#include "win32\_window.h"

#include <dwmapi.h>

#include <flutter\_windows.h>

#include "resource.h"

namespace {

/// Window attribute that enables dark mode window decorations.

///

/// Redefined in case the developer's machine has a Windows SDK older than

/// version 10.0.22000.0.

/// See: https://docs.microsoft.com/windows/win32/api/dwmapi/ne-dwmapi-dwmwindowattribute

#ifndef DWMWA\_USE\_IMMERSIVE\_DARK\_MODE

#define DWMWA\_USE\_IMMERSIVE\_DARK\_MODE 20

#endif

constexpr const wchar\_t kWindowClassName[] = L"FLUTTER\_RUNNER\_WIN32\_WINDOW";

/// Registry key for app theme preference.

///

/// A value of 0 indicates apps should use dark mode. A non-zero or missing

/// value indicates apps should use light mode.

constexpr const wchar\_t kGetPreferredBrightnessRegKey[] =

L"Software\\Microsoft\\Windows\\CurrentVersion\\Themes\\Personalize";

constexpr const wchar\_t kGetPreferredBrightnessRegValue[] = L"AppsUseLightTheme";

// The number of Win32Window objects that currently exist.

static int g\_active\_window\_count = 0;

using EnableNonClientDpiScaling = BOOL \_\_stdcall(HWND hwnd);

// Scale helper to convert logical scaler values to physical using passed in

// scale factor

int Scale(int source, double scale\_factor) {

return static\_cast<int>(source \* scale\_factor);

}

// Dynamically loads the |EnableNonClientDpiScaling| from the User32 module.

// This API is only needed for PerMonitor V1 awareness mode.

void EnableFullDpiSupportIfAvailable(HWND hwnd) {

HMODULE user32\_module = LoadLibraryA("User32.dll");

if (!user32\_module) {

return;

}

auto enable\_non\_client\_dpi\_scaling =

reinterpret\_cast<EnableNonClientDpiScaling\*>(

GetProcAddress(user32\_module, "EnableNonClientDpiScaling"));

if (enable\_non\_client\_dpi\_scaling != nullptr) {

enable\_non\_client\_dpi\_scaling(hwnd);

}

FreeLibrary(user32\_module);

}

} // namespace

// Manages the Win32Window's window class registration.

class WindowClassRegistrar {

public:

~WindowClassRegistrar() = default;

// Returns the singleton registrar instance.

static WindowClassRegistrar\* GetInstance() {

if (!instance\_) {

instance\_ = new WindowClassRegistrar();

}

return instance\_;

}

// Returns the name of the window class, registering the class if it hasn't

// previously been registered.

const wchar\_t\* GetWindowClass();

// Unregisters the window class. Should only be called if there are no

// instances of the window.

void UnregisterWindowClass();

private:

WindowClassRegistrar() = default;

static WindowClassRegistrar\* instance\_;

bool class\_registered\_ = false;

};

WindowClassRegistrar\* WindowClassRegistrar::instance\_ = nullptr;

const wchar\_t\* WindowClassRegistrar::GetWindowClass() {

if (!class\_registered\_) {

WNDCLASS window\_class{};

window\_class.hCursor = LoadCursor(nullptr, IDC\_ARROW);

window\_class.lpszClassName = kWindowClassName;

window\_class.style = CS\_HREDRAW | CS\_VREDRAW;

window\_class.cbClsExtra = 0;

window\_class.cbWndExtra = 0;

window\_class.hInstance = GetModuleHandle(nullptr);

window\_class.hIcon =

LoadIcon(window\_class.hInstance, MAKEINTRESOURCE(IDI\_APP\_ICON));

window\_class.hbrBackground = 0;

window\_class.lpszMenuName = nullptr;

window\_class.lpfnWndProc = Win32Window::WndProc;

RegisterClass(&window\_class);

class\_registered\_ = true;

}

return kWindowClassName;

}

void WindowClassRegistrar::UnregisterWindowClass() {

UnregisterClass(kWindowClassName, nullptr);

class\_registered\_ = false;

}

Win32Window::Win32Window() {

++g\_active\_window\_count;

}

Win32Window::~Win32Window() {

--g\_active\_window\_count;

Destroy();

}

bool Win32Window::Create(const std::wstring& title,

const Point& origin,

const Size& size) {

Destroy();

const wchar\_t\* window\_class =

WindowClassRegistrar::GetInstance()->GetWindowClass();

const POINT target\_point = {static\_cast<LONG>(origin.x),

static\_cast<LONG>(origin.y)};

HMONITOR monitor = MonitorFromPoint(target\_point, MONITOR\_DEFAULTTONEAREST);

UINT dpi = FlutterDesktopGetDpiForMonitor(monitor);

double scale\_factor = dpi / 96.0;

HWND window = CreateWindow(

window\_class, title.c\_str(), WS\_OVERLAPPEDWINDOW,

Scale(origin.x, scale\_factor), Scale(origin.y, scale\_factor),

Scale(size.width, scale\_factor), Scale(size.height, scale\_factor),

nullptr, nullptr, GetModuleHandle(nullptr), this);

if (!window) {

return false;

}

UpdateTheme(window);

return OnCreate();

}

bool Win32Window::Show() {

return ShowWindow(window\_handle\_, SW\_SHOWNORMAL);

}

// static

LRESULT CALLBACK Win32Window::WndProc(HWND const window,

UINT const message,

WPARAM const wparam,

LPARAM const lparam) noexcept {

if (message == WM\_NCCREATE) {

auto window\_struct = reinterpret\_cast<CREATESTRUCT\*>(lparam);

SetWindowLongPtr(window, GWLP\_USERDATA,

reinterpret\_cast<LONG\_PTR>(window\_struct->lpCreateParams));

auto that = static\_cast<Win32Window\*>(window\_struct->lpCreateParams);

EnableFullDpiSupportIfAvailable(window);

that->window\_handle\_ = window;

} else if (Win32Window\* that = GetThisFromHandle(window)) {

return that->MessageHandler(window, message, wparam, lparam);

}

return DefWindowProc(window, message, wparam, lparam);

}

LRESULT

Win32Window::MessageHandler(HWND hwnd,

UINT const message,

WPARAM const wparam,

LPARAM const lparam) noexcept {

switch (message) {

case WM\_DESTROY:

window\_handle\_ = nullptr;

Destroy();

if (quit\_on\_close\_) {

PostQuitMessage(0);

}

return 0;

case WM\_DPICHANGED: {

auto newRectSize = reinterpret\_cast<RECT\*>(lparam);

LONG newWidth = newRectSize->right - newRectSize->left;

LONG newHeight = newRectSize->bottom - newRectSize->top;

SetWindowPos(hwnd, nullptr, newRectSize->left, newRectSize->top, newWidth,

newHeight, SWP\_NOZORDER | SWP\_NOACTIVATE);

return 0;

}

case WM\_SIZE: {

RECT rect = GetClientArea();

if (child\_content\_ != nullptr) {

// Size and position the child window.

MoveWindow(child\_content\_, rect.left, rect.top, rect.right - rect.left,

rect.bottom - rect.top, TRUE);

}

return 0;

}

case WM\_ACTIVATE:

if (child\_content\_ != nullptr) {

SetFocus(child\_content\_);

}

return 0;

case WM\_DWMCOLORIZATIONCOLORCHANGED:

UpdateTheme(hwnd);

return 0;

}

return DefWindowProc(window\_handle\_, message, wparam, lparam);

}

void Win32Window::Destroy() {

OnDestroy();

if (window\_handle\_) {

DestroyWindow(window\_handle\_);

window\_handle\_ = nullptr;

}

if (g\_active\_window\_count == 0) {

WindowClassRegistrar::GetInstance()->UnregisterWindowClass();

}

}

Win32Window\* Win32Window::GetThisFromHandle(HWND const window) noexcept {

return reinterpret\_cast<Win32Window\*>(

GetWindowLongPtr(window, GWLP\_USERDATA));

}

void Win32Window::SetChildContent(HWND content) {

child\_content\_ = content;

SetParent(content, window\_handle\_);

RECT frame = GetClientArea();

MoveWindow(content, frame.left, frame.top, frame.right - frame.left,

frame.bottom - frame.top, true);

SetFocus(child\_content\_);

}

RECT Win32Window::GetClientArea() {

RECT frame;

GetClientRect(window\_handle\_, &frame);

return frame;

}

HWND Win32Window::GetHandle() {

return window\_handle\_;

}

void Win32Window::SetQuitOnClose(bool quit\_on\_close) {

quit\_on\_close\_ = quit\_on\_close;

}

bool Win32Window::OnCreate() {

// No-op; provided for subclasses.

return true;

}

void Win32Window::OnDestroy() {

// No-op; provided for subclasses.

}

void Win32Window::UpdateTheme(HWND const window) {

DWORD light\_mode;

DWORD light\_mode\_size = sizeof(light\_mode);

LSTATUS result = RegGetValue(HKEY\_CURRENT\_USER, kGetPreferredBrightnessRegKey,

kGetPreferredBrightnessRegValue,

RRF\_RT\_REG\_DWORD, nullptr, &light\_mode,

&light\_mode\_size);

if (result == ERROR\_SUCCESS) {

BOOL enable\_dark\_mode = light\_mode == 0;

DwmSetWindowAttribute(window, DWMWA\_USE\_IMMERSIVE\_DARK\_MODE,

&enable\_dark\_mode, sizeof(enable\_dark\_mode));

}

}