CEETEPS
Centro Estadual de Educação Tecnológica Paula Souza
Governo do Estado de São Paulo
Faculdade de Tecnologia de Americana

DirectX 8.1 SDK

Marcos Aurelio da Silva

Orientador: Prof. Antonio Alfredo Lacerda

Americana – SP
Segundo Semestre de 2002

CEETEPS
Centro Estadual de Educação Tecnológica Paula Souza
Governo do Estado de São Paulo
Faculdade de Tecnologia de Americana

DirectX 8.1 SDK

Marcos Aurelio da Silva
RA: 002231-4

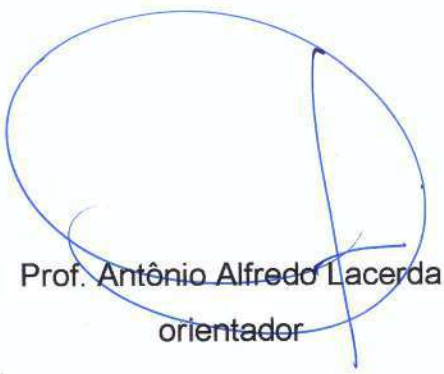
Trabalho de graduação apresentado à Faculdade de Tecnologia de Americana, com parte dos requisitos para obtenção do título de Tecnólogo em Processamento de Dados.

Orientador: Prof. Antonio Alfredo Lacerda

FATEC – AM

Americana – SP
Segundo Semestre de 2002

Trabalho de Graduação sob o título: DirectX 8.1 SDK, defendida por Marcos Aurélio da Silva e aprovada em 21 novembro de 2002, em Americana, Estado de São Paulo, pela banca examinadora constituída pelos professores:



Prof. Antônio Alfredo Lacerda
orientador

Prof. José Mário Scafi
Co-orientador

Prof. Mário José de Souza
Presidente da banca

Dedicatória

À minha querida Priscila.

Sumário

Resumo

O Microsoft DirectX 8.1 SDK é um kit de desenvolvimento de aplicativos para plataforma Windows, a principal função é a criação de uma camada intermediária entre a aplicação que está sendo desenvolvida e as APIs.

O kit hoje é utilizado no desenvolvimento de jogos por computador e videogames, contudo, é um poderoso software gerador de gráficos 2D/3D. O DirectX trabalha com API do Windows, fácil de ser utilizado, acessa placas aceleradoras de vídeo, joysticks, gamepad; além do fato de ser totalmente gratuito.

Pode ser usado com as principais linguagens de programação: C/C++, C++ BUILDER, Visual Basic e Delphi. Precisamos apenas adicionar as bibliotecas do DirectX ao projeto.

Abstract

Microsoft DirectX 8.1 SDK is a kit of development of applications for platform Windows, the main function is the creation of an intermediate layer among the application that is being developed and APIs.

The kit today is used in the development of games by computer and videogames, however, it is a powerful generating software of graphs 2D/3D. DirectX works with API of Windows, easy of being used, it accesses accelerating plates of video, joysticks, gamepad; besides the fact of being totally free.

It can be used with the main programming languages: C/C++, C++ BUILDER, Visual Basic and Delphi. We just needed to add the libraries of DirectX to the project.

1. Introdução

No início, os jogos eram feitos para o sistema operacional DOS, e não ofereciam muitos recursos, especialmente gráficos. Contudo, com o surgimento do Windows, a história mudou, os jogos começaram a ser aperfeiçoados e chamaram a atenção do público, bem como a dos grandes empresários com relação à lucratividade. Mas os jogos necessitavam de melhor performance e melhor utilização do hardware, assim surgiu o DirectX da Microsoft. A partir daí os jogos começaram a evoluir e não pararam mais.

Neste trabalho, procuramos descrever as principais características do kit DirectX 8.1 SDK, bem como seus componentes.

A princípio, apresentamos um breve histórico do surgimento do DirectX, além das suas diversas versões, destacando suas principais funções.

A seguir, descrevemos cada parte do kit Directx 8.1 SDK, começando com o DirectX Graphics (combinação do Microsoft DirectDraw e Microsoft Direct3D), quando oferecemos um exemplo prático em C++. Logo a seguir, apresentaremos o DirectInput, o DirectPlay, o DirectShow e por último o DirectSetup.

Procuramos utilizar exemplos em C++, porque é a linguagem mais utilizada para criação de jogos por computador, a maioria dos exemplos encontrados na Internet e sugeridos no próprio kit DirectX 8.1 SDK são feitos na mesma.

Finalizamos, então, com uma breve discussão sobre as vantagens e desvantagens da utilização do kit, procurando não compará-lo com outras ferramentas existente no mercado e tão pouco o apresentar como melhor opção, simplesmente procuramos demonstrar que o DirectX é uma ferramenta pouco utilizada em outras áreas, contudo tem uma grande performance que não devia ser deixada de lado.

2. Evolução do DirectX

Desde a sua primeira versão em 1995, o DirectX passou de uma rotina opcional para algo imprescindível. As evoluções foram constantes e o DirectX 2 surgiu em 1996, suportando a arquitetura Direct3D, dando início ao acesso direto ao hardware, sem precisar passar pelo Windows. Foi a partir daí que os jogos e outros aplicativos começaram a evoluir, e não pararam mais.

2.1. DirecX 5

Durante a evolução do DirectX, a arquitetura desenvolvida para Windows 95, NT 4.0 apoiava um subconjunto do DirectX para NT Service Pack 3. A partir do Windows 98 e o NT 5.0 o DirectX passou a ser parte integrante de ambos os sistemas operacionais, inclusive do Microsoft Internet Explorer 4.0. O DirectX 5 (1998) introduziu várias características novas, que não tinham sido incluídas em versões anteriores. Adicionou novas camadas no DirectX API, uma camada de baixo nível que comunica diretamente com a hardware de multimídia e uma camada de Mídia que permite aos programadores manipular objetos multimídias e streams.

Podemos citar as seguintes características do DirectX 5:

- O componente Direct3D API oferece melhor qualidade de imagem e maior facilidade de uso, o DrawPrimitive oferece ao programador maior flexibilidade para passar informações, polígonos, diretamente ao hardware em lugar de usar um buffer, além de Meshes progressivos e animações melhoradas;

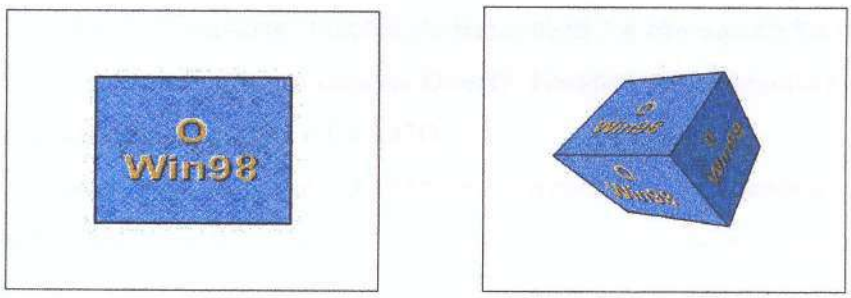


Figura 2.1,2.2: Exemplo de Meshes utilizando DrawPrimitive

- Características avançadas da tecnologia chamada atualmente de "TALISMAN" como sort-independent, anti-aliasing, range-based fog, anisotropic texture filtering e bufferless hidden surface elimination;

- O componente DirectDraw API suporta aceleração gráfica (AGP) e modos novos de baixa resolução e otimização para MMX;
- O DirectInput API suporta dispositivos de force-feedback e o game controller control panel;
- O DirectPlay suporta Windows NT security, client/server e lobby client API;
- O DirectSound3D captura e notifica API's, simplifica o uso de audio streams, além de suportar aceleração de hardware áudio 3D;
- Setup do DirectX é um banco de dados de configurações conhecidas que simplificam o setup. Se o hardware do usuário não for compatível, o setup do DirectX informa possíveis conflitos e oferece a possibilidade de não instalar ou de, após a instalação, reverter à configuração prévia;
- DirectX 5.0 foundation também inclui muitas características novas que são suportadas na versão de Windows NT 5.0, como suporte a vários monitores, porta gráfica avançada AGP, porta de extensão de vídeo (VPE), Universal Serial Bus (BUS) áudio, barramento IEEE 1394, suporta USB(Universal Serial Bus) e joysticks.

2.1.1. DirectX Foundation

DirectX Foundation é uma camada fundamental, é o coração e a alma do Microsoft DirectX. É feito de várias tecnologias que oferecem acesso a hardwares diferentes e lhes proporciona acesso melhorado para as características avançadas de hardware de alto-desempenho como aceleração de gráficos 3D, chips e placas de som. Estas APIs controlam as chamadas "funções de baixo nível," e são suportadas pelos quatro componentes que compõem a camada DirectX Foundation: O DirectDraw da Microsoft, DirectInput, DirectSound e Direct3D.

O DirectDraw é essencialmente um gerenciador de memória para gráficos e superfícies de vídeo que provêem acesso a funções de aceleração de hardware como blitting e overlays.

O DirectInput provê acesso de alto desempenho, para introduzir dispositivos inclusive o mouse, o teclado, o joystick, e o force-feedback (input/output).

O DirectSound capacita o hardware e o software de mixagem de som a capturar efeitos como posicionamento 3D e panning.

Enquanto, o Direct3D é uma interface de desenho para hardware 3D, sendo projetado para habilitar jogos de world-class e gráficos interativos 3D para computador com Windows.

2.1.2. DirectX Media

A DirectX Media é a camada acima do DirectX Foundation e provê serviços de alto-nível que apóiam animação, media streaming (transmissão e visualização de áudio e vídeo, como downloads da Internet) e interatividade.

O DirectX Foundation e o DirectX Media são compostos por vários componentes integrados, que incluem Microsoft DirectShow, DirectAnimation, Direct3D e Microsoft DirectPlay. Suportam DirectShow e DirectAnimation, que estão presentes nas mais recentes versões de Microsoft Internet Explorer, além do VRML.

O Microsoft DirectShow (antigamente chamado Microsoft ActiveMovie) é uma arquitetura media-streaming para a plataforma de Microsoft Windows que habilita a captura de alta qualidade e playback de fluxos de multimídias.

O DirectAnimation unificado, suporta animação, streaming e integração de tipos de mídia diversos, como gráficos de vetor 2D, gráficos 3D, sprites, áudio e vídeo.

O Direct3D é um gerenciador de gráficos 3D de alto-nível, que simplifica a construção de animação de palavras e dados 3D.

O DirectPlay faz facilmente a conexão de jogos na Internet com um modem ou uma rede. É um software de interface que simplifica aplicações de acesso a serviços de comunicação.

2.2. DirectX 7.0

O DirectX 7.0 fornece melhor otimização de performance, gráficos e sons aperfeiçoados e uma grande facilidade de uso, reforçando a posição do Windows como a plataforma preferida para desenvolvimento de jogos.

Dentre as características podemos destacar:

- Ampliado suporte para transformações de aceleração de hardware e iluminação artificial através de aceleradores 3D, liberando a CPU para tarefas como inteligência artificial ou cálculos de física;
- Mapeamento de ambiente com Cubic Environment Maps para reflexões e efeitos de luz mais sofisticados;
- Suporte para texturas projetadas e transformações de texturas para objetos 3D mais realistas;
- Aceleração de hardware para a API do DirectMusic para a criação de trilhas sonoras mais ricas e complexas;
- Algoritmos melhores para som 3D e flexibilidade para gerenciar as capacidades de mixagem do hardware;
- Suporte para o ambiente de desenvolvimento Visual Basic;
- Modelo de interface do Universal Serial Bus (USB) para dispositivos force-feedback.

2.3. DirectX 8.0

A partir da versão 8.0, o DirectX estabeleceu-se como parte integrante da manipulação de multimídia baseada no Windows. Tal como as APIs, o DirectX foi projetado para facilitar a vida dos responsáveis pelo desenvolvimento de software. Em vez de escrever códigos para várias combinações possíveis de equipamentos, os programadores podem escrever para as APIs, ou seja, uma camada intermediaria que traduz os comandos. Essencialmente, na maioria dos jogos modernos, a leitura de áudio e vídeo ou a edição de aplicações, e as ferramentas de desenho 3D, dependem do DirectX. Isto porque a API é responsável pelo áudio posicional entre um jogo e uma placa de som, aplicando as funções gráficas avançadas, tais como o mapa dos acidentes geográficos do ambiente do jogo numa imagem construída a 3D, ou até mesmo ativando uma janela secundária de visualização de DVD.

Embora o DirectX 8.0 seja compatível com as novas tecnologias é necessário que assegure a compatibilidade com versões anteriores, mas os problemas relacionados a hardware continuam acontecendo. Isto se aplica, especialmente, a produtos que não foram atualizados (drivers) regularmente e algumas placas gráficas 3D que parecem ter uma revisão semanal. Se um problema surgir, não é possível desinstalar o DirectX nem reverter a uma versão prévia; a menos que se execute um reinstalação total do seu software do Windows. Contudo atualmente, na internet

existe componentes que alegam, segundo seus criadores, retirar completamente do sistema o DirectX.

Entretanto, o DirectX 8.0 oferece algumas melhorias interessantes, na medida em que aperfeiçoou todos os seus sete componentes principais: Direct3D, DirectDraw, DirectInput, DirectMusic, DirectPlay, DirectSound e DirectShow.

Particularmente, acreditamos que a discussão a respeitos das melhorias trazidas pelo DirectX 8.0 se torna mais relevante se o dividirmos em três categorias: jogos, gráficos 3D e áudio/vídeo.

2.3.1. Jogos

O DirectX tem sido tradicionalmente um elemento muito importante no campo dos jogos. Controladores, placas de som, placas gráficas 3D, e jogos on-line, todos se apóiam no DirectX para providenciar características avançadas. De modo geral, os jogadores beneficiam-se de quase todos os novos aspectos do DirectX 8.0, com a possível exceção do grupo DirectShow que controla principalmente aspectos relacionados com formatos de vídeo e DVD.

Em termos de jogos, os aperfeiçoamentos do DirectPlay e do DirectInput foram impressionantes. A Microsoft desenvolveu o sistema de comunicação Sidewinder GameVoice. O GameVoice é uma combinação que permite o diálogo entre os companheiros de uma mesma equipe e entre os adversários durante jogo. Também permite a utilização de comandos de voz para desencadear determinadas ações básicas como: parar, atacar, entre outras. O DirectPlay Voice torna a comunicação de voz mais suave através de um modem ou, até mesmo através do IP, devido às suas aperfeiçoadas características de rede: ao efetuar a transmissão, utilizando uma largura de banda mais baixa para o áudio, o processo torna-se mais rápido e mais claro.

O DirectX 8.0 também trabalha com o DirectSound para administrar tanto o jogo como o sinal áudio de voz mais eficaz. Tradicionalmente, a comunicação de voz em jogos foi, quando muito, uma novidade entre jogadores com acesso a ligações de alta velocidade. Jogar através de um modem era absurdo devido à pobre qualidade do som, aos longos atrasos de transmissão, e do impacto geral na performance.

O GameVoice utiliza tecnologia de compressão em conjunção com a administração simplificada das comunicações de entrada e de saída do DirectX 8.0,

proporcionando uma comunicação de voz razoavelmente rápida e clara. Está longe da qualidade do telefone - o som é um pouco áspero e existe um pouco demorado - mas há uma melhoria significativa em relação a tentativas prévias da Microsoft e outros.

O DirectPlay anuncia uma evolução significativa no código de ligação em rede para jogos com vários adversários. O DirectInput pode determinar que tipo de jogo se está jogando.

2.3.2. Gráficos 3D

A nVidia lançou um kit de desenvolvimento de software no seu site Web (www.nvidia.com/developer.nsf) que apresenta aperfeiçoamentos que poderemos ver ao utilizar o DirectX 8.0. A Microsoft licenciou grande parte da funcionalidade 3D diretamente da nVidia. Isto significa que todos os fabricantes de placas gráficas que otimizem os seus controladores para DirectX 8.0 podem usar estas melhorias, desde que o hardware satisfaça as exigências mínimas, como a quantidade de memória e de compressão.

A adição mais significativa ao componente Direct3D do DirectX 8.0 é a sua capacidade de programação. Define uma linguagem que é usada para programar o hardware. Para funções com gráficos 3D, esta linguagem é simples e dá aos programadores a possibilidade de programar utilizando vértices (dados de geometria tridimensional), também conhecidos por sombreamento de vértice e sombreamento de pixel.

Os sombreamentos de vértice dão aos programadores a oportunidade de manipular vértices como grupos (componentes triangulares e cuneiformes), isto faz com que os personagens se movam de forma mais realista, ao invés de terem movimentos irregulares tipo robô como se passa hoje em dia. Outra técnica, chamada deformação de superfícies, cria superfícies e terrenos mais realistas. O entrelaçamento de vértices dá às malhas reticulares a capacidade de se dobrarem e de se entrelaçarem (morph) definindo movimentos mais realistas. O sombreamento em camadas acrescenta ainda mais realismo à cena, uma vez que o sombreamento mostra graduações que induzem a percepção de profundidade.

O DirectX 8.0 permite a criação de mapas de relevo, multiamostragem de forma a reduzir distorções nas arestas das imagens, e melhores texturas a 3D. Contando com bastante memória - 32MB ou mais - e suporte de compressão de

hardware, as texturas 3D alcançam um nível de realismo nunca visto antes em jogos 3D.

O MechWarrior 4 aplica bem a técnica do sombreamento em camadas, especialmente nas suas cenas 3D mais elaboradas. As máquinas de guerra gigantes (Meshes) saem de uma nave mãe, ao reduzirem a velocidade de descida, as Meshes produzem efeitos de propulsão inversa completos, muito reais.

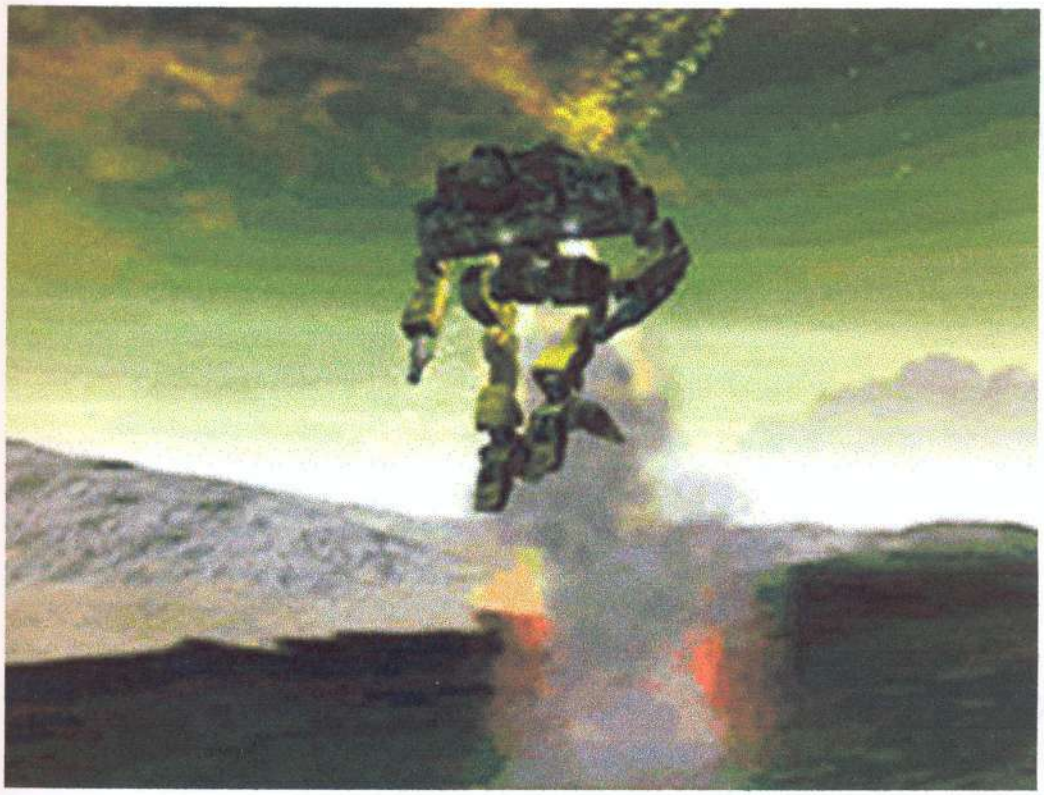


Figura 2.1: Técnica do Sombreamento(MechWarrior 4)

Outro efeito particularmente notável, atualmente suportado no DirectX 8.0, é a difusão de movimento. A difusão de movimento acrescenta um efeito de desenho animado aos modelos 3D. Arrasta uma sombra transparente do modelo pelo caminho do seu movimento. O resultado pode ser dramático em jogos 3D, desenho de títulos Web, e outros elementos 3D. No lançamento do DirectX 8.0 3dfx já o incorporou na sua linha de placas gráficas Voodoo5; e a nVidia iniciava a sua implentação.

2.3.3. Áudio/Vídeo

Um dos aspectos mais interessantes do DirectX 8.0 é a sua capacidade sonora. O formato de controlador áudio WDM surgiu pela primeira vez no DirectX 7.0, mas não era nada mais que um simples codec - o equivalente a um Winmodem.

Com o DirectX 8.0, o WDM dá mais poder aos fabricantes de placas de som, sob a forma de um forte suporte ao áudio ambiental e posicional, assim como os efeitos que são processados instantaneamente. Com esta capacidade, já presente no sistema operacional, os fabricantes de placas podem concentrar-se na potência pura do trabalho, tornando os circuitos áudio de baixo preço com melhor qualidade. Entre as características inclui a capacidade de gerar efeitos de som instantâneos, os quais podem ser utilizados na edição áudios e na edição de jogos. Potencialmente, a música e os sons podem mudar dinamicamente. Imagine uma página da Web cuja música de fundo aumente ou diminua a intensidade dependendo de onde e da ordem em que o visitante dá o clique. Esta é uma das aplicações possíveis com os efeitos de som instantâneos.

A placa de som que utiliza o chip de áudio Canyon 3D-2 da ESS Technology, juntamente com uma versão alfa do seu controlador de dispositivo tem um aspecto interessante, é o tamanho do controlador, com 570K, o que constitui uma mudança significativa comparando com de 10MB observados no passado. De acordo com a ESS, isto se deve ao fato de todas as funções, que normalmente dependem do software do fabricante, terem sido integradas no DirectX 8.0. Com tal nível de integração, a placa de som já pode usar o espaço que está sobrando para melhorar o som, o que significa ter áudio de maior qualidade.

3. DirectX 8.1

3.1. DirectX Graphics

O DirectX Graphics é essencialmente um gerenciador de memória vídeo. Sua capacidade mais importante é permitir ao programador armazenar e manipular bitmaps diretamente na memória vídeo, permitindo aproveitar-se do vídeo hardware blitter para utilizar técnicas como blitting de bitmaps; sendo possível o blit de uma memória de vídeo para outra.

Usando o vídeo hardware blitter, contudo, é mais rápido que utilizar o blitting da memória do sistema para a de vídeo. Isso é verdade hoje com placas de pelo menos 64 bits de vídeo e não esquecendo que as operações de hardware blit são independentes da CPU e então a deixa livre para continuar trabalhando.

A implementação do DirectX Graphics é um pouco diferente do que existia no DirectX 7.0 ou mais velho, o DirectDraw está morto, substituído completamente pelo Direct3D. Contudo consideramos mais fácil dar uma visão do que era o DirectDraw e depois descrever o novo Direct3D.

3.1.1. DirectDraw

O DirectDraw é o componente do DirectX que trabalha com a aceleração gráfica nas aplicações. Tudo que tiver necessidade de ser exibido na tela do monitor será função deste objeto. Podemos acrescentar, que o DirectDraw suporta outros tipos de aceleração de hardware da placa de vídeo, como suporte de hardware para sprites e z-buffering.

3.1.1.1. Páginas de Flipping – Acesso para Animação

Podemos criar superfícies com o DirectDraw como já foi mencionado, ou seja, locais onde armazenamos toda a informação de uma página gráfica. Isto é, antes de iniciarmos uma animação desenhamos todas as imagens necessárias e depois exibimos quadro a quadro, dando a impressão de movimento.

Normalmente criamos três superfícies:

- Superfície Primária (Primary Surface) que é a superfície que representa a tela do monitor. É o que está na tela. Se colocarmos uma

imagem nessa superfície, a imagem será imediatamente exibida na tela do monitor;

- Superfície Secundária (BackBuffer) é a superfície onde as imagens serão alteradas. Nesta superfície desenhamos todo o quadro de animação antes de copiá-lo para a superfície primária, de modo que a imagem forma-se na tela inteiramente no mesmo momento que a superfície primária, ou seja, basta depois transformarmos a primária em secundária e vice-versa. Na superfície primária é desenhado o primeiro quadro de animação e no backbuffer está sendo desenhado o segundo e assim ao trocá-las damos a impressão de movimento, essa técnica é chamada de flipping;
- Superfície Terciária (Off-Screen Surface) são as superfícies adicionais, que são usadas para armazenar imagens, como os quadros de uma animação, os tiles, o mapa, o papel de fundo, etc. Esse tipo de superfície não é exibido na tela, só serve para armazenar imagens, que mais tarde serão copiadas para a tela, contudo, ao copiarmos o off-screen para o backbuffer, ou diretamente para superfície primária estamos realizando um blitting e não um flipping. São técnicas semelhantes, mas que usam funções diferentes.

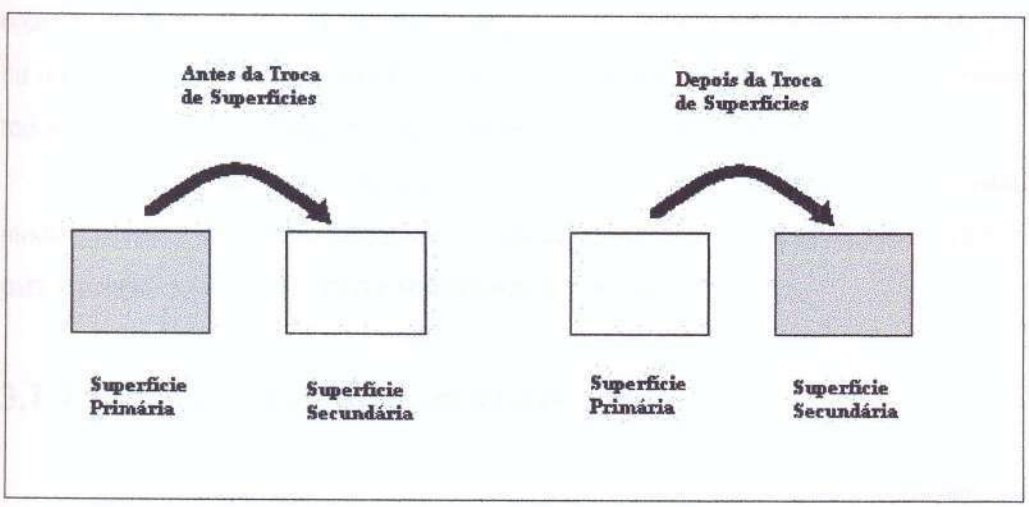


Figura 3.1: Processo de flipping entre as superfícies

Outro fato importante é que o DirectX pode usar objetos COM, isto é, possibilita o uso de exportação de DLLs e polimorfismo.

3.1.1.2. Superfície DirectDraw – Como Acessar a Memória de Vídeo

No DirectDraw, o objetivo é colocar tantos bitmaps na memória de vídeo quanto possível. Com o DirectDraw, todas as memórias de vídeo estão disponíveis. Podemos usar ambos, a primária e o buffer offscreen de vídeo para armazenar vários bitmaps. Todos esses pedaços da memória de vídeo são referidas como superfície no DirectDraw. Quando abrimos um bitmap que representa um sprite, e armazenamos na memória de vídeo, sua primeira superfície é criada, a qual é um pedaço da memória de vídeo, e então efetuamos o blit do bitmap nessa superfície, através disso, efetivamente copiamos o bitmap na memória de vídeo. Esse bitmap agora pode permanecer na memória de vídeo para usarmos o quando necessário.

A memória de vídeo que está corretamente exibida na tela é uma superfície, e é chamada de superfície primária. Essa superfície é tão grande quanto à quantidade de memória necessária para o corrente modo de tela. Se o modo de tela é 640x480 com 256 cores (8 bits por pixel), então a superfície primária usará 307.200 bytes de memória de vídeo. Usualmente criamos uma outra superfície offscreen que será do mesmo tamanho da superfície primária para usá-la como página de flipping. Dessa forma precisamos de 614.400 bytes de memória para iniciar as telas, um dos mais importantes pré-requisitos necessários de hardware para a velocidade de games com DitrectDraw é o tamanho da memória de vídeo. Quando executamos um game ou uma aplicação com DirectDraw criamos as superfícies na memória do sistema, e todos os benefícios de blits de hardware serão avaliados nessas superfícies.

As placas de vídeo tem pelo menos 1 mega de memória, de qualquer modo 1 mega é suficiente, entretanto, 2 megas é provavelmente o mínimo necessário para iniciarmos uma aplicação e trabalharmos com maior rapidez.

3.1.1.3. Programando com DirectDraw

Para utilizar o exemplo abaixo será necessário ter instalado os drives do DirectX, além de um compilador de C/C++, no caso utilizamos o MS Visual C++ 6.0, contudo devemos mencionar que os exemplos utilizados aqui e nos outros capítulos foram retirados do Kit DirectX SDK da Microsoft, cabendo a mesma os direitos autorais.

Para compilar o projeto é necessário criar uma aplicação, no caso do MS Visual C++ 6.0, uma simples WIN 32 Application (NÃO o console application), acrescentar ao projeto as bibliotecas DDRAW.DLL, DDRAW.H além do IMPLIB.EXE utility.

3.1.1.3.1. Iniciando o DirectDraw

A primeiro passo para iniciar uma aplicação gráfica com DirectX, é criar o objeto DirectDraw. Então:

```
// Exemplo do artigo original de:
// DirectDraw Programming Tutorial
// por Lan Mader
// site: GameDev.net

// Variável Global (ugh)
LPDIRECTDRAW lpDD; // Objeto DirectDraw definido no DDRAW.H

/*
 * Funções para inicializar o DirectDraw:
 *  1) Criando o  objeto DirectDraw
 *  2) Fixando o modo de cooperação
 *  3) Fixando a resolução de vídeo
*/

bool DirectDrawInit(HWND hwnd)
{
  HRESULT ddrval;

  /*
  * Criando o função main com o objeto DirectDraw.
  *
  * Iniciando o objeto DirectDraw.
  */
  ddrval = DirectDrawCreate( NULL, &lpDD, NULL );
```

```
  if( ddrval != DD_OK )
  {
    return(false);
  }

  /*
  * O modo de cooperação determina como a aplicação será exibida (full-screen ou
  * janela), e como será a interação com outras aplicações Windows.
  * No caso de nossas aplicações, vamos trabalhar apenas com tela cheia; assim,
  * trabalharemos com as flags DDSCL_EXCLUSIVE e DDSCL_FULLSCREEN.
  */

  ddrval = lpDD->SetCooperativeLevel( hwnd, DDSCL_EXCLUSIVE |
DDSCL_FULLSCREEN );
  if( ddrval != DD_OK )
  {
    lpDD->Release();
    return(false);
  }

  /*
  * Fixamos a resolução de vídeo e o número de cores para 640x480x8
  */

  ddrval = lpDD->SetDisplayMode( 640, 480, 8);
  if( ddrval != DD_OK )
  {
    lpDD->Release();
    return(false);
  }

  return(true);
}
```

3.1.1.3.2. Criando a Superfície Primária e Determinando a Página de Flipping

Criaremos uma superfície com um backbuffer. É chamada de superfície complexa, primária no DirectDraw. O backbuffer é atachado a superfície primária. Quando limpamos e finalizamos o programa, precisamos apenas chamar o Release para o ponteiro da superfície primária e backbuffer será liberado.

```
// Variável Global (ugh)
LPDIRECTDRAWSURFACE lpDDSPrimary; // Superfície Primária DirectDraw
LPDIRECTDRAWSURFACE lpDDSBack; // Superfície back DirectDraw

/*
 * Criando a superfície primária que será usada no flipping, com o
 * atachado backbuffer
 */
bool CreatePrimarySurface()
{
  DDSURFACEDESC ddsd;
  DDSCAPS ddscaps;
  HRESULT ddrval;

  // Criando a descrição da superfície primária com 1 backbuffer
  memset( &ddsd, 0, sizeof(ddsd) );
  ddsd.dwSize = sizeof( ddsd );

  ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
  ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP |
DDSCAPS_COMPLEX;
  ddsd.dwBackBufferCount = 1;

  ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
  if( ddrval != DD_OK )
  {
    lpDD->Release();
```

```
    return(false);
  }

  // atachando o backbuffer à superfície primária
  ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
  ddrval = lpDDSPrimary->GetAttachedSurface(&ddscaps, &lpDDSBack);
  if( ddrval != DD_OK )
  {
    lpDDSPrimary->Release();
    lpDDD->Release();
    return(false);
  }

  return true;
}
```

3.1.1.3.3. Abrindo um Bitmap na Memória de Vídeo

O próximo passo é abrir os bitmaps do game. Para que possamos utilizar estes bitmaps, devemos armazená-los em uma superfície terciária (off-screen). Assim, devemos criar uma descrição parecida com a da superfície primária. Em tal descrição fixamos as flags com DDSD_CAPS, DDSD_HEIGHT e DDSD_WIDTH. Com isto temos condição de fixar os membros dwHeight e dwWidth de acordo com o tamanho do bitmap. O membro ddsCaps.dwCaps deve ser DDSCAPS_OFFSCREENPLAIN, indicando que será uma superfície terciária.

```
/*
 * Função para criar  uma superfície offscreen e abrir um bitmap do
 * disco. O campo szBitmap é o nome do Bitmap.
 */
IDirectDrawSurface * DDLoadBitmap(IDirectDraw *pdd, LPCSTR szBitmap)
{
  HBITMAP hbm;
  BITMAP bm;
  IDirectDrawSurface *pdds;
```

```
    // LoadImage tem como principal funcionalidade abrir o arquivo do disco,
    // contudo, funciona para Windows 95 ou superior.
    hbm = (HBITMAP)LoadImage(NULL, szBitmap, IMAGE_BITMAP, 0, 0,
    LR_LOADFROMFILE|LR_CREATEDIBSECTION);

    if (hbm == NULL)
        return NULL;

    GetObject(hbm, sizeof(bm), &bm); // Tamanho do bitmap

    /*
    * Criando a DirectDrawSurface para o bitmap
    * com a função CreateOffScreenSurface()
    */

    pdds = CreateOffScreenSurface(pdd, bm.bmWidth, bm.bmHeight);
    if (pdds) {
        DDCopyBitmap(pdds, hbm, bm.bmWidth, bm.bmHeight);
    }

    DeleteObject(hbm);

    return pdds;
}

/*
 * Criando a superfície doDirectDraw especificando o tamanho
 * A superfície estará na memória de vídeo, se não,
 * será criado no sistema de memória
 */
IDirectDrawSurface * CreateOffScreenSurface(IDirectDraw *pdd, int dx, int dy)
{
    DDSURFACEDESC ddsd;
    IDirectDrawSurface *pdds;
```

```
    //
    // Criando o DirectDrawSurface para o bitmap
    //
    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwWidth = dx;
    ddsd.dwHeight = dy;

    if (pdd->CreateSurface(&ddsd, &pdds, NULL) != DD_OK)
    {
        return NULL;
    } else {
        return pdds;
    }
}

/*
 * Essa função copia previamente e abre o bitmap na superfície DirectDraw.
 * Notamos que podemos obter o mesmo resultado usando a GDI para
 * a superfície DirectDraw, com a qual a função BitBlt colocará o bitmap
 * na superfície.
 */
HRESULT DDCopyBitmap(IDirectDrawSurface *pdds, HBITMAP hbm, int dx, int
dy)
{
    HDC hdcImage;
    HDC hdc;
    HRESULT hr;
    HBITMAP hbmOld;

    //
    // Selecionamos o bitmap na memória DC que estamos utilizando.
```

```
    //
    hdcImage = CreateCompatibleDC(NULL);
    hbmOld = (HBITMAP)SelectObject(hdcImage, hbm);

    if ((hr = pdds->GetDC(&hdc)) == DD_OK)
    {
        BitBlt(hdc, 0, 0, dx, dy, hdcImage, 0, 0, SRCCOPY);
        pdds->ReleaseDC(hdc);
    }

    SelectObject(hdcImage, hbmOld);
    DeleteDC(hdcImage);

    return hr;
}
```

Obs: Os DCs (Device Context) representam superfícies nas quais podemos desenhar qualquer coisa. Estas superfícies representam as saídas gráficas, ou seja, o monitor, a impressora, etc. É através deles que vamos desenhar linhas, polígonos, bitmaps, e outros objetos gráficos.
Em termos técnicos, vamos designar um DC para um determinado controle (superfície ou objeto gráfico). Assim feito, desenhamos qualquer gráfico neste DC, e então o atualizamos, mostrando as alterações realizadas.

3.1.1.3.4. Abrindo uma Paleta

As paletas de cores são mecanismos pelos quais uma grande quantidade de cores pode ser acessada sem grandes dispêndios de memória. Os modos paletizados vão até 8 bits, ou seja, 256 cores; acima disto, não há vantagens em trabalharmos com as paletas.

Quando utilizamos as paletas de cores temos vantagens e desvantagens. A maior vantagem é a grande performance de nossas aplicações pelo menor uso de memória, pois deixamos de enumerar mais de 16 milhões de cores. Outra grande vantagem é utilizarmos efeitos que são obtidos com a utilização dos modos paletizados, como por exemplo: animação por cores gradativas, como fade que

consiste em uma tela aparecer ou desaparecer por uma transição gradativa para uma cor sólida, normalmente preta, logo podemos destacar o fade in, no qual saímos da cor sólida e montamos a imagem, e o fade out, no qual saímos da imagem, e terminamos em uma tela de cor sólida, entre outros. A grande desvantagem é que todas as imagens a serem utilizadas na aplicação devem ser desenhadas com o número de cores da paleta, o que causa um planejamento completo na parte de arte e design. O preço é pequeno perto dos recursos adquiridos.

Em que consiste trabalhar com as paletas? Quando paletizamos uma aplicação, geramos um array contendo ponteiros para as cores verdadeiras em RGB. Logo, convertemos cada cor dos pontos da nossa imagem em um membro deste array. Ao analisarmos o valor de um determinado pixel, direcionamos para a cor verdadeira com decomposições de vermelho, verde e azul.

```
/*
 * Criando uma paleta com DirectDraw para determinado bitmap no disco.
 * O parâmetro szBitmap é o nome do arquivo bitmap.
 *
 */
IDirectDrawPalette * DDLoadPalette(IDirectDraw *pdd, LPCSTR szBitmap)
{
  IDirectDrawPalette* ddpal;
  int i;
  int n;
  int fh;
  PALETTEENTRY ape[256];

  if (szBitmap && (fh = _lopen(szBitmap, OF_READ)) != -1)
  {
    BITMAPFILEHEADER bf;
    BITMAPINFOHEADER bi;

    _lread(fh, &bf, sizeof(bf));
    _lread(fh, &bi, sizeof(bi));
    _lread(fh, ape, sizeof(ape));
    _lclose(fh);
```

```
    if (bi.biSize != sizeof(BITMAPINFOHEADER))
        n = 0;
    else if (bi.biBitCount > 8)
bsp;        n = 0;
    else if (bi.biClrUsed == 0)
        n = 1 << bi.biBitCount;
    else
        n = bi.biClrUsed;

    //
        nbsp;       // a tabela de cores DIB tem cores iniciadas em BGR e não RGB
    // então somente aproximamos com o flip.
    //
    for(i=0; i<n; i++ )
    {
        BYTE r = ape[i].peRed;
        ape[i].peRed = ape[i].peBlue;
        ape[i].peBlue = r;
    }
  }
  if (pdd->CreatePalette(DDPCAPS_8BIT, ape, &ddpal, NULL) != DD_OK)
  {
    return NULL;
  } else {
    return ddpal;
  }
}
```

3.1.1.3.5. Chaveamento de Cor – Color Keying

O chaveamento de cor consiste em escolhermos uma cor não utilizada nos frames da sprite, e preenchermos o background da imagem com ela. Através de funções do DirectX, definimos esta cor como sendo uma cor de chaveamento fonte ou destino. A cor chaveamento fonte desaparece quando copiamos o frame para a

superfície, deixando o fundo da sprite transparente. A cor de chaveamento destino é a única cor que recebe, na superfície destino, a imagem copiada.

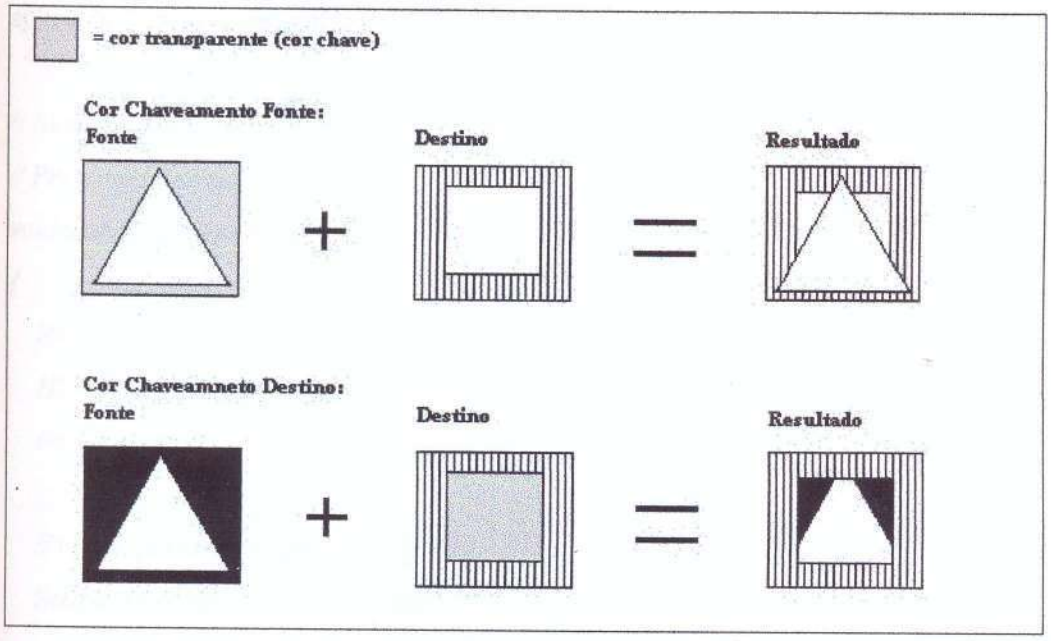


Figura 3.2: Cor Chaveamento Fonte e Cor Chaveamento Destino

```
// Color key para os bitmap.
//
// Escolhendo a cor preta como cor de chaveamento,
// usamos a flag DDPCAPS_ALLOW256 quando criamos a paleta
//

DDCOLORKEY ddck;
ddck.dwColorSpaceLowValue = 0xff;
ddck.dwColorSpaceHighValue = 0xff;

// lpDDSSomeBitmapSurface é a superfície DirectDraw com o
// bitmap aberto. Precisaremos inciar para cada
// superfície contendo o bitmap.
lpDDSSomeBitmapSurface->SetColorKey( DDCKEY_SRCBLT, &ddck );
```

3.1.1.3.6. Compondo a Cena

Nesse ponto podemos iniciar a criação do game, simulador ou outra aplicação baseada em gráficos.

```
// Essa função será chamada repetidamente até o final do
// PeekMessage()loop.
void updateFrame( void )
{
  RECT rcRect;
  HRESULT ddrval;
  int xpos, ypos;

  // O Blit preenche o próximo frame
  SetRect(&rcRect, 0, 0, 640, 480);

  // blit o bitmapa no background. Este é um bitmap de  640x480,
  // somente sera  preenchida a tela (anteriormente, setamos o modo
  // video para 640x480x8.
  // O parâmetro lpDDSOne é uma hipotetica superficie com o
  // bitmap background aberto. LpDDSBack é seu backbuffer.
  ddrval = lpDDSBack->BltFast( 0, 0, lpDDSOne, &rcRect,
  DDBLTFAST_NOCOLORKEY | DDBLTFAST_WAIT);

  if( ddrval != DD_OK )
  {
    return;
  }
  SetRect(&rcRect, 0, 0, 32, 32); // fictício sprite bitmap com 32x32 de tamanho
  // xpos e ypos representam alguma possível localização do sprite.
  // lpDDSMySprite é justamente a superfície com o bitmap para o sprite.
  ddrval = lpDDSBack->BltFast( xpos, ypos, lpDDSMySprite,
  &rcRect, DDBLTFAST_SRCCOLORKEY | DDBLTFAST_WAIT );
  if( ddrval != DD_OK )
  {
```

```
    return;
  }

  // Flip a superficie
  ddrval = lpDDSPrimary->Flip( NULL, DDFLIP_WAIT );
} /* updateFrame */
```

Cumpre lembrar, que as funções BltFast(), e Flip() têm o ultimo parâmetro a flag DDXX_WAIT. Isto é necessário porque cada uma dessas funções é assíncrona. Ou seja, elas retornam quando a operação é iniciada com sucesso (mas ambas ainda não terminam), ou quando um erro ocorre e a função não é capaz de iniciar com sucesso. O erro mais comum retornado é indicar que a superfície está ocupada completando uma prévia tarefa. Usando o flag DDXX_WAIT informamos ao DirectDraw para deter a penosa operação ao longo da superfície ocupada, ou até ocorrer outro erro.

Note que a primeira chamada de BltFast(), é para utilizar o blitting do background, usamos a flag DDBLTFAST_NOCOLORKEY. A chamada é para BltFast ignorar o color key, e podemos melhorar a performance do blit. Dessa maneira, algumas áreas transparentes de nosso bitmap serão copiadas e então serão transparentes, mas isso está certo desde que esse é o bitmap background e não um sprite.

Quando a função Flip é chamada, o objeto da superfície associado com a memória de vídeo será trocado. Essa é a maneira pela qual um buffer primário é agora o backbuffer e vice-versa. O resultado é que podemos reter o objeto usando o ponteiro backbuffer compondo a cena, e chamando o Flip (), tendo o caminho de onde os buffers estão.

3.1.1.3.7 Limpando

Para limparmos o objeto DirectX, a melhor maneira é chamar a função Release():

```
/*
 * Chamando o Release para todos os objetos criados e limpando-os
```

```
 */
void finiObjects( void )
{
  if( lpDD != NULL )
  {
    if( lpDDSPrimary != NULL )
    {
      lpDDSPrimary->Release();
      lpDDSPrimary = NULL;
    }

    if( lpDDSOne != NULL )
    {
      lpDDSOne->Release();
      lpDDSOne = NULL;
    }

    if( lpDDPal != NULL )
    {
      lpDDPal->Release();
      lpDDPal = NULL;
  }
  lpDD->Release();
  lpDD = NULL;
} /* fim Objetos */
```

Quando chamamos o Release no objeto DirectDraw atual (a instancia do LPDIRECTDRAW) será referenciada a zero, e o objeto será destruído. E o modo de vídeo original será restaurado.

3.1.2. Direct3D

Direct3D contém somente 12 interfaces:

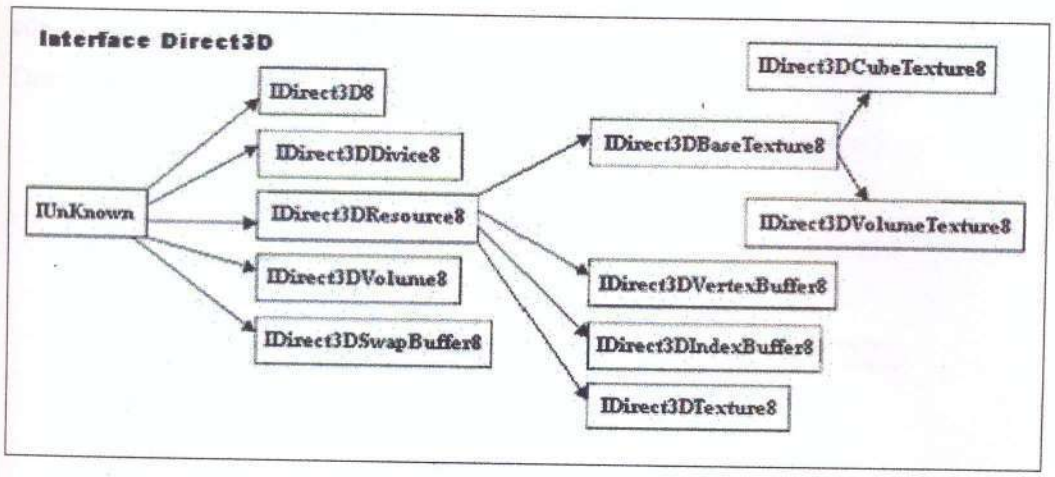


Figura 3.3: As interfaces do Direc3D

Uma das novas características é a adição do sombreamento. Além do Direct3D vir com a biblioteca chamada D3DX, que contém APIs para criar variedades de sprites, para fontes, para texturas.

Há muitas semelhanças entre a D3DX e o OpenGL. Porém, operações de matriz são muito simples com a D3DX; e mais fácil de trabalhar que o OpenGL:

Direct3D	OpenGL
BeginScene	GlBegin
EndScene	GlEnd
DrawPrimitive	GlDrawElements
SetRenderState	GlEnable
SetTexture	GlBindTexture
Clear	GlClear

A programação 2D não está nem sequer morta com a remoção de DirectDraw. DirectX 8.1 tem uma interface de sprites em sua biblioteca de D3DX. Porém, o modo preferido para fazer gráficos 2D está com texturas simples. Desde que chaveamento de cor foi retirado, o único modo para fazer transparência é com alpha blending.

3.2.2.1 Visão Geral da Arquitetura do Direct3D

Existem duas maneiras de programar na arquitetura Direct3D: sombreamento de vértices (vertex shaders) e sombreamento de pixel (pixel shaders). Vertex shaders são comandos executados antes de um vertex assembly e operações

com vetores. O pixel shaders são comandos executados depois de chamadas DrawPrimitive e operações em pixels.

O seguinte diagrama simplifica e ilustra a Arquitetura Direct3D:

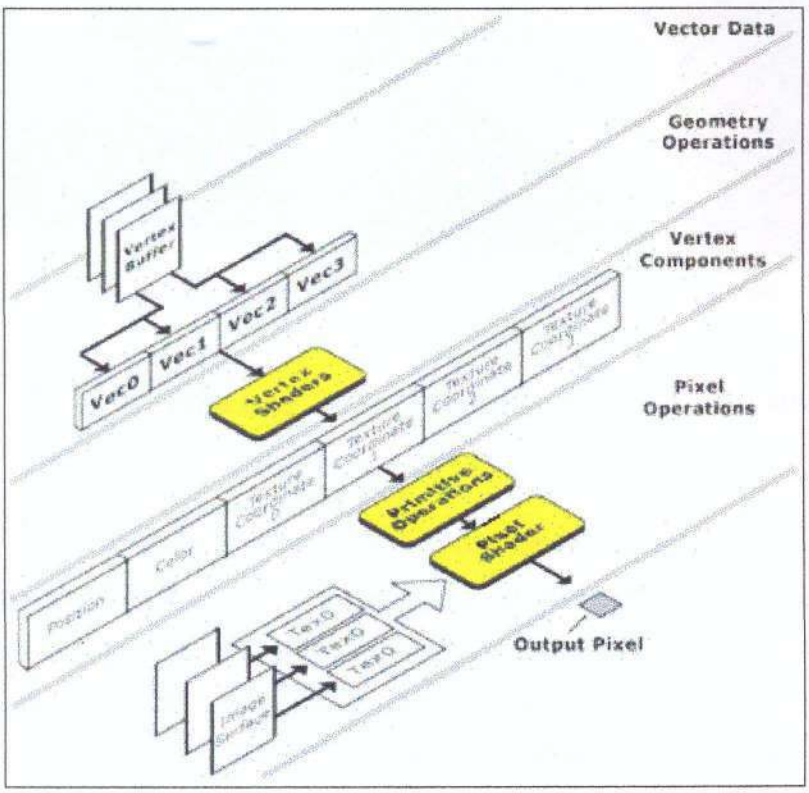
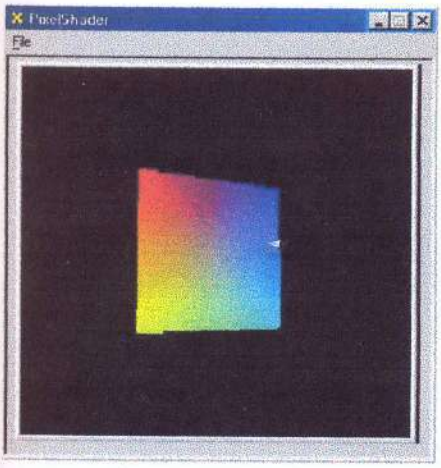


Figura 3.4: Diagrama da Arquitetura Direct3D

Como mostra o diagrama, o vertex buffers espalha vertex data no vertex shader. O vertex shader realiza operações geométricas usando instruções definidas pelo vertex shader assembler da biblioteca D3DX.

Os dados espalham para o vertex shader, mas não são contidos no vertex buffers, este é o caso ideal. Depois o vertex assembly, junta todos os dados no vertex data e os mesmos são extraídos usando o método DrawPrimitive. Nesse ponto, cada pixel é retirado e roteado pelo pixel shader, incluindo posição, cor, textura, e coordenada da textura. O pixel shader realiza operações usando instruções definidas pelo Direct3DX pixel shader assembler.

A saída (input) do pixel processado pelo pixel shader inclui superfície de textura e a coordenada de textura associada, controlando a retirada da superfície. A saída do pixel é roteada pelo buffer frame.



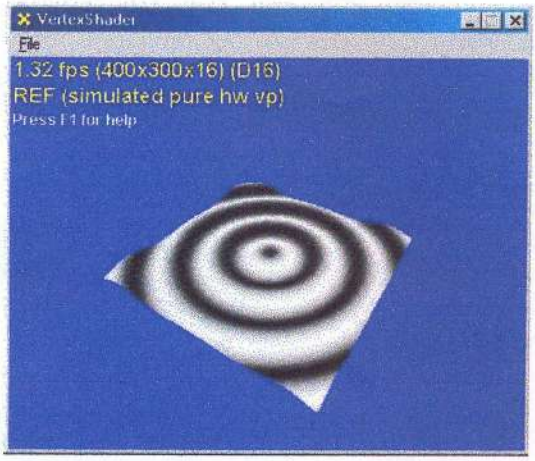


Figura 3.5,3.6: Exemplos de pixel shader e vertex shader respectivamente.

3.2.2.2. HAL Hardware Accelerator – Aceleração por Hardware

O HAL Hardware Accelerator é, basicamente, a capacidade de trabalhar com diversos tipos de hardware existentes em diferentes computadores. O Direct3D consiste num conjunto de interfaces e métodos que explicam como usar determinados tipos de gráficos. A capacidade de trabalhar com diferentes tipos de computadores implementados pelo HAL, pode chegar a ponto de trabalhar com códigos de 16 bits e 32 bits, sendo compatível com Windows NT e Windows 2000, o HAL sempre implementa por padrão em 32 bits.

O Direct3D HAL é implementado pelo fabricante do chip, e no DirectX 8.1, o HAL pode ser de três modos: software vertex processing, que utiliza aceleração por hardware, contudo é muito lento; hardware vertex processing que as placas mais recentes devem suportar, esta opção usa a aceleração por hardware tanto quanto possível e por último a mixed vertex processing que usa o hardware quando possível, mas se o hardware não suportar, o software faz o trabalho.

O diagrama abaixo mostra a relação entre o Direct3D, GDI e HAL:

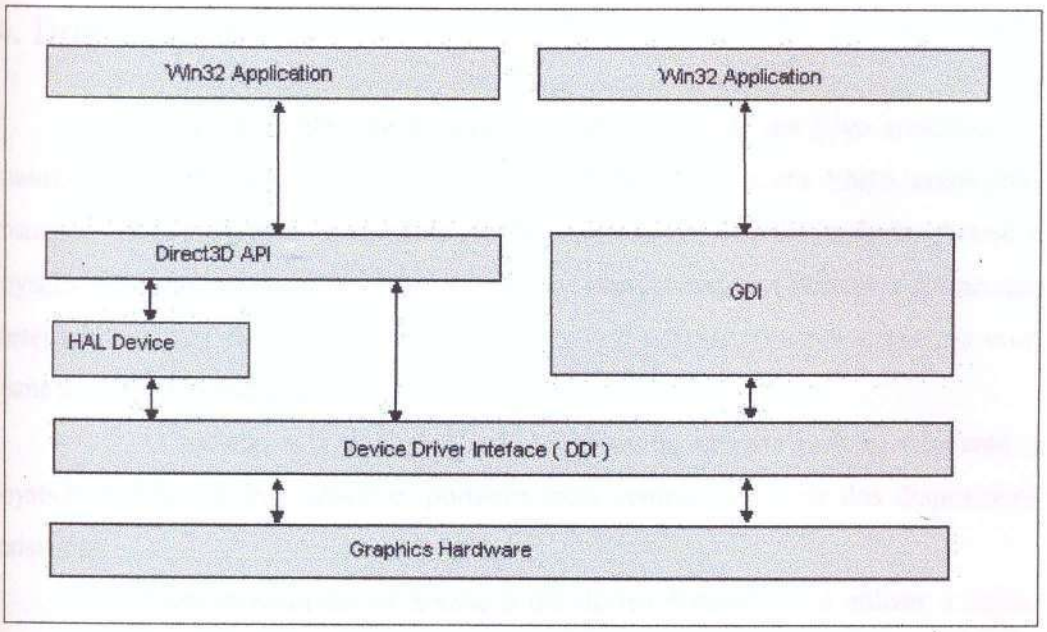


Figura 3.7: Relação entre Direct3D, GDI e HAL

Cumpre lembrar que a peça fundamental para trabalharmos com gráficos é a GDI (Graphical Device Interface), ou seja, o Dispositivo de Interface Gráfica. Tal dispositivo é responsável por toda e qualquer exibição de gráficos dentro do Windows. A GDI comunica-se com o hardware através de um grande número de funções internas. Tudo é desenhado pela GDI, desde botões até textos.

A principal característica desta interface é que ela processa independente do hardware acoplado. É claro que todas estas facilidades devem acarretar alguma desvantagem: ela tem uma performance lenta. Nas funções mais básicas seu funcionamento é perfeito, mas quando temos a necessidade de alta performance, como no caso de games, ela não é a melhor ferramenta. Então, o Direct3D seria uma saída.

4. DirectInput

O que normalmente uma pessoa faz ao comprar um novo aplicativo ou mesmo um game? Ela tenta trocar os controles da forma que gosta. Então temos duas maneiras de implementar o controle de dispositivos de entrada (teclado; mouse e joystick): a primeira seria fixar os controles e obrigar todos a utilizar a forma que desejamos; ou dar flexibilidade, dando opções que possam tornar o aplicativo ou o game mais fácil de ser utilizado.

O teclado e o mouse são dispositivos de entrada padrão; enquanto o joystick é direcionado a jogos e, portanto, nem sempre faz parte dos dispositivos existentes.

Uma maneira de ter acesso a um destes dispositivos é utilizar o objeto DirectInput.

4.1. Trabalhando com Dispositivos de Entrada (Teclado e Mouse)

Podemos acessar a fila do dispositivo de entrada que o usuário está utilizando, mas quase todas às vezes precisamos de informação de primeira mão para os dispositivos que usamos.

A subdivisão de DirectInput de DirectX pode adquirir controle exclusivo de dispositivos de entrada comuns como o teclado, mouse e joystick (se presente). O programador cria um objeto de DirectInput e adquire ponteiros às interfaces desejadas. Ultimamente estas interfaces são usadas para adquirir entrada exclusiva dos dispositivos e recobrar o estado do dispositivo (arraste do mouse, teclas pressionadas, etc).

DirectX está baseado em COM (Component Object Model). Quando criamos um objeto DirectX o instanciamos como objetos COM e obtemos interfaces com eles. Uma vez adquirida estas interfaces, podemos usá-las para chamar métodos que proporcionam acesso direto aos dispositivos de hardware que são bastante importantes para um jogo. Uma vez usadas, já não precisando mais destas interfaces, devemos liberá-las para prevenir acúmulos na memória e assegurar a própria execução, evitando as mensagens fatais de fechar ou fechar.

Podemos encapsular os objetos DirectInput com ponteiros para a classe disponível, e quando terminarmos automaticamente liberamos memória ou utilizamos

o método destructor. O constructor, porém é deixado vazio, só existem alguns ponteiros para NULO para manter os ponteiros limpos e garantir a ocorrência da limpeza. Isto tem que ser feito (constructor vazio) porque para construir a entrada do DirectInput e interfaces precisamos de um handles para a aplicação e uma janela principal (é possível que no momento da construção não tenha sido criada a janela principal da aplicação). Então somamos um adicional método Create () que tem dois parâmetros de handle e tomamos conta da criação dos objetos de entrada do DirectInput obtendo dispositivo conectado, mouse e teclado como é mostrado abaixo:

```
//Exemplo do artigo original:
//Flexible User Input Programming - An Idea
//por Yordan Gyurchev
//retirado do site GameDev.net

HRESULT wj_controls::Create(HINSTANCE hInst, HWND hWnd)
{
 hInstance=hInst;
 hWindow=hWnd;
 HRESULT hr;
 // Criando o objeto DirectInput
 hr = DirectInputCreate( hInst, DIRECTINPUT_VERSION, &m_pDI, NULL );
 if ( FAILED(hr) )
  return hr;

 hr=InitMouse();
 if ( FAILED(hr) )
  return hr;
 hr=InitKeyboard();
 if ( FAILED(hr) )
  return hr;

 AddSysKeys();
 return S_OK;
}
```

A função DirectInputCreate() é original da biblioteca do DirectX e cria um objeto DirectInput que suporta a interface IDirectInput COM. Claro que podemos fazer isto de modo mais fácil, chamando diretamente o CoCreateInstance obtendo, então, o ponteiro para o objeto. InitMouse () e InitKeyboard () são quase idênticos, só usam identificadores de interface diferentes e um conjunto de dados de diferentes formatos para o dispositivo obtido. Todas estas interfaces devem ser armazenadas na classe membro. Tendo sido feita a iniciação, é necessário fixar algumas propriedades. No caso, são quatro:

- Eixo (para o mouse, determina se informa sua posição absoluta ou relativa);
- Valores máximos e mínimos para o eixo;
- Menor valor para incremento do eixo;
- Tamanho do buffer usado para o buffer de entrada do teclado.

Se estivermos construindo um game 3D de tiro, para escolher o melhor método a ser utilizado – coordenadas do mouse relativas ou absolutas - é recomendável usarmos o relativo; se for um jogo de estratégia em tempo real o absoluto será perfeito. Fica claro que devemos ter um conjunto inicial, um setup inicial, porque podemos não saber qual dispositivo foi adquirido.

Devemos ficar atentos à fila de mensagem da aplicação e quando detectamos a desativação/ativação da aplicação, dependendo da situação devemos "não adquirir" / "adquirir" dispositivos de entrada - caso contrário podemos obter resultados indesejáveis. A recuperação também deve ser sincronizada com a flag "active" da classe, porque o estado de um dispositivo não é único e pode resultar em um estranho travamento. Assim, somamos no método MsgProg () para nossa classe e chamamos esta rotina de mensagem principal.

Pode se parecer:

```
switch (uMsg) {
 case WM_ACTIVATE:   //envia mensagem quando trocamos o estado da janela
  if ( WA_INACTIVE == wParam )
  {
   m_bactive = FALSE;
  }
```

```
  else
  {
   m_bactive = TRUE;
  }
  SetAcquire(); // Modo exclusivo de acesso de acordo com a classe membro
                // m_bactive
  break;
}
```

De acordo com o dispositivo do teclado do DirectInput, temos um buffer de teclado (uma array de teclas de 256 (bytes)), com coordenadas relativas ou absolutas para o mouse, e tem um buffer (4 bytes) para os botões. Precisamos declarar alguns dados na classe membro e armazenar esta informação no dispositivo. Claro que começamos com três long integer para a posição do mouse. Na realidade podemos implementar um pequeno truque, enquanto estamos lidando com a recuperação de dados de dispositivo entrada, isto é colocando o buffer dos botões do mouse no final da array do teclado de bytes, que fazem 260 bytes longos (eu uso um buffer default de 256 para o buffer do teclado – c_dfDIKeyboard - e 4 byte para o buffer dos botões do mouse - c_dfDIMouse). Assim produzimos um buffer comum para todas as teclas do jogo que o jogador pode usar. Não importa o que o jogador faça (pressionando o teclado ou clicando o botão do mouse), obterá as informações do mesmo lugar.

Logo, implementamos um array de strings para as teclas – caso o jogador queira ver como os controles são configurados, e é melhor visualizar "Space" que o código (57). E novamente teremos no código 256 um nome "Botão Esquerdo Mouse Pressionado".

4.2. Implementação de Controle – Sistema de Controle

Pode ser necessário controlar mais de uma tecla do teclado, ou seja, a combinação de teclas. Logo, uma forma simples, e montar um "Sistema de Controle". Segue uma possível estrutura:

```
typedef struct {
 WORD id;
```

```
 BYTE flags;
 BYTE key[4];
 BYTE state;
 BYTE oldstate;
 BYTE *external;
} control;
```

Precisamos de um ID para identificar o controle dentre outros controles. Então, as flags nomeiam algumas propriedades no sistema de controle: teclas exclusivas (alguns controles exigem teclas exclusivas e outros não) e operadores lógicos para as teclas (se temos teclas alternativas podemos usar o operador OU e se temos a combinação fundamental usamos o operador AND). O array de teclas representa as teclas para controle, naturalmente, limitamos o número delas a 4, mas podemos estender, porém, não é comum alguém jogar com seis teclas para a mesma ação.

O state e o oldstate contêm o estado do controle de acordo com as teclas especificadas. O seguinte código demonstra como o estado é calculado:

```
if (control.flags&WJ_CTRLS_OPAND2)
 control.state = keys[control.key[0]] &
         keys[control.key[1]];
else
 control.state = keys[control.key[0]] | keys[control.key[1]] |
         keys[control.key[2]] | keys[control.key[3]];
```

Temos um ponteiro externo para armazenar o estado do controle (embora não tenhamos utilizado isto na implementação, o que faz muito mais sentido, porque armazenando o estado diretamente em outro lugar, é melhor que repetir a cada implementação todos os ID de controle para recobrar o estado de controle querido).

Assim, agora que temos uma estrutura de controle, faremos alguns exemplos e os organizaremos.

Temos que escolher uma estrutura de dados para armazenar seus objetos de controle. Escolhemos o vetor STL porque é mais rápido para repetir o acesso e não precisamos de qualquer suplemento para operações de inserção ou deleção na coleção

(se não está familiarizado com STL, pense em um vetor com um array de tamanho dinâmico).

Neste momento, temos que pôr alguns controles reais nele. Por exemplo, para criar um "Primeiro Controle" precisamos de um método AddControl () implementado pela classe. Usamos um ID e uma flag, assim temos: AddControl(id,flags). Só então adicionamos um método AddKey(id,key), embora seja possível personalizar controle sem teclas associadas. O método de AddKey tem que ter um pouco de informação sobre as teclas que compartilham e corretamente tem que controlar a flag exclusiva. Isto significa que, quando uma tecla é acionada devemos conferir se há outro controle associado com esta tecla e se aquele controle tem a flag do jogo de tecla exclusivo.

Assim, terminamos esta seção com uma coleção de estruturas de controles e um par de métodos de controle. Para nossa conveniência, adicionamos outro método chamado FormatControlString(szstr,id) para recobrar o controle de informação da tecla na seqüência de string, e novamente mantemos os olhos nas flag para usar o operador Boolean. Quando conseguimos alternar a string de teclas olhamos para "Alt, Space", mas para o operador AND é necessário mudar para "Alt + Space".

4.3. Personalizando Controle

Como personalizar controles? Na realidade poderíamos fazer isto com o método AddKey, mas caso o usuário queira escolher uma tecla será necessário obtê-la através de códigos. Isto presume que temos um jogo alternativo de objetos de DirectInput ou algum outro modo de adquirir a escolha do usuário. Tudo que precisamos implementar é o método RecordControl(ID). Quando fizermos uma chamada, teremos a entrada do usuário, a primeira tecla pressionada será acrescentada ao controle especificado usando o método AddKey. Assim não aborreceremos o jogador durante o jogo com as teclas que ele queira usar, há pouco chamamos o método de RecordControl quando o usuário quis mudar um controle e continuamente atualizamos a informação sobre os controles que usam FormatStringControl.

O problema aqui é como entender o que foi a primeira tecla pressionada, depois que o método de registro foi chamado. Neste caso o buffer comum é bastante importante. Há poucas diferenças no decorrer do tempo, sendo atualizado e quando descobrimos uma mudança deixamos que seja registrada, enquanto passamos a tecla

detectada a outro método. (Se lembra que o mouse armazena suas teclas no buffer do teclado assim nenhuma checagem adicional é necessária).

4.4. Retorno ao Estado de Controle Corrente

A tarefa que queremos executar é obter informações de dispositivos de entrada, filtrar em nossa coleção de controles, e transferir a uma coleção de estados de controle que podemos usar. Selecionamos o método UpdateState () e será chamado pelo menos uma vez por repetição de jogo. Bem, o que faz isso?

Os objetos do DirectInput são responsáveis pelo resgate do estado do mouse e teclado no nosso buffer. Tendo aquela informação, iteramos nossa coleção de controles e calculamos o estado de controle.

```
vector<control>::iterator it=controls.begin();
for(; it != controls.end(); it++) {
 it->oldstate = it->state;
 if (it->flags&WJ_CTRLS_OPAND2)
  it->state = keys[it->key[0]] & keys[it->key[1]];
 else
  it->state = keys[it->key[0]] | keys[it->key[1]] |
        keys[it->key[2]] | keys[it->key[3]];
}
```

Ao término de UpdateState () temos nossa coleção atualizada com os estados de controle apropriados. Entretanto, precisamos destes estados fora da classe.

Há duas soluções possíveis para este problema: a primeira seria implementar um método GetControlStatus(ID), e a segunda, seria implementar um ponteiro externo de estado. A primeira é mais simples, mas às custas de algum esforço (para cada iteração o programa procura os controles na coleção de ID). A segunda é bastante rápida (somente uma tarefa), mas deixa uma abertura para erros lógicos como liberar memória determinada pelo ponteiro ou outros problemas de memória.

E finalmente o mouse: alguns métodos GetDeltaX () ,GetDeltaY () e GetDeltaZ () serão usados, limpando os dados das classes membros do mouse para guardar um novo valor.

4.5. Controles Persistentes

Persistência de controles é algo importante. Em algum lugar, temos que armazenar a configuração de controles até as próximas interações do jogo. Caso contrário o jogador ficará enfadado, pois a cada jogada precisará configurar as teclas.

Não discutiremos operações de arquivo com coleções, uma vez que há poucos assuntos interessantes envolvidos. Quando salvamos a coleção de arquivos não salvamos o sistema de controle, podemos recuperá-los pelo código. Isto economiza espaço e previne que controles de sistemas que sejam violados acidentalmente ou propositalmente pelo usuário. Quando abrimos os controles já carregamos a coleção, adicionando os controles de sistemas e então fazendo o arquivo de entrada.

Se pensarmos em termos de persistência, os usuários têm um talento incrível para confundir todas as teclas, assim é bom manter uma configuração "default" arquivada.

5. DirectPlay

DirectPlay é uma camada sobre os protocolos de rede normais (IPX, TCP/IP etc). Uma vez que a conexão é feita, podemos enviar mensagens sem precisar saber com qual usuário estamos conectando. Este é uma das melhores características (IMO) e, entretanto alguns podem propor protocolos de mensagem mais eficientes como WinSock

Como outros componentes do DirectX, DirectPlay podem ser usados com C, C++, Microsoft Basic Visual, Delphi e outras linguagens, é necessário que sejam apresentadas informações sobre Microsoft DirectPlay nas seções seguintes.

5.1. O Que Há Novo de DirectPlay

O componente de gerenciamento de redes do Microsoft DirectX sofreu uma revisão principal. Microsoft DirectPlay introduz novas interfaces que permitem aos jogos terem acesso mais direto ao hardware, obtendo um desempenho melhor.

A complexidade de criar uma aplicação transmitida por rede foi simplificada dramaticamente separando as interfaces para criar peer-to-peer e sessões de cliente/servidor.

Uma sessão de DirectPlay é um canal de comunicação entre vários computadores. Uma aplicação deve estar em uma sessão antes de comunicar-se com outras máquinas. Uma aplicação pode unir-se uma sessão existente ou pode criar uma sessão nova. Cada sessão tem somente um servidor que é a aplicação criada para isto. Só o servidor pode mudar as propriedades da sessão.

O modo default de uma sessão é o peer-to-peer, de maneira que o estado completo da sessão é replicado para todas as máquinas. Quando um computador muda algo, todos os outros computadores são notificados.

O outro modo, cliente/servidor significa que tudo é rotiado pelo servidor. Devemos administrar os dados neste modo que provavelmente é o melhor programa de chat. Porém, podemos usar um tipo híbrido.

As interfaces para criar sessões DirectPlay que eram definidas na biblioteca Dplay8.h agora são:

- IDirectPlay8Peer: Provê métodos para criar sessões de peer-to-peer;
- IDirectPlay8Client: Provê métodos para criar aplicação cliente de uma aplicação de cliente/servidor;

- IDirectPlay8Server: Provê métodos para criar a aplicação servidora de uma aplicação de cliente/servidor.

Entretanto, uma aplicação tem que criar um jogador (usuário) para enviar e receber mensagem. Sempre são dirigidas mensagens a um jogador e não ao computador. Neste momento, sua aplicação não precisa sequer saber o local do computador. Toda mensagem que for enviada será dirigida a um jogador específico e toda mensagem recebida será dirigida a um jogador local específico (com exceção de mensagens do sistema). Somente são identificados os jogadores como locais (existentes no mesmo computador) ou remoto (existentes em outro computador). Contudo, podemos criar mais de um jogador local, não é útil, mas existe a possibilidade.

DirectPlay oferece métodos adicionais para armazenar dados específicos da aplicação, assim não temos que implementar uma lista de jogadores.

Também podemos agrupar os jogadores na mesma sessão, assim qualquer mensagem enviada será dirigida ao grupo. Isto é ótimo para jogos de equipes onde podemos enviar mensagens somente para nossos aliados.

Cada mensagem enviada é marcada por um jogador local específico e pode ser enviada a qualquer jogador. Cada mensagem recebida é colocada em uma fila para aquele jogador local. Considerando que criamos só um jogador, todas as mensagens pertencem àquele jogador. Não precisamos nos preocupar com esta fila; tudo que necessitamos é extrair as mensagens e agir. A aplicação pode receber a fila de mensagens ou usar separadamente thread e eventos. Com o método thread e o trabalho é exaustivo, contudo quando se usamos MessageBox para notificar o usuário, é mais simples. Se quisermos usar threads, precisamos interromper a thread quando a aplicação exibir uma MessageBox e de alguma maneira sincronizar. Assim, é recomendável optarmos pelo outro método.

A biblioteca Dplobby8.h define:

- IDirectPlay8LobbyClient: Esta interface é usada para administrar um cliente (jogador), para enumerar e lançar requisições à aplicação;
- IDirectPlay8LobbiedApplication: Esta interface é usada para registrar um lançamento entre a aplicação e o sistema. Também é usada para obter as informações de conexão e habilitar o jogo sem examinar o usuário.

5.2. Transmissão de Voz foi Somada

DirectPlay Voz provê interfaces para acrescentar comunicação de voz em tempo real a uma aplicação. As interfaces seguintes estão definidas na biblioteca Dvoice.h:

- IdirectPlayVoiceClient: Provê métodos para criar e administrar os clientes em uma sessão DirectPlay Voice;
- IdirectPlayVoiceServer: Provê métodos para ser o servidor e administrar uma sessão DirectPlay Voice;
- IdirectPlayVoiceTest: Usando teste de configurações de áudio no DirectPlay Voice.Usa dados baseados em GUID e formato em URL.

Versões prévias do DirectPlay usavam blocos binários de dados com endereços de GUID que eram difíceis de implementar e que não eram possíveis de serem lidas. Em DirectX 8.1, DirectPlay introduz a representação de endereços em formato de URL. Um jogo de interfaces, definido em Dpaddr.h, é usado para criar e manipular o formato novo:

- IDirectPlay8Address: Provê métodos genéricos direcionados a criar e manipular endereços do DirectPlay;
- IDirectPlay8AddressIP: Provê serviços de IP direcionados a específicos provedores.

Foram somadas melhores escalas de gerenciamento de memórias mais eficientes e aumentos no tamanho da banda afetando o desempenho do jogo em rede. DirectPlay gerencia thread melhorando e facilitando projetar escalas, aplicações mais robustas que podem apoiar multiplayer e volumosas aplicações on-lines. Melhorou o suporte a Firewalls e adicionou o Network Address Translators (Tradução de Endereço de Rede).

Porém, jogos de rede escritos (NATs), Firewalls, e métodos de Conexão Compartilhada de Internet (ICS) podem ser difíceis, particularmente não garantem transmissão (UDP). Porque DirectPlay 8.1 foi desenvolvido com este intuito, apoiando soluções NAT onde possível; e o servidor de serviços DirectPlay 8.1 TCP/IP é simples, seleciona porta UDP para dados do jogo, fazendo possíveis configurações de Firewalls e NATs adequadamente. Adicionalmente, DirectPlay faz uso de UDP de forma que, para jogos cliente/servidor, os clientes que buscam algumas NATs poderão conectar-se aos jogos sem configuração adicional.

5.3. Aplicação

A primeira coisa a fazer é incluir as bibliotecas do DirectPlay:

```
//Exemplo do artigo original:
//A DirectPlay Tutorial
//por Sobeit Void
//retirado do site: GameDev.net

#include <dplay.h>
#include <dplobby.h>
```

Também acrescentar DPLAYX.LIB ao projeto e definir INITGUID ou adicionar o dxguid.lib. Definimos isto no começo do projeto.

```
#define INITGUID // usar  os predefinidos IDs
```

Precisamos ter um GUID (Unico Global ID) em nossa aplicação. Este ID distingue a aplicação no computador. Caso contrario somente nossa aplicação envia mensagens pelo browser. Podemos usar guidgen.exe para criar um ID para nossa aplicação. A Microsoft garante que nunca criará matematicamente duas vezes o mesmo GUID. Logo temos algo como:

```
DEFINE_GUID(our_program_id,
0x5bfdb060, 0x6a4, 0x11d0, 0x9c, 0x4f, 0x0, 0xa0, 0xc9, 0x5, 0x42, 0x5e);
```

Agora definimos nossas variáveis globais

```
LPDP DE LPDIRECTPLAY3A = NULO;      // ponteiro de interface para directplay
LPDPLOBBY DE LPDIRECTPLAYLOBBY2A = NULO;  // ponteiro de interface
                                          // lobby
```

A próxima coisa é o loop do programa principal.

```
// Obtenha informações de jogador local
```

```
// Inicialize conexão de Directplay com as informação acima

while(1)
{
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)
  {
    // Tradução normal da mensagem

  } // fim do if PeekMessage (..)
  else
  {
    // Loop do jogo

    // Recebe mensagens
  } // /fim else
} // fim do while(1)

// Fechamento da conexão do Directplay
```

Para obtermos informações do usuário podemos usar uma dialog Box ou outro meio que julgarmos melhor. O principal é o nome e a definição de servidor ou cliente. Se for o servidor, adquire o nome de sessão, se for o cliente, adquire um endereço de TCP/IP para conectar. De qualquer maneira, armazenamos esses dados dentro de uma variável global, como, por exemplo:

```
GAMESERVER DE BOOL;        // flag para cliente/servidor
chamusque player_name[10]; // nome de jogador local, limite de 10 caracteres
chamusque session_name[10]; // nome de sessão, também limite de10 caracteres
chamusque tcp_address[15]; // endereço de TCP/ IP para conectar
```

Podemos encapsular para evitar o uso de variáveis globais, mas variáveis globais simplificam o processo de aprendizagem. Também não faremos muitas verificações de erros; teoricamente deveríamos testar o resultado de toda chamada de função.

Antes de mudarmos, deveríamos implementar uma lista de jogadores na sessão atual. Embora não seja necessário aqui, precisaremos em aplicações maiores.

Criar uma lista com os seguintes elementos:

```
Class DP_PLAYER_LIST_ELEM {

DPID DE DPID;      // o id do jogador DirectPlay
chamusque name[10]; // nome de jogador
DWORD sinaliza;    // flags do jogador do DirectPlay

// qualquer outra informação que poderíamos precisar
};
```

Precisaremos implementar uma classe que adicione um jogador e apague um outro com um dpid específico. Não devemos nos referir a jogadores pelos nomes porque os jogadores podem ter o mesmo nome. Usaremos o ID para diferenciar os jogadores.

Então, definimos um ponteiro global para a classe:

```
DP_PLAYER_LIST *dp_player_list; // lista de jogadores na sessão atual
```

Isso serão as informações globais das quais necessitamos. Também devemos criar uma struct de jogadores para adicionarmos somente informações para os jogadores locais.

5.3.1. Montando a Conexão

Para montar a conexão, escreveremos uma função que pega uma string TCP/IP e cria uma conexão de TCP/IP. Também ocorrendo quando um lobby começa.

Embora, o DirectPlay tenha um método para enumerar as conexões disponíveis, o método enumerará todas as conexões mesmo que elas não estejam disponíveis. Somente se não tivermos uma conexão IPX, o enumerador devolverá um opcional IPX para conectar e a conexão somente falhará quando o usuário tentar faze-

la. Isto parece redundante, assim é recomendáve que seja omitida esta parte e dê uma opção certa ao usuário.

```
int Create_TCP_Connection(char *IP_address)
{
  OLD_LPDPLOBBYA DE LPDIRECTPLAYLOBBYA = NULL;   // ponteiro
                                                 // antigo de
                                                 // lobby
  DPCOMPOUNDADDRESSELEMENT Address[2]; // criar componente addr
  DWORD ADDRESSSIZE = 0;      // tamanho do componente de endereço
  LPCONNECTION DE LPVOID = NULO;   // ponteiro para fazer conexão

  CoInitialize(NULL); // registrando COM

  // criando objeto de DirectPlay
  if (CoCreateInstance(CLSID_DirectPlay, NULO, CLSCTX_INPROC_SERVER,
     IID_IDIRECTPLAY3A,(LPVOID *) &LPDP) != S_OK)
  {
    // retorno um erro no MessageBox
    CoUninitialize ();
    return(0);
  }

  // criando objeto lobby
  DIRECTPLAYLOBBYCREATE(NULL, &OLD_LPDPLOBBYA, NULL, NULL, 0);

  // iniciando nova interface lobby
  OLD_LPDPLOBBYA->QUERYINTERFACE(IID_IDIRECTPLAYLOBBY2A,
(LPVOID *) &LPDPLOBBY));

  old_lpdplobbyA->Release (); // liberando interface velha desde que nós temos
uma
                              // nova

  // preencha dados de endereço
```

```
    Address[0] .guidDataType = DPAID_ServiceProvider;
    Address[0] .dwDataSize = sizeof(GUID);
    ADDRESS[0] .LPDATA = (LPVOID)&DPSPGUID_TCPIP; // TCP ID

    Address[1] .guidDataType = DPAID_INet;
    Address[1] .dwDataSize = lstrlen(IP_address)+1;
    Address[1] .lpData = IP_address;

    // iniciando tamanho para criar endereço
    // este método retornará DPERR_BUFFERTOOSMALL-não havendo erro
    LPDPLOBBY->CREATECOMPOUNDADDRESS(ADDRESS, 2, NULL,
&ADDRESS_SIZE);

    lpConnection = GlobalAllocPtr(GHND, AddressSize); // alocando memoria

    // criando o endereço
    lpdplobby->CreateCompoundAddress(Address, 2, lpConnection, &Address_Size);

    // inicializando a conexão TCP
    lpdp->InitializeConnection(lpConnection, 0);

    GlobalFreePtr(lpConnection); // memória alocada liberada

    return(1); // sucesso

} // fim da int Create_TCP_Connection (..)
```

Primeiramente iniciamos o COM, incrementando em 1, e criamos o objeto DirectPlay usando CoCreateInstance. Isto é outro método de criação de objetos DirectX, o qual é atualmente o método que de fato envolve o uso da função. Temos que passar o identificador de classe e o principal para notarmos é o parâmetro IID_DirectPlay3A. Este é o identificador da interface de IDirectPlay3A, assim se quisermos adquirir uma interface de IDirectPlay2A, fixamos o parâmetro a IID_DirectPlay2A. Faremos da mesma forma se quisermos uma interface de IDirectPlay4A

Então criamos o DirectPlay lobby e examinamos versão 2A. Desde que exista um macro DirectPlayLobbyCreate, não precisaremos iniciar o COM como acima. Esta função faz o mesmo (COM e CoCreateInstance) exceto o fato de que adquire a mais baixa interface, isto é, a IDirectPlayLobbyA. Assim, precisamos iniciar a versão 2A, a qual é feita para identificarmos a interface (notamos que todos os objetos de DirectX são iniciados da mesma maneira). Então, fechamos o lobby anterior desde que não temos um novo.

Logo, criamos um endereço que contém a informação sobre a conexão de TCP. Desde que não usamos EnumConnections, precisamos construir as informações. Podemos usar a informação diretamente da enumeração e saltar para iniciação, mas preferimos reduzir as funções de callback ao mínimo. Fixamos os campos da estrutura de endereço e fixamos o tipo TCP/IP que provê o serviço de ID. Isto é definido em dxguid.lib ou incluindo o INITGUID acima. Fixamos o segundo elemento da string de endereço. Podemos obter todas estas informações do MSDN, e quais parâmetros são necessários passar para criar outros tipos de conexão.

Então, adquirimos o tamanho do buffer requerido para a conexão passando um parâmetro NULL. O tamanho requerido será armazenado na variável Address_Size. Note que este método devolverá DPERR_BUFFERTOOSMALL desde que seja iniciado com sucesso. Não devemos interpretar isto como uma condição de erro. Então alocamos memória do sistema usando GlobalAllocPtr, em lugar de usar malloc. Criamos o endereço de informação para passarmos o alocado buffer para a função. Usamos a chamada Initialize do DirectPlay e teremos criado uma conexão de TCP/IP. Se InitalizeConnection retornar DPERR_UNAVAILABLE, significa que o computador não pode fazer tal conexão e está na hora de informar ao usuário que não há tal protocolo no computador.

Utilizamos o método acima para evitar as caixas de dialogo default até não termos alternativa e precisarmos de informações do usuário. Um bom exemplo de um método contrario a esse é o jogo StarCraft.

Note que deveríamos testar toda chamada de função e devolver o erro apropriado. Não fazemos isso aqui para simplificarmos. Também como essa função esta trabalhando, passamos " " como string se for um hosting da sessão, se não passamos o IP da máquina que está conectada. Assim, no comando "iniciando as informações do usuário", teremos várias informações para chamarmos esta função:

if (gameServer) // se for o host da máquina

```
   Create_TCP_Connection (""); // uma string vazia é suficiente
else
   Create_TCP_Connection(tcp_address); // passou o ip global de usuário
```

É imprescindível lembrarmos que precisamos ter uma sessão e um jogador antes de enviarmos mensagens. Mas antes disso, é necessário fechar a conexão.

```
int DirectPlay_Shutdown ()
{
   if (lpdp) // se conexão já existe, assim não será nulo
   {
      if (lpdplobby)
         lpdplobby->Release ();

      lpdp->Release ();

      CoUninitialize ();
   }

   LPDP = NULL;
   LPDPLOBBY = NULL;

   return(1); // sempre sucesso
} // fim da int DirectPlay_Shutdown ();
```

Esta função fecha à seção de conexão.

5.3.2. Sessões

A função main precisará de uma sessão de gerenciamento como:

```
EnumSessions   - enumera todas as sessões disponíveis
Open           - une ou hosts a uma nova sessão
Close          - fecha a sessão
```

GetSessionDesc - adquira as propriedades da sessão
SetSessionDesc - fixa as propriedades da sessão

lpdp->Close ();

Esta função fecha a sessão antes de chamar DirectPlay_Shutdown (): Todos os jogadores locais criados serão destruídos e o DPMSG_DESTROYPLAYERORGROUP será enviado a outros jogadores na sessão.

lpdp->EnumSessions (..); / / enumera todas as sessões

Esta função só será chamada pelo cliente, assim se tiver o host da sessão chamará Open. O problema é que o cliente precisa procurar uma sessão para unir quando iniciou uma conexão, e desde então pode haver mais de uma sessão host, precisamos saber de todas as sessões disponíveis e apresentar ao usuário.

Isto precisará usar de uma função de callback.

```
BOOL FAR PASCAL EnumSessionsCallback2(LPCDPSESSIONDESC2 lpThisCD,
                    LPDWORD  lpdwTimeOut,
                    DWORD    dwFlags,
                    LPVOID   lpContext);
```

Esta função callback que implementamos será chamada uma vez para cada sessão que for encontrada usando EnumSessions. Uma vez que todas as sessões são enumeradas, a função será chamada uma vez mais com a flag DPESC_TIMEOUT.

Qualquer ponteiro devolvido em uma função de callback só é temporário e só é válido na função de callback. Temos que salvar qualquer informação que queiramos enumerar. Isto é aplicado depois à enumeração do jogador. O protótipo de EnumSessions:

```
EnumSessions(LPDPSESSIONDESC2 lpsd, DWORD dwTimeOut,
        LPDPENUMSESSIONSCALLBACK2 lpEnumSessionCallback2,
        LPVOID context, DWORD dwFlags);
```

O primeiro parâmetro é um descriptor de sessão assim precisamos inicializa-lo.

```
DPSESSIONDESC2  session_desc;   //descrição da sessão

ZeroMemory(&session_desc, sizeof(DPSESSIONDESC2)); // limpando descrição
session_desc.dwSize = sizeof(DPSESSIONDESC2);
session_desc.guidApplication = our_program_id;
```

Isto só assegurará que nosso programa devolverá sessões pelo programa. ZeroMemory é semelhante a memset para 0, que depois de muitas chamadas de DirectX exige a determinação do tamanho na estrutura passada.

O segundo parâmetro deveria ser fixado em 0 (recomendado) para um valor default.

O terceiro parâmetro é a função callback, assim, passamos a função.

O quarto parâmetro é um contexto definido pelo usuário que é passado ao callback para enumeração.

O quinto é o tipo de sessão a enumerar. Existem poucos valores default 0, somente enumeraremos sessões disponíveis, conforme o MSDN pede.

```
// nossa função de callback
BOOL FAR PASCAL EnumSessionsCallback(LPCDPSESSIONDESC2 lpThisSD,
                    LPDWORD lpdwTimeOut,
                    DWORD dwFlags, LPVOID lpContext)
{
  HWND  hwnd;     // handle. Eu sugiro uma listbox de handles

  if (dwFlags & DPESC_TIMEOUT)  // se terminou a enumeração parar
    return(FALSE);

  hwnd = (HWND) lpContext;   // adquire window handle

 // lpThisSd-> lpszSessionNameA  // armazena este valor, nome da sessão,
  // lpThis->guidInstance // armazena isto, uma instancia do host
```

```
    return(TRUE); // continue enumerando
} // fim do callback

// nossa função de enumeração
int EnumSessions(HWND hwnd, GUID app_guid, DWORD dwFlags)
{
    DPSESSIONDESC2   session_desc;    // descrição da sessão

    ZeroMemory (..); // como abaixo
    // fixar tamanho da descrição
    // fixar guid para o guid passado

    // enumere a sessão. Cheque erro aqui. Vital
    lpdp->EnumSessions(&session_desc, 0, EnumSessionsCallback, hwnd, dwFlags);

    return(1); // sucesso
} // fim do int EnumSessions
```

Sugerimos o envio do listbox de handle com o contexto, assim podemos reduzira a quantidade de informação e exibir ao usuário. Na função de callback, temos que alocar espaço para segurar o guidInstance. Esta é a instância do programa que está hospedando a sessão. Precisamos passar esta informação ao usuário para que ele selecione qual sessão deseja. Sugerimos um listbox, assim podemos enviar mensagens pelo parâmetro de hwnd. Lembrando, a instância e o nome somente são válidos dentro da função de callback, assim temos que salvar para poder apresenta-los depois.

Se for devolvido falso no callback, a enumeração parará e retornará o controle a EnumSessions. Se isto não for feito, entraremos num loop eterno. Também será retornado falso se encontrar um erro. É imperativo conferir o valor de EnumSessions especialmente se não estamos enumerando somente sessões disponíveis. Também, o programa inteiro ficará bloqueado enquanto estivermos enumerando, porque há necessidade de busca em todas as sessões. Podemos fazer uma enumeração assíncrona também.

```
lpdp->Open(LPDPSESSIONDESC2 lpsd, DWORD dwFlags)
```

Isto funciona em host ou une uma sessão que usa o dwFlags. Se estivermos unindo uma sessão, só precisaremos preencher o dwSize e guidInstance (salvando em algum lugar) do descriptor.

Assim, definimos um descriptor como:

```
DPSESSIONDESC2  session_desc;

ZeroMemory(&session_desc, sizeof(DPSESSIONDESC2));
session_desc.dwSize = sizeof(DPSESSIONDESC2);
session_desc.guidInstance = // a instancia deve ser salva em algum lufgar;

// e une a sessão
lpdp->Open(&session_desc, DPOPEN_JOIN);
```

Se estivermos hospedando uma sessão, precisaremos preencher, além do anterior, o nome, o número de máximo de jogadores permitidos e as flags. Fixamos a flag para Open como DPOPEN_CREATE | DPOPEN_RETURNSTATUS.

Os estados das flags podem esconder uma caixa de diálogo Box, mostrando os estados de progresso e retornando imediatamente. Embora a documentação diga que deveria manter o chamado Open com a flag de estado de retorno, não é necessário fazer assim.

As flags de descrição da sessão deveriam começar como:

DPSESSION_KEEPALIVE – mantém a sessão aberta quando são derrubados os jogadores anormalmente.
DPSESSION_MIGRATEHOST–se a hospedagem não existe, outro computador se tornará o hospedeiro.

Podemos usar OR nas flags, assim: FLAG_1 | FLAG_2 | FLAG_3.

5.3.3. Criação do Jogador

Agora criaremos um jogador local que usa CreatePlayer. Em primeiro lugar temos que definir a struct de nome, assim:

```
DPNAME    name;     // tipo de nome
DPID      dpid;     // o dpid do jogador criado dado pelo DirectPlay

ZeroMemory(&name,sizeof(DPNAME)); // limpa a  struct
name.size = sizeof(DPNAME);
name.lpszShortNameA = player_name;  // o nome do usuário
name.lpszLongNameA = NULL;

lpdp->CreatePlayer(&dpid, &name, NULL, NULL, 0, player_flags);
```

Esta função devolverá um ID único para o jogador local dentro da sessão. Usamos está identificação para diferenciar jogadores com o mesmo nome, salvando isto no struct de jogador local. O player_name passado é obtido primeiro nas informações exigidas do usuário. Os parâmetros intermediários são usados se não quisermos apurar a recebida fila. As flags de jogador são DPPLAYER_SERVERPLAYER ou 0, o qual o jogador não é um servidor de jogadores. Pode haver só um servidor de jogadores na sessão. Também existe um jogador espectador, mas seu significado está definido pela aplicação, assim não usamos isto aqui.

A outra função necessária é EnumPlayers. Lembra-se da lista global de jogadores que foi definida anteriormente? Há pouco jogadores dentro do callback. Trabalhamos da mesma maneira com a enumeração acima. Não temos que enumerar os jogadores neste chat, mas estamos livres para podermos ver quem está ao mesmo tempo conectado.

Não precisamos destruir o jogador porque fechando a sessão faremos, automaticamente, e não há necessidade de destruirmos e criarmos outro jogador enquanto ainda estivermos conectados. Contudo, podemos fazê-lo.

5.3.4. Gerenciando Mensagens

Agora que temos um jogador, precisamos saber como enviar e receber mensagens. Falaremos sobre enviar mensagens primeiro.

5.3.4.1. Enviando Mensagens

Existe somente uma maneira para enviar uma mensagem e isso está definido na função Send. Enquanto enviamos uma mensagem, requeremos um ID do remetente e do receptor. Uma bom procedimento será salvar o ID do jogador local em um struct e os IDs de todos os jogadores em uma lista de jogadores globais. Assim, chamamos a função como segue:

lpdp->Send(idFrom, idTo, dwFlags, lpData, dwDataSize);

O idFrom é o ID do jogador local. Isto só deve ser fixado a um jogador localmente criado (não queremos personalizar outro jogador). O idTo é o ID do receptor. Usamos somente DPID_SERVERPLAYER para enviar ao servidor e DPID_ALLPLAYERS para enviar a todos (exceto nós mesmos). Se quisermos dirigir uma mensagem a um jogador específico, usamos a lista de jogador. Não temos que usar uma lista de jogador em uma conversa; ao invés podemos acrescentar tudo a um listbox. Mas depois no jogo, uma lista de jogadores estará disponível.

O flag de parâmetro mostra como deve ser enviada a mensagem. O default 0, significa que não é garantido. As outras opções são DPSEND_GUARANTTED, DPSEND_ENCRYPTED e DPSEND_SIGNED. A cada três mensagens enviadas, uma será garantida, só usamos uma mensagem garantida para coisas importantes (como texto). As mensagens codificadas requerem um servidor seguro. Também qualquer mensagem recebida será garantida se estiver livre de corrupção (DirectPlay executa cheques de integridade nelas).

Se criarmos uma sessão que não especifique nenhum ID de mensagem, o idFrom de mensagem não fará sentido nenhum e o receptor receberá uma mensagem de DPID_UNKNOWN.

Os últimos dois parâmetros são um ponteiro para os dados a enviar e o tamanho daquele bloco. Notamos que DirectPlay não tem nenhum limite de tamanho que podemos enviar. O DirectPlay quebrará mensagens grandes em pacotes menores e os juntará no final.

Considerando que estamos fazendo um programa de bate papo, mostrarei um exemplo de enviar uma mensagem ao chat:

```
// tipos de mensagens que a aplicação receberá
const   DWORD       DP_MSG_CHATSTRING = 0;  // mensagem de chat

// a estrutura de uma string de uma mensagem enviada
typedef struct DP_STRING_MSG_TYP   // para variável string
{
   DWORD  dwType;     // tipo de mensagem
   char   szMsg[1];   // variável de tamanho da mensagem

} DP_STRING_MSG. *DP_STRING_MSG_PTR;

// função para enviar string de mensagem para o jogador local
int DP_Send_String_Mesg(DWORD type, DPID idTo, LPSTR lpstr)
{
   DP_STRING_MSG_PTR  lpStringMsg;    // ponteiro de mensagem
   DWORD              dwMessageSize;  // tamanho de mensagem

   // se string vazia, retorna,

   dwMessageSize = sizeof(DP_STRING_MSG)+lstrlen(lpstr); // fixa tamanho

   // aloca espaço
   lpStringMsg          =          (DP_STRING_MSG_PTR)GlobalAllocPtr(GHND,
dwMessageSize);

   lpStringMsg->dwType = tipo;         // tipo
   lstrcpy(lpStringMsg->szMsg, lpstr); // cópia da string

   // envie a string
     lpdp->Send(local_player_id,idTo, DP_SEND_GUARANTEED,
       lpStringMsg, dwMessageSize);

   GlobalFreePtr(lpStringMsg); // liberá memória

   return(1); // sucesso
```

} //fim do int DP_Send_String_Mesg (..)

Definimos os tipos de mensagens que podemos ter. Considerando que este é um programa de bate papo, podemos somente ter um tipo, que fixamos em DP_MSG_CHATSTRING. Podemos somar outros e fixando o tipo, assim podemos usar uma string para enviar o conteúdo. Isso porque a struct da mensagem string tem um tipo diferente de conteúdo. Ao enviar a função basicamente alocamos espaço e enviamos a mensagem ao jogador desejado. Notamos que o local_player_id é armazenado em qualquer lugar, ou podemos fixar isto, para passar outra variável local_player_flag.

5.3.4.2. Recebimento de Mensagens

O recebimento de mensagens requer mais trabalho que enviar. Há dois tipos de mensagens que podemos receber – uma mensagem de jogador e uma mensagem de sistema. Uma mensagem de sistema é enviada quando ocorrer uma mudança no estado de sessão. As mensagens de sistema que recebemos são:

DPSYS_SESSIONLOST - a sessão foi perdida
DPSYS_HOST - o hospedeiro atual partiu e você é o anfitrião novo
DPSYS_CREATEPLAYERORGROUP–um jogador novo foi inserido
DPSYS_DESTROYPLAYERORGROUP–um jogador saiu

Algumas dessas mensagens só são enviadas se são especificadas certas flags quando o jogador criar a sessão.

A função Receive tem sintaxe semelhante a função Send. A única coisa diferente que devemos mencionar é o terceiro parâmetro. Em vez do parâmetro enviando, existe um parâmetro receptor. Fixamos 0 para o valor default, de maneira a extrair a primeira mensagem da fila e deletar a última da fila.

O que não devemos nos esquecer é que precisamos cortar a mensagem DPMSG_GENERIC para evitar sujeira. Assim,

```
void Receive_Mesg()
{
  DPID   idFrom, idTo;              // id de jogador  para
```

```
LPVOID lpvMsgBuffer = NULL;  // o ponteiro buffer receptor
DWORD  dwMsgBufferSize;      // sizeof sobre buffer
HRESULT hr;                  // resultado do temp

DWORD  count = 0;            // temp contendo a mensagem

// Fixa o número da mensagem na fila
lpdp->GetMessageCount(local_player_id, &count);

if (count == 0)    // se nenhuma mensagem
   return;         // não faça nada

do  // leia todas as mensagens em fila
{
   do // loop até que uma única mensagem seja lida com sucesso
   {
      idFrom = 0;   // iniciar variável
      idTo = 0;

      // fixa tamanho do buffer requerido
      hr = lpdp->Receive(&idFrom, &idTo, 0, lpvMsgBuffer, &dwMsgBufferSize);
      if (hr == DPERR_BUFFERTOOSMALL)
      {
         if (lpvMsgBuffer)   // memória antiga liberada
            GlobalFreePtr(lpvMsgBuffer);

         // aloca nova memória
         lpvMsgBuffer = GlobalAllocPtr(GHND, dwMsgBufferSize);

      } // fim do if (hr ==DPERR_BUFFERTOOSMALL)
   } while(hr == DPERR_BUFFERTOOSMALL);

   // mensagem é recebida no buffer
   if (SUCCEEDED(hr) && (dwMsgBufferSize >= sizeof(DPMSG_GENERIC)
   {
```

```
        if (idFrom == DPID_SYSMSG)    // se mensagem de sistema
          Handle_System_Message((LPDPMSG_GENERIC)lpvMsgBuffer,
              dwMsgBuffersize, idFrom, idTo);
        else  // else deve ser aplicado à mensagem
          Handle_Appl_Message((LPDPMSG_GENERIC)lpvMsgBuffer,
              dwMsgBufferSize,idFrom,idTo);
      }
    } while (SUCCEEDED(hr));

    if (lpvMsgBuffer)  // memória liberada
      GlobalFreePtr(lpvMsgBuffer);

} // fim de void Receive_Mesg()
```

Primeiramente, conferimos se existe alguma mensagem no loop. Se não houver, saímos desta função. Então continuamos tentando receber a mensagem no buffer alocando um novo buffer. Quando o valor de retorno não for DPERR_BUFFERTOOSMALL, significa que não recebemos a mensagem ou outro erro sério aconteceu. Assim conferimos se o hresult retornou sucesso antes de determinar se é uma mensagem de sistema ou de aplicação, pelas quais chamamos as funções apropriadas. Mensagens de sistema vêm de DPID_SYSMSG, que é um valor reservado no DirectPlay.

As 2 funções deveriam ser implementadas como segue:

```
int Handle_System_Message(LPDPMSG_GENERIC lpMsg, DWORD dwMsgSize,
            DPID idFrom, DPID idTo)
{
  switch(lpMsg->dwType)
  {
  case DPSYS_SESSIONLOST:
    {
      // informe o usuário
      // PostQuitMessage(0)

    } break;
```

```
  case DPSYS_HOST:
    {
      // informe o usuário

    } break;
  case DPSYS_DESTROYPLAYERORGROUP:  // jogador novo
    {
      // construir mensagem
      LPDPMSG_CREATEPLAYERORGROUP lp =
(LPDPMSG_CREATEPLAYERORGROUP)lpMsg;

      // informe usuário que um novo jogador entrou na sessão
      // nome deste jogador novo é lp->dpnName.lpszShortNameA

    } break;
  case DPSYS_DESTROYPLAYERORGROUP:  // jogador saiu
    {
      // construir mensagem
      LPDPMSG_DESTROYPLAYERORGROUP lp =
(LPDPMSG_DESTROYPLAYERORGROUP)lpMsg;

      // informe usuário que um jogador saiu
      // nome deste jogador novo é lp->dpnName.lpszShortNameA

    } break;
  default:
    // uma mensagem não capturada. Erro
  } // fim do switch


return(1); // sucesso
} // fim da int Handle_System_Message (..)

int Handle_Appl_Message(LPDPMSG_GENERIC lpMsg, DWORD dwMsgSize,
            DPID idFrom, DPID idTo)
```

```
{
  switch(lpMsg->dwType)
  {
  case DP_MSG_CHATSTRING:
    {
      // todo as receberam a mensagem que nós definimos
      DP_STRING_MSG_PTR lp = (DP_STRING_MGS_PTR)lpMsg;


      if (gameServer)      // se servidor, retransmita a mensagem a todos os
jogadores
            {
        lpdp->Send(local_player_id,DPID_ALLPLAYERS,
          DPSEND_GUARANTEED, lp, dwMsgSize);
      }

      // update de nossa window chat


    } break;
  default:
    // mensagem de aplicação desconhecida, ruim,

  } // fim switch

  return(1); // sucesso
} // fim da int Handle_Appl_Message (..)
```

As duas funções são bem parecidas. Para fixar a mensagem atual, precisamos receber a mensagem do tipo apropriado antes de extrairmos os componentes individuais. Até mesmo se a mensagem de aplicação passar, precisamos receber a mensagem de dados. O parâmetro de dwType que definimos no struct de string de chat permite diferenciar as mensagens definidas na aplicação. Embora o chat requeira somente uma mensagem, que é a mensagem que está no chat de mensagens, existe definitivamente uma necessidade por mais tipos de mensagens que a aplicação

deveria controlar. Todo tipo de mensagem nova que definimos deverá incluir um parâmetro de dwType, assim a aplicação poderá diferenciar as mensagens.

5.3.5. Resumindo

Agora sabemos como deveria ser implementado, vamos juntar os vários pedaços (em WinMain).

```
// Obtenha informações de jogador local
// Criação de conexão
if (gameServer)
   Create_TCP_Connection("");
else
   Create_TCP_Connection(tcp_address);

// Parte da sessão
if (gameServer) // se sou hospedeiro
   // conexão aberta
else  // se o cliente
{
   // EnumSessions
   // Conexão aberta
}

// Criação de jogador
if (gameServer)  // flags de jogo para criar serverplayer
   // flags de jogo para DPID_SERVERPLAYER
// criar o jogador local
while(1)
{
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)
   {
      if (msg.message == WM_QUIT)
         break;
```

```
        // traduz e despacha a mensagem

    } // fim do if PeekMessage (..)
    else
    {
        // game loop

        Receive_Mesg ();
    } // /fim do else

} // fim do while(1)

// sessão fechada
DirectPlay_Shutdown ();
```

6. DirectShow

O DirectShow tem muito potencial, contudo muitas das características que poderiam ser usadas com game e outras aplicações, são tão obscuras ou há tão pouca informação que torna inútil e muito complexo de aprender, realmente não os mencionarei. Ao invés disto, focalizarei principalmente na montagem do DirectShow e alcançar um uso básico que poderia produzir uma boa funcionalidade.

6.1. Introdução ao DirectShow

Antes do DirectX 8.1 teríamos que fazer o download do DirectShow separado do SDK. Sem o DirectShow seria difícil, embora não impossível, criar um vídeo e visualizá-lo na tela. Uma possibilidade seria criar um objeto 3D e executar um áudio visual intercalado com textura. Porém, isto criaria um problema como ao tocar áudio, e sincronizar quando necessário. Com DirectShow, tudo que temos de fazer é criar uma interface de controle para podermos executar um vídeo. O DirectShow, então terá o cuidado de dividir o vídeo e o áudio, sincronizando isto, e controlando a maioria dos eventos que poderiam surgir dentro do playback do arquivo. Claro que poderíamos tentar escrever os próprios sincronizadores, e a biblioteca para executarmos e tocar o vídeo com o áudio em uma superfície de DirectDraw, ou de qualquer outra maneira; mas por que fazer isto se o DirectShow já provê está funcionalidade?

6.2. Fundamentos

Um exemplo básico seria criar uma aplicação Windows para usar DirectShow; neste caso necessitaríamos da biblioteca dshow.h e incluir ao projeto a strmiids.lib.

Então, precisaremos criar algumas variáveis globais que usaremos para controlar o playback e criar uma interface para o mesmo.

Assim, temos:

//Exemplo do artigo original:
//The Basics to Using DirectShow
//por Kurifu Roushu

```
//retirado do site: GameDev.net
source.cpp:

#include <windows.h>
#include <dshow.h>

//Variáveis Globais:
static IGraphBuilder *pGB = NULL;
static IMediaControl *pMC = NULL;
static IVideoWindow *pVW = NULL;
static IMediaEventEx *pME = NULL;
static HWND g_hwnd = 0;
```

Agora precisamos iniciar nossa interface COM, juntamente com a função WinMain, adicionando os seguintes comandos:

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
        LPSTR lpCmdLine, int mCmdShow){
 CoInitialize(NULL);

 ...
}
```

Quando terminarmos, chamaremos CoUninitialize (); para reiniciar novamente.

Agora precisamos criar a interface do componente no qual iniciaremos todos os outros objetos e eventualmente tocaremos o arquivo. Para iniciar o DirectShow faremos o seguinte:

```
//Isto cria um filtro gerenciador de gráficos
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC,
        IID_IGraphBuilder, (void **)&pGB);

//Interface COM
pGB->QueryInterface(IID_IMediaControl, (void **)&pMC);
pGB->QueryInterface(IID_IVideoWindow, (void **)&pVW);
```

Agora construiremos o gráfico. Aqui passamos o nome de arquivo no qual gostaríamos de tocar, e o DirectShow conectará todos os filtros para tocar o arquivo.

```
pGB->RenderFile(L"test.avi",NULL);
```

O "L" é usado para converter o nome do arquivo de uma string ASCII para uma string de caracteres longos (Wide Character string).

Os seguintes tipos de mídia são reconhecidos pelo DirectShow:

- Microsoft Windows Media Video codec versão 7.0*;
- ISO MPEG-4 video version 1.0*;
- Microsoft MPEG-4 version 3*;
- Sipro Labs ACELP*;
- Windows Media Audio*;
- MPEG Audio Layer-3 (MP3) (descompactado somente);
- Digital Video (DV);
- MPEG-1;
- MJPEG;
- Indeo;
- Voxware*;
- Cinepak.

Tudo que teremos que fazer agora é fixar a janela que DirectShow usará para executar o vídeo e a janela filha já criada, e então tocar o arquivo.

```
//Fixar a janela
pVW->put_Owner((OAHWND)g_hwnd);
//Fixar a janela filha
pVW->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS);
//Aqui criamos um RECT no qual escolhemos uma janela filha
RECT vwrect;
//Tamanho da janela pai
GetClientRect(g_hwnd,&vwrect);
```

```
//Fixar a posição da janela filha
pVW->SetWindowPosition(0,0,vwrect.right,vwrect.bottom);
//Executando
pGB->Run();
```

Agora temos que limpar:

```
pVW->put_Visible(OAFALSE);
pVW->put_Owner(NULL);
pMC->Release();
pVW->Release();
pGB->Release();
```

6.2.1. Controlando Eventos

Agora com as informações que foram passadas, é possível abrir e tocar um arquivo vídeo na tela com mínimo esforço. O único problema, é que quando o arquivo é executado, a aplicação para, nada mais acontece. Isto reforça o uso de handle para ajudar a controlar o playback do vídeo.

É onde ImediaEventEx entra em jogo, logo podemos acrescentar o seguinte:

```
#define WM_P_GRAPHNOTIFY WM_APP + 1
```

O Windows permitirá especificar mensagens privadas à função de callback com os valores de WM_APP, ou seja, 0xBFFF. A definição acima mostra um desses eventos callback.

Quando iniciamos o componente de interface, teremos que adicionar o seguinte código:

```
pGB->QueryInterface(IID_IMediaEventEx, (void **)&pME);
```

Fixando WM_P_GRAPHNOTIFY como uma mensagem callback de uma janela válida é como executar a seguinte linha:

```
pME->SetNotifyWindow((OAHWND)g_hwnd, WM_P_GRAPHNOTIFY, 0);
```

O primeiro parâmetro seria a janela na qual enviamos as mensagens, neste caso a janela pai, também g_hwnd. O segundo argumento é a mensagem que será enviada, e o terceiro argumento instancia o DirectShow para chamá-lo.

Porque o Windows não inclui informação útil neste momento no wparam ou lparams, temos que criar outra função que confira a mensagem DirectShow na fila.

```
case(msg) WM_P_GRAPHNOTIFY:
 HandleEvent();
 break;

...

void HandleEvent(){
 long evCode, param1, param2;

 HRESULT hr;

 while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0),
SUCCEEDED(hr))
 {
  hr = pEvent->FreeEventParams(evCode, param1, param2);

  if ((EC_COMPLETE == evCode) || (EC_USERABORT == evCode))
  {
   CleanUp();
   break;
  }
 }
}
```

Sendo as mensagens de Windows e as mensagens do DirectShow assíncronas, se verificarmos a fila de mensagens, nunca teremos um acumulo de

mensagens encontradas, até ocorrer um erro, quando saberemos que está vazio e seguro para parar. Porque saberemos que existe uma mensagem que espera ser usada, quando chamarmos GetEvent, passaremos 0 como o quarto argumento para não especificar nenhum intervalo. Depois que corrigirmos um evento, chamaremos FreeEventParams para libertar qualquer recurso checado na fila.

7. DirectSetup

Aplicações e jogos que dependem de Microsoft DirectX usam o DirectXSetup para instalar os componentes do sistemas necessários. A função opcional de atualização de drivers de display e áudio são otimizadas para suportar o DirectX. Tipicamente, chamaríamos o DirectXSetup do programa que estivermos usando para instalar seus próprios arquivos de aplicação.

Notamos que a função de DirectXSetup descreve de forma elaborada os componentes do sistemas e de versões prévias do DirectX. Por exemplo, se instalarmos DirectX 8.1 em um sistema que já tenha os componentes do DirectX 7.0, todos os componentes DirectX 7.0 serão reescritos. Porque todos os componentes de DirectX obedecem ao Modelo de Objeto de Componente (COM) regras de compatibilidade a versões anteriores, os software escrito para DirectX 7.0 continuará funcionando corretamente.

A função do DirectXSetup pode saber quando os componentes de DirectX, drivers de display, e drivers de áudio necessitam ser atualizados. Também pode distinguir se os componentes podem ou não ser atualizados sem afetar o sistema operacional do Windows. Podemos dizer que é uma atualização segura para o sistema operacional; não necessariamente para as aplicações que funcionam no computador. Algumas aplicações dependentes do hardware e podem ser afetadas negativamente por uma atualização que é segura para Windows.

Por default, a função de DirectXSetup executa só atualizações seguras. Se a atualização de um dispositivo de drivers puder afetar a operação de Windows, a atualização não será executada.

Durante o processo de setup, o DirectSetup cria uma cópia auxiliar dos componentes do sistemas e drivers que são substituídos. Estes podem ser restabelecidos se problemas acontecerem.

Quando os drivers de display áudio são atualizados, a função DirectXSetup usa um banco de dados criado pela Microsoft para administrar o processo. O banco de dados contém informações sobre drivers existentes que são compatíveis a Microsoft. Este banco de dados descreve o estado de atualização de cada driver, baseado em testes feitos na Microsoft e em outros locais.

Se o valor devolvido pelo DirectXSetup for DSETUPERR_SUCCESS_RESTART, notificará o usuário que as mudanças não entrarão em vigor sem um reiniciar, e oferecendo imediatamente a escolha reiniciar.

7.1. Preparando o Diretório Utilizado pelo Setup

A função DirectXSetup pega o parâmetro, lpszRootPath que aponta ao diretório de raiz de instalação. Pode ser NULO para indicar que o caminho do path é o diretório atual, o diretório onde seu programa de setup reside.

O diretório conterá os arquivos Dsetup.dll, e Dsetup32.dll. Também terá pasta nomeada "DirectX" (este nome não é sensitive case), contendo todos os arquivos de redistribuição e drivers. Para criar a própria estrutura no disco de setup, copie o conteúdo todo do diretório Dxf\Redist\DirectX8 do Microsoft DirectX SDK no disco com o caminho de seu programa de setup. Esta pasta não foi copiada para seu computador quando você instalou o SDK, assim você tem que obter isto no disco original.

7.2. Personalizando o Setup

O Microsoft DirectSetup permite definir uma função de callback para personalizar o processo de setup da Microsoft DirectX. Na documentação do DirectSetup, esta função de callback é chamada de DirectXSetupCallbackFunction, mas podemos dar qualquer nome.

Se uma função de callback não é compatível ao programa de setup, a função DirectXSetup mostra o estado e uma informação de erro em uma caixa de diálogo e obtém entrada de dados do usuário, chamando a MessageBox da função Microsoft Win32. Se um callback é criado, as informações que teriam sido usadas para criar a caixa de dialogo ou caixa de mensagem é passada ao invés do callback. A função de callback é chamada uma vez para cada componente de DirectX e dispositivos de driver que podem ser instalados ou podem ser atualizados.

Podemos usar as funções callback para fazer o seguinte:

- Empregar uma interface default, pois o programa de setup pode exibir mensagens de modos diferente usando MessageBox ou caixas de dialogo padrão Microsoft Windows. A função de callback permite integrar mensagens para o componente do DirectX de sua instalação em sua própria interface;
- Atualizar um indicador de progresso;

- Suprimir a exibição de status e mensagens de erro. Designers de programas para usuários novatos poderiam suprimir mensagens de erro e deixar a organização do programa de controle erros e fazer escolhas de atualização silenciosamente. Esta aproximação requer um maicr esforço de desenvolvimento para o programa de setup, mas pode ser apropriado para o publico alvo.

Os tópicos seguintes ajudam a personalizar o setup:

- Criando uma Função Callback DirectSetup;
- Fixando uma Função Callback;
- Usando Flags de atualização na Função Callback;
- Cancelar Default na Função Callback.

7.2.1 Criando uma Função Callback DirectSetup

Para personalizar o processo de setup, primeiro criamos uma função callback conforme o protótipo da DirectXSetupCallbackFunction, abaixo:

```
DWORD WINAPI DirectXSetupCallbackFunction(
    DWORD dwReason,
    DWORD dwMsgType,
    LPSTR szMessage,
    LPSTR szName,
    void *pInfo);
```

O nome da função é igual a do protótipo, mas isto é opcional. Os nomes dos parâmetros diferem ligeiramente do protótipo declarado na biblioteca Dsetup.h e são usados ao longo da discussão seguinte.

7.2.1.1. Parâmetros

O parâmetro de dwReason indica porque a função callback foi chamada.

O parâmetro dwMsgType recebe flags equivalente para o Microsoft DirectSetup, por default, passa a MessageBox, mostra como controlar esses botões e

que ícones serão exibidos. O interessante é que são as flags do botão que usamos para determinar que valores de retorno são esperados. Se este valor for 0, o evento nunca requer entradas do usuário, e DirectSetup regularmente exibe uma mensagem de status.

O parâmetro szMessage recebe o mesmo texto que DirectSetup, caso contrário passa uma caixa de diálogo ou uma MessageBox.

Quando um driver for um candidato a atualização, seu nome é passado no parâmetro de szName, e o pInfo aponta a informação de como a atualização será no handled – por exemplo, se DirectSetup recomenda que o antigo driver seja mantido ou atualizado.

A maneira como a função callback interpreta os parâmetros está completamente explicada. Tipicamente escolhemos quais mensagens exibir (baseado em dwReason) e quando apresentamos ao usuário alternativas, e podemos modificar a interface adequadamente.

7.2.1.2. Valor de Retorno

O valor devolvido pela função callback tem que obedecer às seguintes regras:

Quando dwMsgType for 0, o valor de retorno deve ser IDOK. Neste caso, não há nenhuma escolha a ser feita pelo usuário (Sua aplicação exibe uma mensagem de status, embora nenhuma entrada seja requerida.).

Se dwMsgType for nonzero, o valor de retorno deve ser igual ao devolvido pelo MessageBox, dando a equivalente escolha ao usuário.

Ao determinar o valor de retorno apropriado no segundo caso, testamos dwMsgType para os botões que normalmente teriam sido postos em uma MessageBox. O exemplo seguinte da função GetReply em Dinstall mostra com isto é feito:

```
/ * A variável global g_wReply identifica o botão de diálogo
  que teria sido selecionada pelo usuário. * /

switch (dwMsgType & 0x0000000F)
{
  / * Normalmente teria sido um OK e um botão de Cancela.
```

```
    IDBUT1 é equivalente ao botão de OK. Se o usuário não fizesse
    escolha, seria um Cancela. * /
  case MB_OKCANCEL:
    wDefaultButton = (g_wReply == IDBUT1) ? IDOK : IDCANCEL;
    break;

  / * E assim por diante com as outras combinações de botão. * /
   case MB_OK:
    wDefaultButton = IDOK;
    break;
  case MB_RETRYCANCEL:
    wDefaultButton = (g_wReply == IDBUT1) ? IDRETRY : IDCANCEL;
    break;
  case MB_ABORTRETRYIGNORE:
    if (g_wReply == IDBUT1)
      wDefaultButton = IDABORT;
    else if (g_wReply == IDBUT2)
      wDefaultButton = IDRETRY;
    else
      wDefaultButton = IDIGNORE;
    break;
  case MB_YESNOCANCEL:
    if (g_wReply == IDBUT1)
      wDefaultButton = IDYES;
    else if (g_wReply == IDBUT2)
      wDefaultButton = IDNO;
    else
      wDefaultButton = IDCANCEL;
    break;
  case MB_YESNO:
    wDefaultButton = (g_wReply == IDBUT1) ? IDYES : IDNO;
    break;
  default:
    wDefaultButton = IDOK;
}
```

Esta rotina traduz o clique no botão para o equivalente identificador de diálogo standard que o DirectSetup criaria. A variável wDefaultButton é fixada ao equivalente identificador do botão da caixa de diálogo standard , e este valor é devolvido no final para a função callback.

Necessariamente não temos que dar ao usuário uma escolha, até mesmo quando dwMsgType é nonzero. Por exemplo, poderiamos decidir atualizar os drivers automaticamente até mesmo se DirectSetup pedisse confirmação normalmente do usuário. A amostra de Dinstall faz isto quando o usuário pede para ver somente mensagens de problema; e a atualização do driver é considerada segura, como segue:

```
case DSETUP_CB_UPGRADE_SAFE:
  switch (dwMsgType & 0x0000000F)
  {
    case MB_YESNO:
    case MB_YESNOCANCEL:
      return IDYES;
    case MB_OKCANCEL:
    case MB_OK:
    default:
      return IDOK;
  }
  break;
```

A função callback devolve IDYES ou IDOK, dependendo qual botão tenha sido escolhido na caixa de diálogo, que pergunta ao usuário se deveria ou não proceder à atualização.

7.2.2. Fixando a Função Callback

Antes de chamar o DirectXSetup, notamos que se usarmos uma callback para chamar a função DirectXSetupSetCallback, então passaremos um ponteiro à callback como um parâmetro. O exemplo seguinte mostra como isto é feito:

```
DirectXSetupSetCallback(
```

(DSETUP_CALLBACK) DirectXSetupCallbackFunction);

7.2.3. Usando Flags de Atualização na Função Callback

Quando definimos uma aplicação como uma função callback, o DirectXSetupCallbackFunction é chamado pela função de DirectXSetup, é passado um parâmetro que contém a razão que a função callback foi invocada. Se a razão for DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, o parâmetro pInfo aponta a uma estrutura de DSETUP_CB_UPGRADEINFO que contém a flag que resume a função DirectXSetup e oferece recomendações de como executar a atualização de componentes e drivers da Microsoft de DirectX. A estrutura membro contém as flags e é chamada UpgradeFlags.

As flags têm as seguintes categorias:

- Flags de Atualização primária:
  Estas flags são mutuamente exclusivas. Um delas sempre está presente na estrutura membro UpgradeFlags. Podemos executá-la usando o operador AND com DSETUP_CB_UPGRADE_TYPE_MASK.
  DSETUP_CB_UPGRADE_FORCE
  DSETUP_CB_UPGRADE_KEEP
  DSETUP_CB_UPGRADE_SAFE
  DSETUP_CB_UPGRADE_UNKNOWN

- Flags de Atualização secundárias:
  Qualquer um dos dois ou ambos destas bandeiras podem estar presentes.
  DSETUP_CB_UPGRADE_CANTBACKUP
  DSETUP_CB_UPGRADE_HASWARNINGS

- Dispositivo Flag Ativado:
  Esta flag está presente se o dispositivo cujo driver está sendo atualizado for ativo. Pode ser combinado com quaisquer dos outros.
  DSETUP_CB_UPGRADE_DEVICE_ACTIVE

- Flags de dispositivo de Classe:

  Estas flags são mutuamente exclusivas. Um delas está sempre presente.

  DSETUP_CB_UPGRADE_DISPLAY

  DSETUP_CB_UPGRADE_MEDIA

Em resumo, toda vez o parâmetro de Reason tiver o valor DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, a estrutura membro UpgradeFlags apontará para o pInfo contendo uma Flag de Atualização Primária, zero ou mais Flags de Atualização Secundárias, zero ou um Dispositivo Flag Ativo e uma Flag de dispositivo de Classe.

Se o membro UpgradeFlags é fixado a DSETUP_CB_UPGRADE_KEEP, o componente de DirectX ou driver de dispositivo não pode ser atualizado sem o Microsoft Windows deixar de funcionar corretamente. A função DirectXSetup não executa uma atualização no componente ou driver.

Se o valor de DSETUP_CB_UPGRADE_FORCE está em UpgradeFlags significa que o componente ou o driver deve ser atualizado para o Windows funcionar corretamente. O DirectXSetup funciona atualizando o driver ou componente. É possível que a atualização possa afetar alguns programas no sistema. Quando a função DirectXSetup descobrir esta condição, o membro UpgradeFlags é fixado (DSETUP_CB_UPGRADE_FORCE | DSETUP_CB_UPGRADE_HAS_WARNINGS). Quando isto acontece, a função DirectXSetup executa a atualização e também executa uma advertência ao usuário.

Componentes e drivers são considerados seguros para a atualização se eles não afetarem a operação de Windows quando são instalados. Neste caso, o membro UpgradeFlags é fixado a DSETUP_CB_UPGRADE_SAFE. É possível que a atualização possa estar segura para o Windows, mas ainda pode causar problemas para programas instalados no sistema. Quando DirectXSetup descobrir esta condição, o membro UpgradeFlags contém o valor (DSETUP_CB_UPGRADE_SAFE | DSETUP_CB_UPGRADE_HAS_WARNINGS). Se esta condição acontecer, ocorre a ação default da função DirectXSetup é não executará a atualização.

7.2.4. Cancelar Default na Função Callback.

A aplicação definida como função DirectXSetupCallbackFunction pode anular alguns dos comportamentos default da função DirectXSetup retornando um valor.

Por exemplo, o padrão default do DirectXSetup é "não instalar" um componente se o tipo de atualização apontada pelo parâmetro pInfo for fixado em (DSETUP_CB_UPGRADE_SAFE | DSETUP_CB_UPGRADE_HAS_WARNI NGS). Neste caso, o parâmetro MsgType da função callback é fixado (MB_YESNO | MB_DEFBUTTON2). Sem uma função callback, o Microsoft DirectSetup apresentaria ao usuário uma caixa de diálogo cujo botão default seria NO. Se a função callback não vê a entrada do usuário, mas aceita a falta, devolve o IDNO. Para anular o default, a função callback devolve IDYES. Se anular o default, a função DirectXSetup notifica o usuário.

8. Conclusão

O DirectX é largamente usado no campo dos jogos, contudo a facilidade de controlarmos placas de som, placas gráficas 3D, e formatos de vídeo e DVD é impressionante. È possível conversar com outra pessoa utilizando componentes do DirectX, além de permitir executar algumas ações com comando de voz.

Seria possível então criar uma aplicação num ambiente de rede que simularia desde um simples recurso a uma aula de alfabetização a uma simulação de um aparelho ou máquina que normalmente custaria muito caro, simplesmente para ser utilizada como meio de capacitação profissional, ou seja, é possível desenvolver uma simulação 3D de uma situação real e a partir daí ensinar. Acredito que exista outras ferramentas que podem fazer a mesma coisa, contudo o DirectX já vem com o Windows, é totalmente gratuito e pode ser utilizado com várias linguagens de programação.

Poderíamos imaginar uma aula onde o professor através de um modem envia mensagens as aplicações individuais de seus alunos, podendo interagir com eles, explicando o que deve ser feito e podendo responder perguntas individualmente, uma verdadeira sala de chat.

As aplicações podem utilizar toda a capacidade do hardware, podendo integrar um computador com uma placa aceleradora de vídeo e um computador que não tenha, ou seja, a aplicação adapta-se ao hardware, não necessitando de qualquer interferência do usuário.

Contudo, pudemos observar que a performance cai muito sem uma placa aceleradora de vídeo; a aceleração por hardware é sofrível, quanto mais recurso melhor será seu desempenho; a memória RAM também é muito exigida na versão 8.1, como é compatível com o Windows XP, seria improvável ter menos de 128 MB.

Não existe um processador que tenha melhor performance com o DirectX, contudo alguns processadores já vem com alguma tecnologia voltada especificamente para o DirectX.

Para utilizar todas as características do Kit é necessário um Pentium III, AMD K6 II, ou outro compatível, mas quanto maior melhor.

Uma grande deficiência se refere à dependência em relação à plataforma Windows, uma vez que o DirectX não é compatível com outras plataformas, por exemplo, não existe uma versão para LINUX.

Acreditamos que apesar das limitações citadas acima, o DirectX é um software que pode ser melhor explorado, principalmente, na capacitação profissional através de simuladores, e no auxílio ao professor durante o processo de alfabetização.

9. Referências Bibliográficas

Livros

[FER02] Maicris Fernandes. *Programação de Jogos com Delphi usando DirectX.* Editora Relativo, 2002.

[TRU02] Stan Trujillo. *Tudo o que Você Precisa para Programar em Direct 3D.* Editora Ciência Moderna, 1997.

WWW (Word Wide Web)

[PER98] Adam Perer. *The DirectX Experience.* http://www.geocities.com/SiliconValley/Way/3390. (22/05/2002).

[GAM02] Gamedev.net. *GameDev.net* http://www.gamedev.net.(27/04/2002).