Navigation

Home
DirectX for Windows 8
  DirectX 11.2
  DirectX 11.1
DirectX for Desktop
  DirectX 11
  DirectX 9
Useful Resources
About DirectXTutorial
DirectXTutorial Premium
Testimonials

Contact

Contact me here:
Twitter: @dastopher
Email: chris@directxtutorial.com

Or leave me feedback:
Quick Feedback

Lesson 5: The Real-Time Message Loop

Previous Next

Lesson Overview

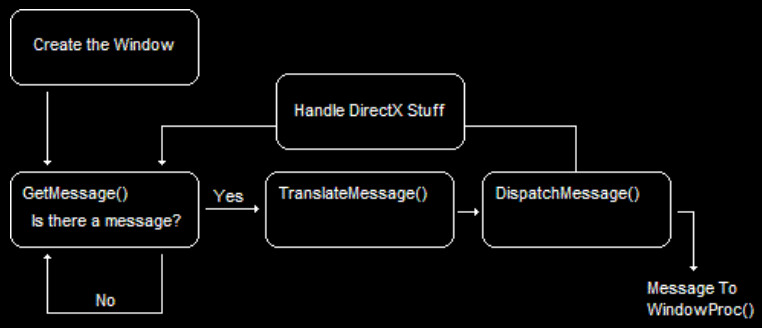
In this lesson we will cover a single function, PeekMessage(), and how this function differs from its evil twin, GetMessage().

Actually, there's nothing wrong with GetMessage(), the way it works just doesn't have spectacular results on games and their continuous activity. We'll go over how this is and how PeekMessage() is a solution.

The Structure of the GetMessage() Loop

In the previous lesson, we built a simple Windows application while using the function GetMessage(). We used GetMessage() and two other functions to create a loop that handled all the Windows message sent. However, there was a catch we didn't talk about at the time.

The following diagram shows how the event loop we wrote works:



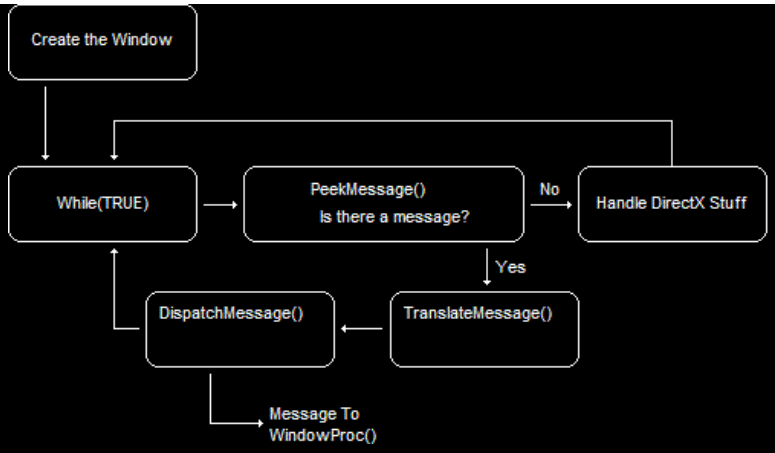
The Structure of a GetMessage() Loop

Once we create the window, we get into the event loop, where we see the function GetMessage(). GetMessage() then waits for a message and, upon recieving one, sends it to the next step, TranslateMessage(). This is perfectly logical for Windows programming, because generally speaking Windows applications, Word for example, tend to sit and do nothing until you make a move.

However, this doesn't work well for us. While all this waiting is going on, we need to be creating thirty to sixty fully-rendered 3D images per second and putting them on the screen without any delay at all. And so we are presented with a rather interesting problem, because Windows, if it sends any messages, will most definitely not be sending thirty of them per second.

A New Function, PeekMessage()

What we will do to solve this dilema is replace our current GetMessage() function with a new function, PeekMessage(). This function does essentially the same thing, but with one important difference: it doesn't wait for anything. PeekMessage() just looks into the message queue and checks to see if any messages are waiting. If not, the program will continue on, allowing us to do what we need.



The Structure of a PeekMessage() Loop

Before we go any further, let's take a good look at PeekMessage(). Here is it's prototype.

```
BOOL PeekMessage(LPMSG lpMsg,
                 HWND hWnd,
                 UINT wMsgFilterMin,
                 UINT wMsgFilterMax,
                 UINT wRemoveMsg);
```

The first four parameters should be familiar to you. They are identical to the four parameters of GetMessage(). However, the fifth one, wRemoveMsg, is new.

What it does is indicate whether or not the message retrieved from the event queue should stay on the event queue or come off. We can put either PM_REMOVE or PM_NOREMOVE. The first one takes the messages off the queue when they are read, while the second one leaves the messages there for later retrieval. We will use the PM_REMOVE value here, and keep things simple.

So how do we implement this into our program? Following is the main loop from the last program we made, modified to use PeekMessage().

```
// enter the main loop:

// this struct holds Windows event messages
MSG msg = {0};

// Enter the infinite message loop
while(TRUE)
{
    // Check to see if any messages are waiting in the queue
    if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // translate keystroke messages into the right format
        TranslateMessage(&msg);

        // send the message to the WindowProc function
        DispatchMessage(&msg);

        // check to see if it's time to quit
        if(msg.message == WM_QUIT)
            break;
    }
    else
    {
        // Run game code here
        // ...
        // ...
    }
}
```

Now our program can handle things as timely as we please, without having to worry about Windows and its tedious messages. Let's quickly go over the changes we made.

while(TRUE)

Naturally, this creates an infinite loop. We will escape out of it later when it comes time to quit the game.

if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))

Because we are no longer waiting for a message, but rather just peeking in to see what's there, we will use PeekMessage(). PeekMessage() returns TRUE if there is a message and FALSE if

there isn't. So, if there is a message, it runs TranslateMessage() and DispatchMessage(), and just continues on to the game code if there isn't.

if(msg.message == WM_QUIT)

If our message turns out to be a WM_QUIT, that means its time to exit our "infinite" loop and return to Windows. Remember, we didn't have this before because GetMessage() would always return '0' if it was time to quit, thus breaking the while() loop. We don't have that mechanism here unfortunately, and we have to look for it ourselves.

Running the New Loop

Here is our new code, after we've modified the program with PeekMessage(). The parts that changed are now in **bold**.

[Main.cpp]

Just like in the previous lesson, running this program appears no different than it did before. It only has an effect on how often you can make *changes* to your window.

I'd say we're doing a fairly good job. You now have under your belt all the Windows programming tools you will need for basic game programming. With what you know, you can create a window, prepare it for a game, and have everything ready for the DirectX code.

So, now the real fun begins! Let's dive into DirectX and build ourselves a 3D game!

Next Lesson: Understanding Graphics Concepts

GO! GO! GO!

© 2006-2020 DirectXTutorial.com. All Rights Reserved. Expand