivos midis da mesma maneira que os arquivos wave, ou seja, é preciso armazenar um identificador para poder acessar as músicas posteriormente.

Protótipo: **int Carregar_Midi(char *nome_arq);**

Comentários: Carrega um arquivo de som no formato MIDI e retorna um identificador que será usado para manuseá-lo.

Exemplo:
```
int som_mid;
som_mid = Carregar_Midi("musica.mid");
```

Protótipo: **int Tocar_Midi(int id);**

Comentários: Toca um som MIDI que foi previamente carregado.

Exemplo:
```
Tocar_Midi(som_mid);
```

Protótipo: **int Parar_Midi(int id);**

Comentários: Pára um som MIDI.

Exemplo:
```
Parar_Midi(som_mid);
```

Protótipo: **int Midi_Tocando(int id);**

Comentários: Retorna '1' se o som MIDI estiver tocando.

Exemplo:
if(Midi_Tocando(som_mid)) ...

3.5 - Recursos adicionais

Para fazer o controle da velocidade do jogo, o MRDX disponibiliza as funções “Iniciar_Relogio( )” e “Esperar_Relogio( )”. Antes da execução do código principal, deve ser chamada a função “Iniciar_Relogio( )” para que seja inicializado um contador interno do MRDX, no final do código será chamada a função “Esperar_Relogio( )” e passado como parâmetro a quantidade de milisegundos que um ciclo deve durar.

Protótipo: **int Iniciar_Relogio(void);**

Comentários: Inicia um contador interno do MRDX. É usado em conjunto com a função Esperar_Relogio( ).

Exemplo:

```
Iniciar_Relogio( );
```

Protótipo: **int Esperar_Relogio(unsigned int tempo_espera);**

Comentários: Deve-se enviar como parâmetro o tempo em milisegundos que a função deve esperar. Este tempo é contado a partir da última chamada de Iniciar_Relogio( ). Estas funções são usadas para que o jogo seja mantido em uma velocidade fixa. Um valor comum é de 40 milisegundos, que é equivalente a 25 quadros por segundo.

Exemplo:

```
Iniciar_Relogio( );
//Processar jogo...
Esperar_Relogio(40);
```

O MRDX disponibiliza duas outras funções de desenho, uma para desenhar retângulos e a outra para escrever textos na tela.

Protótipo: **int Desenhar_Retangulo(int x1, int y1, int x2, int y2, int cor);**

Comentários: Desenha um retângulo que começa em (x1, y1) e termina em (x2, y2), preenchido com a cor especificada.

Exemplo:

```
Desenhar_Retangulo(100,100,200,200,VERMELHO);
```

Protótipo: **int Escreve_Texto(char *texto, int x, int y, int cor);**

Comentários: Escreve o texto na posição especificada. Se for preciso escrever valores de variáveis, utiliza-se a função sprintf( ) para construir o texto esperado, depois passa-se este texto como parâmetro para “Escreve_Texto( )”.

Exemplo:

```
Escreve_Texto("Funciona",100,100,BRANCO);
```

```
//Para escrever variáveis use:
char texto[80];
int pontos = 10;
sprintf(texto, "Pontos = %d", pontos);
Escreve_Texto(texto, 100,100,BRANCO);
```

Finalmente, há uma função para finalizar o jogo e outra que lida com números aleatórios, um recurso muito utilizado em jogos.

Protótipo: **int Finalizar_Jogo(void);**

Comentários: Encerra a aplicação. É aconselhável que após a chamada a esta função o programa saia de Jogo_Main( ) utilizando um "return", pois Finalizar_Jogo( ) não encerra imediatamente a aplicação, o que ela faz é mandar uma mensagem para o Windows solicitando o fim da aplicação.

Exemplo:

```
Finalizar_Jogo( );
return;
```

Protótipo: **int Aleatorio(int min, int max);**

Comentários: Retorna um número aleatório entre "min" e "max" inclusive.

Exemplo:

```
int valor;
valor = Aleatorio(1,6);
```

A seguir está um exemplo de código utilizando o MRDX que cria uma variável do tipo IMAGEM, carrega um quadro de imagem, faz com que essa IMAGEM se mova de acordo com os movimentos do mouse, em seguida ele carrega um som WAV que é tocado quando o botão esquerdo do mouse for pressionado. Se for pressionada a tecla ESC o jogo é finalizado.

```
1   #include "mrdx.h"
2   IMAGEM jogador;
3   int inicio = 1;
4   int som_wav;
5   void Jogo_Main(void) {
6   if( inicio == 1 ) {
7     inicio = 0;
8     Carregar_Bitmap("arte.bmp");
9     Criar_Imagem(jogador, 32, 32, 1);
10    Carregar_Quadro(jogador,0,0,0);
11    som_wav = Carregar_Wav("bip.wav");
12  }
13  if( Testar_Tecla(DIK_ESCAPE) ) {
14    Destruir_Imagem(jogador);
15    Finalizar_Jogo( );
```

```
16   return;
17  }
18  Iniciar_Relogio( );
19  Limpar_Tela( );
20  jogador.x += Mouse_Xvar( );
21  jogador.y += Mouse_Yvar( );
22  if(Mouse_Bot_Esquerdo())
23    Tocar_Wav(som_wav);
24  Desenhar_Imagem(jogador);
25  Mostrar_Tela( );
26  Esperar_Relogio(40);
27  }
```

Código 3.1 – Exemplo de uso do MRDX

As linhas 2 até 4 definem três variáveis. Na linha 2, a variável "jogador" do tipo IMAGEM representa um *sprite* que é movido e desenhado na tela. Na linha 3, a variável "inicio" é usada para garantir que o bloco de inicialização do jogo seja executado uma única vez. Na linha 4, a variável "som_wav" define um identificador que será usado para manipular um som de formato WAV.

Os códigos que estão entre as linhas 6 e 12 representam o bloco de inicialização. Nesta parte é que são carregados na memória todas as imagens e sons que serão usados durante o jogo. O comando condicional "if(inicio == 1)" é usado para que o bloco de inicialização seja executado uma única vez. O comando da linha 8 serve para carregar um bitmap na memória. Os próximos dois comandos configuram a variável "jogador" (que é do tipo IMAGEM) com o tamanho de 32x32 pixels e um quadro de imagem que é carregado a partir do bitmap que está na memória. Na linha 11 é carregado um arquivo de som no formato WAV e o identificador que será usado para manipulá-lo é guardado na variável "som_wav".

As linhas de 13 a 17 são o bloco de finalização que é chamado quando a tecla ESC é pressionada. Antes de encerrar o programa é preciso liberar a memória utilizada pelas variáveis do tipo IMAGEM, isto é feito na linha 14. O comando "Iniciar_Relogio( )", que está à linha 18, é usado em conjunto com "Esperar_Relogio( )", encontrado na linha 26, para determinar a velocidade que o jogo irá rodar. Neste exemplo é usado um valor de espera de 40 milisegundos para um quadro. Isto significa que a velocidade do jogo é de 25 quadros-por-segundo (pois 40x25 = 1000 milisegundos ou 1 segundo).

As linhas de 20 a 23 fazem com que o jogador se mova de acordo com os movimentos do mouse e que um som seja tocado quando o botão do mouse for pressionado. Finalmente na linha 24 o jogador é desenhado e em seguida todos os desenhos feitos durante o quadro atual são mostrados na tela.

4 A ESTRUTURA DE UM JOGO

Neste capítulo examina-se a estrutura de um jogo simples desenvolvido com o MRDX. Este jogo encontra-se na pasta "progjogos \ ex1" do MRDX. Uma imagem da tela do jogo pode ser vista na figura 4.1.

À medida que os jogos crescem em complexidade, faz-se necessário uma maior formalidade na definição da estrutura de um jogo. O livro de Rollings [ROL2000] propõe abordagens de Engenharia de Software voltadas para o desenvolvimento de jogos.

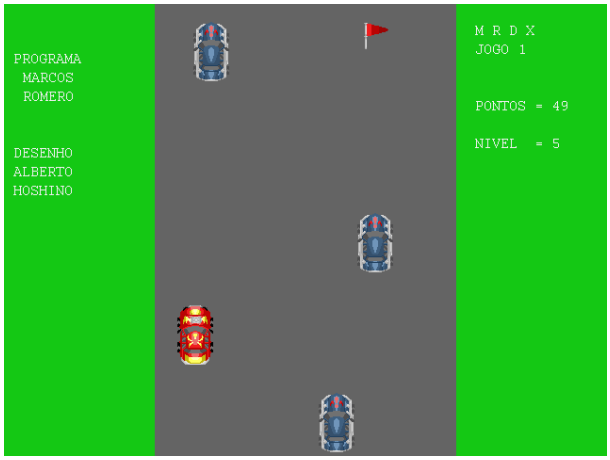

Figura 4.1 – Um jogo usando o MRDX.

4.1 – O jogo e sua função principal

Neste jogo, o jogador controla um carro e deve pegar as bandeiras que aparecem na pista e desviar dos outros carros. O jogador, os carros e a bandeira são variáveis do tipo IMAGEM definidas da seguinte forma:

```
#define MAX_CARROS      3
IMAGEM carros[MAX_CARROS];
IMAGEM jogador;
IMAGEM bandeira;
```

Os carros são definidos na forma de matriz para que possam aparecer diversos carros simultaneamente na pista.

A função principal do jogo está listada no código 4.1. A partir dela é que são chamadas as outras funções. É preciso efetuar a inicialização do jogo (carregar imagens e sons) na primeira vez que é executada. Isto é feito na função Jogo_Inicio( ) e é usada uma variável chamada "inicio" para que esta função seja chamada apenas uma vez. Depois disso é iniciado o relógio do MRDX em cada ciclo. Se a tecla ESC for pressionada então é chamada a

função Jogo_Fim( ) que irá executar os procedimentos de saída do jogo. É feito um teste condicional para verificar se a música (midi) está tocando, senão estiver então começa a tocar.

O cenário de fundo é feito apenas limpando a tela com a cor VERDE e então desenhado um retângulo CINZA no centro. Depois é escrito algumas informações na tela com a função “Desenha_Dados( )”. Então é feito um teste para ver se o menu está ativo. Se o menu não estiver ativo então o jogo está rodando, e são feitas as chamadas às funções que controlam as variáveis do tipo IMAGEM do jogo que são os carros, a bandeira e o jogador. Por último é mostrada a tela onde foram feitos os desenhos e é feita uma pausa para sincronizar o tempo do jogo.

```
void Jogo_Main(void)
{
	if(inicio) {
		inicio = 0;
		Jogo_Inicio();
	}
	Iniciar_Relogio();

	if(Testar_Tecla(DIK_ESCAPE)) {
		//encerrar o jogo
		Jogo_Fim();
		return;
	}
	//tocar música
	if(!Midi_Tocando(musica_mid)) {
		Tocar_Midi(musica_mid);
	}
	//desenhar pista
	Limpar_Tela(VERDE);
	Desenhar_Retangulo(160,0,479,479,CINZA);

	Desenha_Dados();

	if(menu) {
		Menu();
	}
	else {
		// está no jogo, processar objetos
		Bandeira();
		Carros();
		Jogador();
	}
	Mostrar_Tela();
	Esperar_Relogio(40); //40 milisegundos -> 25 quadros por segundo
}
```

Código 4.1 – Função “Jogo_Main( )”

4.2 – Inicialização do jogo

A função “Jogo_Inicio( )”, listada no código 4.2, é responsável por todo o processo de inicialização. Nela são carregados as imagens e sons que serão usados no jogo. As imagens do jogador e dos carros encontram-se no arquivo "carros.bmp". Para carregar este arquivo na memória é utilizado a função “Carregar_Bitmap( )”. Em seguida é criada a IMAGEM jogador de tamanho 40x64 com espaço para um quadro. Este quadro é copiado da posição (0,0) do bitmap previamente carregado e armazenado na estrutura IMAGEM através da função “Carregar_Quadro( )”.

Os carros do jogo são mantidos em uma matriz. Eles têm o mesmo tamanho do jogador. O quadro deles está na posição (1, 0) do bitmap. A imagem da bandeira está em um arquivo bitmap diferente, então teremos que carregar este outro arquivo na memória. O tamanho da bandeira é de 32x32. Há apenas dois sons neste jogo. Um bip no formato WAV e uma música no formato MIDI. São criadas duas variáveis inteiras no começo do arquivo para armazenar os identificadores destes sons. Por último é atribuído o valor '1' à variável "menu" para indicar que o menu está ativo.

```
void Jogo_Inicio(void)
{
        int i;

        Carregar_Bitmap("carros.bmp");
        Criar_Imagem(jogador,40,64,1);
        Carregar_Quadro(jogador,0,0,0);

        for(i=0; i< MAX_CARROS; i++)    {
                Criar_Imagem(carros[i],40,64,1);
                Carregar_Quadro(carros[i],0,1,0);
        }

        Carregar_Bitmap("bandeira.bmp");
        Criar_Imagem(bandeira,32,32,1);
        Carregar_Quadro(bandeira,0,0,0);

        //carregar som
        musica_mid = Carregar_Midi("arpe.mid");
        bip_wav = Carregar_Wav("bip.wav");

        menu = 1;  // ir p/ menu
}
```

Código 4.2 – Função “Jogo_Inicio( )”

4.3 Etapas Adicionais

Existem outras etapas do jogo que são representadas por funções, neste exemplo mostra-se a finalização, o menu e uma função para reinicializar os dados do jogo quando uma nova partida for solicitada.

A função do código 4.3 é responsável pela finalização. Há a liberação da memória alocada pelas variáveis do tipo IMAGEM através da função “Destruir_Imagem( )”. Depois é chamada a função do MRDX que finaliza o jogo.

```
void Jogo_Fim(void)
{
	//liberar memória das imagens
	Destruir_Imagem(jogador);
	Destruir_Imagem(bandeira);
	for(i=0; i< MAX_CARROS; i++)
	{
		Destruir_Imagem(carros[i]);
	}
	Finalizar_Jogo();
}
```

Código 4.3 – Função “Jogo_Fim( )”

A função “Menu( )” (código 4.4) apenas escreve na tela o menu e testa se o jogador pressionou ENTER para começar uma nova partida.

```
void Menu(void)
{
	Escreve_Texto("ENTER - Iniciar",200,100,BRANCO);
	Escreve_Texto("ESC   - Sair",200,150,BRANCO);

	if(Testar_Tecla(DIK_RETURN))
	{
		menu = 0;
		Novo_Jogo();
	}
}
```

Código 4.4 – Função “Menu( )”

A função “Novo_Jogo( )” (código 4.5) reinicia os valores do jogo para uma nova partida. Coloca o estado da bandeira e dos carros em INATIVO, move o jogador para posição inicial e atribui os valores iniciais de outras variáveis do jogo.

```
void Novo_Jogo(void)
{
        int i;

        //iniciar valores
        bandeira.estado = INATIVO;
        for( i=0; i < MAX_CARROS; i++)
        {
                carros[i].estado = INATIVO;
        }
        jogador.x = 300;
        jogador.y = 400;
        pontos = 0;
        dist_carro=0;
        vel_carro = 4;
        nivel=1;
}
```

Código 4.5 – Função "Novo_Jogo( )"

4.4 Os elementos do jogo

A seguir estão as funções que controlam os elementos do jogo, tais como : jogador, carros e bandeiras.

A função "Bandeira( )" (código 4.6) controla o movimento da bandeira. No começo é feito um teste se a bandeira está no estado INATIVO, se estiver então o estado passa para ATIVO e ela retorna ao topo da tela em uma posição X aleatória. O movimento dela é reto para baixo, a sua velocidade foi definida em uma constante chamada BAND_VEL. Se ela passar da base da tela então seu estado passa para INATIVO. No final da função ela é desenhada.

```
void Bandeira(void)
{
        if( bandeira.estado IGUAL INATIVO) {
          bandeira.estado = ATIVO;
          bandeira.x = Aleatorio(160, 479 - bandeira.largura);
          bandeira.y = - bandeira.altura;
        }
        bandeira.y += BAND_VEL;

        if(bandeira.y > 479)  {
          bandeira.estado = INATIVO;
        }
        Desenhar_Imagem(bandeira);
}
```

Código 4.6 – Função "Bandeira( )"

A função “Carros( )” (código 4.7) controla os carros dos oponentes. Ela está dividida em duas partes. Na primeira é feito um teste para ver se deve iniciar um novo carro. Na segunda parte são processados todos os carros que estiverem ativos.

Há uma variável (dist_carro) que controla a distância do último carro para o topo da tela. Quando essa distância for maior que 200 então é iniciado um outro carro. Para isso procura-se na matriz "carros[ ]" um elemento que esteja INATIVO para então iniciá-lo no topo da tela em uma posição X aleatória.

Para processar todos os carros é utilizado um laço "for" que percorre toda a matriz dos carros. Se o carro estiver ATIVO então ele é processado. A velocidade vertical dos carros está armazenada na variável "vel_carro" que altera de acordo com o nível do jogo. Quanto ao seu movimento horizontal, ele pode está indo RETO, para ESQUERDA ou para DIREITA. Estas palavras de direções foram definidas para facilitar o entendimento do código. Ele mantém uma direção por um período de tempo, quando este tempo acaba então seleciona outra direção e outro período de tempo aleatoriamente. Para armazenar estes dados extras foi utilizado a matriz "dados[MAX_DADOS]" da estrutura IMAGEM. Se o carro tocar na borda da pista então é invertida a direção que ele estava. Se o carro passar da base da tela então seu estado passa para INATIVO. No final o carro é desenhado.

```
void Carros(void)
{
	int i;
	dist_carro += vel_carro;
	if(dist_carro > 200)   {
		dist_carro = 0;

		//inicia um carro
		//procurar um disponível
		i=0;
		while((carros[i].estado IGUAL ATIVO) E (i < MAX_CARROS)) i++;

		if( i < MAX_CARROS) {
			//inicia o carro
			carros[i].estado = ATIVO;
			carros[i].x = Aleatorio(160,479-carros[i].largura);
			carros[i].y = - carros[i].altura;
			carros[i].dados[TEMPO] = 0;
		}
	}
```

```
        //processar os carros
        for(i=0; i <MAX_CARROS; i++)
        {
                if(carros[i].estado IGUAL ATIVO)
                {
                        carros[i].y += vel_carro;
                        carros[i].dados[TEMPO]--;
                        if(carros[i].dados[TEMPO]<0)
                        {
                                carros[i].dados[TEMPO] = 10 * Aleatorio(1,5);
                                // 0 - reto; 1 - esquerda ; 2 direita
                                carros[i].dados[DIR] = Aleatorio(0,2);
                        }

                        if(carros[i].dados[DIR] == DIREITA)
                        {
                                carros[i].x+=vel_carro;
                                if(carros[i].x > (479 - carros[i].largura))
                                {
                                        carros[i].dados[DIR]= ESQUERDA;
                                }
                        }
                        else //esquerda
                        if(carros[i].dados[DIR] == ESQUERDA)
                        {
                                carros[i].x-=vel_carro;

                                if(carros[i].x < 160) carros[i].dados[DIR]= DIREITA;
                        }

                        if(carros[i].y > 479) carros[i].estado = INATIVO;

                        Desenhar_Imagem(carros[i]);
                }
        }
}
```

Código 4.7 – Função “Carros( )”

A função “Jogador( )” processa os movimentos do jogador. Consiste em três partes: ler os comandos do jogador, testar colisão com a bandeira e testar colisão com os carros. No final o jogador é desenhado.

O jogador pode ser controlado pelo teclado ou pelo joystick. É feito o teste das teclas que controlam os movimentos nas quatro direções. Além disso, não é permitido ao jogador sair da pista. A velocidade do jogador está definida em uma constante chamada JOG_VEL. Como exemplo temos no código 4.8 o teste de movimento para cima.

```
if( Testar_Tecla(DIK_UP) OU Joy_Cima( ) )
    {
        jogador.y - = JOG_VEL;
        if(jogador.y < 0) jogador.y = 0;
    }
```

Código 4.8 – Movimento para cima do jogador

Se o jogador colidir com uma bandeira então aumenta o número de bandeiras pegas, aumenta os pontos de acordo com o nível, coloca a bandeira no estado INATIVO e toca um som. A cada cinco bandeiras pegas aumenta o nível e a velocidade dos carros. Se a colisão for com alguns dos carros então a partida encerra e o jogo retorna para o menu. O teste de colisão só é feito se os carros estiverem no estado ATIVO (código 4.9).

```
if(Colisao_Imagens(jogador, bandeira))
    {
        n_bandeiras++;
        pontos+=nivel;
        bandeira.estado = INATIVO;
        Tocar_Wav(bip_wav);

        if((n_bandeiras % 5 ) IGUAL 0)
        {
            nivel++;
            vel_carro++;
        }
    }

    for(i=0; i< MAX_CARROS; i++)
    {
        if(carros[i].estado IGUAL ATIVO)
        {
            if(Colisao_Imagens(jogador, carros[i]))
            {
                //fim de jogo
                menu = 1;
                return;
            }
        }
    }
```

Código 4.9 – Testes de colisões do jogador

5 TÉCNICAS DE JOGOS 2D

Neste capítulo examina-se o código de outro jogo que se encontra na pasta "progjogos \ ex2" do MRDX. O jogo é baseado em um labirinto. Deve-se pegar os "bônus" que aparecem na tela e evitar ser pego pelos monstros que se movem pelo labirinto. Ele tem a mesma estrutura do jogo anterior, então são comentadas apenas as características específicas deste jogo (figura 5.1) e técnicas gerais usadas em jogos 2D. Para mais informação sobre técnicas de jogos 2D, sugere-se o livro de Gruber [GRU1994].

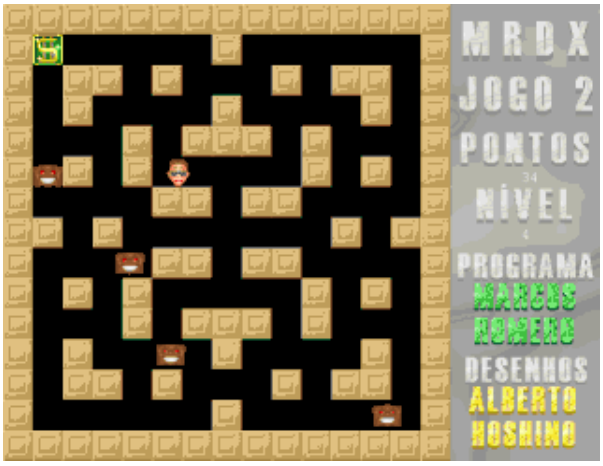

Figura 5.1 – Um jogo de labirinto com o MRDX

5.1 – Sprites

Nos jogos 2D é muito utilizado o termo *sprite*. *Sprites* são os personagens que aparecem no jogo, que interagem uns com os outros e com o cenário. Eles podem correr, pular, voar, nadar, lutar e atirar. Geralmente, o jogo fornece um *sprite* que representa o personagem principal controlado pelo jogador. Os outros *sprites* podem ser monstros e inimigos que perseguem o jogador, ou objetos que o jogador pode pegar, tal como moedas e armas.

Em vários jogos, diferentes *sprites* são introduzidos à medida que o jogador vai avançando no jogo, de tal forma que prende a atenção do jogador por estar sempre encontrando novidades. Um jogo é dividido em várias fases, e cada fase tem um tema predominante que irá determinar o conjunto de *sprites* usados, por exemplo, podem-se ter fases ambientadas no deserto, floresta, cavernas e oceano.

A parte mais desafiadora de programar os sprites está relacionada à animação. Um *sprite* pode executar diversos tipos de movimentos que devem ser representados visualmente para dar mais realismo ao jogo. Desta forma, a estrutura de dados que representa os *sprites* deve permitir que vários quadros sejam armazenados para que seja possível escolher no decorrer do jogo qual o quadro que representa o estado atual do *sprite.* O capítulo 6 irá demonstrar formas de programação para a animação de sprites.

No jogo de exemplo, o personagem principal possui 4 quadros que são alternados durante a partida para dar a impressão de animação. O código 5.1 demonstra como foi executada a operação de carregamento dos 4 quadros que estão armazenados em um único bitmap chamado “arte.bmp”.

```
Carregar_Bitmap("arte.bmp");

Criar_Imagem(jogador,32,32,4);
for(i=0; i< 4; i++)
{
        Carregar_Quadro( jogador, i, i, 0);
}
```

Código 5.1 – Carregando vários quadros para um sprite.

5.2 – Interação com o cenário

Os sprites geralmente interagem com o cenário em um jogo, mas para tornar isto possível, é preciso armazenar informações extras que indicam posições onde os sprites podem se mover, obstáculos e áreas que causam dano aos sprites.

Uma estrutura bem adequada para armazenar informações de um cenário 2D é uma matriz bidimensional, onde cada célula representa uma posição do cenário do jogo. No exemplo temos um labirinto que é representado por um matriz de tamanho 15 x 15. Cada célula pode armazenar apenas dois valores: 0 ou 1. O valor '1' indica que há um bloco nessa posição, o '0' indica caminho.

O preenchimento do mapa foi feito direto no código. Mas seria ideal construir um simples editor para fazer os mapas, este tópico será abordado no capítulo 7.

A função “Desenha_Mapa( )” (código 5.2) desenha o mapa na tela, sendo que uma célula da matriz representa uma posição com tamanho de 32x32 pixels na tela:

```
void Desenha_Mapa(void)
{
 int i,j;
 for(i=0; i<15; i++)
        for(j=0; j < 15; j++)
        {
                if(mapa[i][j])
                {
                        mapa_im.x = j * 32;
                        mapa_im.y = i * 32;
                        Desenhar_Imagem(mapa_im);
                }
        }
}
```

Código 5.2 – Função “Desenha_Mapa( )”

O código de movimento do jogador deverá levar em conta a direção que se deseja ir e testar se não há um obstáculo que impeça o movimento. O código 5.3 é a parte que faz o teste de movimento quando o jogador quer se mover para cima. As variáveis “mapa_x” e “mapa_y” representam a célula da matriz que corresponde ao labirinto. No movimento para cima, é preciso testar se estar livre tanto o lado esquerdo quanto o lado direito do *sprite* para poder ser executado o movimento. A constante “JOG_VEL” guarda o valor da velocidade na qual o jogador deve se mover a cada ciclo.

```
int mapa_x, mapa_y;

if(Testar_Tecla(DIK_UP) OU Joy_Cima( ) )
{
        mapa_y = jogador.y / 32; //topo

        mapa_x = jogador.x  / 32; //lado esquerdo

         if(mapa[mapa_y][mapa_x] = = 0)
         {
                mapa_x = ( jogador.x + 31 ) / 32; //lado direito

                if(mapa[mapa_y][mapa_x] = = 0)
                {
                                jogador.y = jogador.y - JOG_VEL;
                }
         }
}
```

Código 5.3 – Movimento para cima do jogador no labirinto

5.3 – Controladores

Além da programação dos *sprites* individuais em um jogo é geralmente necessário o uso de controladores que servem para gerenciar um grupo de *sprites*. Eles podem ser usados para determinar quando um *sprite* deve aparecer e também podem ter informações que não pertencem a uma instância em particular. Este conceito é muito utilizado em um ambiente orientado a objetos. Um exemplo seria para organizar uma formação de aeronaves. Se for escolhida uma espécie de *sprite* líder que os outros deverão seguir, haverá problema se este líder for eliminado no jogo, pois os outros *sprites* ficarão literalmente perdidos, pois não terão mais acesso aos dados de referência que se encontrava no líder. O ideal é manter estes dados de referência ao encargo de um controlador.

Como o exemplo do jogo de labirinto é bem simples, há uma função chamada "Monstros( )" (código 5.4) que faz o papel de controlador de todos os monstros do jogo. O controlador acessa a matriz que armazena os monstros e executa o código de movimento de cada um deles. O número de monstros presentes na tela é igual ao número do nível do jogo. Eles apenas se movimentam pelo labirinto de uma forma bem simples. Basicamente ele escolhe uma direção para se mover, quando chega em uma célula ele tende a manter a mesma direção se estiver livre, mas há uma chance que ele mude de direção. Quando houver mudança de direção, ele escolhe uma aleatoriamente e testa se está livre o caminho, se não então escolhe outra direção até encontrar uma livre. Para facilitar o código, foram definidas constantes para as direções: ESQUERDA, DIREITA, CIMA, BAIXO. A velocidade do monstro se encontra em MON_VEL. A direção atual de um monstro é armazenada no campo "dados[DIR]".

```
void Monstros(void)
{
        int i, dir;
        int mapa_x, mapa_y;
        int caminho_livre;
        //o número de monstros é igual ao nivel
        for( i=0; i < nivel; i++) {
                dir = monstros[i].dados[DIR];
                //mover monstro
                switch(dir){
                        case ESQUERDA     : monstros[i].x -= MON_VEL; break;
                        case DIREITA      : monstros[i].x += MON_VEL; break;
                        case CIMA         : monstros[i].y -= MON_VEL; break;
                        case BAIXO        : monstros[i].y += MON_VEL; break;
                }
```

```
                //se monstro estiver totalmente em uma célula então
                //decidir próxima direção
                if( ( monstros[i].x % 32 IGUAL 0) E
                   ( monstros[i].y % 32 IGUAL 0) )
                {
                        caminho_livre = 0;

                        mapa_x = monstros[i].x / 32;
                        mapa_y = monstros[i].y / 32;

                        //talvez mudar direção
                        if(Aleatorio(1,10) IGUAL 10) dir = Aleatorio(0,3);

                        do
                        {
                         switch(dir)
                         {
                           case ESQUERDA:
                                if(mapa[mapa_y][mapa_x - 1] IGUAL 0)    {
                                        caminho_livre = 1;
                                }
                                break;
                          case DIREITA:
                                if(mapa[mapa_y][mapa_x + 1] IGUAL 0)  {
                                        caminho_livre = 1;
                                }
                                break;
                          case CIMA:
                                if(mapa[mapa_y - 1][mapa_x] IGUAL 0) {
                                        caminho_livre = 1;
                                }
                                break;
                          case BAIXO:
                                if(mapa[mapa_y + 1][mapa_x] IGUAL 0)  {
                                        caminho_livre = 1;
                                }
                                break;
                         }

                          //se não está livre, tentar outra direção
                         if(caminho_livre IGUAL 0) dir = Aleatorio(0,3);

                        } while(caminho_livre IGUAL 0);

                        monstros[i].dados[DIR] = dir;
                }//if
                Desenhar_Imagem(monstros[i]);
        }//for
}
```

Código 5.4 – Função “Monstros( )”

6 ANIMAÇÃO

Nesta parte é explorado o tema de animação. São apresentadas noções básicas e uma classe que facilita a programação da animação. Finalmente, é feito um estudo de caso do uso da classe proposta para animação.

6.1 – Noções Básicas

Para a produção de animações em jogos 2D é preciso armazenar todos os quadros que serão utilizados, de forma semelhante à um desenho animado (figura 6.1). A estrutura IMAGEM tem espaço para armazenar até 128 quadros por variável. O campo de IMAGEM que indica o quadro corrente é o “quadro_atual”. Então basta alterar o valor dessa variável para que outro quadro seja desenhado na tela.

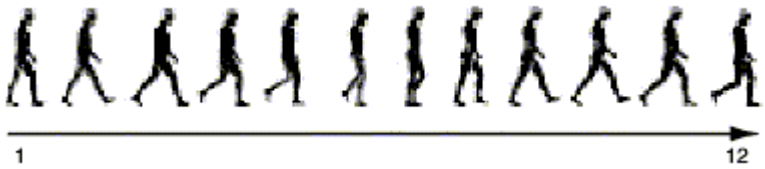

Figura 6.1 – Uma animação é composta por vários quadros

A seguir explicam-se alguns casos de animações, começando pelo mais simples. Considera-se que haja uma variável do tipo IMAGEM chamada "jogador" nos exemplos.

1º Caso: *Animar os quadros de uma IMAGEM na seqüência em que eles estão armazenados* (Código 6.1).

É utilizada uma variável "anim_cont" que irá contar o número de ciclos desde o último quadro selecionado. Quando este valor for igual a '3' (nº de quadros de espera) então será passado para o próximo quadro. Se passar do último quadro então retorna ao primeiro quadro.

```
//no começo do código definição das variáveis
int anim_cont = 0;

//No Jogo_Main( ):

//aumenta anim_cont e testa se alcançou o valor de espera
anim_cont ++;

if(anim_cont = = 3)
{
  anim_cont = 0;       //reinicia anim_cont
  jogador.quadro_atual ++; //passa para o próximo quadro
```

```
  //testa se passou do último quadro. Se verdadeiro então zera quadro_atual
   if(jogador.quadro_atual = = jogador.num_quadros)
   {
       jogador.quadro_atual = 0;
   }
}
Desenhar_Imagem(jogador);
```

Código 6.1 – Alterna os quadros de uma IMAGEM

2º Caso: *Animação baseada em uma matriz que contém as seqüências dos quadros (código 6.2).*

Existe uma matriz com os números dos quadros da animação. É preciso uma variável chamada "anim_ind" que mantém o índice do quadro de animação atual. A animação é reiniciada quando passar o último índice da matriz.

```
//no começo do código, definição da matriz e variáveis
int anim_seq[6] = {0,1,2,3,2,1};  //seqüencia de quadros
int anim_cont = 0;
int anim_ind = 0;       //índice da matriz

//no Jogo_Main( ):

//aumenta anim_cont e testa se alcançou o valor de espera
anim_cont ++;

if(anim_cont = = 3)
{
  anim_cont = 0;
  anim_ind++;          //passa para o próximo quadro
  if(anim_ind = = 6) anim_ind = 0;  //Se passou último índice então reinicia

  jogador.quadro_atual = anim_seq[anim_ind];  //pegar o quadro atual na matriz
}
Desenhar_Imagem(jogador);
```

Código 6.2 – Animação baseada em uma matriz.

3º Caso: *Várias animações (código 6.3).*

Para implementar várias animações pode-se usar uma matriz bidimensional. Um dos índices indica a animação atual (anim_atual) e o outro índice é usado para os quadros da animação (anim_ind). Iremos utilizar o valor '-1' para indicar o fim de uma animação. Para mudar de animação, basta alterar o valor da variável "anim_atual" e reiniciar a variável "anim_ind" para zero. No exemplo a seguir serão criadas três animações com um valor máximo de oito quadros para cada.

```
//no começo do código, definição da matriz e variáveis

//cada linha representa uma animação
int anim_seq[3][8] = {0,1,2,3,2,1,-1,0,
                      3,4,5,-1,0,0,0,0,
                      5,4,3,2,-1,0,0,0};

int anim_cont = 0;
int anim_ind = 0;      //índice dos quadros
int anim_atual = 0;    //índice de animação

//no Jogo_Main( ) :

//aumenta anim_cont e testa se alcançou o valor de espera
anim_cont ++;
if(anim_cont = = 3)
{
  anim_cont = 0;

  anim_ind++;         //passa para o próximo quadro

  //se alcançar o fim da animação representado por '-1' então reinicia animação
  if(anim_seq[anim_atual][anim_ind] = = -1)
  {
       anim_ind = 0;
  }
}

//se pressionar a tecla ENTER então passa para a próxima animação
if(Testar_Tecla[DIK_RETURN])
{
  anim_ind = 0;        //reinicia contadores
  anim_cont = 0;

  anim_atual++;       //próxima animação

 //se passou da última animação, volta para a primeira
  if(anim_atual = = 3) anim_atual = 0;
}

// Ler o quadro atual na matriz.
// Ele depende da animação atual e do índice de quadros.
jogador.quadro_atual = anim_seq[anim_atual][anim_ind];

Desenhar_Imagem(jogador);
```

Código 6.3 – Várias animações.

6.2 – Organizando em uma classe

Uma classe chamada “CAnim” foi programada para encapsular as complexidades da animação, para facilitar o seu uso em jogos. Ela apenas trabalha com valores inteiros e matrizes e não tem nenhuma dependência com o MRDX. Isto significa que a classe pode ser usada em qualquer programa C++.

Para utilizá-la, é preciso definir as matrizes que contém as seqüências dos quadros da animação e armazená-las em um objeto da classe CAnim. Estão disponíveis vários métodos que são usados para seleção da animação, atualização e definição de parâmetros. O cabeçalho da classe CAnim encontra-se no código 6.4.

```
class CAnim
{
        public:
                CAnim(void);
                ~CAnim(void);
                int Atualizar(void);
                int Quadro_Atual(void);
                int Atribuir_Anim_Freq(int freq);
                int Anim_Freq(void);
                int Adiciona_Anim(int num_pos, int *sequencia);
                int Anim_Ind(void);
                int Anim_Atual(void);
                int Atribuir_Anim_Atual(int anim, bool trava=false);
                bool Anim_Feita(void);
                void Reinicia_Anim(void);
                int Num_Anim(void);
                void Prox_Quadro(void);

        protected:
                int anim_cont;
                int anim_freq;
                int quadro_atual;
                int anim_ind;
                int *anim_seq[MAX_ANIM];
                int anim_atual;
                bool anim_feita;
                int num_anim;
                bool trava_ult_quadro;
};
```

Código 6.4 – Classe CAnim.

6.3 – Animando um lutador

Para demonstrar o uso da classe CAnim, foi elaborado um programa onde o usuário controla um lutador que pode executar diversos movimentos, tais como andar, pular, abaixar e chutar. A figura 6.2 é uma imagem da tela do programa.

Figura 6.2 – Exemplo de animação

No programa principal é preciso definir um objeto da classe CAnim e as matrizes com as seqüências de animação e variáveis inteiras que irão armazenar um identificador retornado pela classe para ser usado na manipulação da animação. Para facilitar o entendimento, estarei usando como exemplo apenas três das animações do lutador (código 6.5).

```
int  anim_andando, anim_chute, anim_pulo;

int ANIM_ANDANDO[ ] = {6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21};
int ANIM_CHUTE[ ] = {37,38,39,39,38,37};
int ANIM_PULO[ ] = {25,26,27,28,29,29,30,31,32,33};

CAnim anim;
```

Código 6.5 – Definição das matrizes de animação

O lutador é representado por uma variável do tipo IMAGEM. Seu tamanho é de 104x124 pixels e possui 45 quadros que são usados em suas diversas animações. Os quadros estão armazenados em um arquivo bitmap chamado “lutador.bmp”. Eles estão distribuídos como em uma grade de 9 colunas e 5 linhas. O código 6.6 é responsável pela inicialização do lutador.

```
Carregar_Bitmap("lutador.bmp");
Criar_Imagem(lutador,104,124,45);
int ind=0;
```

```
for(int lin=0; lin < 5; lin++)
        for(int col=0; col < 9; col++)
        {
                Carregar_Quadro(lutador,ind,col,lin);
                ind++;
        }
```

Código 6.6 – Inicialização do lutador

Em seguida é precisar armazenar os dados da animação através do método "Adiciona_Anim" da classe CAnim. É preciso informar o tamanho de cada matriz que representa uma animação. O método retorna um identificador que deverá ser armazenado (código 6.7).

```
anim_andando = anim.Adiciona_Anim(16, ANIM_ANDANDO);
anim_chute   = anim.Adiciona_Anim(6, ANIM_CHUTE);
anim_pulo    = anim.Adiciona_Anim(10, ANIM_PULO);
```

Código 6.7 – Armazenando as animações na classe

As animações são selecionadas de acordo com o comando do jogador e o estado do lutador. No final do ciclo é feito uma chamada ao método "Atualizar" da classe CAnim, que faz os cálculos necessários para a atualização da animação retornando o índice do quadro atual para ser selecionado no lutador (código 6.8).

```
if(Testar_Tecla(DIK_RIGHT))
{
     anim.Atribuir_Anim_Atual(anim_andando);
}

if(Testar_Tecla(DIK_LCONTROL))
{
    anim.Atribuir_Anim_Atual(anim_chute);
}

lutador.quadro_atual = anim.Atualizar( );
Desenhar_Imagem(lutador);
```

Código 6.8 – Seleção das animações

7 EDITOR DE MAPAS

Neste capítulo descreve-se um editor de mapas e comenta-se sobre as características dos *Tile Based Games.* Também será apresentada uma classe que implementa uma versão bem simples de mapas. Então, o editor é desenvolvido baseado na classe proposta.

7.1 – Tile Based Games

A maioria dos jogos 2D são conhecidos como *Tile Based Games* porque usam uma técnica muito comum de construção de cenários baseados em mapa de *tiles.* A tradução da palavra *"tile"* é de azulejo. A idéia é que se têm vários blocos de mesmo tamanho mas com diferentes imagens e que o cenário é montado a partir da junção desses blocos de uma forma organizada. Esta técnica é usada em jogos de plataforma, como o Mario da Nintendo, e até em complexos jogos de estratégia em tempo real como o Warcraft II da Blizzard.

O tamanho mais comum para um bloco é de 32x32 pixels. Então um mapa que ocupe a tela toda (640x480) teria 20 posições no eixo de X e 15 no eixo de Y. Uma forma fácil de programar os mapas e que serve para a maioria dos casos é dividi-lo em três camadas que tenham o seguinte conteúdo:

Camadas	Conteúdo
Base	Esta é a camada que fica por baixo de todas as outras. Geralmente representa o solo. Ex: Areia, grama, pisos, água.
Intermediária	Itens que não são móveis nem podem ser pegos pelo jogador. Ex: Rochas, mesas, árvores.
Objetos	Os objetos são móveis ou podem ser pegos pelo jogador. Ex: Espadas, livros, chaves, pessoas.

Base Intermediária Objetos

Figura 7.1 – Exemplo de conteúdo para cada camada

Cada uma dessas camadas pode ser representada por matrizes bidimensionais, e o seu conteúdo indica qual o índice do bloco que deve ser desenhado naquela posição. É possível também ter uma quarta camada que seria usada para objetos que estão acima do jogador. Esta opção é boa para dar uma impressão de profundidade. Por exemplo, se tivermos a ilustração de uma árvore que fosse dividida em duas partes: o tronco e a copa. O tronco ficaria na camada intermediária e teria um atributo que indicaria que o jogador não pode passar, enquanto que a copa estaria na quarta camada de forma que o jogador passaria por baixo dela conforme a figura 7.2.

Figura 7.2 – Personagem atrás da árvore.

Para cada bloco é possível ter diversas informações associadas para indicar se ele é um obstáculo, se causa algum dano ao jogador ou se há alguma ação especial que deva ocorrer quando forem tocados. Em certas situações é interessante permitir que um bloco tenha animação, isto é muito usado em cenários que ocorrem na beira de mares para ilustrar o movimento da água. Outra questão relativa à implementação diz respeito à variação do tamanho dos mapas. É mais fácil manter um padrão fixo para o tamanho dos mapas, porém é mais conveniente que o tamanho seja variável para cada mapa.

7.2 – A Classe CMapa

Para por em prática essas noções de mapas de *tiles* será definida uma classe que executa alguns serviços básicos que são necessários para a programação de mapas em jogos 2D. O objetivo não é de criar uma classe genérica para ser usado em qualquer jogo 2D, mas sim demonstrar um exemplo de fácil entendimento para que seja usada como base para a construção de uma classe de mapas que satisfaça suas necessidades.

Nesta classe já estão predefinidos em constantes o tamanho do bloco e a altura e largura do mapa. Ele possui duas camadas para desenho e a opção para atribuição e leitura de valores de uma determinada posição do mapa. Já existe um método que desenha o mapa na tela e o mapa pode ser salvo em arquivo para ser recuperado posteriormente. A classe não guarda informações adicionais para os blocos para o processamento de colisão. O arquivo cabeçalho da classe, chamado “CMapa.h” está exposto no código 7.1.

```
#define CMAPA_LARGURA       14
#define CMAPA_ALTURA        14
#define TAM_BLOCO           32
#define NUM_TILES           48

class CMapa
{
        public:

                void Carregar(void);
                void Salvar(void);

                void Atribuir_Imagem_Base(IMAGEM & im);
                void Atribuir_Imagem_Objetos(IMAGEM & im);

                void Atribuir(int camada, int cel_x, int cel_y, int valor);
                int Ler(int camada, int cel_x, int cel_y);

                void Desenhar(int x, int y);

        protected:

                int mapa_base[CMAPA_LARGURA][CMAPA_ALTURA];
                int mapa_objetos[CMAPA_LARGURA][CMAPA_ALTURA];

                IMAGEM *im_base;
                IMAGEM *im_objetos;

};
```

Código 7.1 – Classe CMapa.

A classe possui duas variáveis de referência para estruturas do tipo IMAGEM. Uma mantém as imagens usadas na camada base e a outra tem as imagens da camada de objetos. Estas imagens são carregadas externamente e apenas a referência é passada para a classe através dos dois métodos descritos no código 7.2.

```
void CMapa::Atribuir_Imagem_Base(IMAGEM & im)
{
  im_base = &im;
}

void CMapa::Atribuir_Imagem_Objetos(IMAGEM & im)
{
  im_objetos = &im;
}
```

Código 7.2 – Métodos para armazenar as referências de IMAGEM

Para modificar os valores do mapa deve-se utilizar o método “Atribuir( )” (código 7.3). É preciso informar as coordenada x e y da célula que será alterada, a camada a que pertence e o novo valor. Este valor é o índice da imagem que será desenhada naquela posição. O método usado para a leitura deste valor chama-se “Ler( )”.

```
void CMapa::Atribuir(int camada, int cel_x, int cel_y, int valor)
{
	//Atribui o valor a uma determinada posição do mapa
	// camada 0 -> Base; 1 -> Objetos;

	if(cel_x < 0 || cel_x >= CMAPA_LARGURA) return;
	if(cel_y < 0 || cel_y >= CMAPA_ALTURA) return;
	if(camada < 0 || camada >= 2) return;
	if(valor < 0 || valor >= NUM_TILES) return;

	if(camada = = 0)
	{
		mapa_base[cel_x][cel_y] = valor;
	}
	else
	{
		mapa_objetos[cel_x][cel_y] = valor;
	}
}
```

Código 7.3 – Método “Atribuir( )”

Há o método que desenha todas as camadas do mapa na tela (código 7.4). Ele percorre a matriz do mapa, calcula a posição correspondente à célula atual, e seleciona as imagens corretas de acordo com o conteúdo de cada célula. É importante notar que a imagem de índice “0” da camada de objetos deve ser totalmente transparente para possibilitar que a camada base seja vista.

```
void CMapa::Desenhar(int x, int y)
{
        int cel_x,cel_y;
        //desenha as duas camadas do mapa
        for(cel_x = 0; cel_x < CMAPA_LARGURA; cel_x++)
          for(cel_y = 0; cel_y < CMAPA_ALTURA; cel_y++)
          {
                im_base->x = im_objetos->x = x + (cel_x * TAM_BLOCO);
                im_base->y = im_objetos->y = y + (cel_y * TAM_BLOCO);

                //pegar índice do quadro que deve ser desenhado.
                im_base->quadro_atual = mapa_base[cel_x][cel_y];
                im_objetos->quadro_atual = mapa_objetos[cel_x][cel_y];

                //desenha as duas camadas da célula
                Desenhar_Imagem(*im_base);
                Desenhar_Imagem(*im_objetos);
          }
}
```

Código 7.4 – Método "Desenhar( )"

Finalmente, há a opção de salvar (código 7.5) e carregar o mapa a partir de arquivos. Estes métodos demonstram uma forma básica que pode ser adaptada para implementar a opção de salvar um jogo para que o usuário possa retornar ao local onde parou na última partida.

```
void CMapa::Salvar(void)
{
         //Salvar os dados do mapa no arquivo mapa.rhg
        FILE *fp;
        int cel_x,cel_y;

        //abrir arquivo em modo binário p/ escrita
        if( (fp = fopen("mapa.rhg","wb") ) != NULL )
        {
          for(cel_x = 0; cel_x < CMAPA_LARGURA; cel_x++)
         {
                for(cel_y = 0; cel_y < CMAPA_ALTURA; cel_y++)
                {
                  putw(mapa_base[cel_x][cel_y], fp);
                  putw(mapa_objetos[cel_x][cel_y], fp);
                }
          }
          //fechar arquivo
          fclose(fp);
        }
}
```

Código 7.5 – Método "Salvar( )".

7.3 – O Editor

Figura 7.3 – Um simples editor de mapas

Baseado na classe CMapa descrita no item anterior, é construído um simples editor (figura 7.3) para a criação de cenários. A edição é feita através do mouse e a tecla “control” serve para alternar entre as camadas. O código do editor encontra-se no arquivo “editor.cpp”. Nele são definidos três variáveis do tipo IMAGEM, sendo duas para guardar as imagens que serão usadas pelas duas camadas do mapa, e a outra é usada como o cursor do programa. Também há a definição de uma variável da classe CMapa (código 7.6).

```
IMAGEM objetos;
IMAGEM base;
IMAGEM cursor;

CMapa mapa;
```

Código 7.6 – Declaração das variáveis usadas no editor

Na inicialização são carregadas as imagens do cursor e dos blocos que serão usados no mapa. Os blocos de cada camada estão organizados em um arquivo bitmap. Cada camada possui 48 blocos que estão distribuídos como uma grade de 8 linhas por 6 colunas no bitmap. Temos o código 7.7 de preparação da variável IMAGEM que representa a camada base. Além disso é invocado o método “Carregar( )” da classe CMapa que irá verificar se existe um arquivo com informações do mapa salvos previamente, se não houver, o mapa é iniciado com valores padrões.

```
Carregar_Bitmap("base.bmp");
Criar_Imagem( base, 32, 32, 48 );

ind=0;
for(lin=0; lin < 8; lin++)
        for(col=0; col < 6; col++)
        {
                Carregar_Quadro( base, ind, col, lin );
                ind++;
        }

mapa.Atribuir_Imagem_Base( base );
mapa.Carregar( );
```

Código 7.7 – Fragmentos da inicialização do editor.

O comando associado ao pressionamento da tecla “control” modifica o conteúdo da variável “camada_atual” que indica a camada que está sofrendo alteração. O cursor ajusta sua posição de acordo com a variação da posição do mouse. Testes são feitos para manter o cursor sempre visível. Quando o botão do mouse é pressionado é verificado em que local o cursor se encontra, se estiver na área do mapa então a célula para a qual ele está apontando receberá o valor do bloco atual. Se o cursor estiver na área de seleção dos blocos então o bloco atual é modificado (código 7.8).

```
cursor.x += Mouse_Xvar();
if(cursor.x <0) cursor.x = 0;
if(cursor.x >=640) cursor.x = 639;
cursor.y += Mouse_Yvar();
if(cursor.y <0) cursor.y = 0;
if(cursor.y >=480) cursor.y = 479;

if(Mouse_Bot_Esquerdo()) {
    if(cursor.x < 448) {
        if(cursor.y < 448) {
                //area do mapa
                mapa.Atribuir( camada_atual,  cursor.x / TAM_BLOCO,
                                cursor.y / TAM_BLOCO, tile_atual);
        }
    }
    else {
        if(cursor.y < 256) {
                //area dos tiles
                tile_atual  =   (cursor.x - 448) / TAM_BLOCO +
                                (cursor.y / TAM_BLOCO) * 6;
        }
    }
}
```

Código 7.8 – Controle do mouse

Para mostrar o mapa na tela basta chamar o método “Desenhar( )” da classe CMapa. Porém o editor também desenha na lateral os blocos para que o usuário possa selecioná-los, conforme código 7.9.

```
mapa.Desenhar(0,0);

//desenhar os tiles a serem selecionados
int quadro_ind=0;
int tile_lin,tile_col;

if(camada_atual = = 0)
{
  for( tile_lin =0; tile_lin < 8; tile_lin++)
        for(tile_col =0; tile_col < 6; tile_col++)
        {
                base.quadro_atual = quadro_ind;
                base.x = 448 + tile_col * TAM_BLOCO;
                base.y = tile_lin * TAM_BLOCO;
                Desenhar_Imagem(base);
                quadro_ind++;
        }
}
```

Código 7.9 – Desenhando o mapa e a interface do editor

8 FUNDAMENTOS DE JOGOS 3D

O MRDX é deixado de lado agora para que sejam descritos os jogos 3D. Comenta-se sobre a biblioteca gráfica OpenGL. São apresentados conceitos sobre a movimentação 3D e características da renderização de um cenário usando o OpenGL.

8.1 – OpenGL

A principal tecnologia utilizada na indústria de computação gráfica é uma biblioteca chamada OpenGL. O OpenGL é uma biblioteca gráfica poderosa e de baixo nível que fornece ao programador uma interface para o hardware gráfico. Foi originalmente desenvolvido pela Silicon Graphics, empresa responsável pelos efeitos especiais vistos em filmes como Exterminador do Futuro 2 e Parque dos Dinossauros.

Desde 1992, o OpenGL tem sido mantido pelo OpenGL Architecture Review Board [OPE2004], que é composto por grandes empresas como a Compaq, IBM, Intel, Microsoft, HP, nVidia e diversas outras.

O OpenGL é multiplataforma, estando disponível em todos os principais sistemas operacionais como o Windows e Linux, e ambientes de desenvolvimento como C/C++, Delphi e Java. Ele foi projetado para renderizações 3D com aceleração de hardware. Atualmente, placas gráficas para PC como a NVIDIA GeForce processa comandos de OpenGL diretamente no hardware. Diversos jogos profissionais utilizam o OpenGL para a renderização, tais como os jogos da série Quake. O livro de Hawkins [HAW2001] é um excelente guia para o uso de OpenGL na programação de jogos 3D.

Como o OpenGL trabalha principalmente com a definição de primitivas para a construção de ambientes 3D, surgiu a necessidade de se implementar bibliotecas auxiliares para facilitar a programação. As duas principais estão comentadas a seguir:

1. GLU (OpenGL Utility Library) : suplementa o OpenGL fornecendo funções de alto nível para a execução de certas tarefas tais como o controle de câmera; manipulação das matrizes de projeção e de orientação da visualização; suporte para superfícies curvas e definição de objetos 3D como esferas e cilindros.
2. GLUT (OpenGL Utility Toolkit) : O OpenGL não suporta diretamente nenhum tipo de janelas, menus ou entrada de dados. Por isso foi desenvolvido o GLUT que providencia uma implementação básica desses itens enquanto se mantém independente de plataforma.

Para demonstrar o uso do OpenGL foi desenvolvido um programa usando o GLUT que desenha um simples labirinto 3D. O usuário pode se mover pelo labirinto através das setas do teclado. É feito o teste de colisão com as paredes para impedir que o usuário atravesse-as. Há uma esfera que se movimenta pelo labirinto desviando das paredes. É possível visualizar o labirinto em modo *wireframe* pressionando a tecla 'W'. A imagem do programa está na figura 8.1. O código da função main( ) está na listagem de código 8.1.

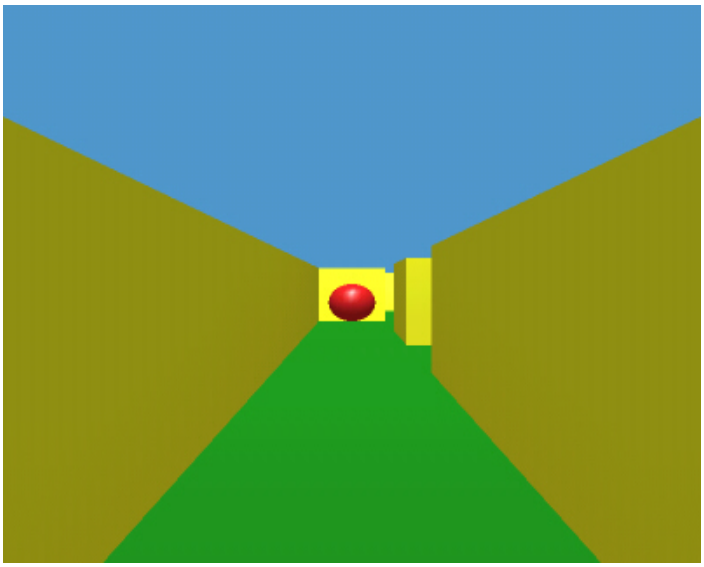

Figura 8.1 – Programa de Labirinto 3D

```
void main(int argc, char **argv)
{
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
 glutCreateWindow("Labirinto 3D");
 glutFullScreen( );

 Inicializa( );

 glutDisplayFunc(display);
 glutKeyboardFunc(Keyboard_Function);
 glutSpecialFunc(Special_Function);
 glutIdleFunc(Move_Esfera);

 glutMainLoop( );
}
```

Código 8.1 – Função main( ) do programa Labirinto 3D

A função main( ) do programa é composta quase toda por chamadas às funções do GLUT para a configuração do ambiente. A seguir está exposto o que cada uma delas faz:

- **glutInit(&argc, argv) :** Função de inicialização do GLUT.
- **glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH) :** Indica o modo de exibição que será usado no programa. Neste exemplo está sendo criado buffer duplo para o desenho (GLUT_DOUBLE), o sistema de cores usado é o RGB (GLUT_RGB), e o buffer de profundidade (GLUT_DEPTH), que serve para ordenar os objetos na renderização, está sendo usado.
- **glutCreateWindow("Labirinto 3D") :** Este comando cria a janela do programa com o título de “Labirinto 3D”.
- **glutFullScreen( ) :** Especifica que o programa deve ser executado em modo de tela cheia.
- **Inicializa( ) :** Esta função é definida em outra parte do programa. Nela são feitas as inicializações do OpenGL e do programa em si.
- **glutDisplayFunc(display) :** Indica qual a função que será usada para a renderização.
- **glutKeyboardFunc(Keyboard_Function) :** Indica a função usada para a leitura de teclas convencionais, que tem um valor ASCII associado.
- **glutSpecialFunc(Special_Function) :** Indica a função usada para a leitura das teclas especiais tais como as setas e teclas de função (F1, F2...).
- **glutIdleFunc(Move_Esfera) :** Indica a função que deverá ser chamada constantemente durante a execução do programa.
- **glutMainLoop( ) :** Esta função processa todos os eventos que ocorrem enquanto seu programa está rodando, como mensagens enviadas pelo sistema operacional e teclas pressionadas.

8.2 – Movimentação 3D

Existem dois conceitos essenciais necessários para que seja entendida a movimentação em um mundo virtual 3D: o ponto de vista e o corpo. O ponto de vista é uma espécie de olho no mundo virtual. Os movimentos que modificam o ponto de vista são conhecidos como angulares e não alteram a posição em que se encontra o ponto de vista no mundo 3D.

Segundo Gradecki [GRA1995], são três os movimentos angulares: Guinada, Arfada e Rolamento. Para olhar alguma coisa à esquerda ou direta, a cabeça vira nesta direção, este é o movimento chamado de guinada. O movimento de arfada é quando se olha para cima ou para baixo. Finalmente, o rolamento equivale à inclinação da cabeça para esquerda ou para direita.

Os movimentos associados ao corpo permitem que o ponto de vista seja deslocado para qualquer outro local do mundo virtual. A execução do movimento consiste na alteração das coordenadas (x, y, z) que são usadas para definir a posição do corpo no mundo tridimensional.

Na prática, os movimentos mais usados são o de guinada e deslocamento para frente ou para trás em uma determinada direção. Para a implementação destes movimentos é necessária uma variável que guarde o ângulo indicando a direção para onde o jogador está apontando. É preciso também armazenar o valor do passo que o jogador se move. Isto lembra o sistema polar de coordenadas, porém nos programas utiliza-se o sistema cartesiano de coordenadas, sendo necessário a conversão destes valores, conforme a figura 8.2.

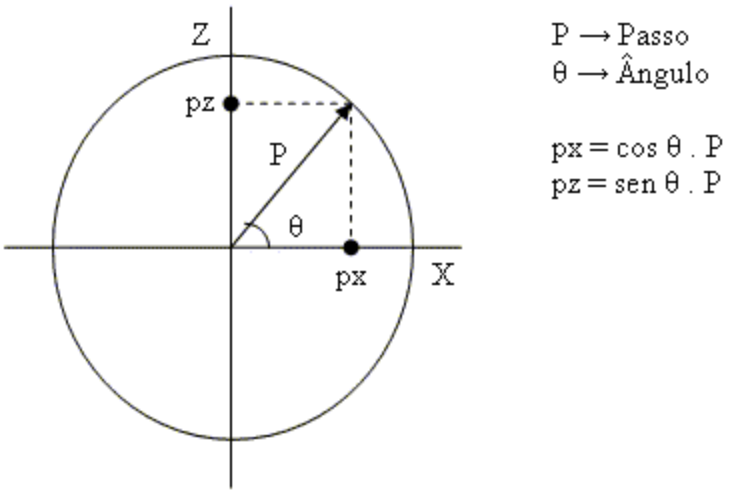

Figura 8.2 – Conversão para coordenadas cartesianas.

No programa do Labirinto 3D o movimento é feito através do teclado. O GLUT especifica funções para a leitura de teclas convencionais e para as teclas especiais, como as setas. As únicas teclas convencionais usadas neste programa é o ESCAPE para encerrar o aplicativo e a tecla “W” que é usado para alternar o modo de renderização entre normal ou em wireframe (Código 8.2).

```
void Keyboard_Function(unsigned char key, int x, int y)
{
  switch (key)
  {
   case  27:     exit(0);  //ESC -> encerra aplicativo...
                 break;

   case 'w':
   case 'W':     wire =!wire;
                 break;
  }
}
```

Código 8.2 – Função Keyboard_Function( )

Na função de leitura das teclas especiais são processados os comandos associados às setas do teclado. As setas para esquerda e direita executam o movimento de guinada alterando o valor do ângulo para o qual o jogador está direcionado e calcula os componente X e Z do movimento.As setas para frente e trás fazem o movimento do corpo, porém antes da execução é verificado se o caminho está livre através da função “pode_mover( )” (Código 8.3).

```
void Special_Function(int key, int x, int y)
{
 float rad;

 switch (key) {
   case GLUT_KEY_RIGHT:

        angulo += 10;
        if(angulo >= 360) angulo -=360;

         rad =  (float) (3.14159 * angulo / 180.0f);

        mov_x = cos(rad) * PASSO;
        mov_z = sin(rad) * PASSO;
        break;
```

```
 case GLUT_KEY_UP:

          if(pode_mover(jog_x, jog_z,  mov_x, mov_z))
          {
           jog_x += mov_x;
           jog_z += mov_z;
          }
          break;
 ...

 }
}
```

Código 8.3 – Fragmento da função Special_Function( )

O labirinto é representado no programa como uma matriz bidimensional onde cada célula armazena ‘0’ ou ‘1’. Se o valor for igual a ‘1’ então há um cubo naquela posição. A função “pode_mover( )” recebe como parâmetro a posição do jogador e o movimento que deseja fazer. A partir destes valores é calculada a célula na matriz correspondente e verificado se há algum cubo impedindo o movimento (código 8.4).

```
int pode_mover(float pos_x, float pos_z, float vet_x, float vet_z)
{
        float mundo_x = pos_x + vet_x ;
        float mundo_z = pos_z + vet_z ;

        int ind_x = (int) ((mundo_x + TAM_BLOCO/2) / TAM_BLOCO);
        int ind_z = (int) ((mundo_z + TAM_BLOCO/2) / TAM_BLOCO);

        if(mapa[ind_x][ind_z]) return 0;
        else return 1;
}
```

Código 8.4 – Função pode_mover( )

8.3 - Renderização

A câmera é o objeto que determina a visão do mundo 3D. São 3 vetores que definem a câmera: um de posição; outro que indica para onde está sendo direcionada; o último determina que parte da câmera está para cima. A câmera pode ser especifica usando a função “gluLookAt( )”. A alteração no vetor de posição caracteriza o movimento de corpo. O vetor de direção está associado aos movimentos angulares e o último vetor corresponde ao movimento de rolamento.

Para definir cores no OpenGL deve-se especificar a intensidade (um valor entre 0.0 e 1.0) de cada cor primária, que são o vermelho (Red), verde (Green) e azul (Blue). Um quarto elemento chamado alfa é geralmente usado na definição de cores para o controle da transparência.

O OpenGL tenta modelar a luz do mundo real através de alguns componentes:

- **Luz do Ambiente** : Não vem de uma direção em particular.
- **Luz Difusa** : Vem de uma certa direção, mas quando atinge uma superfície é refletida em todas as direções.
- **Luz Especular** : Vem de uma certa direção e reflete em uma única direção.

A cor de material de um objeto é que determina como a luz será refletida. O material possui cor ambiente, difusa e especular diferentes que indicam a forma que a cada uma das componentes da iluminação será refletida. Os valores usados no programa do labirinto para a definição da luz e dos materiais estão no código 8.5. Para mais informações sobre cores, iluminação e materiais, sugere-se o livro de Wells [WEL1995].

```
GLfloat mat_specular[ ] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[ ] = { 50.0 };
GLfloat mat_amarelo[ ] = {0.8, 0.8, 0.1, 1.0};
GLfloat mat_verde[ ] = { 0.1, 0.6, 0.1, 1.0 };
GLfloat mat_vermelho[ ] = { 0.7, 0.1, 0.1, 1.0 };
GLfloat light_position[ ] = { 0.0, 500.0, 0.0, 1.0 };
GLfloat luz_branca[ ] = {1.0,1.0,1.0,1.0};
GLfloat lmodel_ambient[ ] = {0.6,0.6,0.6,1.0};
```

Código 8.5 – Definição de valores usados na iluminação

Na função “Inicializa( )”(código 8.6), é selecionada através da função “glShadeModel ( )” o modo de colorização com várias tonalidades. A função “glClearColor( )” especifica a cor usada para limpar a tela. A iluminação é definida através de várias chamadas à função “glLightfv( )” para especificar suas componentes e sua posição. A projeção em perspectiva é selecionada através da função “gluPerspective( )”. O comando “glEnable(GL_DEPTH_TEST)” habilita a ordenação em profundidade de forma que as imagens que estiverem mais próximas ao observador irão sobrepor as mais distantes.

```
void Inicializa(void)
{
 glShadeModel(GL_SMOOTH);

 glClearColor(0.3, 0.6, 0.8, 0.0);

 glLightfv(GL_LIGHT0, GL_POSITION, light_position);
 glLightfv(GL_LIGHT0, GL_DIFFUSE, luz_branca);
 glLightfv(GL_LIGHT0, GL_SPECULAR, luz_branca);
 glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);

 //Especifica sistema de coordenadas de projecao
 glMatrixMode(GL_PROJECTION);
 // Inicializa sistema de coordenadas de projecao
 glLoadIdentity();

 //Especifica a projecao perspectiva
 gluPerspective(90,1,0.1,3000);

 //Especifica sistema de coordenadas do modelo
 glMatrixMode(GL_MODELVIEW);

 // Inicializa sistema de coordenadas de projecao
 glLoadIdentity();

 glEnable(GL_DEPTH_TEST);

 // inicializa numeros aleatorios
 srand(GetTickCount());

 //posicao inicial da esfera
g_esfera.x_pos = 3 * TAM_BLOCO;
g_esfera.z_pos = TAM_BLOCO;
g_esfera.dir = LESTE;
}
```

Código 8.6 – Função Inicializa( )

Finalmente, a função “display( )” (código 8.7) renderiza todos os objetos na tela. A posição da câmera é ajustada de acordo com a posição do jogador. Um plano verde representando o solo é desenhado a partir do uso de primitivas do OpenGL. A esfera é desenhada através da função “glutSolidSphere( )”. O labirinto é desenhado bloco a bloco depois de verificar se na posição correspondente da matriz existe o valor “1”.

```
void display(void)
{
  int x, z;
  int x_mun, z_mun;

  //limpa todos os pixels
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

 glLoadIdentity();

 //câmera
 gluLookAt(jog_x,25   ,jog_z, jog_x+mov_x,25,jog_z+mov_z, 0,1,0);

 //plano
 glPushMatrix();

  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_verde);

  glBegin ( GL_QUADS);

          glVertex3f(-10000, -TAM_BLOCO/2, -10000);
          glVertex3f(-10000, -TAM_BLOCO/2, 10000);
          glVertex3f(10000, -TAM_BLOCO/2, 10000);
          glVertex3f(10000, -TAM_BLOCO/2, -10000);

  glEnd();

 glPopMatrix();

 //desenha esfera
 glPushMatrix();

  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_vermelho);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

  glTranslatef(g_esfera.x_pos, 5 ,g_esfera.z_pos);

  glutSolidSphere(20,20,16);

 glPopMatrix();
 //labirinto
  for(x=0; x < 15; x++)
  {
          for(z=0; z < 15; z++)
          {
            if(mapa[x][z]) //tem um bloco
            {
               x_mun = x * TAM_BLOCO;
               z_mun = z * TAM_BLOCO;
```

```
            //cubo grande
            glPushMatrix();

            glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_amarelo);
            glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
            glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

            glTranslatef(x_mun, 5 ,z_mun);

            if(wire) glutWireCube(TAM_BLOCO);
            else glutSolidCube(TAM_BLOCO);

            glPopMatrix();

          }//if(mapa[x][z])
        }//for
 }//for
 glutSwapBuffers();
}
```

Código 8.7 – Função display( )

9 CONCLUSÃO

O desenvolvimento de jogos eletrônicos é uma das atividades mais interessantes e mais complexas na área da computação pois integra os mais diversos assuntos como Computação Gráfica, Inteligência Artificial, Sistemas Distribuídos, Engenharia de Software, Estrutura de Dados e Multimídia. Os jogos podem motivar os estudantes a se interessarem pelas mais diversas matérias que compõem um curso de Bacharelado em Ciência da Computação. Espera-se que este TCC sirva como um estímulo para o desenvolvimento de outros trabalhos voltados para a programação de jogos. Principalmente em regiões onde ainda não há empresas de jogos, a Universidade deve desempenhar um papel fundamental incentivando pesquisas e promovendo discussões para despertar a sociedade para o potencial da área de jogos.

Como um exemplo da característica interdisciplinar dos jogos, o autor desenvolveu o Projeto JEDI – Jogos Eletrônicos Distribuídos e Inteligentes, com o objetivo de servir como um laboratório de aplicação dos conceitos das disciplinas de Sistemas Distribuídos e Inteligência Artificial aplicadas no desenvolvimento de jogos. Informações sobre o projeto JEDI, incluindo apresentações e demos, podem ser encontradas na página de jogos do autor [ROM2004].

A versão inicial do MRDX foi desenvolvida com um propósito didático, o ensino de programação de jogos. Como o MRDX implementa os serviços básicos, que são comuns para diferentes jogos, um programador inicia diretamente a programação dos aspectos criativos do jogo. Outras vantagens desta biblioteca são: é livre para download, com documentação em português, e com código fonte aberto. Graças às funcionalidades do MRDX, neste TCC, foi possível tratar de assuntos de mais alto nível relacionados à programação de jogos.

REFERÊNCIAS BIBLIOGRÁFICAS

[CON2004] **Continuum.** Disponível em http://www.continuum.com.br. Acesso em janeiro de 2004.

[DAV1995] DAVIS, Stephen. **C++ para Leigos**. São Paulo: Berkeley, 1995.

[DEI2000] DEITEL, Harley. **C++: Como Programar**. Porto Alegre: Bookman, 2000.

[GRA1995] GRADECKI, Joe. **Kit de Programação da Realidade Virtual**. São Paulo : Berkeley, 1995.

[GRU1994] GRUBER, Diana. **Action Arcade Adventure Set.** Arizona: Coriolis, 1994.

[HAT2004] HATTAN, John. **What language do I use?.** Disponível em http://www.gamedev.net/reference/articles/article895.asp. Acesso em janeiro de 2004.

[HAW2001] HAWKINS, Kevin; ASTLE, Dave. **OpenGL Game Programming**. USA: Prima Publishing, 2001.

[IN22004] **IN2Games – Congresso Internacional de Inovação em Jogos para Computadores**. Disponível em http://www.gamenetpr.com.br/in2games/. Acesso em janeiro de 2004.

[LAM1999] LAMOTHE, André. **Tricks of the Windows Game Programming Gurus**. Indianapolis: SAMS, 1999.

[OPE2004] **OpenGL Architecture Review Board.** Disponível em http://www.opengl.org. Acesso em janeiro de 2004.

[ROL2000] ROLLINGS, Andrew; MORRIS, Dave. **Game Architecture and Design.** Arizona: Coriolis, 2000.

[ROM2004] ROMERO, Marcos. **Programação de Jogos.** Disponível em http://www.geocities.com/progjogos. Acesso em janeiro de 2004.

[SAL2000] SALTZMAN, Marc. **Game Design Secrets of the Sages.** Indianapolis: Macmillan, 2000.

[SCH1988] SCHILDT, Herbert. **Turbo C - Guia do Usuário**. São Paulo: McGraw-Hill, 1988.

[WEL1995] WELLS, Drew; YOUNG, Chris. **Criações em Ray Tracing**. São Paulo : Berkeley, 1995.

[WJO2004] **Wjogos – Workshop Brasileiro de Jogos e Entretenimento Digital.** Disponível em http://www.icad.puc-rio.br/wjogos. Acesso em janeiro de 2004.

ANEXO 1: CURSOS DE JOGOS

- *Pós-Graduação em Desenvolvimento de Jogos para Computadores – Unicenp- PR*

Link: http://www.posunicenp.edu.br/index.asp?strCurso=Jogos
Acesso em janeiro de 2004.

Objetivos do Curso:

Formar profissionais capazes de desenvolver jogos para computador levando em consideração tanto a análise quanto a programação. Fornecer uma forte base de programação em C/C++ para Windows com a utilização da biblioteca de jogos DirectX. Apresentar aspectos modernos da construção de jogos, como game design, técnicas de inteligência artificial e programação avançada.

Ementa :

Grupo I: Gerenciais

- Empreendedorismo

Mercado de trabalho: emprego e trabalho. O empreendedor. Geração da idéia de um novo negócio. Validação da idéia. Análises preliminares. Escala e recursos. Projeto de produto. Plano de Marketing. Tecnologia e produção. Plano de Operações. Plano de Finanças. Criação e legalização do negócio.

- Gerenciamento de Equipes de Jogos

Bons tempos aqueles em que apenas uma boa idéia bastava. Uma pessoa era o roteirista, designer, ilustrador, programador e, para encerrar, o responsável pelos testes. Hoje não é possível pensar em um jogo sem a formação de uma equipe. Quem vai gerenciar esta equipe? Afinal, desenvolvimento de jogos é um negócio e, como tal, possui prazos, orçamento, relatório para os investidores, etc. O foco desta disciplina é o gerenciamento, mais especificamente, o gerenciamento de equipes de jogos.

Grupo II: Projeto e Modelagem

- Análise e Projeto de Sistemas OO com UML

UML. RUP. Design Patterns. Ferramentas CASE. Especificação de Sistemas Orientados a Objetos. Estudo de Caso.

- Projeto de Jogos – Game Design I e II

Introdução a Game Design. Jogadores e Público-alvo. Brainstorm. Foco. Elementos de Jogabilidade. Tramas, história. Inteligência Artifical. Mecânica da Jogabilidade. Documentação, Design Bible. Ferramentas de Game Design. Level Design. Feedback e Playtest.

Conceito Inicial. Design Essencial. Jogabilidade. Design Detalhado. Balanceamento de Games. Explorando a Interação. Análise de Especialistas. Futuro do Game Design. Tópicos Avançados.

Grupo III: Teoria para Jogos

- Algoritmos para Jogos

Listas, pilhas, filas, grafos, árvores, etc. Jogos são ambientes ricos para a aplicação de diversas estruturas de dados. Nesta disciplina vamos "revisitar" estas estruturas e conhecer outras usadas normalmente durante o desenvolvimento de um jogo.

- Matemática 3D para Jogos

Teoria matemática necessária para o entendimento de como funciona um mundo virtual 3D dentro do computador. Matrizes. Trigonometria. Análise Matemática.

- Inteligência Artificial em Jogos

Técnicas e algoritmos de IA aplicadas ao desenvolvimento de jogos. Algoritmos de Busca. Máquinas de Estado.

Grupo IV: Programação

- Programação em C/C++

Classes e Objetos, Herança, Polimorfismo, Templates.

- Programação para Windows

Esta disciplina, além de ser uma introdução à programação Windows, apresentará a programação básica para este sistema operacional, sem o uso de bibliotecas ou recursos que encapsulam as funcionalidades básicas do sistema. O objetivo é programar para windows conhecendo as estruturas internas deste sistema operacional. Uso avançado das APIs Windows. Otimização de Código. Assembler Inline. Técnicas de Depuração e Profile. Programação direta de Hardware.

- Programação com DirectX – Introdução

O que é DirectX. Porque usar DirectX. Aplicação Básica usando DirectX. Entrada de Dados com DirectInput. Modos de Vídeo. Parametrizando aplicações. Noções de DirectAudio e DirectVideo.

- Programação em 3D com DirectX

Introdução ao Direct3D. Inicialização do Direct3D. Transformações. Luzes. Texturas. Efeitos Especiais. Detecção de Colisão. Objetos Complexos e Animação.

- Programação em 3D com OpenGL

Introdução ao OpenGL. Aplicação Básica com OpenGL. Desenhando Objetos Simples. Manipulando Objetos. Luzes. Texturas. Efeitos Especiais. Objetos Complexos e Animação.

- Programação com DirectX – Network e Internet

Introdução ao DirectPlay. Sessões. Lobbies. Enviando e Recebendo Dados. Técnicas de Comunicação. Estudo de Caso.

- *Design e Planejamento de Games – Anhembi – Morumbi (Graduação)*

Link: http://www.anhembi.br/portal/nossa_u/cursos/grad-mod-zip/cur_design_plan.htm
Acesso em janeiro de 2004.

Duração: 4 anos

Proposta :

Formar profissionais para o desenvolvimento de games, nos seus mais variados formatos - jogos de estratégia, ação, aventura, esporte e RPG, para a aplicação em projetos de entretenimento, cultura e arte, experimentação, educação e treinamento corporativo para diversas plataformas, incluindo web e dispositivos móveis.

Mercado de Trabalho :

Nos últimos anos, com a crescente valorização do lazer, os games adquiriram importância na indústria de entretenimento. Porém, o mercado de trabalho para o profissional de design de games é muito mais amplo. Os jogos adquiriram importância em outras áreas, como na educação, cinema e televisão. Além disso, são utilizados em ambientes corporativos como ferramentas de motivação, treinamento e vendas. O campo de atuação, portanto, é bastante promissor. O profissional pode atuar em empresas de desenvolvimento de games e softwares, escritórios de design, portais e sites da internet, e departamentos de treinamento e comunicação.

Formação :

O curso prepara o aluno para desenvolver projetos de games a partir de uma sólida formação nas disciplinas fundamentais no processo de criação e planejamento de um jogo: design de hipermídia, computação gráfica, lógica de programação, banco de dados, redes de computadores, roteirização, estratégia de jogos, animação, multimídia, linguagem sonora, cinema e vídeo, história em quadrinhos, realidade virtual, planejamento e custos.

Principais Disciplinas :

Fundamentos do Game Design, Metodologia de Projeto para Design de Games, Linguagem de Programação, Animação, Laboratório de Modelagem, Roteiro, Design de Som, Gestão de Negócios de Games, Desenho, Inteligência Artificial, Cálculo, Física, Banco de Dados, Redes de Computador, Direção de Arte, Segurança de Informação, Linguagem Visual, História da Arte e Tecnologia.

- *Curso de Extensão em Desenvolvimento e Design de Jogos 3D/PUC-RJ.*

Link: http://www.cce.puc-rio.br/artes/designjogos.htm
Acesso em janeiro de 2004.

Carga Horária : 135 horas.

Objetivo :

Permitir que o aluno desenvolva um jogo 3D para computadores utilizando as tecnologias mais atuais do mercado; Introduzir o aluno aos principais conceitos de Computação Gráfica, Inteligência Artificial, Sistemas Distribuídos e Design envolvidos em jogos; Explorar o potencial criativo do aluno e trabalhar sua habilidade para criar roteiros, projetar interfaces, desenhar personagens, cenários e outros elementos que compõem um jogo 3D, incluindo áudio; e Possibilitar ao aluno conhecer a dinâmica de uma equipe de desenvolvimento de Jogos, através de trabalhos práticos em uma equipe.

Público :

Programadores, designers, estudantes, entusiastas por games.

Programa :

- Introdução: História dos Jogos, Tipos ou Gêneros de Jogos, Arte Conceitual, Partes de um Jogo: Roteiro e Estória, Arte, Motor.
- Mercado de Jogos: Estudo do Mercado de Jogos (nacional e internacional), Tendências do mercado, plano de negócios.
- Princípios Matemáticos para Desenvolvimento de jogos 3D.- Teoria da Computação Gráfica.
- Princípios de programação (em C / C++).
- API's Gráficas: Introdução a OpenGL, Direct X.
- Tópicos de Inteligência Artificial.
- Tópicos de Sistemas Distribuídos: Paradigmas de Jogos Multi-Jogador, Ponto-a- Ponto x Cliente-Servidor, Servidores de Jogos, Clientes de Jogos, Compactação, Servidores-Mestre para Jogos Multi-Jogador na Internet, Direct Play.
- Princípios de Design para Jogos, Conceituação de cenários e personagens, storyboards.
- Interface para Jogos.
- 3DS MAX - cenários estáticos e personagens.
- Produção de Áudio e Vídeo para Jogos.
- Arquitetura de Engines.
- Engines: Fly3D, 3D Game Studio, Discreet GMAX.
- Games com Flash MX.

ANEXO 2: A IMPLEMENTAÇÃO DO MRDX

O MRDX foi desenvolvido usando a linguagem C/C++ [SCH1988] [DAV1995]. Ele está dividido em dois arquivos: “mrdx.h” e “mrdx.cpp”. O arquivo “mrdx.h” é conhecido como arquivo cabeçalho e possui os protótipos das funções do MRDX e as definições de constantes e de algumas estruturas de dados. O arquivo “mrdx.cpp” possui a definição de variáveis de módulo e das funções do MRDX.

Para que seja entendida a forma como o MRDX foi desenvolvido, é preciso conhecer algumas características que programas para o sistema operacional Windows devem ter e os fundamentos da biblioteca de programação de jogos da Microsoft chamada DirectX.

Segundo LaMothe [LAM1999], o DirectX é um sistema de software que abstrai o hardware do computador para que seja utilizado o mesmo código independentemente da configuração de uma máquina em particular. O DirectX usa a tecnologia COM (Component Object Model) e um conjunto de bibliotecas escritas pela própria Microsoft e pelos fabricantes de hardware. A Microsoft estabeleceu uma série de convenções que devem ser seguidas pelos fabricantes de hardware para implementar as drivers que se comunicam com o hardware.

Os componentes principais do DirectX, na versão 6.0, são:

- **DirectDraw** - Este é o componente que controla a tela de vídeo. Todos os gráficos são conduzidos através dele. É o mais importante de todos os componentes do DirectX.
- **DirectInput** - Este sistema controla todos os dispositivos de entrada, tais como teclado, mouse e joystick.
- **DirectSound** - Componente que controla o som digital do DirectX.
- **DirectMusic** - Responsável pelo som no formato MIDI, muito utilizado para músicas.
- **DirectPlay** - Componente usado para conexões em rede usando a Internet, modens ou conexão direta.
- **Direct3D** - É o sistema utilizado para a criação de ambientes em 3D no DirectX.

a)Programação para Windows

Para a criação de um programa para o sistema operacional Windows, usando a Win32 API, é preciso realizar as seguintes tarefas:

- **Criar uma classe Windows** : Há uma estrutura de dados chamada "WNDCLASSEEX" que contém as informações básicas de todas as janelas. Deve-se especificar os dados particulares da janela que será usada na aplicação que está sendo desenvolvida.
- **Registrar a sua classe Windows** : Depois de especificado os dados da classe Windows, é preciso registrá-la para que o Sistema Operacional Windows saiba da existência dela. Para isso utiliza-se a função "RegisterClassEx( )".
- **Criar a janela** : Feito o registro, pode-se criar uma janela baseada na classe registrada. A função usada é a "CreateWindowEx( )". Informa-se uma série de parâmetros tais como tamanho, posição e estilo. A função irá retornar um identificador da janela criada.
- **Receber as mensagens :** Faz-se necessário a definição de um laço principal que ficará em execução até que receba uma mensagem de finalização. Nele são recebidas as mensagens enviadas por outras aplicações e mandadas para o Manipulador de Eventos.
- **Manipular os eventos** : É preciso definir uma função que irá processar os eventos ocorridos durante a execução da aplicação. Determinam-se quais são os eventos que devem ser processados e de que forma, ou pode ser usado o processamento padrão do Sistema Operacional Windows.

O MRDX encapsula essas operações em duas funções : “WinMain( )” e “WindowProc( )”. A função “WindowProc( )” é o manipulador de eventos. Na função “WinMain( )” (código A2.1) são executadas as demais operações. A partir dela é que são chamadas as funções do MRDX responsáveis pela inicialização (“Directx_Iniciar( )”) e encerramento (“Directx_Encerrar( )”) do DirectX. É definido um laço onde as mensagens são recebidas e é feita a chamada à função principal (“Jogo_Main( )”) de um jogo feito no MRDX.

```
// construção da janela ...

Directx_Iniciar();

// laço principal de eventos
while(TRUE)
{
  // testa se há uma mensagem na fila, se tiver então retira
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
  {
        if (msg.message == WM_QUIT)
              break;

        // traduz teclas aceleradoras
        TranslateMessage(&msg);

        // manda a mensagem para o manipulador de evento (WindowProc)
        DispatchMessage(&msg);
  }
  //atualizar dispositivos de entrada - MRDX
  Atualizar_Teclado( );
  Atualizar_Mouse( );
  Atualizar_Joystick( );

  //Processamento principal do jogo
  Jogo_Main( );
 }
Directx_Encerrar( );
...
```

Código A2.1 – Fragmentos da função "WinMain( )"

b) DirectDraw

O DirectDraw é composto pelas seguintes interfaces:

- **IDirectDraw** – Interface principal que deve ser instanciada para se trabalhar com DirectDraw. Ela representa a placa de vídeo. No MRDX ela é instanciada na função "Directx_Iniciar( )". Várias tarefas são executadas a partir desta interface, como a atribuição do modo de vídeo.
- **IDirectDrawPalette** - Interface que lida com a palheta de cores nos modos de vídeo que precisam de palheta, tal como o de 256 cores usado pelo MRDX. No MRDX ela também é instanciada na função "Directx_Iniciar( )". Toda vez que um arquivo bitmap é carregado na memória através da função "Carregar_Bitmap( )", a palheta que está presente no bitmap passa a ser a palheta atual.

- **IDirectDrawClipper** - Usada para que sejam cortadas as partes das imagens que estiverem fora da área especificada para desenho. O MRDX define uma única área de clipper (corte), através da função “Anexar_Clipper( )”, que representa toda a tela de vídeo.
- **IDirectDrawSurface** - As superfícies representam as imagens que serão manipuladas pelo DirectDraw. A superfície primária eqüivale à área da memória que está sendo mostrada na tela de vídeo. A superfície secundária tem as mesmas características da superfície primária, sendo usada para a preparação do próximo quadro do jogo. Depois que o quadro estiver pronto, é feito uma troca entre as superfícies de forma que a superfície secundária passa ser a superfície primária, mostrando o conteúdo do quadro atual na tela de vídeo. Esta operação é feita no MRDX através da função “Mostrar_Tela( )”. As superfícies também são usadas para armazenar os desenhos dos diversos objetos do jogo. A estrutura IMAGEM do MRDX possui uma matriz de superfícies que é usada para armazenar os vários quadros dos personagens de um jogo.

c) DirectInput

O DirectInput é composto pelas seguintes interfaces:

- **IDirectInput** - Interface principal que deve ser instanciada para se trabalhar com DirectInput. A partir dela é que serão solicitados os dispositivos de entrada.
- **IDirectInputDevice** - Interface usada para a comunicação com os dispositivos de entrada. Para cada dispositivo como teclado, mouse e joystick, será associada uma “IDirectInputDevice”.

Todas as etapas de inicialização do DirectInput e dos dispositivos de entrada, são realizadas na função “Directx_Iniciar( )”. A leitura dos estados atuais dos dispositivos é feita na função “WinMain( )” (código A2.1), uma única vez por quadro, antes da chamada à função principal do jogo “Jogo_Main( )”. Feita a leitura, as informações dos estados dos dispositivos ficam armazenadas em estruturas internas do MRDX, específicas para cada dispositivo, de forma que quando o programador for verificar algum tipo de entrada do jogador, serão consultadas apenas as estruturas internas do MRDX ao invés de serem feitas chamadas ao hardware.

d) DirectSound e DirectMusic

As principais interfaces do DirectSound são :

- **IDirectSound -** Interface principal que deve ser instanciada para se trabalhar com do DirectSound. Representa a placa de som. No MRDX ela é instanciada na função “Directx_Iniciar( )”.
- **IDirectSoundBuffer** – Interface que representa os arquivos de sons. Os buffers permitem que os sons sejam armazenados na memória para que sejam usados posteriormente.

O MRDX possui uma matriz interna para armazenar os buffers de sons. A criação dos buffers no MRDX são feitas a partir da função “Carregar_Wav( )”, que cria o buffer de som baseado no arquivo *wav*, passado como parâmetro, em um espaço livre na matriz de buffers, retornando um identificador que será usado para posterior manipulação do buffer de som.

O DirectMusic é o componente do DirectX usado para manipulação de músicas no formato MIDI. Além disso, ele possui diversos recursos avançados que permitem a manipulação dinâmica da música. O MRDX lida com o DirectMusic de uma forma semelhante ao DirectSound, através do uso de uma matriz interna para o armazenamento das músicas. As principais interfaces do DirectMusic são:

- **IDirectMusicPerformance** – Principal interface que controla e manipula a reprodução das músicas. No MRDX ela é criada na função “Directx_Iniciar( )”.
- **IDirectMusicLoader** – Interface usada para carregar os arquivos MIDI na memória. No MRDX ela é instanciada na função “Directx_Iniciar( )” e usada na função “Carregar_Midi( )”.
- **IDirectMusicSegment** – Um “segmento” representa os dados de um arquivo de música MIDI.
- **IDirectMusicSegmentState** – Contém informações relacionadas ao estado atual de um “segmento” de música.

ANEXO 3: CONFIGURAÇÕES DO VISUAL C++

a) Para adicionar os diretórios (pastas) das bibliotecas do DirectX, acesse o menu :

Tools → Options → Directories

Selecione "Include files" e coloque o diretório "progjogos\dxsdk\include" (Figura A3.1).

Selecione "Library files" e coloque o diretório "progjogos\dxsdk\lib" (Figura A3.2).

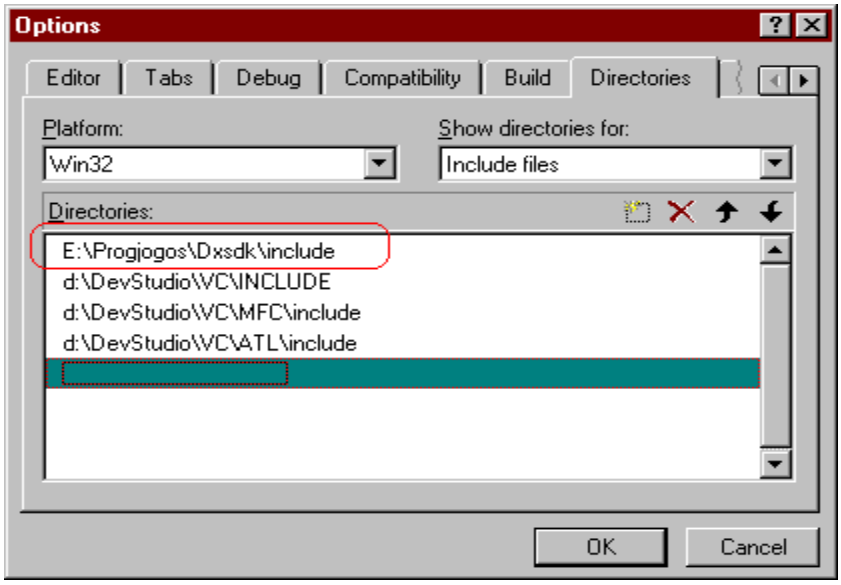

Figura A3.1 – Pasta dos arquivos com cabeçalhos do DirectX.

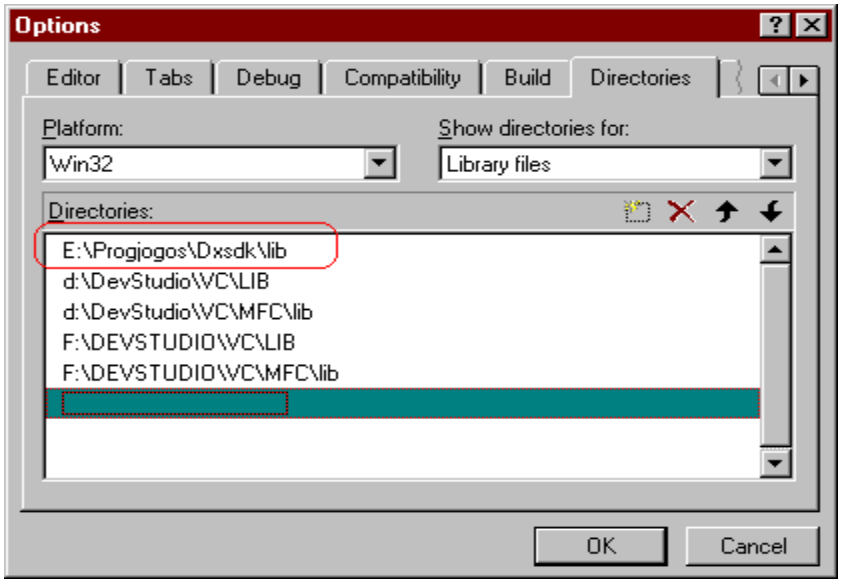

Figura A3.2 – Pasta dos arquivos com bibliotecas do DirectX.

b) Para adicionar as bibliotecas do DirectX em um projeto no Visual C++, acesse o menu :

Project → Settings →Link →Object / Library modules

Então acrescente os seguintes nomes de arquivos: ddraw.lib dsound.lib dinput.lib dxguid.lib winmm.lib. (Figura A3.3)

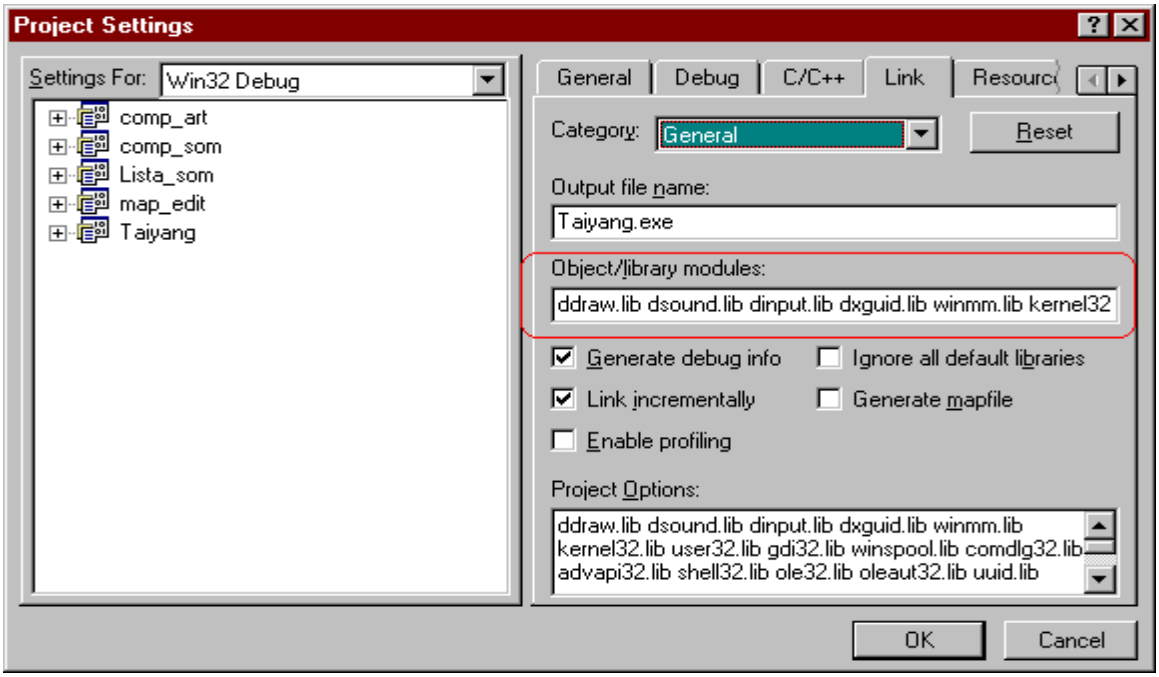

Figura A3.3 – Adição das bibliotecas do DirectX ao projeto.