버전 1.0

2022.10.23



※현재 포토폴리오에서는 ‘코드를 작성한 이유’와 ‘설계’에 관해서 중점적으로 작성하였습니다.

작성자: 김경주

## WInAPI(윈도우 창)

### WinAPI 클래스를 main.cpp에서 분리한 이유:

→ 코드의 **재사용성**을 위해. 멀티 플랫폼이 대세이고, 혹시 WinAPI를 제외한 다른 플랫폼을 이용하고 싶을 수도 있기 때문에 코드를 분리하여서 사용합니다.

### 힘들었던 점:

졸업과제로 당시 directx9으로 엔진 만들 당시, 이런 구조의 코드를 작성하고 싶어서 mfc 코드를 뜯어서 적용시켜 봤으나 링크 에러로 고민을 많이 했습니다.

### 에러 및 해결방법:

WindowProc 링크 에러  
→ win 프로시저 함수에 static을 적용하여서 this-> 라는 포인트를 제거함으로 해결  
 lnk2019 error  
→ (속성/링커/시스템) 하위 시스템을 콘솔에서 창으로 변경합니다

### 코드 및 설계

#pragma once //미리컴파일된 해더

#ifndef STDAFX\_H // STDAFX\_H가 정의 안되어 있다면

#define STDAFX\_H // STDAFX\_H를 정의

#endif //가정 끝(pragma once)와 비슷한 효과 헤더파일이 중복되어 읽어도 괜찮도록 만들어 줌

#include <windows.h> //winAPI

#include <string> // string변수

stdafx.h(c++)

#pragma once

#include "DirectX12Base.h"

class WinAPI

{

public:

WinAPI(); //생성자

~WinAPI(); //소멸자

static bool Init( HINSTANCE hInstance, int nCmdShow); //초기화

static int Run(); //실행

static HWND GetHwnd(); //window 핸들 얻기

protected:

static LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam); //윈도우 프로시져

private:

static HWND WinAPI\_hwnd; //윈도우 핸들

};

WinAPI.h(c++)

#include "stdafx.h"

#include "WinAPI.h"

HWND WinAPI::WinAPI\_hwnd = nullptr; //핸들 값 초기화

WinAPI::WinAPI() //생성자

{

}

WinAPI::~WinAPI() //소멸자

}

bool WinAPI::Init( HINSTANCE hInstance, int nCmdShow) //초기값

{

//명령줄 매게변수 구분

int argc;

LPWSTR\* argv = CommandLineToArgvW(GetCommandLineW(), &argc);

LocalFree(argv);

// 윈도우 클라스 초기화

WNDCLASSEX windowClass = { 0 };

windowClass.cbSize = sizeof(WNDCLASSEX);

windowClass.style = CS\_HREDRAW | CS\_VREDRAW;

windowClass.lpfnWndProc = WindowProc;

windowClass.hInstance = hInstance;

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

windowClass.lpszClassName = L"DirectX12MiniEngine";

RegisterClassEx(&windowClass);

RECT windowRect = { 0, 0, 1200, 900 }; //윈도우 창범위

AdjustWindowRect(&windowRect, WS\_OVERLAPPEDWINDOW, FALSE);

// 창과 핸들 만듦

WinAPI\_hwnd = CreateWindow(

windowClass.lpszClassName,

L"DirectX12MiniEngine",

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT,

CW\_USEDEFAULT,

windowRect.right - windowRect.left,

windowRect.bottom - windowRect.top,

nullptr, // 부모창 없음

nullptr, // 메뉴 사용하지 않음

hInstance,

nullptr);

ShowWindow(WinAPI\_hwnd, nCmdShow); //윈도우 보여주기

return true;

}

WinAPI.cpp(c++)

//WinAPI실행

int WinAPI::Run( )

{

// 메인 루프

MSG msg = { 0 };

while (msg.message != WM\_QUIT) //메시지가 winAPI종료가 아니라면

{

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE)) // 메시지가 있으면 처리

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

else //그렇지 않으면 에니메이션/게임 작업을 수행

{

}

}

// WM\_QUIT 메시지로 반환

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

}

//핸들값을 얻기 위해서

HWND WinAPI::GetHwnd()

{

return WinAPI\_hwnd;

}

//윈 프로시져

LRESULT WinAPI::WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

//메시지

switch (message)

{

case WM\_CREATE: //창이 만들어졌으면

{

// 윈도우 만들시 DirectX12Base 를 저장

LPCREATESTRUCT pCreateStruct = reinterpret\_cast<LPCREATESTRUCT>(lParam);

SetWindowLongPtr(hWnd, GWLP\_USERDATA, reinterpret\_cast<LONG\_PTR>(pCreateStruct->lpCreateParams));

}

return 0;

case WM\_KEYDOWN: //키버튼이 눌렸으면

return 0;

case WM\_KEYUP: //키버튼이 때어젔으면

return 0;

//case WM\_PAINT: //사용하지 않는 이유: 이함수는 다시 그리는용인데 run()함수에서 처리하고 있기 때문

// return 0;

case WM\_DESTROY: //파괴되었을때

PostQuitMessage(0);

return 0;

}

// 디폴트값 대신 모든 메시지 처리

return DefWindowProc(hWnd, message, wParam, lParam);

}

WinAPI.cpp(c++)

LRESULT WinAPI::WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

}

한동안 윈 프로시저 안에 메시지를 처리하는 switch문에서 break와 return을 사용하는 문제에 대해서 시끄러웠을 때가 있었습니다.

//윈 프로시져

LRESULT WinAPI::WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

//메시지

switch (message)

{

case WM\_CREATE: //창이 만들어졌으면

return 0;

case WM\_KEYDOWN: //키버튼이 눌렸으면

return 0;

case WM\_KEYUP: //키버튼이 때어젔으면

return 0;

//case WM\_PAINT: //사용하지 않는 이유: 이함수는 다시 그리는용인데 run()함수에서 처리하고 있기 때문

// return 0;

case WM\_DESTROY: //파괴되었을때

PostQuitMessage(0);

return 0;

}

// 디폴트값 대신 모든 메시지 처리

return DefWindowProc(hWnd, message, wParam, lParam);

}

위의 return 사용되신 아래 break사용으로 바꿀 수 있습니다.

//윈 프로시져

LRESULT WinAPI::WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

//메시지

switch (message)

{

case WM\_CREATE: //창이 만들어졌으면

break;

case WM\_KEYDOWN: //키버튼이 눌렸으면

break;

case WM\_KEYUP: //키버튼이 때어젔으면

break;

case WM\_DESTROY: //파괴되었을때

break;

default:

DefWindowProc(hWnd, message, wParam, lParam);

break;

return 0

}

return, break를 사용하는 입장의 차가 첨예하게 대립하는데 return의 경우는 함수의 종료를 신경 쓰고, break의 경우에는 switch의 종료를 신경 쓰는 편입니다. 개인적인 생각으로는 return, break 둘의 명확한 사용법만 알고 있으면 된다고 생각합니다 또한 switch 문에 하나로 같이 쓰지 않으면 된다고 생각합니다.

LPCREATESTRUCT pCreateStruct = reinterpret\_cast<LPCREATESTRUCT>(lParam);

원래 c를 쓰던 유저라 강제 캐스트를 c++스타일로 쓰는 걸 별로 좋아하지는 않습니다. 예전부터 코드작업을 했기에 c 스타일에 코드를 읽는데도 무리는 없다고 생각합니다. 하지만 c++ 스타일로 바꾸려는 것은 이 코드를 읽는 사람이 내가 아닌 다른 사람일 수도 있기 때문입니다. 연습하지 않으면 외워지지 않음으로 코드 연습 겸 정리합니다.  
**명시적 타입 4가지**  
1) static\_cast<> // 정적 캐스팅, 컴파일시 검사  
→ (c) float a = 1.0f / int b = (int)a;  
→ (c++) float a = 1.0f / int b = static\_cast<int>a;  
⇒ 값, 객체의 타입을 변환시킬 때 사용합니다.  
2) reinterpret\_cast<> // 포인트 안 데이터를 변환시킬 시  
ex)  
temp\* a = new Temp(0,0);  
unsigned int b = reinterpret\_cast<unsigned int>(a); //주소값으로 변경  
temp \*c = reinterpret\_cast<temp \*>(b); //다시 포인트로 변경  
⇒ 포인터 사이 형변환, 포인터와 변수로 변환시킬 시 사용합니다.  
3) const\_cast<> // const 함수 제거할 시  
⇒ 변경권이 없는 외부 DLL함수라던가 const 함수를 제거 할 시 사용합니다.  
4) dynamic\_cast<> // 런타임시 판별  
⇒ 포인터는 참조형을 캐스팅할 때만 사용가능. 호환되지 않는 자식형으로 캐스팅되면 NULL로 반환 RTTI(real time type information)을 사용하지 않으면 static\_cast<>와 동일 합니다.  
c의 강제 캐스팅 ( ) 와 c++ 명시적 캐스팅의 차이점:  
구글링으로 확인해보면 가끔 c++이 안전하다고 적어 놓은 사이트들이 있는데, 토이 프로젝트로 진행해보면 똑같이 런타임시 크래쉬 나는 경우가 대부분입니다. 따라서 안전성은 똑같다고 생각합니다. 하지만 c++의 장점은 프로젝트 검색할 때 ctrl + f 로 검색할 쉽게 검색하여 찾을 수 있고, 가독성이 좋다는 장점이 있습니다.  
(관련 글) <https://stackoverflow.com/questions/103512/why-use-static-castintx-instead-of-intx>

#include "stdafx.h"

#include "WinAPI.h" //다이렉트 x 파이프라인

\_Use\_decl\_annotations\_ //error c28213 해결 사용 이유: 정적분석 도구 에서 주석을 가져오도록 사용

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow)

{

if (!WinAPI::Init( hInstance, nCmdShow)) //초기화 실패하면

return 0; // 0으로 리턴

return WinAPI::Run( ); //성공시 run 코드 실행

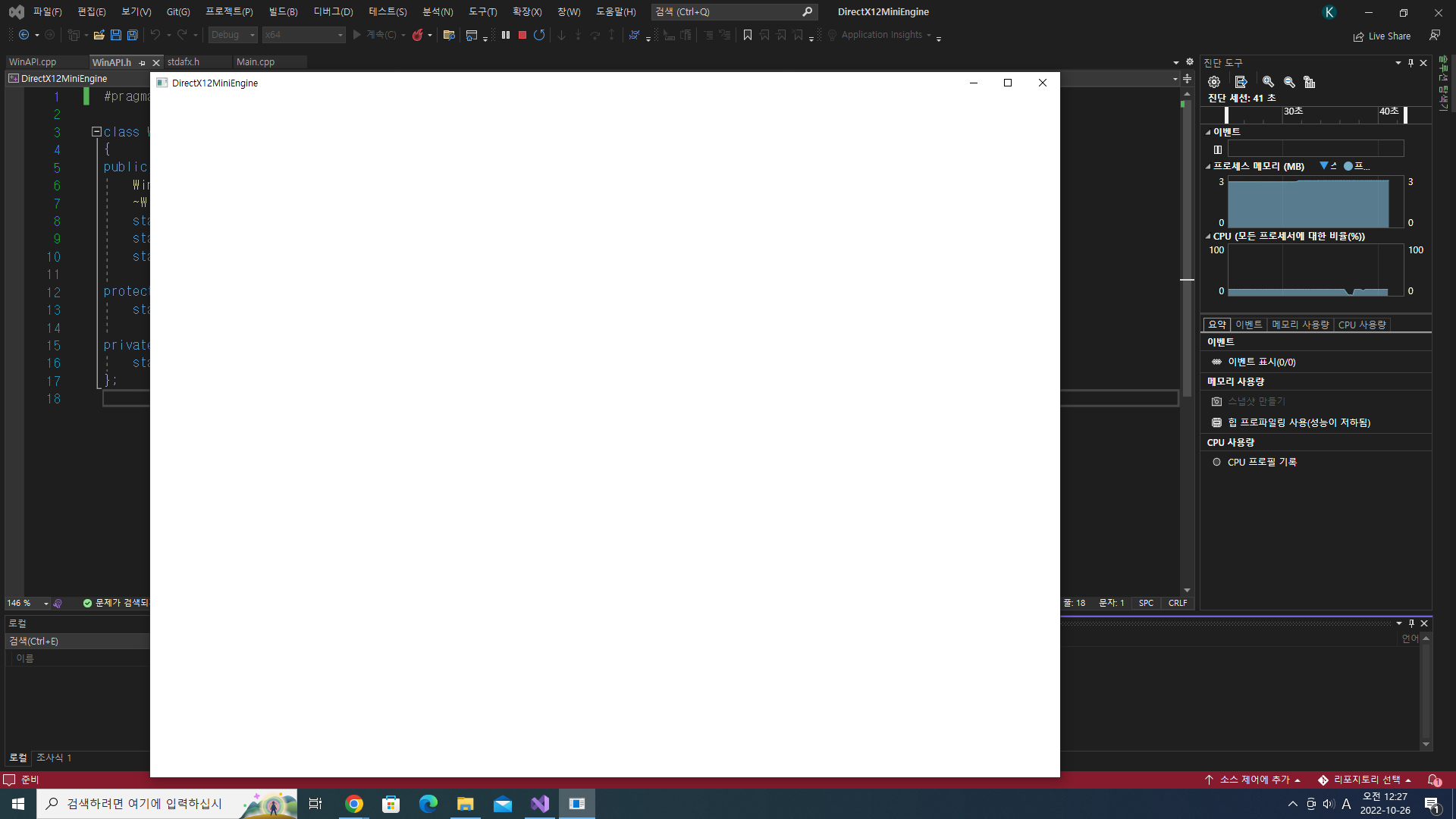
}

Main.cpp(c++)

\_Use\_decl\_annotations\_

**Warning C28213 해결**  
함수의 주석을 가저오도록 정적 코드 분석도구에 지시. 바이너리에 제공되는 API의 주석 중복 되어있는 것을 방지하기 위해서 사용합니다.

### 결과



## directX 초기화

### 이 파트를 정리하는 이유

→ 졸업과제에서는 점수를 위해 구현하기 바빠서 생각보다 하드웨어에 대해서 깊게 생각해 볼 수 없음. Directx12의 장점은 하드웨어 연동이라고 생각하기 때문에 좀더 심도 깊게 탐구해보려고 합니다

### 에러 및 해결 방법

c2102 error  
→ noexpcet의 제너릭 함수를 만들어서 예외 처리

### 코드 및 설계

#pragma once

#include "DirectX12Function.h"

#include "WinAPI.h"

class DirectX12Base

{

public:

DirectX12Base(UINT width, UINT height, std::wstring name); //생성자

virtual ~DirectX12Base(); //소멸자

virtual void OnInit() = 0; //초기값

virtual void OnUpdate() = 0; //업데이트

virtual void OnRender() = 0; //랜더러

virtual void OnDestroy() = 0; //파괴

// 이벤트 핸들러를 재정의하여 특정 메시지 처리

virtual void OnKeyDown(UINT8);// 매개변수 키값

virtual void OnKeyUp(UINT8);// 매개변수 키값

// 접근자

UINT GetWidth() const; //넓이

UINT GetHeight() const; //높이

const WCHAR\* GetTitle() const; //타이틀

void ParseCommandLineArgs(\_In\_reads\_(argc) WCHAR\* argv[], int argc); //명령줄 인수 구분

protected:

std::wstring GetAssetFullPath(LPCWSTR assetName);

void GetHardwareAdapter(\_In\_ IDXGIFactory1\* pFactory, \_Outptr\_result\_maybenull\_ IDXGIAdapter1\*\* ppAdapter, bool requestHighPerformanceAdapter = false);

void SetCustomWindowText(LPCWSTR text);

// 뷰포트

UINT directX12\_width = 0;

UINT directX12\_height = 0;

float directX12\_aspectRatio = 0.0f;

// 어뎁터 정보

bool directX12\_useWarpDevice = false;

private:

std::wstring directX12\_assetsPath; // 루트자산경로

std::wstring directX12\_title; // 윈도우 타이틀

};

DirectX12Base.h(c++)

void GetHardwareAdapter(\_In\_ IDXGIFactory1\* pFactory, \_Outptr\_result\_maybenull\_ IDXGIAdapter1\*\* ppAdapter, bool requestHighPerformanceAdapter = false);

**포인트 관련된 주석(매개변수, 구조체 버퍼)**  
이 주석을 추가하면 포인터가 null일 때, 오류를 알려줌 원래는 잘 안 사용했으나, 연습하기 위해서 사용. 이러한 이유는 포인터로 여러가지 일을 할 수 있는데 함수를 구성한 프로그래머의 의도대로 다른 사람이 사용 안 할 수 도 있기 때문에 사용합니다.  
\_In\_ : 입력 매개 변수를 의미하는 주석  
\_Out\_ : 출력 매개 변수를 의미하는 주석  
\_Outptr\_result\_maybenull\_ : 매개변수가 null일 수는 없고 함수가 종료된 후 가르키는 위치는 null 일 가능성이 있음을 의미합니다.

#include "stdafx.h"

#include "DirectX12Base.h"

//생성자

DirectX12Base::DirectX12Base(UINT width, UINT height, std::wstring name) : directX12\_width(width), directX12\_height(height), directX12\_title(name), directX12\_useWarpDevice(false)

{

WCHAR assetsPath[512];

GetAssetsPath(assetsPath, \_countof(assetsPath)); //위치 알아내기

directX12\_assetsPath = assetsPath; //에셋을 불러오는 현재 위치 저장

directX12\_aspectRatio = static\_cast<float>(width) / static\_cast<float>(height); //종횡비 사이즈 저장

}

//소멸자

DirectX12Base::~DirectX12Base()

{

}

void DirectX12Base::OnKeyDown(UINT8) //키를 눌렀을때

{

}

void DirectX12Base::OnKeyUp(UINT8) //키를 떘을때

{

}

UINT DirectX12Base::GetWidth() const //넓이 얻어오기(변경금지)

{

return directX12\_width;

}

UINT DirectX12Base::GetHeight() const //높이 얻어오기(변경금지)

{

return directX12\_height;

}

const WCHAR\* DirectX12Base::GetTitle() const //타이틀 얻어오기(변경금지)

{

return directX12\_title.c\_str();

}

void DirectX12Base::ParseCommandLineArgs(\_In\_reads\_(argc) WCHAR\* argv[], int argc) //명령줄 인수 구분

{

for (int i = 1; i < argc; ++i)

{

// '-warp' 또는 '/warp'를 argv와 비교하여 일치한다면

if (\_wcsnicmp(argv[i], L"-warp", wcslen(argv[i])) == 0 || \_wcsnicmp(argv[i], L"/warp", wcslen(argv[i])) == 0)

{

directX12\_useWarpDevice = true; //warp 디바이스를 true로 바꿈

directX12\_title = directX12\_title + L" (WARP)";

}

}

}

DirectX12Base.cpp(c++)

std::wstring DirectX12Base::GetAssetFullPath(LPCWSTR assetName)

{

return directX12\_assetsPath + assetName;

}

void DirectX12Base::GetHardwareAdapter(\_In\_ IDXGIFactory1\* pFactory, \_Outptr\_result\_maybenull\_ IDXGIAdapter1\*\* ppAdapter, bool requestHighPerformanceAdapter)

{

\*ppAdapter = nullptr; // 어댑터 포인트 초기화

ComPtr<IDXGIAdapter1> adapter;

ComPtr<IDXGIFactory6> factory6;

if (SUCCEEDED(pFactory->QueryInterface(IID\_PPV\_ARGS(&factory6))))

{

for (

UINT adapterIndex = 0;

SUCCEEDED(factory6->EnumAdapterByGpuPreference(

adapterIndex,

requestHighPerformanceAdapter == true ? DXGI\_GPU\_PREFERENCE\_HIGH\_PERFORMANCE : DXGI\_GPU\_PREFERENCE\_UNSPECIFIED,

IID\_PPV\_ARGS(&adapter)));

++adapterIndex)

{

DXGI\_ADAPTER\_DESC1 desc;

adapter->GetDesc1(&desc);

if (desc.Flags & DXGI\_ADAPTER\_FLAG\_SOFTWARE)

{

// 기본 랜더 드라이버 어뎁터 선택 x

// 소프트웨어 어뎁터가 필요하면 /warp 명령을 보냄

continue;

}

// 어뎁터가 directx12 지원하는지 확인 실제 장치라 생성 x

if (SUCCEEDED(D3D12CreateDevice(adapter.Get(), D3D\_FEATURE\_LEVEL\_11\_0, \_uuidof(ID3D12Device), nullptr)))

{

break;

}

}

}

if (adapter.Get() == nullptr)

{

for (UINT adapterIndex = 0; SUCCEEDED(pFactory->EnumAdapters1(adapterIndex, &adapter)); ++adapterIndex)

{

DXGI\_ADAPTER\_DESC1 desc;

adapter->GetDesc1(&desc);

if (desc.Flags & DXGI\_ADAPTER\_FLAG\_SOFTWARE)

{

// 기본 랜더 드라이버 어뎁터 선택 x

// 소프트웨어 어뎁터가 필요하면 /warp 명령을 보냄

continue;

}

// 어뎁터가 directx12 지원하는지 확인 실제 장치라 생성 x

if (SUCCEEDED(D3D12CreateDevice(adapter.Get(), D3D\_FEATURE\_LEVEL\_11\_0, \_uuidof(ID3D12Device), nullptr)))

{

break;

}

}

}

\*ppAdapter = adapter.Detach();

}

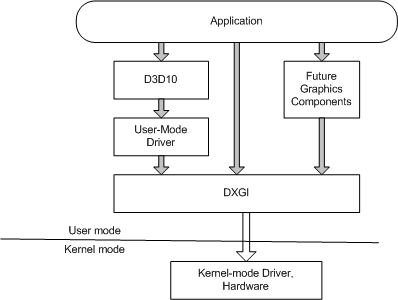
void DirectX12Base::SetCustomWindowText(LPCWSTR text) //윈도우 상태표시창 표현

{

std::wstring windowText = directX12\_title + L": " + text; //유니코드

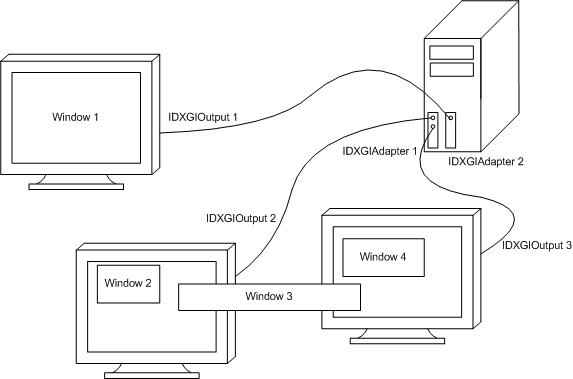
SetWindowText(WinAPI::GetHwnd(), windowText.c\_str()); // 핸들값으로 윈도우 설정하기}

DirectX12Base.cpp(c++)

  
DXGI Directx10 부터 생겼는데, 그 목적은 커널과 드라이버 통신하는 통신하여 미드웨이 적인 특성을 띄는 것입니다. DXGI가 만들어진 배경을 생각해 봐야 하는데 DirectX9일 때는 winAPI 플랫폼이라 사실상 pc그래픽 카드만 적용하여 알면 되기에 상관없었습니다.   
DirectX9 + winAPI → DirectX10 + mfc → DirectX11 + WFP → DirectX12 + UWP으로 발전을 했는데, 사실 상 pc이외에도 다른 하드웨어 접근할 방법이 필요로 해졌기 때문에 DXGI 만들어 졌습니다.   
**DXGI 1.1**: DirectX10, DirectX11 버전, 하이 컬러 및 BGRA(blue green red alpha)색상 지원  
**DXGI 1.2**: 스테레오코스픽(3d 영화관 같은 원리) 지원, ★플립 모델 스왑 체인 지원, 16비트(BPP)지원  
- 플립 모델 이란: 모니터 화면을 갱신하다 보면 재생률 낮은 모니터에서는 화면 ‘찢어짐’ 이라는 위, 아래로 잘리는 현상이 나타나는데, 이를 해결하기 위해서 백 버퍼라는 것을 만듭니다. 모니터 화면에 비출 때 백버퍼에 새로 갱신할 이미지를 담고, 이때 바꾸는 것을 플리핑이라고 합니다. 백버퍼를 여러 개 생성하여 순서를 지정해 놓고 있는 것을 스왑체인라고 합니다.  
**DXGI 1.3**: 어뎁터 메모리 플러시(캐쉬 데이터 지움) 및 해제, 스왑체인 크기 조정  
**DXGI 1.4**: DirectX12 기능 어댑터 기능분리 및 간소화, ★비디오 메모리 비용 추적(budget tracking), 다중어댑터(스왑체인 단일 어댑터 백버퍼 1에서만 만들어 지던거 해결) 지원  
- 비디오 메모리 비용 추적 이란(budget tracking): 비디오 메모리는 절전모드 같은 경우도 존재하기 때문에 메모리가 동적으로 변경될 수 있습니다. 그렇기 때문에 항상 메모리의 사용할 수 있는 크기를 알아야 합는데 이것을 비디오 메모리 비용 추적이라고 합니다. 동적 메모리 크기보다 사용량이 높으면 API실패를 반환하거나 다른 어플리케이션이 실행됩니다.  
**DXGI 1.5**: ★동적 범위가 넓은 디스플레이색(High Dynamic Range, Wide Color Gamut), ★가변 새로고침 빈도표시, 리소스 회수(리소스 삭제 말고 메모리를 해제할 수 있습니다.)  
- 동적범위가 넓은 디스플레이어 고급색 이란: 디스플레이어 HDR인 광도의 수준을 더해서 레이저 색상에서 처리할 수 없는 65% 색처리가 증가함  
- 가변 새로고침 빈도 표시 란: 수직동기화(Vsync) 해제 지원. 수직동기화란 그래픽 카드의 프레임과 모니터의 주사율을 일치시켜주는 기능. 즉, 모니터가 60hz이고 그래픽 카드의 프레임이 59프레임이면 찢김 현상 발생. 하지만 수직동기화를 키게 되면 빠른 입력을 요하는 게임이나 고사양 게임에서는 인풋랙 발생하게 됩니다. 하지만 수직동기화는 불필요하게 디스플레이를 새로고침을 하지 않음으로 전력 관리를 할 수 있습니다.

**DXGI 1.6**: HDR 디스플레이를 지원하는지 검색하기 위한 기능 추가

ComPtr<IDXGIAdapter1> adapter;

디스플레이의 하위 시스템(GPU, DAC, 및 비디오 메모리)을 나타냄  


그래픽 어댑터를 열거하는 단일 매서드

ComPtr<IDXGIFactory6> factory6;

#pragma once

#include "DirectX12Base.h"

class DirectX12EnginePipline : public DirectX12Base

{

public:

DirectX12EnginePipline(UINT width, UINT height, std::wstring name); //생성자

~DirectX12EnginePipline(); //소멸자

virtual void OnInit(); //초기화

virtual void OnUpdate(); //업데이트

virtual void OnRender(); //랜더러

virtual void OnDestroy(); //파괴할떄

private:

static const UINT FrameCount = 2; //프레임 카운트

// 파이프라인 객체

ComPtr<IDXGISwapChain3> directX12\_swapChain; //스왑체인 백버퍼 -> 프론트버퍼

ComPtr<ID3D12Device> directX12\_device; //디바이스

ComPtr<ID3D12Resource> directX12\_renderTargets[FrameCount]; //랜더타겟(프레임카운트)

ComPtr<ID3D12CommandAllocator> directX12\_commandAllocator; //커멘드 할당

ComPtr<ID3D12CommandQueue> directX12\_commandQueue; //커맨드 큐

ComPtr<ID3D12DescriptorHeap> directX12\_rtvHeap; //rtv 힙

ComPtr<ID3D12DescriptorHeap> directX12\_dsvHeap; // dsv 힙

ComPtr<ID3D12PipelineState> directX12\_pipelineState; // 파이프라인 스테이트

ComPtr<ID3D12GraphicsCommandList> directX12\_commandList; // 커멘드 리스트

UINT directX12\_rtvDescriptorSize = 0; // rtv 서술자 사이즈

UINT directX12\_dsvDescriptorSize = 0; // dsv 서술자 사이즈

// 동기화 객체

UINT directX12\_frameIndex = 0; //프레임 인댁스

HANDLE directX12\_fenceEvent = nullptr; //팬스 이벤트

ComPtr<ID3D12Fence> directX12\_fence; // 팬스

UINT64 directX12\_fenceValue = 0; // 팬스 값

//싱글톤

static DirectX12EnginePipline\* s\_app;

void LoadPipeline(); //파이프라인 로드

void LoadAssets(); //파이프라인 에셋

void PopulateCommandList(); // 커멘드 리스트 마들기

void WaitForPreviousFrame(); //이전 프레임 대기(나중에 지울꺼)

};

DirectX12EnginePipline.h(c++)

include "stdafx.h"

#include "DirectX12EnginePipline.h"

//생성자

DirectX12EnginePipline::DirectX12EnginePipline(UINT width, UINT height, std::wstring name):DirectX12Base(width, height, name),directX12\_frameIndex(0),directX12\_rtvDescriptorSize(0), directX12\_fenceValue(0)

{

}

//소멸자

DirectX12EnginePipline::~DirectX12EnginePipline()

{

}

//초기화

void DirectX12EnginePipline::OnInit()

{

LoadPipeline(); //파이프라인 로드

LoadAssets(); //에셋 로드

}

//업데이트

void DirectX12EnginePipline::OnUpdate()

{

}

//랜더링

void DirectX12EnginePipline::OnRender()

{

//장면을 랜더링하는데 필요한 모든 명령 목록을 기록

PopulateCommandList();

//커맨드 리스트를 실행

ID3D12CommandList\* ppCommandLists[] = { directX12\_commandList.Get() };

directX12\_commandQueue->ExecuteCommandLists(\_countof(ppCommandLists), ppCommandLists);

// 프레임 제시

ThrowIfFailed(directX12\_swapChain->Present(1, 0));

WaitForPreviousFrame();

}

void DirectX12EnginePipline::OnDestroy()

{

//gpu가 더이상 리소스를 참조하지 않는지 확인

WaitForPreviousFrame();

CloseHandle(directX12\_fenceEvent);

}

void DirectX12EnginePipline::LoadPipeline() //파이프라인 로드

{

UINT dxgiFactoryFlags = 0; //dxgi 팩토리 플레그

#if defined(\_DEBUG) || defined(\_DEBUG)

// 디버그 레이어 활설 (앱 및 기능/선택적 기능/그래픽도구)

// 장치설정 디버그 계층 활성 활성자치 무효

{

ComPtr<ID3D12Debug> debugController; //디버그 컨트롤러

if (SUCCEEDED(D3D12GetDebugInterface(IID\_PPV\_ARGS(&debugController))))

{

debugController->EnableDebugLayer(); //디버깅 레이어 활성

dxgiFactoryFlags |= DXGI\_CREATE\_FACTORY\_DEBUG; //추가 디버그 레이어 활설

}

}

#endif

ComPtr<IDXGIFactory4> factory; //팩토리

//ThrowIfFailed(CreateDXGIFactory1(IID\_PPV\_ARGS(&factory)));

ThrowIfFailed(CreateDXGIFactory2(dxgiFactoryFlags, IID\_PPV\_ARGS(&factory))); //팩토리 만들기 dxgi.dll 가저오기

DirectX12EnginePipline.cpp(c++)

if (directX12\_useWarpDevice) //WARP 장치로 대체합니다.

{

ComPtr<IDXGIAdapter> warpAdapter; // 레스터라이즈 Windows Advanced Rasterization Platform 세이더 기반 렌더링

ThrowIfFailed(factory->EnumWarpAdapter(IID\_PPV\_ARGS(&warpAdapter))); //어뎁터

// 하드웨어 장치 및 시도

ThrowIfFailed(D3D12CreateDevice(

warpAdapter.Get(),

D3D\_FEATURE\_LEVEL\_11\_0, //레벨

IID\_PPV\_ARGS(&directX12\_device)

));

}

else //아니면 하드웨어 장비로 대체

{

ComPtr<IDXGIAdapter1> hardwareAdapter;

GetHardwareAdapter(factory.Get(), &hardwareAdapter); //하드웨어 어뎁터

ThrowIfFailed(D3D12CreateDevice(

hardwareAdapter.Get(),

D3D\_FEATURE\_LEVEL\_11\_0, //레벨

IID\_PPV\_ARGS(&directX12\_device)

));

}

//명령 대기열으(큐)를 정의하고 생성

D3D12\_COMMAND\_QUEUE\_DESC queueDesc = {};

queueDesc.Flags = D3D12\_COMMAND\_QUEUE\_FLAG\_NONE;

queueDesc.Type = D3D12\_COMMAND\_LIST\_TYPE\_DIRECT;

ThrowIfFailed(directX12\_device->CreateCommandQueue(&queueDesc, IID\_PPV\_ARGS(&directX12\_commandQueue)));

//스왑 체인을 정의하고 생성

ComPtr<IDXGISwapChain1> swapChain;

swapChain.Reset(); //기본쓰레기 값이 들어가 있는경우도 있어서 리셋한번 해줌

//스왑체인 정의

DXGI\_SWAP\_CHAIN\_DESC1 swapChainDesc = {};

swapChainDesc.BufferCount = FrameCount; //버퍼 갯수

swapChainDesc.Width = directX12\_width;

swapChainDesc.Height = directX12\_height;

swapChainDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

swapChainDesc.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

swapChainDesc.SwapEffect = DXGI\_SWAP\_EFFECT\_FLIP\_DISCARD;

swapChainDesc.SampleDesc.Count = 1;

ThrowIfFailed(factory->CreateSwapChainForHwnd(

directX12\_commandQueue.Get(), //스왑체인은 이미지를 처리할때 강제로 비워줘야 되기는 플러시를 해야되기에 대기열이 필요하다.

WinAPI::GetHwnd(),

&swapChainDesc,

nullptr,

nullptr,

&swapChain

));

// 전체화면 지원 안할꺼기 때문에.

ThrowIfFailed(factory->MakeWindowAssociation(WinAPI::GetHwnd(), DXGI\_MWA\_NO\_ALT\_ENTER));

ThrowIfFailed(swapChain.As(&directX12\_swapChain));

directX12\_frameIndex = directX12\_swapChain->GetCurrentBackBufferIndex();

// 설명자 힙을 생성 (cpu 가상 공간에 생성)

{

// 랜더 대상보기 설명자 힙 생성

D3D12\_DESCRIPTOR\_HEAP\_DESC rtvHeapDesc = {}; //render target view 힙

rtvHeapDesc.NumDescriptors = FrameCount; //스왑체인 버퍼 =프레임

rtvHeapDesc.Type = D3D12\_DESCRIPTOR\_HEAP\_TYPE\_RTV;

rtvHeapDesc.Flags = D3D12\_DESCRIPTOR\_HEAP\_FLAG\_NONE;

rtvHeapDesc.NodeMask = 0;

ThrowIfFailed(directX12\_device->CreateDescriptorHeap(&rtvHeapDesc, IID\_PPV\_ARGS(&directX12\_rtvHeap)));

directX12\_rtvDescriptorSize = directX12\_device->GetDescriptorHandleIncrementSize(D3D12\_DESCRIPTOR\_HEAP\_TYPE\_RTV);

DirectX12EnginePipline.cpp(c++)

D3D12\_DESCRIPTOR\_HEAP\_DESC dsvHeapDesc; //depth stencil veiew 힙

dsvHeapDesc.NumDescriptors = 1;

dsvHeapDesc.Type = D3D12\_DESCRIPTOR\_HEAP\_TYPE\_DSV;

dsvHeapDesc.Flags = D3D12\_DESCRIPTOR\_HEAP\_FLAG\_NONE;

dsvHeapDesc.NodeMask = 0;

ThrowIfFailed(directX12\_device->CreateDescriptorHeap(&dsvHeapDesc, IID\_PPV\_ARGS(&directX12\_dsvHeap)));

directX12\_dsvDescriptorSize = directX12\_device->GetDescriptorHandleIncrementSize(D3D12\_DESCRIPTOR\_HEAP\_TYPE\_DSV);

}

// 프레임 리소스를 생성

{

CD3DX12\_CPU\_DESCRIPTOR\_HANDLE rtvHandle(directX12\_rtvHeap->GetCPUDescriptorHandleForHeapStart()); //cpu 가상 주소 공간에 생성

//각 프레임에 대한 rtv를 생성

for (UINT i = 0; i < FrameCount; ++i)

{

ThrowIfFailed(directX12\_swapChain->GetBuffer(i, IID\_PPV\_ARGS(&directX12\_renderTargets[i])));

directX12\_device->CreateRenderTargetView(directX12\_renderTargets[i].Get(), nullptr, rtvHandle);

rtvHandle.Offset(1, directX12\_rtvDescriptorSize);

}

}

ThrowIfFailed(directX12\_device->CreateCommandAllocator(D3D12\_COMMAND\_LIST\_TYPE\_DIRECT, IID\_PPV\_ARGS(&directX12\_commandAllocator)));

}

void DirectX12EnginePipline::LoadAssets()

{

// 커맨드 리스트을 생성

ThrowIfFailed(directX12\_device->CreateCommandList(

0,

D3D12\_COMMAND\_LIST\_TYPE\_DIRECT,

directX12\_commandAllocator.Get(), // 관련 명령 할당자

nullptr, //초기 파이프라인 상태 오브젝트

IID\_PPV\_ARGS(&directX12\_commandList)));

// 커맨드 리스트는 레코딩 상태에서 생성되지만 아무것도 존재하지 않음

// 레코드 상태에서 닫힐것을 예상함으로 지금 닫음.(재설정 해야됨)

ThrowIfFailed(directX12\_commandList->Close());

// 동기화 객체 생성

{

ThrowIfFailed(directX12\_device->CreateFence(0, D3D12\_FENCE\_FLAG\_NONE, IID\_PPV\_ARGS(&directX12\_fence)));

directX12\_fenceValue = 1;

// 프레임 동기화에 사용할 이벤트 핸들을 생성

directX12\_fenceEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);

if (directX12\_fenceEvent == nullptr)

{

ThrowIfFailed(HRESULT\_FROM\_WIN32(GetLastError()));

}

}

}

void DirectX12EnginePipline::PopulateCommandList()

{

// 커맨드 리스트는 연결된 경우만 재설정 가능

// 커멘드 리스트는 GPU처리를 완료했음으로 앱을 실행

// GPU처리를 위해 팬스 설정

ThrowIfFailed(directX12\_commandAllocator->Reset());

// 특정명령에서 ExecuteCommandList() 호출되면, 해당 커맨드 리스트는 재설정 해야 함으로 다시 레코딩

ThrowIfFailed(directX12\_commandList->Reset(directX12\_commandAllocator.Get(), directX12\_pipelineState.Get()));

// 백버퍼가 랜더 타겟으로 사용됨

directX12\_commandList->ResourceBarrier(1, &keep(CD3DX12\_RESOURCE\_BARRIER::Transition(directX12\_renderTargets[directX12\_frameIndex].Get(), D3D12\_RESOURCE\_STATE\_PRESENT, D3D12\_RESOURCE\_STATE\_RENDER\_TARGET)));

CD3DX12\_CPU\_DESCRIPTOR\_HANDLE rtvHandle(directX12\_rtvHeap->GetCPUDescriptorHandleForHeapStart(), directX12\_frameIndex, directX12\_rtvDescriptorSize); //cpu 가상주소 공간에 생성

// 명령을 기록

const float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };

directX12\_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr);

// 백버퍼에서 있던 내용을 화면으로 뿌려줌

directX12\_commandList->ResourceBarrier(1, &keep(CD3DX12\_RESOURCE\_BARRIER::Transition(directX12\_renderTargets[directX12\_frameIndex].Get(), D3D12\_RESOURCE\_STATE\_RENDER\_TARGET, D3D12\_RESOURCE\_STATE\_PRESENT)));

ThrowIfFailed(directX12\_commandList->Close());

}

void DirectX12EnginePipline::WaitForPreviousFrame()

{

//완료할때 까지 기다림

// 시그널 보내면서 팬스값 증가

const UINT64 fence = directX12\_fenceValue;

ThrowIfFailed(directX12\_commandQueue->Signal(directX12\_fence.Get(), fence));

directX12\_fenceValue++;

// 프레임 완료될때까지 기다림

if (directX12\_fence->GetCompletedValue() < fence)

{

ThrowIfFailed(directX12\_fence->SetEventOnCompletion(fence, directX12\_fenceEvent));

WaitForSingleObject(directX12\_fenceEvent, INFINITE);

}

directX12\_frameIndex = directX12\_swapChain->GetCurrentBackBufferIndex(); //현재 백퍼 인덱스의 번호로 바꿔줌

}

DirectX12EnginePipline.cpp(c++)

void DirectX12EnginePipline::WaitForPreviousFrame()

{

//완료할때 까지 기다림

// 시그널 보내면서 팬스값 증가

const UINT64 fence = directX12\_fenceValue;

ThrowIfFailed(directX12\_commandQueue->Signal(directX12\_fence.Get(), fence));

directX12\_fenceValue++;

// 프레임 완료될때까지 기다림

if (directX12\_fence->GetCompletedValue() < fence)

{

ThrowIfFailed(directX12\_fence->SetEventOnCompletion(fence, directX12\_fenceEvent));

WaitForSingleObject(directX12\_fenceEvent, INFINITE);

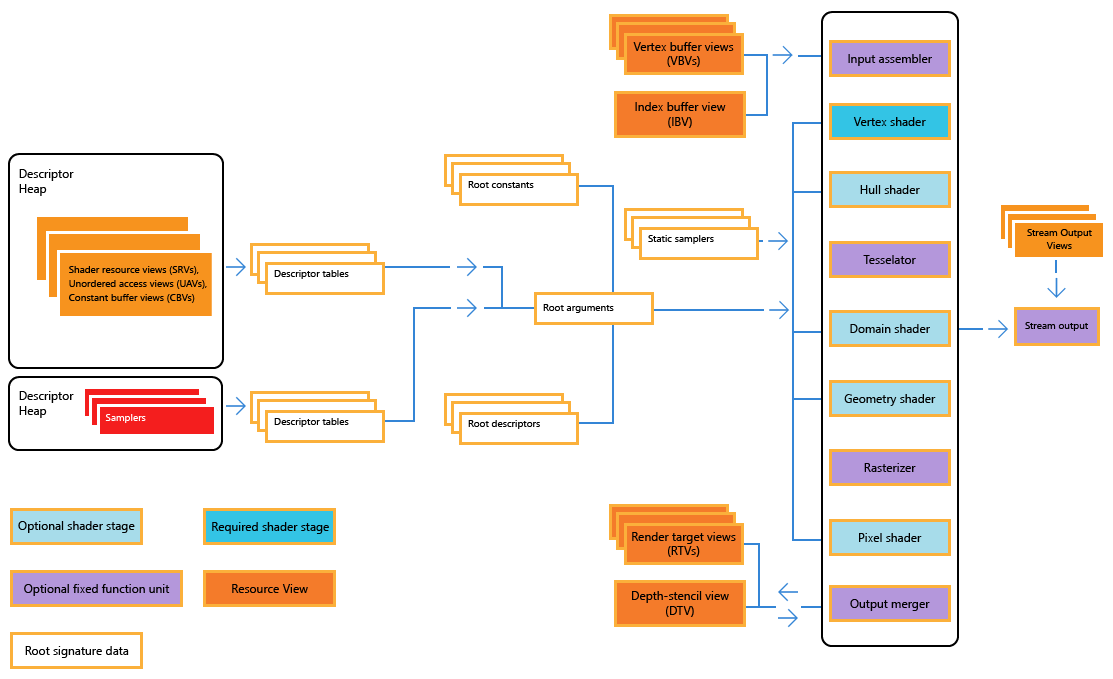
}

directX12\_frameIndex = directX12\_swapChain->GetCurrentBackBufferIndex(); //현재 백퍼 인덱스의 번호로 바꿔줌

}

DirectX12EnginePipline.cpp(c++)

그래픽스 파이프라인(Graphic Pipeline)  
마이크로 소프트의 DXGI 도큐먼트를 보는 도중 ‘맞다고 생각하던 지식’인 학부에서 배운 그래픽스 이론, 전공 책, 구글링등이 틀렸다는 것을 깨달았습니다. 이로 인해 아직도 부족하다는 것을 깨달았습니다.



DirectX9에서는 셰이더를 사용하려면 특별하게 셰이더를 인식하는 코드를 짜야 했습니다. 하지만 DirectX10 부터는 셰이더 중심 파이프라인이 되면서 DirectX12에서는 PSO(파이프라인 상태 객체)가 도입되어서 셰이더의 구성요소를 상태를 저장하고 레지스터에 복사하게 됩니다. 이는 DirectX11의 다른 입력을 처리하기 위해 셰이더 가져오기(fetch shader) 더 이상 사용하지 않는다는 점입니다. DirectX12는 바인딩 없는 리소스에 대한 엑세스를 제공하고 이는 효율적으로 오버헤드를 감소시키기 때문입니다.

3D viewing Pipeline

그래픽스 이론 수업, 전공 책, 특정 사이트에서 전부 그래픽스 파이프 라인으로 소개해서 잘못 알았던 3d 뷰잉 파이프 라인입니다.



3d 모델이 모니터로 출력하는 과정으로서 졸업과제에 이 파이프라인을 이용하여 적용하였습니다.

