5

Event Driven Programming: Making your first 2D game

In chapter, the following recipes will be covere

* Starting to make a Windows Game
* Using Windows classes and handles
* Creating your first window
* Adding keyboard and mouse controls with text output
* Using Windows resources with GDI
* Using Dialogs and Controls
* Using sprites
* Using Animated Sprites

# Introduction

Windows programming is the start of creating proper applications. We need to know how to package our game into one executable file so that all our resources like images, models, sounds are encrypted properly and packaged into one file. By doing this, the files are safe and cannot be illegally copied on distribution. The application however makes use of these files at runtime.

Windows programming also marks the start of understanding the Windows Message Pump. This system is very important to understand as all major programming paradigm will depend on this principle, especially when we are doing event driven programming.

The main principle of event driven programming is that based on events we should process something. The concept to be understood here is how often do we check for events and how often should we process them.

# Starting to make a Windows game

1. The first thing to understand before we start making a windows game is how a window or a message box is drawn. We need to be aware of the numerous inbuilt functions which Windows provide us and the different call-back functions that we can use.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will see how easy it is to create a message box in Windows. There are different types of message boxes we can create and it is only matter of few lines of codes that takes us to achieve this.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 Windows application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#define WIN32\_LEAN\_AND\_MEAN

#include <windows.h>

#include <windowsx.h>

int WINAPI WinMain(HINSTANCE \_hInstance,

HINSTANCE \_hPrevInstance,

LPSTR \_lpCmdLine,

int \_iCmdShow)

{

MessageBox(NULL, L"My first message",

L"My first Windows Program",

MB\_OK | MB\_ICONEXCLAMATION);

return (0);

}

## How it works...

WINMAIN() is the entry point of a Windows program. In this example we have used the inbuilt function to create a message box. The windows.h contains all the necessary files that we need to call the inbuilt functions present in the Windows API. A message box is typically used to display something. We can also the message box to associate with default windows sounds. The display of the message box can also be controlled to a great extent. We need to use the right type of parameter to achieve this.

There are other types of message boxes which we can use as well.

* **MB\_OK** One button, OK.
* **MB\_OKCANCEL** Two buttons, OK, Cancel.
* **MB\_RETRYCANCEL** Two buttons, Retry, Cancel.
* **MB\_YESNO** Two buttons, Yes, No.
* **MB\_YESNOCANCEL** Three buttons, Yes, No, Cancel.
* **MB\_ABORTRETRYIGNORE** Three buttons, Abort, Retry, Ignore.
* **MB\_ICONEXCLAIMATION** An exclamation point icon appears.
* **MB\_ICONINFORMATION** An information icon appears.
* **MB\_ICONQUESTION** A question mark icon appears.
* **MB\_ICONSTOP** A stop sign icon appears.

Like all good Win32 API functions, MessageBox returns a value to let us know what happened.

# Using Windows classes and handles

To write games, we do not need to know a lot about Windows programming. What we need to know is how to open a windows, how to process messages and how to call the main game loop. The starting purpose of a Windows application is to create a window. After the windows is created, we can do various other things, like processing events and handling call-backs. These events are finally used by the game framework to display sprites on the screen and make them movable and interact able so that we can play a game.

## Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will find out how easy it is to use Windows classes and handles.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 Windows application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#define WIN32\_LEAN\_AND\_MEAN

#include <windows.h> // Include all the windows headers.

#include <windowsx.h> // Include useful macros.

#define WINDOW\_CLASS\_NAME L"WINCLASS1"

void GameLoop()

{

//One frame of game logic occurs here...

}

LRESULT CALLBACK WindowProc(HWND \_hwnd,

UINT \_msg,

WPARAM \_wparam,

LPARAM \_lparam)

{

// This is the main message handler of the system.

PAINTSTRUCT ps; // Used in WM\_PAINT.

HDC hdc; // Handle to a device context.

// What is the message?

switch (\_msg)

{

case WM\_CREATE:

{

// Do initialization stuff here.

// Return Success.

return (0);

}

break;

case WM\_PAINT:

{

// Simply validate the window.

hdc = BeginPaint(\_hwnd, &ps);

// You would do all your painting here...

EndPaint(\_hwnd, &ps);

// Return Success.

return (0);

}

break;

case WM\_DESTROY:

{

// Kill the application, this sends a WM\_QUIT message.

PostQuitMessage(0);

// Return success.

return (0);

}

break;

default:break;

} // End switch.

// Process any messages that we did not take care of...

return (DefWindowProc(\_hwnd, \_msg, \_wparam, \_lparam));

}

int WINAPI WinMain(HINSTANCE \_hInstance,

HINSTANCE \_hPrevInstance,

LPSTR \_lpCmdLine,

int \_nCmdShow)

{

WNDCLASSEX winclass; // This will hold the class we create.

HWND hwnd; // Generic window handle.

MSG msg; // Generic message.

// First fill in the window class structure.

winclass.cbSize = sizeof(WNDCLASSEX);

winclass.style = CS\_DBLCLKS | CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

winclass.lpfnWndProc = WindowProc;

winclass.cbClsExtra = 0;

winclass.cbWndExtra = 0;

winclass.hInstance = \_hInstance;

winclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

winclass.hCursor = LoadCursor(NULL, IDC\_ARROW);

winclass.hbrBackground =

static\_cast<HBRUSH>(GetStockObject(WHITE\_BRUSH));

winclass.lpszMenuName = NULL;

winclass.lpszClassName = WINDOW\_CLASS\_NAME;

winclass.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

// register the window class

if (!RegisterClassEx(&winclass))

{

return (0);

}

// create the window

hwnd = CreateWindowEx(NULL, // Extended style.

WINDOW\_CLASS\_NAME, // Class.

L"My first Window", // Title.

WS\_OVERLAPPEDWINDOW | WS\_VISIBLE,

0, 0, // Initial x,y.

400, 400, // Initial width, height.

NULL, // Handle to parent.

NULL, // Handle to menu.

\_hInstance, // Instance of this application.

NULL); // Extra creation parameters.

if (!(hwnd))

{

return (0);

}

// Enter main event loop

while (true)

{

// Test if there is a message in queue, if so get it.

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

// Test if this is a quit.

if (msg.message == WM\_QUIT)

{

break;

}

// Translate any accelerator keys.

TranslateMessage(&msg);

// Send the message to the window proc.

DispatchMessage(&msg);

}

// Main game processing goes here.

GameLoop(); //One frame of game logic occurs here...

}

// Return to Windows like this...

return (static\_cast<int>(msg.wParam));

}

## How it works...

The entire typedef struct \_WNDCLASSEXis defined as follows:

{

UINT cbSize; // Size of this structure.

UINT style; // Style flags.

WNDPROC lpfnWndProc; // Function pointer to handler.

int cbClsExtra; // Extra class info.

int cbWndExtra; // Extra window info.

HANDLE hInstance; // The instance of the app.

HICON hIcon; // The main icon.

HCURSOR hCursor; // The cursor for the window.

HBRUSH hbrBackground; // The Background brush to paint   
 // the window.

LPCTSTR lpszMenuName; // The name of the menu to attach

LPCTSTR lpszClassName;// The name of the class itself.

HICON hIconSm; // The handle of the small icon.

} WNDCLASSEX;

The Windows API provides us multiple API callbacks. We need to decide which message to intercept and process what information in that message pump. For example, the WM\_CREATE is a Windows create. We should do most of our initializations here. Similarly, WM\_DESTROY is where we need to destroy our created objects. We need to use GDI objects alongside to paint boxes and other things on the window. We can also use our own cursors and icons to display to the window.

# Creating your first Window

Creating a Window is the first step in Windows Programming. All our sprites and other objects will be drawn on top of this window. There is a standard way of drawing a window. So this part of the code will be repeated in all programs using Windows programming to draw something.

## Getting ready

1. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

In this recipe we will find out how easy it is to create a window.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 windows application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

1. #define WIN32\_LEAN\_AND\_MEAN
2. #include <windows.h> // Include all the windows headers.
3. #include <windowsx.h> // Include useful macros.
4. #include "resource.h"
5. #define WINDOW\_CLASS\_NAME L"WINCLASS1"
6. void GameLoop()
7. {
8. //One frame of game logic occurs here...
9. }
10. LRESULT CALLBACK WindowProc(HWND \_hwnd,
11. UINT \_msg,
12. WPARAM \_wparam,
13. LPARAM \_lparam)
14. {
15. // This is the main message handler of the system.
16. PAINTSTRUCT ps; // Used in WM\_PAINT.
17. HDC hdc; // Handle to a device context.
18. // What is the message?
19. switch (\_msg)
20. {
21. case WM\_CREATE:
22. {
23. // Do initialization stuff here.
24. // Return Success.
25. return (0);
26. }
27. break;
28. case WM\_PAINT:
29. {
30. // Simply validate the window.
31. hdc = BeginPaint(\_hwnd, &ps);
32. // You would do all your painting here...
33. EndPaint(\_hwnd, &ps);
34. // Return Success.
35. return (0);
36. }
37. break;
38. case WM\_DESTROY:
39. {
40. // Kill the application, this sends a WM\_QUIT message.
41. PostQuitMessage(0);
42. // Return success.
43. return (0);
44. }
45. break;
46. default:break;
47. } // End switch.
48. // Process any messages that we did not take care of...
49. return (DefWindowProc(\_hwnd, \_msg, \_wparam, \_lparam));
50. }
51. int WINAPI WinMain(HINSTANCE \_hInstance,
52. HINSTANCE \_hPrevInstance,
53. LPSTR \_lpCmdLine,
54. int \_nCmdShow)
55. {
56. WNDCLASSEX winclass; // This will hold the class we create.
57. HWND hwnd; // Generic window handle.
58. MSG msg; // Generic message.
59. HCURSOR hCrosshair = LoadCursor(\_hInstance, MAKEINTRESOURCE(IDC\_CURSOR2));
60. // First fill in the window class structure.
61. winclass.cbSize = sizeof(WNDCLASSEX);
62. winclass.style = CS\_DBLCLKS | CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;
63. winclass.lpfnWndProc = WindowProc;
64. winclass.cbClsExtra = 0;
65. winclass.cbWndExtra = 0;
66. winclass.hInstance = \_hInstance;
67. winclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);
68. winclass.hCursor = LoadCursor(\_hInstance, MAKEINTRESOURCE(IDC\_CURSOR2));
69. winclass.hbrBackground =
70. static\_cast<HBRUSH>(GetStockObject(WHITE\_BRUSH));
71. winclass.lpszMenuName = NULL;
72. winclass.lpszClassName = WINDOW\_CLASS\_NAME;
73. winclass.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);
74. // register the window class
75. if (!RegisterClassEx(&winclass))
76. {
77. return (0);
78. }
79. // create the window
80. hwnd = CreateWindowEx(NULL, // Extended style.
81. WINDOW\_CLASS\_NAME, // Class.
82. L"Packt Publishing", // Title.
83. WS\_OVERLAPPEDWINDOW | WS\_VISIBLE,
84. 0, 0, // Initial x,y.
85. 400, 400, // Initial width, height.
86. NULL, // Handle to parent.
87. NULL, // Handle to menu.
88. \_hInstance, // Instance of this application.
89. NULL); // Extra creation parameters.
90. if (!(hwnd))
91. {
92. return (0);
93. }
94. // Enter main event loop
95. while (true)
96. {
97. // Test if there is a message in queue, if so get it.
98. if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))
99. {
100. // Test if this is a quit.
101. if (msg.message == WM\_QUIT)
102. {
103. break;
104. }
105. // Translate any accelerator keys.
106. TranslateMessage(&msg);
107. // Send the message to the window proc.
108. DispatchMessage(&msg);
109. }
110. // Main game processing goes here.
111. GameLoop(); //One frame of game logic occurs here...
112. }
113. // Return to Windows like this...
114. return (static\_cast<int>(msg.wParam));
115. }

## How it works...

In this example, we have used the standard Windows API call-back. We query on the message parameter that is passed and based on that, we intercept and perform suitable actions. We have used the WM\_PAINT message to paint the window for us and the WM\_DESTROY message to destroy the current window. To paint the window, we need a handle to the device context and then appropriately we can BeginPaint and EndPaint. In the main structure, we need to fill up the windows structures and specify the current cursor and icons that needs to be loaded. Here we can specify what colour brush we are going to use to paint the window. Finally, the size of the window is specified and registered. After that we need to continuously peek messages, translate them and finally dispatch them to the windows procedure.

# Adding keyboard and mouse controls with text output

One of the most important things that we require in a video game is a human interface to interact with. The most common interface device is the keyboard and the mouse. Hence it is very important to understand how they work and how we can detect the key presses and movements. It is equally important to know how to display certain text to the screen which can be really useful for debugging and for HUD implementation.

## Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe, we will find out how easy it is to detect Keyboard and Mouse events.

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 Windows application
4. Add a source file called Source.cpp
5. Add the following lines of code.

**Source.cpp**

#define WIN32\_LEAN\_AND\_MEAN // No MFC

#include <windows.h> //Include all the Windows headers.

#include <windowsx.h> //Include useful macros.

#include <strstream>

#include <string>

#include <cmath>

#include "resource.h"

#include "mmsystem.h"

//also uses winmm.lib

using namespace std;

#define WINDOW\_CLASS\_NAME "WINCLASS1"

HINSTANCE g\_hInstance;

//RECT g\_rect;

const RECT\* g\_prect;

POINT g\_pos;

int g\_iMouseX;

int g\_iMouseY;

bool IS\_LEFT\_PRESSED = 0;

bool IS\_RIGHT\_PRESSED = 0;

bool IS\_UP\_PRESSED = 0;

bool IS\_DOWN\_PRESSED = 0;

bool IS\_LMB\_PRESSED = 0;

bool IS\_RMB\_PRESSED = 0;

bool IS\_MMB\_PRESSED = 0;

int LAST\_KEYPRESS\_ASCII = 0;

float ang = 0.0f;

template<typename T>

std::string ToString(const T& \_value)

{

std::strstream theStream;

theStream << \_value << std::ends;

return (theStream.str());

}

//GameLoop

void GameLoop()

{

ang += 0.0005f;

//One frame of game logic goes here

}

//Event handling (window handle, message handle --

LRESULT CALLBACK WindowProc(HWND \_hwnd, UINT \_msg, WPARAM \_wparam, LPARAM \_lparam)

{

//This is the main message handler of the system.

PAINTSTRUCT ps; //Used in WM\_PAINT

HDC hdc; // Handle to a device context.

if ((GetAsyncKeyState(VK\_LEFT) & 0x8000) == 0x8000)

{

IS\_LEFT\_PRESSED = TRUE;

}

else

{

IS\_LEFT\_PRESSED = FALSE;

}

if ((GetAsyncKeyState(VK\_RIGHT) & 0x8000) == 0x8000)

{

IS\_RIGHT\_PRESSED = TRUE;

}

else

{

IS\_RIGHT\_PRESSED = FALSE;

}

if ((GetAsyncKeyState(VK\_UP) & 0x8000) == 0x8000)

{

IS\_UP\_PRESSED = TRUE;

}

else

{

IS\_UP\_PRESSED = FALSE;

}

if ((GetAsyncKeyState(VK\_DOWN) & 0x8000) == 0x8000)

{

IS\_DOWN\_PRESSED = TRUE;

}

else

{

IS\_DOWN\_PRESSED = FALSE;

}

//What is the message?

switch(\_msg)

{

case WM\_CREATE:

{

//Do initialisation stuff here.

//Return success.

return(0);

}

break;

case WM\_PAINT:

{

////Simply validate the window.

hdc = BeginPaint(\_hwnd, &ps);

InvalidateRect( \_hwnd,

g\_prect,

FALSE);

string temp;

int iYDrawPos = 15;

COLORREF red = RGB(255,0,0);

SetTextColor(hdc, red);

temp = "MOUSE X: ";

temp += ToString((g\_pos.x));

while (temp.size() < 14)

{

temp += " ";

}

TextOut(hdc,30,iYDrawPos,temp.c\_str(), static\_cast<int>(temp.size()));

iYDrawPos+= 13;

temp = "MOUSE Y: ";

temp += ToString((g\_pos.y));

while (temp.size() < 14)

{

temp += " ";

}

TextOut(hdc,30,iYDrawPos,temp.c\_str(), static\_cast<int>(temp.size()));

iYDrawPos+= 13;

if (IS\_LEFT\_PRESSED == TRUE)

{

TextOut(hdc,30,iYDrawPos,"LEFT IS PRESSED ", 24);

}

else

{

TextOut(hdc,30,iYDrawPos,"LEFT IS NOT PRESSED ", 20);

}

iYDrawPos+= 13;

if (IS\_RIGHT\_PRESSED == TRUE)

{

TextOut(hdc,30,iYDrawPos,"RIGHT IS PRESSED ", 25);

}

else

{

TextOut(hdc,30,iYDrawPos,"RIGHT IS NOT PRESSED ", 21);

}

iYDrawPos+= 13;

if (IS\_DOWN\_PRESSED == TRUE)

{

TextOut(hdc,30,iYDrawPos,"DOWN IS PRESSED ", 24);

}

else

{

TextOut(hdc,30,iYDrawPos,"DOWN IS NOT PRESSED ", 20);

}

iYDrawPos+= 13;

if (IS\_UP\_PRESSED == TRUE)

{

TextOut(hdc,30,iYDrawPos,"UP IS PRESSED ", 22);

}

else

{

TextOut(hdc,30,iYDrawPos,"UP IS NOT PRESSED ", 18);

}

// TextOut(hdc, static\_cast<int>(200 +(sin(ang)\*200)), static\_cast<int>(200 +(sin(ang)\*200))) , "O", 1);

EndPaint(\_hwnd, &ps);

//Return sucess.

return(0);

}

break;

case WM\_DESTROY:

{

//Kill the application, this sends a WM\_QUIT message.

PostQuitMessage(0);

//Return Sucess.

return(0);

}

break;

case WM\_MOUSEMOVE:

{

GetCursorPos(&g\_pos);

// here is your coordinates

//int x=pos.x;

//int y=pos.y;

return(0);

}

break;

case WM\_COMMAND:

{

}

default:break;

} // End switch.

//Process any messages we didn't take care of...

return(DefWindowProc(\_hwnd, \_msg, \_wparam, \_lparam));

}

int WINAPI WinMain(HINSTANCE \_hInstance, HINSTANCE \_hPrevInstance, LPSTR \_lpCmdLine, int \_nCmdShow)

{

WNDCLASSEX winclass; ///This wil hold the class we create

HWND hwnd; //Generic window handle.

MSG msg; //Generic message.

g\_hInstance = \_hInstance;

//First fill in the window class structure

winclass.cbSize = sizeof(WNDCLASSEX);

winclass.style = CS\_DBLCLKS | CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

winclass.lpfnWndProc = WindowProc;

winclass.cbClsExtra = 0;

winclass.cbWndExtra = 0;

winclass.hInstance = \_hInstance;

winclass.hIcon = LoadIcon(g\_hInstance, MAKEINTRESOURCE(IDI\_ICON1));

winclass.hCursor = NULL;

winclass.hbrBackground = static\_cast<HBRUSH>(GetStockObject(WHITE\_BRUSH));

winclass.lpszMenuName = MAKEINTRESOURCE(IDR\_MENU1);

winclass.lpszClassName = WINDOW\_CLASS\_NAME;

winclass.hIconSm = LoadIcon(g\_hInstance, MAKEINTRESOURCE(IDI\_ICON1));

//Register the window class

if (!RegisterClassEx(&winclass))

{ //perhaps use log manager here

return(0);

}

//Create the window

if (!(hwnd = CreateWindowEx(NULL, //Extended style.

WINDOW\_CLASS\_NAME, //Class

"Recipe4", //Title

WS\_OVERLAPPEDWINDOW | WS\_VISIBLE,

400,300, //Initial X, Y

400,400, //Initial width, height.

NULL, //handle to parent.

NULL, //handle to menu

\_hInstance, //Instance of this application

NULL))) //Extra creation parameters

{

return (0);

}

RECT rect;

rect.left = 0;

rect.right = 400;

rect.top = 0;

rect.bottom = 400;

g\_prect = &rect;

//Enter main event loop

while (TRUE)

{

//Test if there is a message in queue, if so get it.

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

//Test if this is a quit

if (msg.message == WM\_QUIT)

{

break;

}

//Translate any accelerator keys

TranslateMessage(&msg);

//Send the message to the window proc.

DispatchMessage(&msg);

}

//Main game processing goes here.

GameLoop(); //One frame of game logic goes here...

}

//Return to Windows like this...

return(static\_cast<int>(msg.wParam));

}

## How it works...

The main window is created and registered. In the call back function, we use a function called GetAsyncKeyState(VK\_KEYNAME) to detect which key was pressed. After that we perform a bitwise AND operation to check if the last key press was also the same key and whether it is actually pressed. Accordingly we have different Boolean parameters to detect the state of the key press and store them. The code could be structured in a better way but this is the easiest way to understand how to detect key press. To detect the mouse movement co-ordinates, we use a function called GetCursorPos inside WM\_MOUSEMOVE and accordingly get the x and y co-ordinates on screen. Finally we need to display all this information on the screen. To do this we create a rectangle on screen. In that rectangle, we need to use a function called TextOut to display that information. The TextOut function uses the handle to the device context, the x and y co-ordinates, and the message to be displayed.

# Using windows resources with GDI

GDI is the graphics device interface which allow us to do interesting things using bitmaps, icons cursors etc. GDI is used as a rendering alternative if we are not implementing any other rendering alternatives like OpenGL or DirectX.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

In this recipe we will find out how easy it is to use load resources using Windows GDI

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 Windows application
4. Right click on Resource files and add a new cursor.
5. A resource.h file will automatically be created for you
6. Add a source file called Source.cpp
7. Add the following lines of code

Source.cpp

1. #define WIN32\_LEAN\_AND\_MEAN
2. #include <windows.h> // Include all the windows headers.
3. #include <windowsx.h> // Include useful macros.
4. #include "resource.h"
5. #define WINDOW\_CLASS\_NAME L"WINCLASS1"
6. void GameLoop()
7. {
8. //One frame of game logic occurs here...
9. }
10. LRESULT CALLBACK WindowProc(HWND \_hwnd,
11. UINT \_msg,
12. WPARAM \_wparam,
13. LPARAM \_lparam)
14. {
15. // This is the main message handler of the system.
16. PAINTSTRUCT ps; // Used in WM\_PAINT.
17. HDC hdc; // Handle to a device context.
18. // What is the message?
19. switch (\_msg)
20. {
21. case WM\_CREATE:
22. {
23. // Do initialization stuff here.
24. // Return Success.
25. return (0);
26. }
27. break;
28. case WM\_PAINT:
29. {
30. // Simply validate the window.
31. hdc = BeginPaint(\_hwnd, &ps);
32. // You would do all your painting here...
33. EndPaint(\_hwnd, &ps);
34. // Return Success.
35. return (0);
36. }
37. break;
38. case WM\_DESTROY:
39. {
40. // Kill the application, this sends a WM\_QUIT message.
41. PostQuitMessage(0);
42. // Return success.
43. return (0);
44. }
45. break;
46. default:break;
47. } // End switch.
48. // Process any messages that we did not take care of...
49. return (DefWindowProc(\_hwnd, \_msg, \_wparam, \_lparam));
50. }
51. int WINAPI WinMain(HINSTANCE \_hInstance,
52. HINSTANCE \_hPrevInstance,
53. LPSTR \_lpCmdLine,
54. int \_nCmdShow)
55. {
56. WNDCLASSEX winclass; // This will hold the class we create.
57. HWND hwnd; // Generic window handle.
58. MSG msg; // Generic message.
59. HCURSOR hCrosshair = LoadCursor(\_hInstance, MAKEINTRESOURCE(IDC\_CURSOR2));
60. // First fill in the window class structure.
61. winclass.cbSize = sizeof(WNDCLASSEX);
62. winclass.style = CS\_DBLCLKS | CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;
63. winclass.lpfnWndProc = WindowProc;
64. winclass.cbClsExtra = 0;
65. winclass.cbWndExtra = 0;
66. winclass.hInstance = \_hInstance;
67. winclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);
68. winclass.hCursor = LoadCursor(\_hInstance, MAKEINTRESOURCE(IDC\_CURSOR2));
69. winclass.hbrBackground =
70. static\_cast<HBRUSH>(GetStockObject(WHITE\_BRUSH));
71. winclass.lpszMenuName = NULL;
72. winclass.lpszClassName = WINDOW\_CLASS\_NAME;
73. winclass.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);
74. // register the window class
75. if (!RegisterClassEx(&winclass))
76. {
77. return (0);
78. }
79. // create the window
80. hwnd = CreateWindowEx(NULL, // Extended style.
81. WINDOW\_CLASS\_NAME, // Class.
82. L"PacktUp Publishing", // Title.
83. WS\_OVERLAPPEDWINDOW | WS\_VISIBLE,
84. 0, 0, // Initial x,y.
85. 400, 400, // Initial width, height.
86. NULL, // Handle to parent.
87. NULL, // Handle to menu.
88. \_hInstance, // Instance of this application.
89. NULL); // Extra creation parameters.
90. if (!(hwnd))
91. {
92. return (0);
93. }
94. // Enter main event loop
95. while (true)
96. {
97. // Test if there is a message in queue, if so get it.
98. if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))
99. {
100. // Test if this is a quit.
101. if (msg.message == WM\_QUIT)
102. {
103. break;
104. }
105. // Translate any accelerator keys.
106. TranslateMessage(&msg);
107. // Send the message to the window proc.
108. DispatchMessage(&msg);
109. }
110. // Main game processing goes here.
111. GameLoop(); //One frame of game logic occurs here...
112. }
113. // Return to Windows like this...
114. return (static\_cast<int>(msg.wParam));
115. }

## How it works...

Loading up the new cursor is the easiest task to achieve. We need to modify the line winclass.hCursor = LoadCursor(\_hInstance, MAKEINTRESOURCE(IDC\_CURSOR2)).

If we specify null here, the default windows cursor will be loaded. Instead we can load the cursor which we just created. Make sure the reference name of the cursor in resource.h is specified as IDC\_CURSOR2. We can name it to anything but we need to call the appropriate reference from the LoadCursor function. MAKEINTREsource enables us to relate to the resource file from the source code. Similarly we can load multiple cursors and switch them at run-time if we so desire. The same process is for loading other resources like icons and other bitmaps. When we modify a resource file, the corresponding resource.h file must be closed or it will not allow us to edit. Similarly if we want to manually edit the source.h file, we need to close the corresponding .rc or resource file.

# Using dialogs and controls

Dialogs are one of the most mandatory feature in Windows programming. If we are creating a complete application, there will be at some stage when we will require dialogs of some form. Dialogs could be in the form of edit boxes, radio buttons, check boxes, etc. Dialogs are of two forms; modal dialog boxes and modeless. Modal dialog boxes require an immediate response whereas modeless dialog boxes are more of a floating boxes which does not require immediate response.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. In this recipe we will find out how easy it is to create dialog boxes.
2. Open Visual Studio.
   1. Create a new C++ project
   2. Select a win32 windows application
   3. Create a new resource file.
   4. Select dialog as the type of resource.
   5. Edit the box in whatever way we so desire.
   6. A corresponding resource.h file will be created
   7. Add the following files: Source.cpp
   8. Add the following lines of code.

**Source.cpp**

#define WIN32\_LEAN\_AND\_MEAN

#include <windows.h> // Include all the windows headers.

#include <windowsx.h> // Include useful macros.

#include "resource.h"

#define WINDOW\_CLASS\_NAME L"WINCLASS1"

void GameLoop()

{

//One frame of game logic occurs here...

}

BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT msg, WPARAM wparam, LPARAM lparam)

{

switch (msg)

{

case WM\_INITDIALOG:

break;

case WM\_COMMAND:

switch (LOWORD(wparam))

{

case IDOK:

EndDialog(

hDlg, //Handle to the dialog to end.

0); //Return code.

break;

case IDCANCEL:

EndDialog(

hDlg, //Handle to the dialog to end.

0); //Return code.

break;

default:

break;

}

}

return true;

}

LRESULT CALLBACK WindowProc(HWND \_hwnd,

UINT \_msg,

WPARAM \_wparam,

LPARAM \_lparam)

{

// This is the main message handler of the system.

PAINTSTRUCT ps; // Used in WM\_PAINT.

HDC hdc; // Handle to a device context.

// What is the message?

switch (\_msg)

{

case WM\_CREATE:

{

// Do initialization stuff here.

// Return Success.

return (0);

}

break;

case WM\_PAINT:

{

// Simply validate the window.

hdc = BeginPaint(\_hwnd, &ps);

// You would do all your painting here...

EndPaint(\_hwnd, &ps);

// Return Success.

return (0);

}

break;

case WM\_DESTROY:

{

// Kill the application, this sends a WM\_QUIT message.

PostQuitMessage(0);

// Return success.

return (0);

}

break;

default:break;

} // End switch.

// Process any messages that we did not take care of...

return (DefWindowProc(\_hwnd, \_msg, \_wparam, \_lparam));

}

int WINAPI WinMain(HINSTANCE \_hInstance,

HINSTANCE \_hPrevInstance,

LPSTR \_lpCmdLine,

int \_nCmdShow)

{

WNDCLASSEX winclass; // This will hold the class we create.

HWND hwnd; // Generic window handle.

MSG msg; // Generic message.

// First fill in the window class structure.

winclass.cbSize = sizeof(WNDCLASSEX);

winclass.style = CS\_DBLCLKS | CS\_OWNDC | CS\_HREDRAW | CS\_VREDRAW;

winclass.lpfnWndProc = WindowProc;

winclass.cbClsExtra = 0;

winclass.cbWndExtra = 0;

winclass.hInstance = \_hInstance;

winclass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

winclass.hCursor = LoadCursor(NULL, IDC\_ARROW);

winclass.hbrBackground =

static\_cast<HBRUSH>(GetStockObject(BLACK\_BRUSH));

winclass.lpszMenuName = NULL;

winclass.lpszClassName = WINDOW\_CLASS\_NAME;

winclass.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

// register the window class

if (!RegisterClassEx(&winclass))

{

return (0);

}

// create the window

hwnd = CreateWindowEx(NULL, // Extended style.

WINDOW\_CLASS\_NAME, // Class.

L"My first Window", // Title.

WS\_OVERLAPPEDWINDOW | WS\_VISIBLE,

0, 0, // Initial x,y.

1024, 980, // Initial width, height.

NULL, // Handle to parent.

NULL, // Handle to menu.

\_hInstance, // Instance of this application.

NULL); // Extra creation parameters.

if (!(hwnd))

{

return (0);

}

DialogBox(\_hInstance, MAKEINTRESOURCE(IDD\_DIALOG1), hwnd, AboutDlgProc);

// Enter main event loop

while (true)

{

// Test if there is a message in queue, if so get it.

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

// Test if this is a quit.

if (msg.message == WM\_QUIT)

{

break;

}

// Translate any accelerator keys.

TranslateMessage(&msg);

// Send the message to the window proc.

DispatchMessage(&msg);

}

// Main game processing goes here.

GameLoop(); //One frame of game logic occurs here...

}

// Return to Windows like this...

return (static\_cast<int>(msg.wParam));

}

## How it works...

After the resource.h file is automatically created for us, we can manually edit to name the dialog appropriately. After the main window is created, we need to get a handle to the window and then call the dialog box function like this: DialogBox(\_hInstance, MAKEINTRESOURCE(IDD\_DIALOG1), hwnd, AboutDlgProc). Very much like the main window call-back, the dialog box has its own callback. We need to intercept the messages accordingly and perform our actions. BOOL CALLBACK AboutDlgProc is the call-back that we have at our disposal. We have a similar initialise message. For our dialog boxes, most of the intercept will take place in the WM\_COMMAND.Based on the wparam parameter, we need to switch so that know whether we have pressed the OK button or the CANCEL button and accordingly take appropriate steps.

# Using sprites

To develop any 2D game we need sprites. Sprites are elements of computer graphics which can stay on screen, be manipulated with or be animated. GDI allows us to use sprites for creating our game. All the assets in the game will probably be sprites, starting from the UI to the main characters and so on.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. In this recipe we will find out how to use sprites in our game
2. Open Visual Studio.
3. Create a new C++ project
4. Create a new resource type
5. Select the sprite option as the new resource type
6. Add the following source files : backbuffer.h/cpp, Clock.h/cpp, Game.h/.cpp, sprite.h/cpp, Utilities.h
7. Add the following lines of code.
8. **backbuffer.h**

#pragma once

#if !defined(\_\_BACKBUFFER\_H\_\_)

#define \_\_BACKBUFFER\_H\_\_

// Library Includes

#include <Windows.h>

// Local Includes

// Types

// Constants

// Prototypes

class CBackBuffer

{

// Member Functions

public:

CBackBuffer();

~CBackBuffer();

bool Initialise(HWND \_hWnd, int \_iWidth, int \_iHeight);

HDC GetBFDC() const;

int GetHeight() const;

int GetWidth() const;

void Clear();

void Present();

protected:

private:

CBackBuffer(const CBackBuffer& \_kr);

CBackBuffer& operator= (const CBackBuffer& \_kr);

// Member Variables

public:

protected:

HWND m\_hWnd;

HDC m\_hDC;

HBITMAP m\_hSurface;

HBITMAP m\_hOldObject;

int m\_iWidth;

int m\_iHeight;

private:

};

#endif // \_\_BACKBUFFER\_H\_\_

1. **backbuffer.cpp**
2. // Library Includes
3. // Local Includes
4. // This include
5. #include "BackBuffer.h"
6. // Static Variables
7. // Static Function Prototypes
8. // Implementation
9. CBackBuffer::CBackBuffer()
10. : m\_hWnd(0)
11. , m\_hDC(0)
12. , m\_hSurface(0)
13. , m\_hOldObject(0)
14. , m\_iWidth(0)
15. , m\_iHeight(0)
16. {
17. }
18. CBackBuffer::~CBackBuffer()
19. {
20. SelectObject(m\_hDC, m\_hOldObject);
21. DeleteObject(m\_hSurface);
22. DeleteObject(m\_hDC);
23. }
24. bool
25. CBackBuffer::Initialise(HWND \_hWnd, int \_iWidth, int \_iHeight)
26. {
27. m\_hWnd = \_hWnd;
28. m\_iWidth = \_iWidth;
29. m\_iHeight = \_iHeight;
30. HDC hWindowDC = ::GetDC(m\_hWnd);
31. m\_hDC = CreateCompatibleDC(hWindowDC);
32. m\_hSurface = CreateCompatibleBitmap(hWindowDC, m\_iWidth, m\_iHeight);
33. ReleaseDC(m\_hWnd, hWindowDC);
34. m\_hOldObject = static\_cast<HBITMAP>(SelectObject(m\_hDC, m\_hSurface));
35. HBRUSH brushWhite = static\_cast<HBRUSH>(GetStockObject(LTGRAY\_BRUSH));
36. HBRUSH oldBrush = static\_cast<HBRUSH>(SelectObject(m\_hDC, brushWhite));
37. Rectangle(m\_hDC, 0, 0, m\_iWidth, m\_iHeight);
38. SelectObject(m\_hDC, oldBrush);
39. return (true);
40. }
41. void
42. CBackBuffer::Clear()
43. {
44. HBRUSH hOldBrush = static\_cast<HBRUSH>(SelectObject(GetBFDC(), GetStockObject(LTGRAY\_BRUSH)));
45. Rectangle(GetBFDC(), 0, 0, GetWidth(), GetHeight());
46. SelectObject(GetBFDC(), hOldBrush);
47. }
48. HDC
49. CBackBuffer::GetBFDC() const
50. {
51. return (m\_hDC);
52. }
53. int
54. CBackBuffer::GetWidth() const
55. {
56. return (m\_iWidth);
57. }
58. int
59. CBackBuffer::GetHeight() const
60. {
61. return (m\_iHeight);
62. }
63. void
64. CBackBuffer::Present()
65. {
66. HDC hWndDC = ::GetDC(m\_hWnd);
67. BitBlt(hWndDC, 0, 0, m\_iWidth, m\_iHeight, m\_hDC, 0, 0, SRCCOPY);
68. ReleaseDC(m\_hWnd, hWndDC);
69. }
70. **Clock.h**
71. #pragma once
72. #if !defined(\_\_CLOCK\_H\_\_)
73. #define \_\_CLOCK\_H\_\_
74. // Library Includes
75. // Local Includes
76. // Types
77. // Constants
78. // Prototypes
79. class CClock
80. {
81. // Member Functions
82. public:
83. CClock();
84. ~CClock();
85. bool Initialise();
86. void Process();
87. float GetDeltaTick();
88. protected:
89. private:
90. CClock(const CClock& \_kr);
91. CClock& operator= (const CClock& \_kr);
92. // Member Variables
93. public:
94. protected:
95. float m\_fTimeElapsed;
96. float m\_fDeltaTime;
97. float m\_fLastTime;
98. float m\_fCurrentTime;
99. private:
100. };
101. #endif // \_\_CLOCK\_H\_\_
102. **Clock.cpp**
103. // Library Includes
104. #include <windows.h>
105. // Local Includes
106. #include "Clock.h"
107. // Static Variables
108. // Static Function Prototypes
109. // Implementation
110. CClock::CClock()
111. : m\_fTimeElapsed(0.0f)
112. , m\_fDeltaTime(0.0f)
113. , m\_fLastTime(0.0f)
114. , m\_fCurrentTime(0.0f)
115. {
116. }
117. CClock::~CClock()
118. {
119. }
120. bool
121. CClock::Initialise()
122. {
123. return (true);
124. }
125. void
126. CClock::Process()
127. {
128. m\_fLastTime = m\_fCurrentTime;
129. m\_fCurrentTime = static\_cast<float>(timeGetTime());
130. if (m\_fLastTime == 0.0f)
131. {
132. m\_fLastTime = m\_fCurrentTime;
133. }
134. m\_fDeltaTime = m\_fCurrentTime - m\_fLastTime;
135. m\_fTimeElapsed += m\_fDeltaTime;
136. }
137. float
138. CClock::GetDeltaTick()
139. {
140. return (m\_fDeltaTime / 1000.0f);
141. }
142. **Game.h**
143. #pragma once
144. #if !defined(\_\_GAME\_H\_\_)
145. #define \_\_GAME\_H\_\_
146. // Library Includes
147. #include <windows.h>
148. // Local Includes
149. #include "clock.h"
150. // Types
151. // Constants
152. // Prototypes
153. class CBackBuffer;
154. class CGame
155. {
156. // Member Functions
157. public:
158. ~CGame();
159. bool Initialise(HINSTANCE \_hInstance, HWND \_hWnd, int \_iWidth, int \_iHeight);
160. void Draw();
161. void Process(float \_fDeltaTick);
162. void ExecuteOneFrame();
163. CBackBuffer\* GetBackBuffer();
164. HINSTANCE GetAppInstance();
165. HWND GetWindow();
166. // Singleton Methods
167. static CGame& GetInstance();
168. static void DestroyInstance();
169. protected:
170. private:
171. CGame();
172. CGame(const CGame& \_kr);
173. CGame& operator= (const CGame& \_kr);
174. // Member Variables
175. public:
176. protected:
177. CClock\* m\_pClock;
178. CBackBuffer\* m\_pBackBuffer;
179. //Application data
180. HINSTANCE m\_hApplicationInstance;
181. HWND m\_hMainWindow;
182. // Singleton Instance
183. static CGame\* s\_pGame;
184. private:
185. };
186. #endif // \_\_GAME\_H\_\_
187. **Game.cpp**
188. // Library Includes
189. // Local Includes
190. #include "Clock.h"
191. #include "BackBuffer.h"
192. #include "Utilities.h"
193. // This Include
194. #include "Game.h"
195. // Static Variables
196. CGame\* CGame::s\_pGame = 0;
197. // Static Function Prototypes
198. // Implementation
199. CGame::CGame()
200. : m\_pClock(0)
201. , m\_hApplicationInstance(0)
202. , m\_hMainWindow(0)
203. , m\_pBackBuffer(0)
204. {
205. }
206. CGame::~CGame()
207. {
208. delete m\_pBackBuffer;
209. m\_pBackBuffer = 0;
210. delete m\_pClock;
211. m\_pClock = 0;
212. }
213. bool
214. CGame::Initialise(HINSTANCE \_hInstance, HWND \_hWnd, int \_iWidth, int \_iHeight)
215. {
216. m\_hApplicationInstance = \_hInstance;
217. m\_hMainWindow = \_hWnd;
218. m\_pClock = new CClock();
219. VALIDATE(m\_pClock->Initialise());
220. m\_pClock->Process();
221. m\_pBackBuffer = new CBackBuffer();
222. VALIDATE(m\_pBackBuffer->Initialise(\_hWnd, \_iWidth, \_iHeight));
223. ShowCursor(false);
224. return (true);
225. }
226. void
227. CGame::Draw()
228. {
229. m\_pBackBuffer->Clear();
230. // Do all the game’s drawing here...
231. m\_pBackBuffer->Present();
232. }
233. void
234. CGame::Process(float \_fDeltaTick)
235. {
236. // Process all the game’s logic here.
237. }
238. void
239. CGame::ExecuteOneFrame()
240. {
241. float fDT = m\_pClock->GetDeltaTick();
242. Process(fDT);
243. Draw();
244. m\_pClock->Process();
245. Sleep(1);
246. }
247. CGame&
248. CGame::GetInstance()
249. {
250. if (s\_pGame == 0)
251. {
252. s\_pGame = new CGame();
253. }
254. return (\*s\_pGame);
255. }
256. void
257. CGame::DestroyInstance()
258. {
259. delete s\_pGame;
260. s\_pGame = 0;
261. }
262. CBackBuffer\*
263. CGame::GetBackBuffer()
264. {
265. return (m\_pBackBuffer);
266. }
267. HINSTANCE
268. CGame::GetAppInstance()
269. {
270. return (m\_hApplicationInstance);
271. }
272. HWND
273. CGame::GetWindow()
274. {
275. return (m\_hMainWindow);
276. }
277. **sprite.h**
278. #pragma once
279. #if !defined(\_\_SPRITE\_H\_\_)
280. #define \_\_SPRITE\_H\_\_
281. // Library Includes
282. #include "windows.h"
283. // Local Includes
284. // Types
285. // Constants
286. // Prototypes
287. class CSprite
288. {
289. // Member Functions
290. public:
291. CSprite();
292. ~CSprite();
293. bool Initialise(int \_iResourceID, int \_iMaskResourceID);
294. void Draw();
295. void Process(float \_fDeltaTick);
296. int GetWidth() const;
297. int GetHeight() const;
298. int GetX() const;
299. int GetY() const;
300. void SetX(int \_i);
301. void SetY(int \_i);
302. void TranslateRelative(int \_iX, int \_iY);
303. void TranslateAbsolute(int \_iX, int \_iY);
304. protected:
305. private:
306. CSprite(const CSprite& \_kr);
307. CSprite& operator= (const CSprite& \_kr);
308. // Member Variables
309. public:
310. protected:
311. //Center handle
312. int m\_iX;
313. int m\_iY;
314. HBITMAP m\_hSprite;
315. HBITMAP m\_hMask;
316. BITMAP m\_bitmapSprite;
317. BITMAP m\_bitmapMask;
318. static HDC s\_hSharedSpriteDC;
319. static int s\_iRefCount;
320. private:
321. };
322. #endif // \_\_SPRITE\_H\_\_
323. **sprite.cpp**
324. // Library Includes
325. // Local Includes
326. #include "resource.h"
327. #include "Game.h"
328. #include "BackBuffer.h"
329. #include "Utilities.h"
330. // This include
331. #include "Sprite.h"
332. // Static Variables
333. HDC CSprite::s\_hSharedSpriteDC = 0;
334. int CSprite::s\_iRefCount = 0;
335. // Static Function Prototypes
336. // Implementation
337. CSprite::CSprite()
338. : m\_iX(0)
339. , m\_iY(0)
340. {
341. ++s\_iRefCount;
342. }
343. CSprite::~CSprite()
344. {
345. DeleteObject(m\_hSprite);
346. DeleteObject(m\_hMask);
347. --s\_iRefCount;
348. if (s\_iRefCount == 0)
349. {
350. DeleteDC(s\_hSharedSpriteDC);
351. s\_hSharedSpriteDC = 0;
352. }
353. }
354. bool
355. CSprite::Initialise(int \_iSpriteResourceID, int \_iMaskResourceID)
356. {
357. HINSTANCE hInstance = CGame::GetInstance().GetAppInstance();
358. if (!s\_hSharedSpriteDC)
359. {
360. s\_hSharedSpriteDC = CreateCompatibleDC(NULL);
361. }
362. m\_hSprite = LoadBitmap(hInstance, MAKEINTRESOURCE(\_iSpriteResourceID));
363. VALIDATE(m\_hSprite);
364. m\_hMask = LoadBitmap(hInstance, MAKEINTRESOURCE(\_iMaskResourceID));
365. VALIDATE(m\_hMask);
366. GetObject(m\_hSprite, sizeof(BITMAP), &m\_bitmapSprite);
367. GetObject(m\_hMask, sizeof(BITMAP), &m\_bitmapMask);
368. return (true);
369. }
370. void
371. CSprite::Draw()
372. {
373. int iW = GetWidth();
374. int iH = GetHeight();
375. int iX = m\_iX - (iW / 2);
376. int iY = m\_iY - (iH / 2);
377. CBackBuffer\* pBackBuffer = CGame::GetInstance().GetBackBuffer();
378. HGDIOBJ hOldObj = SelectObject(s\_hSharedSpriteDC, m\_hMask);
379. BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s\_hSharedSpriteDC, 0, 0, SRCAND);
380. SelectObject(s\_hSharedSpriteDC, m\_hSprite);
381. BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s\_hSharedSpriteDC, 0, 0, SRCPAINT);
382. SelectObject(s\_hSharedSpriteDC, hOldObj);
383. }
384. void
385. CSprite::Process(float \_fDeltaTick)
386. {
387. }
388. int
389. CSprite::GetWidth() const
390. {
391. return (m\_bitmapSprite.bmWidth);
392. }
393. int
394. CSprite::GetHeight() const
395. {
396. return (m\_bitmapSprite.bmHeight);
397. }
398. int
399. CSprite::GetX() const
400. {
401. return (m\_iX);
402. }
403. int
404. CSprite::GetY() const
405. {
406. return (m\_iY);
407. }
408. void
409. CSprite::SetX(int \_i)
410. {
411. m\_iX = \_i;
412. }
413. void
414. CSprite::SetY(int \_i)
415. {
416. m\_iY = \_i;
417. }
418. void
419. CSprite::TranslateRelative(int \_iX, int \_iY)
420. {
421. m\_iX += \_iX;
422. m\_iY += \_iY;
423. }
424. void
425. CSprite::TranslateAbsolute(int \_iX, int \_iY)
426. {
427. m\_iX = \_iX;
428. m\_iY = \_iY;
429. }
430. **Utilities.h**
431. // Library Includes
432. // Local Includes
433. #include "resource.h"
434. #include "Game.h"
435. #include "BackBuffer.h"
436. #include "Utilities.h"
437. // This include
438. #include "Sprite.h"
439. // Static Variables
440. HDC CSprite::s\_hSharedSpriteDC = 0;
441. int CSprite::s\_iRefCount = 0;
442. // Static Function Prototypes
443. // Implementation
444. CSprite::CSprite()
445. : m\_iX(0)
446. , m\_iY(0)
447. {
448. ++s\_iRefCount;
449. }
450. CSprite::~CSprite()
451. {
452. DeleteObject(m\_hSprite);
453. DeleteObject(m\_hMask);
454. --s\_iRefCount;
455. if (s\_iRefCount == 0)
456. {
457. DeleteDC(s\_hSharedSpriteDC);
458. s\_hSharedSpriteDC = 0;
459. }
460. }
461. bool
462. CSprite::Initialise(int \_iSpriteResourceID, int \_iMaskResourceID)
463. {
464. HINSTANCE hInstance = CGame::GetInstance().GetAppInstance();
465. if (!s\_hSharedSpriteDC)
466. {
467. s\_hSharedSpriteDC = CreateCompatibleDC(NULL);
468. }
469. m\_hSprite = LoadBitmap(hInstance, MAKEINTRESOURCE(\_iSpriteResourceID));
470. VALIDATE(m\_hSprite);
471. m\_hMask = LoadBitmap(hInstance, MAKEINTRESOURCE(\_iMaskResourceID));
472. VALIDATE(m\_hMask);
473. GetObject(m\_hSprite, sizeof(BITMAP), &m\_bitmapSprite);
474. GetObject(m\_hMask, sizeof(BITMAP), &m\_bitmapMask);
475. return (true);
476. }
477. void
478. CSprite::Draw()
479. {
480. int iW = GetWidth();
481. int iH = GetHeight();
482. int iX = m\_iX - (iW / 2);
483. int iY = m\_iY - (iH / 2);
484. CBackBuffer\* pBackBuffer = CGame::GetInstance().GetBackBuffer();
485. HGDIOBJ hOldObj = SelectObject(s\_hSharedSpriteDC, m\_hMask);
486. BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s\_hSharedSpriteDC, 0, 0, SRCAND);
487. SelectObject(s\_hSharedSpriteDC, m\_hSprite);
488. BitBlt(pBackBuffer->GetBFDC(), iX, iY, iW, iH, s\_hSharedSpriteDC, 0, 0, SRCPAINT);
489. SelectObject(s\_hSharedSpriteDC, hOldObj);
490. }
491. void
492. CSprite::Process(float \_fDeltaTick)
493. {
494. }
495. int
496. CSprite::GetWidth() const
497. {
498. return (m\_bitmapSprite.bmWidth);
499. }
500. int
501. CSprite::GetHeight() const
502. {
503. return (m\_bitmapSprite.bmHeight);
504. }
505. int
506. CSprite::GetX() const
507. {
508. return (m\_iX);
509. }
510. int
511. CSprite::GetY() const
512. {
513. return (m\_iY);
514. }
515. void
516. CSprite::SetX(int \_i)
517. {
518. m\_iX = \_i;
519. }
520. void
521. CSprite::SetY(int \_i)
522. {
523. m\_iY = \_i;
524. }
525. void
526. CSprite::TranslateRelative(int \_iX, int \_iY)
527. {
528. m\_iX += \_iX;
529. m\_iY += \_iY;
530. }
531. void
532. CSprite::TranslateAbsolute(int \_iX, int \_iY)
533. {
534. m\_iX = \_iX;
535. m\_iY = \_iY;
536. }

## How it works...

As we know the back-buffer is used to draw the image first and then we swap the buffer to present it to the screen. This process is also called “presenting”. We create a generic backbuffer class which help us to swap the buffer. The sprite class is used to load the sprite and push them on to back buffer where it can be processed and finally drawn to the screen. The sprite class is also provided with some basic utility function which help us to get the width and height of the sprite. Most of the functions are just a wrapper on top of the Windows own API functions and call backs. We have also created a clock class which help us to keep a track of time as every time should be implemented as a function of delta time. If we do not do that, then the game might run with varying behaviour based on the machine that is executing it. The game class is used to put all of it together. It has an instance of the backbuffer which is a singleton class and handles and context of the window and other resources.

# Using animated sprites

Using animated sprites are an important part of games programming. Unless there is some kind of animation applied to the sprites, it will not appear realistic enough and the whole mood of immersion in the game will be lost. Although animations can be achieved in a variety of ways, we will only look at sprite strip animation as it is the most used animation form for 2D games.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. In this recipe we will find out how easy it is to create dialog boxes.
2. Open Visual Studio.
   1. Create a new C++ project
   2. Select a win32 windows application
   3. Add the following files: AnimatedSprite.cpp
   4. Add the following lines of code.

**Source.cpp**

// This include

#include "AnimatedSprite.h"

// Static Variables

// Static Function Prototypes

// Implementation

CAnimatedSprite::CAnimatedSprite()

: m\_fFrameSpeed(0.0f)

, m\_fTimeElapsed(0.0f)

, m\_iCurrentSprite(0)

{

}

CAnimatedSprite::~CAnimatedSprite()

{

Deinitialise();

}

bool

CAnimatedSprite::Deinitialise()

{

return (CSprite::Deinitialise());

}

bool

CAnimatedSprite::Initialise(int \_iSpriteResourceID, int \_iMaskResourceID)

{

return (CSprite::Initialise(\_iSpriteResourceID, \_iMaskResourceID));

}

void

CAnimatedSprite::Draw()

{

int iTopLeftX = m\_vectorFrames[m\_iCurrentSprite];

int iTopLeftY = 0;

int iW = GetFrameWidth();

int iH = GetHeight();

int iX = m\_iX - (iW / 2);

int iY = m\_iY - (iH / 2);

HDC hSpriteDC = hSharedSpriteDC;

HGDIOBJ hOldObj = SelectObject(hSpriteDC, m\_hMask);

BitBlt(CGame::GetInstance().GetBackBuffer()->GetBFDC(), iX, iY, iW, iH, hSpriteDC, iTopLeftX, iTopLeftY, SRCAND);

SelectObject(hSpriteDC, m\_hSprite);

BitBlt(CGame::GetInstance().GetBackBuffer()->GetBFDC(), iX, iY, iW, iH, hSpriteDC, iTopLeftX, iTopLeftY, SRCPAINT);

SelectObject(hSpriteDC, hOldObj);

}

void

CAnimatedSprite::Process(float \_fDeltaTick)

{

m\_fTimeElapsed += \_fDeltaTick;

if (m\_fTimeElapsed >= m\_fFrameSpeed &&

m\_fFrameSpeed != 0.0f)

{

m\_fTimeElapsed = 0.0f;

++m\_iCurrentSprite;

if (m\_iCurrentSprite >= m\_vectorFrames.size())

{

m\_iCurrentSprite = 0;

}

}

CSprite::Process(\_fDeltaTick);

}

void

CAnimatedSprite::AddFrame(int \_iX)

{

m\_vectorFrames.push\_back(\_iX);

}

void

CAnimatedSprite::SetSpeed(float \_fSpeed)

{

m\_fFrameSpeed = \_fSpeed;

}

void

CAnimatedSprite::SetWidth(int \_iW)

{

m\_iFrameWidth = \_iW;

}

int

CAnimatedSprite::GetFrameWidth()

{

return (m\_iFrameWidth);

}

## How it works...

For the animation to work, we need to load in a sequence of images as sprite strips. The more the number of images, the smoother the animation will be. For the equivalent number of sprites, we need to load in their masks as well so that they can be blitted together. We need to store all the images in a vector list. For the animation to work properly, all the images must be equally spaced out. After we have stored them correctly, we can run the animations as fast or slow as we want, by controlling how many frames/sprites we want to draw in a certain duration of time. The remaining process of drawing the sprite to the screen remains same.