Programação Windows em C++

A seguir segue-se uma tradução dos capítulos iniciais do *livro OpenGL Game Development*, que descrevem como criar uma janela Windows através de código C++ puro, ao contrário da criação visual.

No livro ainda segue-se o assunto explicando como criar botões, painéis e outras coisas, porém eu traduzi até a parte em que se consegue criar uma janela totalmente funcional, pois meu objetivo é apenas adquirir um contexto para renderização com OpenGL sem usar GLUT.

Se você tem interesse pelo assunto que não traduzi, leia o livro. Ele está no catálogo de livros PROGAMING.

O livro diz que foi usado o MS Visual C++ 6.0, estou usando o MS Visual C++ 2005 (8.0) e não consegui compilar por ele não ter o *header* ***Windows.h***, por isso testei o código no Bloodshed Dev-C++ e no Code::Blocks, nos dois obtive sucesso.

Conceitos de Programação Orientada a Eventos

Programação orientada a eventos é um conceito simples, mas poderoso. Quando algo é feito, o computador envia um evento para o gerenciador, o qual processará esse evento e executará as ações associadas com ele. Na programação Windows, programação orientada a eventos é usada para criar interfaces gráficas para o usuário (GUIs) sem o desgastante trabalho de detectar cada possível clique, pressionamento e movimento que o programa permite. Em vez disso, o software automaticamente enviará uma mensagem dizendo, "O evento X aconteceu." Um simples exemplo é quando o desenvolvedor tem um botão que tocará um som quando pressionado. Em alguns casos, as especificações de software requerem que o desenvolvedor continuamente cheque se o botão foi pressionado; entretanto, quando programamos com eventos, o software pode automaticamente enviar uma mensagem de evento para o gerenciador de mensagens dizendo, "O botão X está pressionado" e nesse caso o som poderá ser tocado.

Infelizmente, o sistema operacional Windows tem mais que apenas um evento *ButtonDown*. Em um típico programa Windows você poderá usar dúzias de diferentes eventos para criar a aplicação ou jogo perfeito. Quase todos os cantos de uma interface gráfica para usuário Win32 estão cobertos com eventos de mensagens, desde a ativação do programa e os pressionamentos de teclas até a exibição do programa e os temporizadores de fundo. E se você precisa de um evento específico que não existe, muitas bibliotecas Windows tem a capacidade de criar eventos de mensagens Windows personalizados.

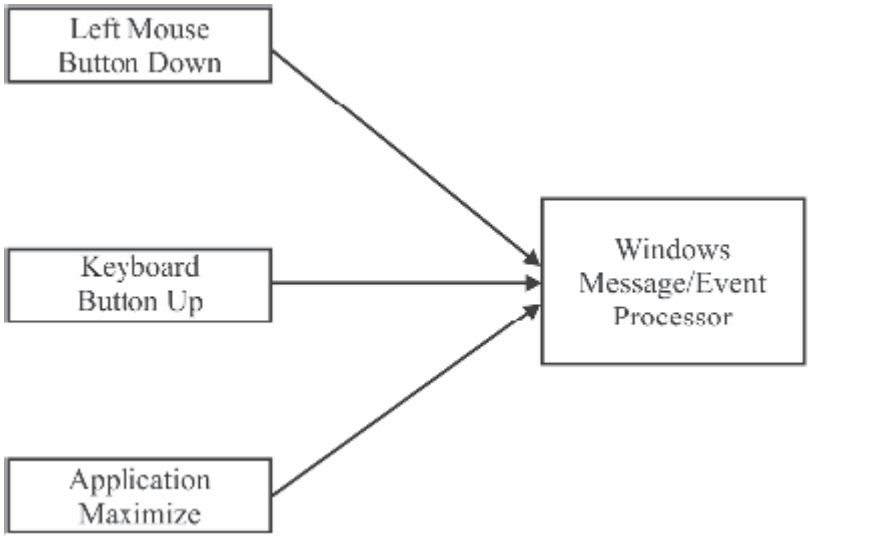

Figura 1: Fluxo de eventos

No Win32 SDK, usaremos a função de *callback Window Procedure* (*WndProc*) para gerenciamento de mensagens. A declaração será discutida em mais detalhes no próximo capitulo.

Introdução ao Win32 SDK

Quando desenvolvemos aplicações Windows no Microsoft Visual C++, há dois métodos diferentes que podemos usar para criar a aplicação. O primeiro metodo é usando o *framework* Microsoft Foundation Classes (MFC). O MFC é uma biblioteca orientada a objetos robusta que encapsula uma grande porção da interface de programação de aplicações (API) Windows numa classe C++. Geralmente quando desenvolvemos pequenas aplicações, o MFC é demais. Para simplificar, neste livro usaremos o método alternativo de escrever aplicações para Windows usando o *kit* de desenvolvimento de software (SDK) Win32. O Win32 SDK em alguns casos é menos robusto no quesito funcionalidades, mas é de longe o mais fácil para iniciantes entenderem.

Quando usamos o Win32 SDK, há dois tipos diferentes de aplicações que você pode criar: Win32 console e Win32. Uma aplicação Win32 console é similar a um programa em C no DOS. Você pode usar a funcionalidade Win32 SDK dentro de uma aplicação console; entretanto, ela deverá ser executada de dentro de um *prompt* MS-DOS. Uma aplicação Win32 é uma aplicação gráfica Windows comum.

Criar aplicações Win32 é muito diferente de crier um tipico programa C que muitos programadores estão familiarizados. Um bom exemplo é o ponto de entrada *main* do código de um programa C. Em aplicações Windows, não é um *main*, mas sim um *WinMain*, e em vez de suprir parâmetros opcionalmente para o WinMain, aqui é obrigatório. O tipo de retorno do *WinMain* ainda é um tipo *int*; entretanto devemos adicionar a convenção padrão de chamadas a Windows API. Os parâmetros padrão são muito diferentes. Agora são quatro parâmetros: dois do tipo HINSTANCE, um LPSTR, e um *int*. Em muitas variáveis Win32 você notará um H na frente do nome. Para instâncias, o tipo HINSTANCE é um gerenciador (***H****andler*) de instâncias. Uma instância é o próprio programa. Para os dois primeiros parâmetros usamos o tipo HINSTANCE com os nomes *hInstance* e *hPrevious*, respectivamente. O *hInstance* é a instância do programa que está rodando atualmente, e o *hPrevious* em aplicações de 32-bit é sempre NULL. O terceiro parâmetro é do tipo LPSTR. O tipo LPSTR é um ponteiro para uma string terminada em *null* de caracteres de 8-bit. A variável passa para o programa argumentos quando iniciado. Isto é similar a variável *argv* em programas C comuns, com a exceção que ela não inclui o nome do programa. O parâmetro final define como o programa deverá ser mostrado quando iniciado.

O seguinte código mostra o ponto de entrada *main* para aplicações Windows.

```
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevious,
                    LPSTR lpCmdString,
                    int CmdShow)
```

Agora que temos um *main*, estamos prontos para começar a construir uma aplicação Windows. Mas primeiro, devemos discutir quais *headers* incluir. Invés de incluir I/O padrão (stdio.h) e a biblioteca padrão (stdlib.h), precisaremos incluir o *header* Windows (windows.h). Este *header* inclui *links* para todos os *headers* relevantes para compilar um programa Windows básico. Com isso em mente, vamos crier uma nova aplicação Windows e usar e código acima para começar a programar. O seguinte código mostra o ponto de entrada *main* para um programa Windows.

```
#include <windows.h>

int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevious,
                    LPSTR lpCmdString,
                    int CmdShow)
{
     return (1);
}
```

Este código compilará no Visual C++; entretanto, estamos longe de criar uma aplicação Windows totalmente funcional. Antes de criarmos um programa Windows totalmente funcional, devemos registrar uma classe *window*, a qual define os atributos da janela.

Registrando uma classe Window

A classe *window* contém detalhes como o nome da classe, menus, a callback para mensagens e vários outros dados. Para declarar uma classe *window*, usamos o tipo WNDCLASS. Uma típica classe *window* pode parecer como se segue.

```
WNDCLASS wc;

wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
wc.hCursor = LoadCursor (NULL, IDC_ARROW);
wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);
wc.hInstance = hInstance;
wc.lpfnWndProc = WndProc;
wc.lpszClassName = "ME";
wc.lpszMenuName = NULL;
wc.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
```

Começando de cima, ajustamos a variável *cbClsExtra* para 0 por quê não queremos alocar bytes extra de memória além do tamanho da classe WNDCLASS. A variável *cbWndExtra* é ajustada para 0 por quê não precisamos alocar memória extra para a instância da janela. A variável *hbrBackground* ajusta a cor de fundo da janela. Usamos a função *GetStockObject* para pegar um pincel armazenado, o qual neste caso foi cinza claro, e dá-se um *typecasting* nele como tipo HBRUSH. O HBRUSH é um gerenciador para um objeto pincel. Em vez de usarmos LTGRAY_BRUSH, poderemos ter usado uma das seguintes cores:

- Cinza escuro (DKGRAY_BRUSH)
- Preto (BLACK_BRUSH)
- Branco (WHITE_BRUSH)

Entretanto, cinza claro é uma das cores de fundo padrão, então ficaremos com o básico.

A variável *hCursor* ajusta o cursor padrão para a janela. No nosso caso carregaremos um cursor padrão usando a função *LoadCursor*. Esta função requer dois parâmetros, o primeiro sendo a instância atual, a qual pode ser NULL, e o segundo sendo a ID do recurso do cursor, o qual neste caso pode ser IDC_ARROW. A variável *hIcon* define o ícone padrão para a aplicação. Para ajustar esta variável devemos usar a função *LoadIcon*, que retorna o gerenciador de uma ícone. O primeiro parâmetro para a função *LoadIcon* é a instância do programa atual, que é NULL. O segundo parâmetro é a ID do recurso do ícone, que é IDI_APPLICATION. Em ambas funções *LoadIcon* e *LoadCursor*, o primeiro parâmetro é do tipo HINSTANCE. Nós ajustamos o primeiro parâmetro como NULL para indicar que queremos usar os ícones/cursores padrão do Windows em vez de especificar recursos customizados. A variável *hInstance* requer o gerenciador desta instância, o qual é o *hInstance* do nosso *WinMain*.

A variável *lpfnWndProc* é o ponteiro para o procedimento *window*, o qual é nosso gerenciador de mensagens. Como discutido antes, nosso gerenciador de mensagens processará qualquer mensagem enviada pelo sistema. A variável *lpszClassName* é o nome da classe *window*, que em nosso caso é ME, abreviação para *Map Editor*. Mais tarde, quando criarmos uma janela, sera necessário reusar esse nome, então é uma boa idéia mantê-lo simples. A próxima variável, *lpszMenuName*, controla o menu que será carregado para a classe *window*. Não precisamos desse parâmetro, então podemos deixá-lo como NULL. E finalmente, a última variável na estrutura WNDCLASS que devemos preencher para indicar o estilo da classe *window*. Uma grande variedade de opções pode ser usada para o estilo. Eu escolhi os valores básicos de CS_HREDRAW e CS_VREDRAW, que redesenharão a tela quando a largura ou altura da janela for alterada, e também requisitei o CS_OWNDC, que dá a cada classe registrada um único contexto de dispositivo (*device context*).

Agora que todos os campos na estrutura WNDCLASS estão preenchidos, é hora de registrar um nova classe *window* chamando a função *RegisterClass* e passando a variável *wc* que preenchemos. Se o registro da classe teve sucesso, um número não-zero será retornado. Se a função falha, será retornado 0. Com isso em mente, o próximo fragmento de código mostra como registrar uma classe *window* e mostrar uma mensagem de erro se o registro falhar.

```
if (!RegisterClass(&wc))
{
     MessageBox (NULL,
                 "Error: Cannot Register Class",
                 "ERROR!",
                 MB_OK);
     return (0);
}
```

Como você pode ver, registrar a classe *window* é razoavelmente simples. Para mostrar a mensagem de erro, eu simplesmente usei a função *MessageBox* para mostrar uma mensagem de erro básica. A função *MessageBox* requer quatro parâmetros: um gerenciador para uma janela (HWND), duas *strings* terminadas em *null* do tipo LPCTSTR, as quais incluem a mensagem principal e o texto da barra de título da mensagem respectivamente, e um *int* para o tipo. Nós não precisamos qualquer caixa de mensagem elaborada por enquanto, então uma mensagem com um botão MB_OK ou OK básicos será o suficiente para as nossas necessidades.

Criando uma Window

Depois de registrar a classe window devemos crier uma janela na tela usando a função *CreateWindow*. Há 11 parâmetros simples nesta função, o primeiro é o LPCTSTR *lpClassName*, que é como o nome sugere o nome da nossa classe, "ME". O segundo parâmetro é outro LPCTSTR chamado *lpWindowName*, que é uma *string* identificando o nome da janela na barra de títulos. Para este parâmetro usaremos o valor "Map Editor". Depois do parâmetro *lpWindowName*, ajustaremos o estiloda janela usando estilos DWORD. Existem muitas opções para escolher quando ajustamos o estilo da janela; usaremos as opções padrão WS_OVERLAPPEDWINDOW e WS_VISIBLE motivos de simplicidade. O estilo WS_OVERLAPPEDWINDOW cria uma janela com uma barra de título com os botões Minimize e Maximize. O estilo WS_VISIBLE permite que a janela seja visível após a criação. Os próximos dois parâmetros são do tipo *int* e identificam as coordenadas iniciais X e Y para a janela. Depois das posições iniciais estão a largura e a altura da janela como tipo *int*. O oitavo parâmetro é *hWndParent*, que é do tipo HWND. Este parâmetro é usado quando uma janela filha deve ser criada. Desde que esta janela é a pai, podemos manter este parâmetro como NULL. O próximo parâmetro é do tipo HMENU. Se quisermos carregar um menu podemos usar a função *LoadMenu* e retornar um valor neste parâmetro; entretanto, não precisamos dessa funcionalidade no momento e assim este parâmetro pode ser NULL por enquanto. Mais tarde discutiremos como criar, mostrar e adicionar funcionalidades aos menus. Depois no parâmetro *hMenu* incluiremos a instância atual, que no nosso caso é *hInstance*. O parâmetro final é um LPVOID chamado *lpParam*. Este parâmetro é usado quando criamos uma interface de documentos múltiplos (MDI - *Multiple-Document Interface*). Isto vai além dos tópicos discutidos neste livro e sendo assim deixaremos este parâmetro como NULL.

Agora que preenchemos a função *CreateWindow*, é uma boa idéia declarar a variável do nosso gerenciador de janelas em cima do código fonte como uma variável global. Na seção global declararemos um HWND chamado *Window*. Colocaremos nosso código *CreateWindow* depois da chamada a função *RegisterClass*.

```
if (!RegisterClass (&wc))
{
     MessageBox (NULL,
                 "Error: Cannot Register Class",
                 "ERROR!",
                 MB_OK);
     return (0);
}

Window = CreateWindow("ME",
                      "Map Editor",
                      WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                      0,
                      0,
                      640,
                      480,
                      NULL,
                      NULL,
                      hInstance,
                      NULL);
```

É uma boa idéia testar se o valor retornado pela janela depois de chamar a função *CreateWindow* é NULL. Se a criação da janela falha, NULL é retornado e devemos mostrar uma mensagem de erro e sair do programa. Como todos os erros, usaremos a função *MessageBox* para mostrar o erro e retornar 0 para indicar que um erro ocorreu. O seguinte código mostra a verificação de erro na criação da nossa janela.

```
if (Window == NULL)
{
     MessageBox (NULL,
                 "Error: Failed to Create Window",
                 "ERROR!",
                 MB_OK);
     return (0);
}
```

O *loop* principal

Depois de criar a janela devemos escrever o *loop* principal do software. O *loop* principal sera criado para rodar indefinidamente. A cada passo do *loop* verificaremos uma mensagem WM_QUIT, com a qual o programa sairá do *loop*. Para fazer isso devemos primeiro declarar uma variável chamada *msg* do tipo MSG. O tipo MSG contém informações sobre mensagens que são enviadas do programa. Usaremos essa variável como um dos parâmetros para procurar mensagens. No *loop while* teremos uma condição verdadeira fixa (1) para permitir a continuidade do *loop*. Dentro do *loop* usaremos a função PeekMessage para procurar mensagens na fila de mensagens.

A função *PeekMessage* tem cinco parâmetros que precisam ser preenchidos. O primeiro é o endereço da variável *msg*, o segundo é o gerenciador de janelas do qual pegaremos as mensagens. Este valor será NULL. Os próximos dois valores são inteiros positivos que especificam o alcance das mensagens que podemos usar, que em nosso caso será 0 por padrão. E finalmente, o quinto parâmetro é um inteiro positive que descreve como a mensagem será gerenciada. Há duas opções para o valor: PM_NOREMOVE, que não remove a mensagem da fila e PM_REMOVE, que remove a mensagem da fila. Usaremos o último dos dois. Quando a *PeekMessage* é chamada e uma mensagem está na fila, ela é colocada dentro da nossa variável *msg* e um valor não-zero é retornado. Se não há dados na fila, o valor retornado é 0.

Quando a *PeekMessage* é bem sucedida devemos verificar se a mensagem enviada é WM_QUIT, que é a mensagem de saída designada para terminar nosso programa. Um simples *if* comparando a mensagem da variável *msg* e o valor WM_QUIT pode ser suficiente; se os valores são iguais, use o comando *break* para quebrarmos o *loop while*. Se a mensagem não é WM_QUIT, então devemos chamar as funções *TranslateMessage* e *DispatchMessage*, que transferirá a mensagem dos pressionamento de teclas virtuais para as mensagens de caractere e despachará a nova mensagem para a função de *callback Window Procedure*. O código do *loop* principal quando completo parecerá com o seguinte:

```
while (1)
{
     if (PeekMessage (&msg,
                      NULL,
```

```
                    0,
                    0,
                    PM_REMOVE))
    {
        if (msg.message == WM_QUIT) break;
        TranslateMessage(&msg);
        DispatchMessage (&msg);
    }
}
```

O gerenciador de mensagens

A próxima seção que precisa ser endereçada antes de nossa aplicação compilar é o gerenciador de mensagens. Como dito anteriormente, o gerenciador de mensagens processa qualquer mensagem como pressionamento de teclas, movimento do mouse, cliques em botões e outros tipos de interatividade. O valor retornado pelo gerenciador de mensagens é do tipo LRESULT, que é um inteiro de 32-bit. Também usaremos o tipo de dado CALLBACK em vez do FAR PASCAL. O gerenciador de mensagens tem quatro parâmetros. O primeiro é do tipo HWND e é um gerenciador para a janela atual. O segundo parâmetro é um inteiro positivo (UINT), que contém o tipo de mensagem sendo enviada. O terceiro parâmetro é um do tipo WPARAM, que é um inteiro de 32-bit. Esta variável contém informações adicionais sobre as mensagens que estão sendo enviadas. O quarto e último parâmetro é o gerenciador de mensagens do tipo de dados LPARAM. O LPARAM é similar ao WPARAM no que é uma variável de 32-bit que contém informações adicionais sobre mensagens. Com o tipo de retorno sendo do tipo LRESULT, é uma boa idéia retornar o valor retornado pela função *DefWindowProc*. A função *DefWindowProc* assegurará que todas as mensagens serão processadas através de seu respectivo gerenciador de mensagens ou como mensagens padrão. Cada mensagem produz diferentes valores de retorno; portanto o uso desta função facilitará o tratamento dos valores retornados pelas mensagens e o gerenciamento de mensagens padrão. Os parâmetros da função *DefWindowProc* são exatamente os mesmos da *callback WndProc*, tornando fácil lembrar os parâmetros da função.

Gerenciar mensagens no gerenciador de mensagens é relativamente simples. Usaremos um *switch* na variável *msg* para controlar cada mensagem. A primeira mensagem a ser gerenciada será a WM_DESTROY, que é enviada pela aplicação quando o botão "X" é clicado no canto superior direito de todas as aplicações Windows. Esta mensagem não significa necessariamente que temos que sair do programa; entretanto, é bom seguir os padrões gerais e adicionar a funcionalidade de saída. Para sair da nossa aplicação usaremos a função *PostQuitMessage*, que enviará uma mensagem WM_QUIT para a aplicação. O único parâmetro requerido pela *PostQuitMessage* é um inteiro, que é o código de erro que você retornará. Em nosso caso este número será o número mágico 0. O gerenciador de mensagens está completo agora e o código é mostrado abaixo.

```
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT msg,
                         WPARAM wParam,
                         LPARAM lParam)
{
    switch (msg)
```

```
     {
          case WM_DESTROY: PostQuitMessage(0); break;
     }
     return (DefWindowProc(hWnd, msg, wParam, lParam));
}
```

Com a adição do gerenciador de mensagens do Windows acabamos nossa primeira aplicação.

A primeira aplicação Windows

Agora que acabamos nossa primeira aplicação, é hora de revisar o código fonte e testar o programa.

Como você pode ver, este exemplo é exatamente como escrevemos as seções individuais anteriormente a este capitulo. Nossas informações de cabeçalho somente requerem o arquivo *windows.h* no momento; entretanto, isto mudará muito. Abaixo das informações de cabeçalho estão nossas variáveis globais, que no momento é somente a *Window*. O gerenciador de mensagens (*WndProc*) está abaixo das variáveis globais. E finalmente, temos o *WinMain* na parte de baixo do código fonte. Você pode alternativamente colocá-lo no topo da aplicação se você escrever os protótipos das funções apropriadamente antes de qualquer função ser definida. Como uma preferência pessoal eu gosto do *WinMain* embaixo do arquivo fonte principal, por que eu o encontro mais facilmente para construir o software de baixo para cima (embaixo do código fonte) ao oposto do *top down* (de cima para baixo). Ambos os estilos tem os mesmos objetivos no final, mas o último requer protótipos de funções para o código fonte compilar.

Example ex1_1.cpp:

```
#include <windows.h>

HWND Window;

LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT msg,
                         WPARAM wParam,
                         LPARAM lParam)
{
     switch (msg)
     {
          case WM_DESTROY: PostQuitMessage(0); break;
     }
     return (DefWindowProc(hWnd, msg, wParam, lParam));
}

int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevious,
                    LPSTR lpCmdString,
                    int CmdShow)
{
     WNDCLASS wc;
     MSG msg;
     wc.cbClsExtra = 0;
```

```
     wc.cbWndExtra = 0;
     wc.hbrBackground =
(HBRUSH)GetStockObject(LTGRAY_BRUSH);
     wc.hCursor = LoadCursor (NULL, IDC_ARROW);
     wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);
     wc.hInstance = hInstance;
     wc.lpfnWndProc = WndProc;
     wc.lpszClassName = "ME";
     wc.lpszMenuName = NULL;
     wc.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;

     if (!RegisterClass(&wc))
     {
          MessageBox (NULL,
                      "Error: Cannot Register Class",
                      "ERROR!",
                      MB_OK);
          return (0);
     }

     Window = CreateWindow("ME",
                           "Map Editor",
                           WS_OVERLAPPEDWINDOW |
                           WS_VISIBLE,
                           0,
                           0,
                           640,
                           480,
                           NULL,
                           NULL,
                           hInstance,
                           NULL);

     if (Window == NULL)
     {
          MessageBox (NULL,
                      “Error: Failed to Create Window”,
                      “ERROR!”,
                      MB_OK);
          return (0);
     }

     while (1)
     {
          if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
               if (msg.message == WM_QUIT) break;
               TranslateMessage(&msg);
               DispatchMessage (&msg);
          }
     }
     return (1);
}
```