Using the DisplayInformation class from a desktop Win32 application, part 2

devblogs.microsoft.com/oldnewthing/20240321-00

March 21, 2024

Raymond Chen

Last time, we tried to create a `DisplayInformation` from an `HWND`, but it failed with the message “A DispatcherQueue is required for DisplayInformation created from an HWND.”

So I guess we need to make sure the thread has a `DispatcherQueue`.

```
#include <DispatcherQueue.h>
#include <windows.graphics.display.interop.h>
#include <winrt/Windows.Graphics.Display.h>
#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.System.h>

namespace winrt
{
    using namespace winrt::Windows::Graphics::Display;
    using namespace winrt::Windows::System;
}

namespace ABI
{
    using namespace ABI::Windows::System;
}
winrt::DispatcherQueueController g_controller{ nullptr };
winrt::DisplayInformation g_info{ nullptr };

OnCreate(HWND hwnd, LPCREATESTRUCT lpcs) noexcept
{
    DispatcherQueueOptions options{
        sizeof(options), DQTYPE_THREAD_CURRENT, DQTAT_COM_NONE };
    winrt::check_hresult(
        CreateDispatcherQueueController(options,
            reinterpret_cast<ABI::IDispatcherQueueController**>(
                controller.put())));

    g_info = wil::capture_interop<winrt::DisplayInformation>
        (&IDisplayInformationStaticsInterop::GetForWindow, hwnd);

    return TRUE;
}
catch (...)
{
    return FALSE;
}

winrt::fire_and_forget
OnDestroy(HWND hwnd)
{
    g_info = nullptr;
    co_await g_controller.ShutdownQueueAsync();
    PostQuitMessage(0);
}
```

We use `CreateDispatcherQueueController` to attach a dispatcher queue to our existing message pump. There is a bit of a hassle with the second parameter because we are mixing the C++/WinRT and Win32 ABI versions of the `IDispatcherQueueController` interfaces.

(Previous discussion.) And since we created the dispatcher queue controller, we also have to shut down the dispatcher queue when we're done, which we take care of in `OnDestroy()`.

Hooray, this code now successfully creates a `DispatcherQueueController` that manages a `DispatcherQueue` on the current thread, and that removes the obstacle that was preventing `GetForWindow` from succeeding.

Now we can hook up the event and start reading the orientation.

```
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs) noexcept
{
    DispatcherQueueOptions options{
        sizeof(options), DQTYPE_THREAD_CURRENT, DQTAT_COM_NONE };
    winrt::check_hresult(
        CreateDispatcherQueueController(options,
            reinterpret_cast<ABI::IDispatcherQueueController**>(
                controller.put())));

    g_info.OrientationChanged([hwnd](auto&&, auto&&)
        {
            InvalidateRect(hwnd, nullptr, TRUE);
        });

    g_info = wil::capture_interop<winrt::DisplayInformation>
        (&IDisplayInformationStaticsInterop::GetForWindow, hwnd);

    return TRUE;
}
catch (...)
{
    return FALSE;
}

void
PaintContent(HWND hwnd, PAINTSTRUCT* pps) noexcept
{
    PCWSTR message;
    switch (g_info.CurrentOrientation()) {
    case winrt::DisplayOrientations::Landscape:
        message = L"Landscape"; break;
    case winrt::DisplayOrientations::Portrait:
        message = L"Portrait"; break;
    case winrt::DisplayOrientations::LandscapeFlipped:
        message = L"LandscapeFlipped"; break;
    case winrt::DisplayOrientations::PortraitFlipped:
        message = L"PortraitFlipped"; break;
    default:
        message = L"Unknown"; break;
    }
    TextOut(pps->hdc, 0, 0, message,
        static_cast<int>(wcslen(message)));
}
```

After creating the `DisplayInformation`, we register for the orientation change event and force a repaint when that happens.

Our paint handler simply prints the current orientation of the monitor that the window is on.

I won't bother demonstrating it here, but you can also use `GetForMonitor` to get a `DisplayInformation` for a specific monitor.