

버전 1.0

2022.10.23

# PORTFOLLO

## DirectX12-Mini Engine

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

작성자: 김경주



## 0. WINAPI(윈도우 창)

### 1) WINAPI 클래스를 MAIN.CPP 에서 분리한 이유:

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

### 2) 힘들었던 점:

졸업과제로 당시 directx9 으로 엔진 만들 당시, 이런 구조의 코드를 작성하고 싶어서 mfc 코드를 뜯어서 적용시켜 봤으나 링크 에러로 고민을 많이 했습니다.

### 3) 에러 및 해결방법:

#### ① WindowProc 링크 에러

→ win 프로시저 함수에 static 을 적용하여서 this-> 라는 포인트를 제거함으로 해결

#### ② lnk2019 error

→ (속성/링커/시스템) 하위 시스템을 콘솔에서 창으로 변경합니다

### 4) 코드 및 설계

#### stdafx.h(c++)

```
#pragma once //미리컴파일된 헤더

#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
```

#### WinAPI.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 프로시저

private:
    static HWND WinAPI_hwnd; //윈도우 핸들
};
```

## WinAPI.cpp(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 매개변수 구분
    int argc;
    LPWSTR* argv = CommandLineToArgvW(GetCommandLineW(), &argc);
    LocalFree(argv);
    // 윈도우 클래스 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 만들
    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 애니메이션/게임 작업을 수행
        {
        }
    }

    // WM_QUIT 메시지로 반환
    return static_cast<char>(msg.wParam);
}

//핸들값을 얻기 위해서
HWND WinAPI::GetHwnd()
{
    return WinAPI_hwnd;
}

//원 프로시저
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 떴어졌으면
            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);
}

```

```
LRESULT WINAPI::WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
}
```

한동안 원 프로시저 안에 메시지를 처리하는 switch 문에서 break 와 return 을 사용하는 문제에 대해서 시끄러웠을 때가 있었습니다.

```
//원 프로시저
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 떴으면
            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 break 사용으로 바꿀 수 있습니다.

```
//원 프로시저
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 떴으면
            break;
        case WM_DESTROY: //파괴되었을때
            break;
        default:
            DefWindowProc(hWnd, message, wParam, lParam);
            break;
    }

    return 0
}
```

return, break 를 사용하는 입장의 차가 첨예하게 대립하는데 return 의 경우는 함수의 종료를 신경 쓰고, break 의 경우에는 switch 의 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 c 를 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 4 가지

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 DLL 함수라던가 const 함수를 제거 할 시 사용합니다.

4) dynamic\_cast<> // 런타임시 판별

⇒ 포인터는 참조형을 캐스팅할 때만 사용가능. 호환되지 않는 자식형으로 캐스팅되면 NULL 로 반환 RTTI(real time type information)를 사용하지 않으면 static\_cast<>와 동일합니다.

c 의 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>

## Main.cpp(c++)

```
#include "stdafx.h"
#include "WinAPI.h" //다이렉트 x 파이프라인

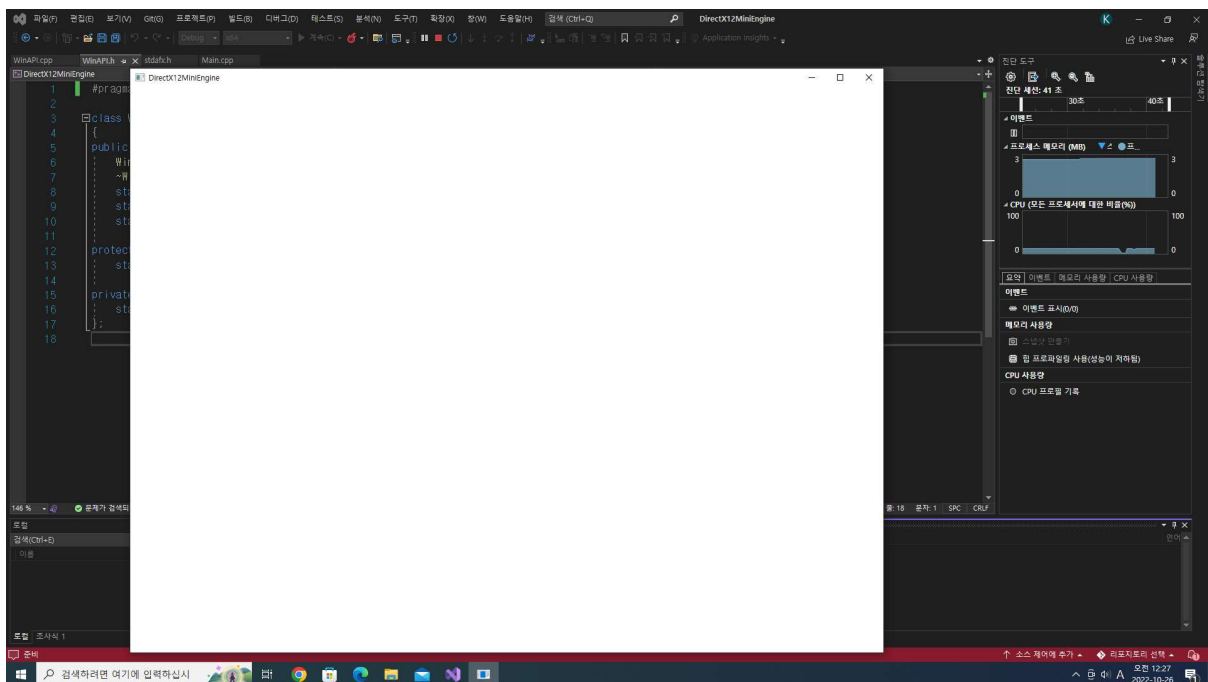
_Use_decl_annotations_ //error c28213 해결 사용 이유: 정적분석 도구 에서 주석을 가져오도록
//사용
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow)
{
    if (!WinAPI::Init( hInstance, nCmdShow)) //초기화 실패하면
        return 0; // 0으로 리턴
    return WinAPI::Run( ); //성공시 run 코드 실행
}
```

\_Use\_decl\_annotations\_

## Warning C28213 해결

함수의 주석을 가져오도록 정적 코드 분석도구에 지시. 바이너리에 제공되는 API 의 주석 중복 되어있는 것을 방지하기 위해서 사용합니다.

## 5) 결과





## 1. DIRECTX 초기화

### 1) 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

### 2) 에러 및 해결 방법

#### ① c2102 error

→ noexcept의 제너릭 함수를 만들어서 예외 처리

### 3) 코드 및 설계

#### DirectX12Base.h(c++)

```
#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; //렌더링
    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;

    // 어댑터 정보
    bool directX12_useWarpDevice = false;

private:
    std::wstring directX12_assetsPath; // 루트자산경로
    std::wstring directX12_title; // 윈도우 타이틀
};
```

```
void GetHardwareAdapter(_In_ IDXGIFactory1* pFactory, _Outptr_result_maybenull_
IDXGIAdapter1** ppAdapter, bool requestHighPerformanceAdapter = false);
```

### 포인트 관련된 주석(매개변수, 구조체 버퍼)

이 주석을 추가하면 포인터가 null 일 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 null 일 수는 없고 함수가 종료된 후 가르키는 위치는 null 일 가능성이 있음을 의미합니다.

### DirectX12Base.cpp(c++)

```
#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 뗄때
{
}

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 렌더 드라이버 어댑터 선택 x
                // 소프트웨어 어댑터가 필요하면 /warp 명령을 보냄
                continue;
            }

            // 어댑터가 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);

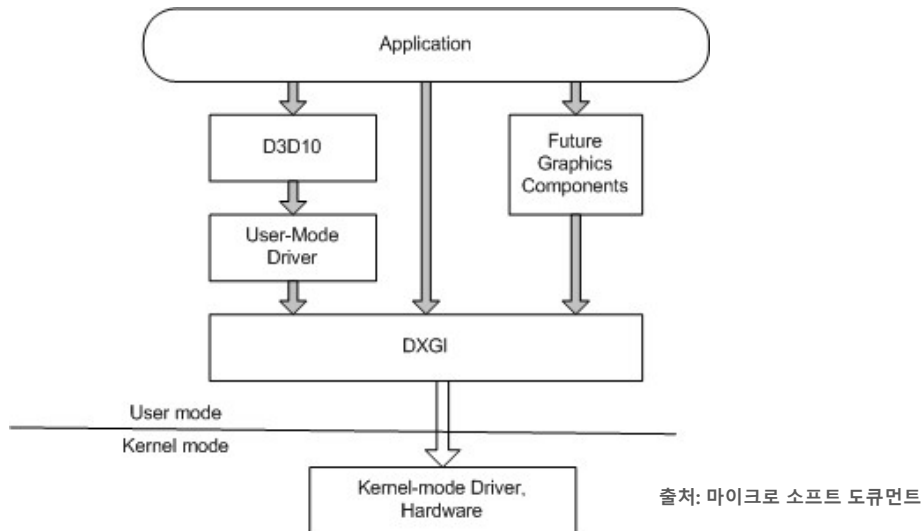
            if (desc.Flags & DXGI_ADAPTER_FLAG_SOFTWARE)
            {
                // 기본 렌더 드라이버 어댑터 선택 x
                // 소프트웨어 어댑터가 필요하면 /warp 명령을 보냄
                continue;
            }

            // 어댑터가 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 상태 표시창 표현
{
    std::wstring windowText = directX12_title + L": " + text; //유니코드
    SetWindowText(WinAPI::GetHwnd(), windowText.c_str()); // 핸들값으로 윈도우 설정하기}

```



DXGI DirectX10 부터 생겼는데, 그 목적은 커널과 드라이버 통신하는 통신하여 미드웨어적인 특성을 띄는 것입니다. DXGI 가 만들어진 배경을 생각해 봐야 하는데 DirectX9 일 때는 winAPI 플랫폼이라 사실상 pc 그래픽 카드만 적용하여 알면 되기에 상관없었습니다. DirectX9 + winAPI → DirectX10 + mfc → DirectX11 + WFP → DirectX12 + UWP 으로 발전을 했는데, 사실 상 pc 이외에도 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, 16 비트(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 스왑체인이라고 합니다.

**DXGI 1.3:** 어댑터 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 1 에서만 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 하는데 이것을 비디오 메모리 비용 추적이라고 합니다. 동적 메모리 크기보다 사용량이 높으면 API 실패를 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.)

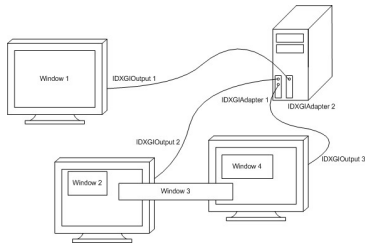
- 동적범위가 넓은 디스플레이어 고급색 이란: 디스플레이어 HDR 인 광도의 수준을 더해서 레이저 색상에서 처리할 수 없는 65% 색처리가 증가함

- 가변 새로고침 빈도 표시 란: 수직동기화(Vsync) 해제 지원. 수직동기화란 그래픽 카드의 프레임과 모니터의 주사율을 일치시켜주는 기능. 즉, 모니터가 60hz 이고 그래픽 카드의 프레임이 59 프레임이면 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



출처: 마이크로 소프트 문서

```
ComPtr<IDXGIFactory6> factory6;
```

그래픽 어댑터를 열거하는 단일 매서드

### DirectX12EnginePipeline.h(c++)

```
#pragma once
#include "DirectX12Base.h"

class DirectX12EnginePipeline : public DirectX12Base
{
public:
    DirectX12EnginePipeline(UINT width, UINT height, std::wstring name); //생성자
    ~DirectX12EnginePipeline(); //소멸자

    virtual void OnInit(); //초기화
    virtual void OnUpdate(); //업데이트
    virtual void OnRender(); //랜더러
    virtual void OnDestroy(); //파괴할때

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; //커맨드 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; // 커맨드 리스트
    UINT directX12_rtvDescriptorSize = 0; // rtv 서술자 사이즈
    UINT directX12_dsvDescriptorSize = 0; // dsv 서술자 사이즈

    // 동기화 객체
    UINT directX12_frameIndex = 0; //프레임 인덱스
    HANDLE directX12_fenceEvent = nullptr; //팬스 이벤트
    ComPtr<ID3D12Fence> directX12_fence; // 팬스
    UINT64 directX12_fenceValue = 0; // 팬스 값

    //싱글톤
    static DirectX12EnginePipeline* s_app;

    void LoadPipeline(); //파이프라인 로드
    void LoadAssets(); //파이프라인 예셋
    void PopulateCommandList(); // 커맨드 리스트 만들기
    void WaitForPreviousFrame(); //이전 프레임 대기(나중에 지울꺼)
};
```

## DirectX12EnginePipeline.cpp(c++)

```

include "stdafx.h"
#include "DirectX12EnginePipeline.h"

//생성자
DirectX12EnginePipeline::DirectX12EnginePipeline(UINT width, UINT height, std::wstring name):DirectX12Base(width, height,
name),directX12_frameIndex(0),directX12_rtvDescriptorSize(0), directX12_fenceValue(0)
{
}

//소멸자
DirectX12EnginePipeline::~DirectX12EnginePipeline()
{
}

//초기화
void DirectX12EnginePipeline::OnInit()
{
    LoadPipeline(); //파이프라인 로드
    LoadAssets(); //에셋 로드
}

//업데이트
void DirectX12EnginePipeline::OnUpdate()
{
}

//렌더링
void DirectX12EnginePipeline::OnRender()
{
    //장면을 렌더링하는데 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 DirectX12EnginePipeline::OnDestroy()
{
    //gpu가 더이상 리소스를 참조하지 않는지 확인
    WaitForPreviousFrame();
    CloseHandle(directX12_fenceEvent);
}

void DirectX12EnginePipeline::LoadPipeline() //파이프라인 로드
{
    UINT dxgiFactoryFlags = 0; //dxgi 팩토리 플레그

#ifdef _DEBUG || defined(_DEBUG)
    // 디버그 레이어 활성화 (앱 및 기능/선택적 기능/그래픽도구)
    // 장치설정 디버그 계층 활성화 활성화자치 무효
    {
        ComPtr<ID3D12Debug> debugController; //디버그 컨트롤러
        if (SUCCEEDED(D3D12GetDebugInterface(IID_PPV_ARGS(&debugController))))
        {
            debugController->EnableDebugLayer(); //디버깅 레이어 활성화
            dxgiFactoryFlags |= DXGI_CREATE_FACTORY_DEBUG; //추가 디버그 레이어 활성화
        }
    }
#endif

    ComPtr<IDXGIFactory4> factory; //팩토리
    ThrowIfFailed(CreateDXGIFactory1(IID_PPV_ARGS(&factory)));
    ThrowIfFailed(CreateDXGIFactory2(dxgiFactoryFlags, IID_PPV_ARGS(&factory))); //팩토리 만들기 dxgi.dll 가져오기

```

## DirectX12EnginePipeline.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))); //어댑터

    // 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 어댑터

    ThrowIfFailed(D3D12CreateDevice(
        hardwareAdapter.Get(),
        D3D_FEATURE_LEVEL_11_0, //레벨
        IID_PPV_ARGS(&directX12_device)
    ));
}

//명령 대기열(큐)을 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
    &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)
{
    // 렌더 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);
}

```

## DirectX12EnginePipeline.cpp(c++)

```

D3D12_DESCRIPTOR_HEAP_DESC dsvHeapDesc; //depth-stencil veiw 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 DirectX12EnginePipeline::LoadAssets()
{
    // 커맨드 리스트를 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
    // 레코딩 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 DirectX12EnginePipeline::PopulateCommandList()
{
    // 커맨드 리스트는 연결된 경우만 재설정 가능
    // 커맨드 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());
}

```



## DirectX12EnginePipeline.cpp(c++)

```

void DirectX12EnginePipeline::WaitForPreviousFrame()
{
    //완료할때 까지 기다림

    // 시그널 보내면서 펜스값 증가
    const UINT64 fence = directX12_fenceValue;
    ThrowIfFailed(directX12_commandQueue->Signal(directX12_fence.Get(), fence));
    directX12_fenceValue++;

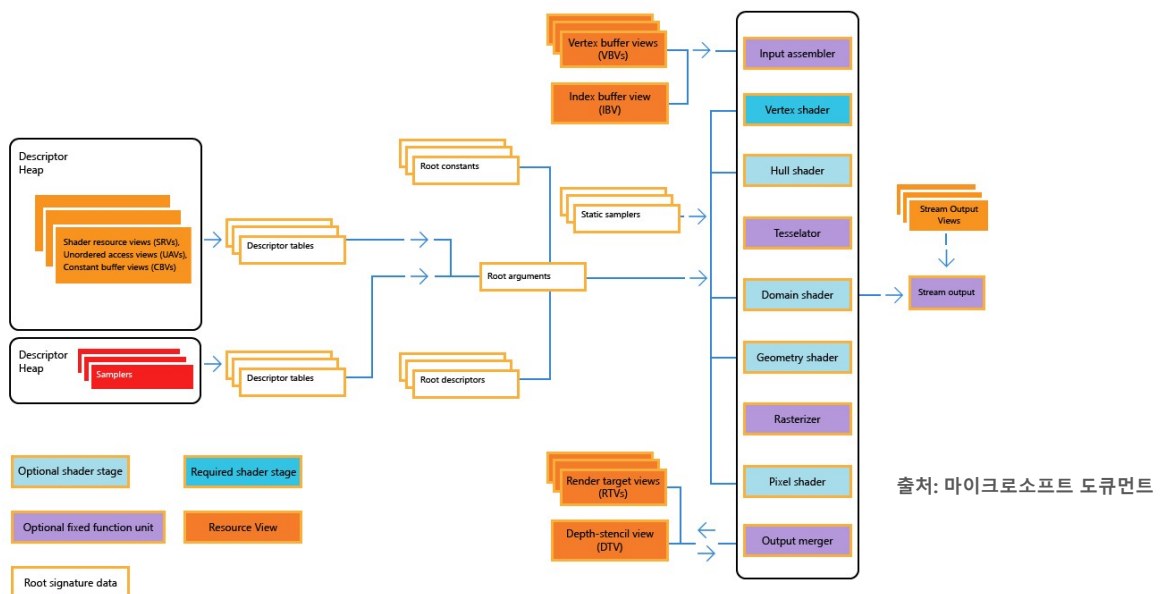
    // 프레임 완료될때까지 기다림
    if (directX12_fence->GetCompletedValue() < fence)
    {
        ThrowIfFailed(directX12_fence->SetEventOnCompletion(fence, directX12_fenceEvent));
        WaitForSingleObject(directX12_fenceEvent, INFINITE);
    }

    directX12_frameIndex = directX12_swapChain->GetCurrentBackBufferIndex(); //현재 백퍼 인덱스의 번호로 바꿔줌
}

```

## ① 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 DirectX10부터는 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 액세스를 제공하고 이는 효율적으로 오버헤드를 감소시키기 때문입니다.

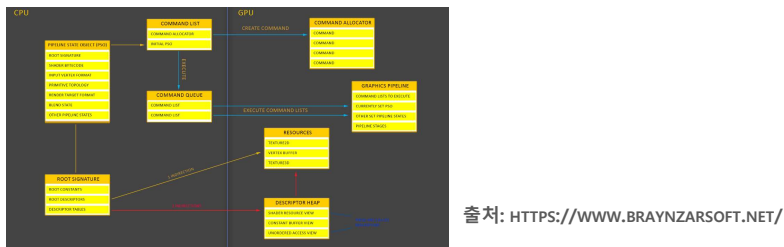
## ② 3D viewing Pipeline

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



3d 모델이 모니터로 출력하는 과정으로서 졸업과제 DirectX9 에 이 파이프라인을 이용하여 적용하였습니다.

## ③ 클래스 다이어그램



### DirectX12Function.h(c++)

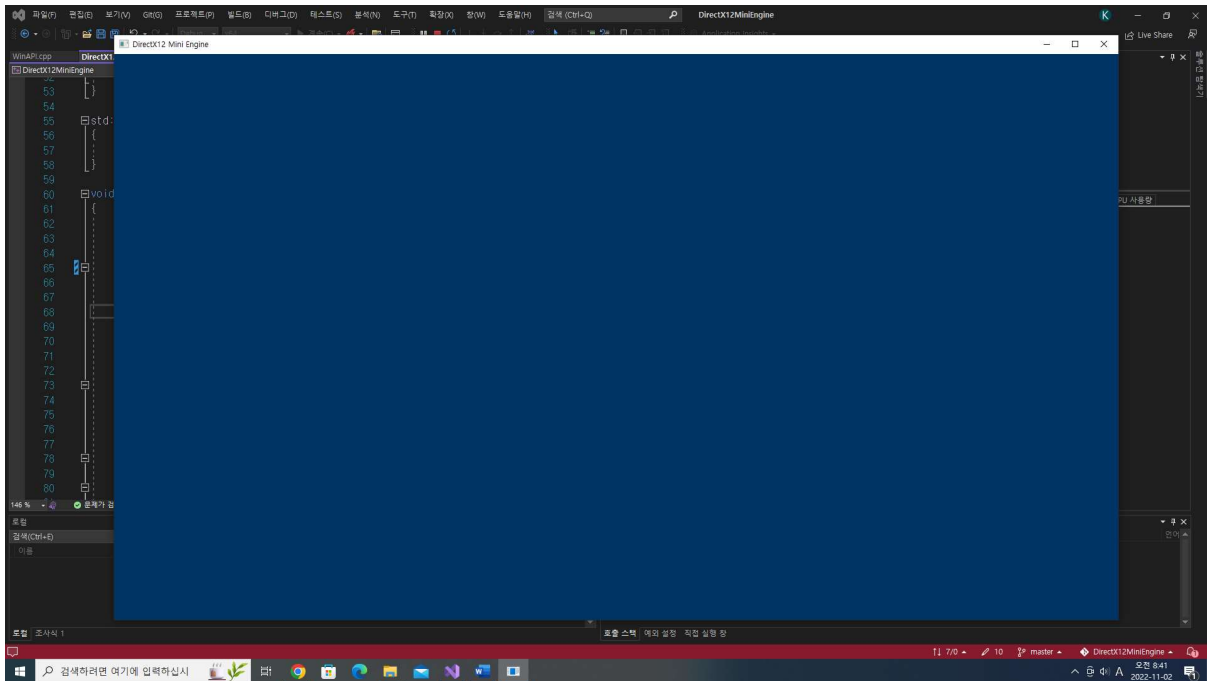
```
pragma once
#include <stdexcept>

//c2102 error 해결법
template <class T>
inline constexpr auto& keep(T&& x) noexcept {
    return x;
}
```

```
inline constexpr auto& keep(T&& x) noexcept
```

noexcept: C++11 이전에서는 컴파일러 에러가 철저히 지켜져야만 했습니다. 그러나 함수 구현을 바꾸면서 예외 지정이 바뀔 수 있고, 기존 예외를 지키던 구문이 깨질 수도 있는 경우들이 생겼습니다. 이러한에 따라 C++11에서는 예외를 사용할 것인지 인지 하는 코드가 필요했고 그 결과 noexcept 라는 코드가 생겼습니다. 이는 C++11 부터 동적 예외 지정자인 throw()를 더 이상 사용하지 않으며(C++17 에서 제거됨), 대신 사용을 합니다. 예외가 붙은 함수에 선언이 되며 컴파일러에 보다 효율적인 성능을 제공합니다. constexpr: const 라는 함수 보다 좀 더 상수에 가까운 함수를 정의할 필요성이 생겼습니다. const 는 변수의 초기화를 런타임까지 지연시킬 수 있습니다. 따라서 C++11에서는 컴파일 단계에서 변수 초기화 해야 할 코드가 필요했고 constexpr 가 등장하게 되었습니다

#### 4) 결과



## 2. TRIANGLE

### 1) TRIANGLE

평면을 이루는 가장 기초적 단위인 삼각형 메쉬를 그리는 방법입니다. 이 메쉬 하나를 프로그래밍 용어로 드로우 콜이라고 하며, 드로우 콜의 최적화에 따라 성능이 달라집니다.

### 2) 에러 및 해결 방법

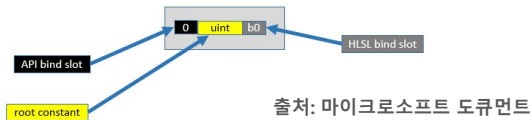
- ① DXcaptureReplay.pdb 등 directx12 그래픽 디버깅 불가(visual studio 2022)  
→ 2017 년부터 vs 에서 그래픽 디버깅 지원 안하고 있음. PIX 유틸리티로 변경

### 3) 코드 및 연구 – 기본 삼각형 메쉬

#### DirectX12EnginePipeline.cpp(c++)

```
{
    CD3DX12_ROOT_SIGNATURE_DESC rootSignatureDesc; //루트서명 정의
    rootSignatureDesc.Init(0, nullptr, 0, nullptr, D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

    ComPtr<ID3DBlob> signature;
    ComPtr<ID3DBlob> error;
    ThrowIfFailed(D3D12SerializeRootSignature(&rootSignatureDesc, D3D_ROOT_SIGNATURE_VERSION_1, &signature, &error));
    ThrowIfFailed(directX12_device->CreateRootSignature(0, signature->GetBufferPointer(), signature->GetBufferSize(),
    IID_PPV_ARGS(&directX12_rootSignature)));
}
```



메쉬를 정의하기전 비어있는 루트 서명(Root Signature)을 만드는 곳입니다. 루트 서명은 셰이더가 접속하는 데이터를 정의하는 곳입니다. 위와 같이 루트 서명에는 index, 슬롯 루트 상수, 바인드 셰이더처럼 하나로 묶여져 있습니다.

#### DirectX12EnginePipeline.cpp(c++)

```
// 셰이더 컴파일 및 로드를 포함하는 파이프라인 만들기
{
    ComPtr<ID3DBlob> vertexShader; //버텍스 셰이더
    ComPtr<ID3DBlob> pixelShader; //픽셀 셰이더

    #if defined(_DEBUG)
        UINT compileFlags = D3DCOMPILER_DEBUG | D3DCOMPILER_SKIP_OPTIMIZATION;
    #else
        UINT compileFlags = 0;
    #endif

    ThrowIfFailed(D3DCompileFromFile(GetAssetFullPath(L"../shaders.hlsl").c_str(), nullptr, nullptr, "VSMMain",
    "vs_5_0", compileFlags, 0, &vertexShader, nullptr)); //버텍스 셰이더에 hlsl VSMMain 담기
    ThrowIfFailed(D3DCompileFromFile(GetAssetFullPath(L"../shaders.hlsl").c_str(), nullptr, nullptr, "PSMain",
    "ps_5_0", compileFlags, 0, &pixelShader, nullptr)); //픽셀 셰이더 PSMain 담기

    // 정점 입력 레이아웃 정의
    D3D12_INPUT_ELEMENT_DESC inputElementDescs[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }, //위치
        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 } //색상
    };

    // PSO(그래픽스 파이프라인 상태 오브젝트)를 설명하고 생성
    D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc = {};
    psoDesc.InputLayout = { inputElementDescs, _countof(inputElementDescs) }; //GPU 프론트 엔드 레이아웃
    psoDesc.pRootSignature = directX12_rootSignature.Get(); //루트 시그널 포인트
    psoDesc.VS = CD3DX12_SHADER_BYTECODE(vertexShader.Get()); // 버텍스 셰이더
    psoDesc.PS = CD3DX12_SHADER_BYTECODE(pixelShader.Get()); //픽셀 셰이더
    psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT); // 레스터 라이즈: 기본
    psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT); // 블렌드 상태: 기본
    psoDesc.DepthStencilState.DepthEnable = FALSE; //덱스: 불가
    psoDesc.DepthStencilState.StencilEnable = FALSE; //스텐실: 불가
    psoDesc.SampleMask = UINT_MAX; //블렌드 상태의 멀티 샘플링 32개(bit)로 최대
    psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE; //기본도형의 구조(테셀레이터) 삼각형
    psoDesc.NumRenderTargets = 1; //렌더 갯수
    psoDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM; //렌더 대상의 형틀
    psoDesc.SampleDesc.Count = 1; //멀티샘플링 표본 품질의 갯수 1개
    ThrowIfFailed(directX12_device->CreateGraphicsPipelineState(&psoDesc, IID_PPV_ARGS(&directX12_pipelineState)));
    //그래픽 파이프라인 상태 만들기
}
```

삼각형을 그리기 위해서 셰이더인 hlsl 에 있는 버텍스 셰이더, 픽셀 셰이더를 찾아서 찾아서 pso(그래픽스 파이프라인 오브젝트)에 정의를 만들어 줍니다.

## DirectX12EnginePipeline.cpp(c++)

```
// 커맨드 리스트를 생성
ThrowIfFailed(directX12_device->CreateCommandList(
    0,
    D3D12_COMMAND_LIST_TYPE_DIRECT,
    directX12_commandAllocator.Get(), // 관련 명령 할당자
    directX12_pipelineState.Get(), // 초기 파이프라인 상태 오브젝트(nullptr -> directX12_pipelineState.Get())
    IID_PPV_ARGS(&directX12_commandList));
```

그리고 커맨드 리스트에 pso 를 저장하기 위해서 nullptr(널 값)이었던 부분에 파이프라인을 적어서 저장시켜 줍니다.

```
HRESULT WINAPI D3DCompileFromFile(
    in LPCWSTR pFileName,
    in_opt const D3D_SHADER_MACRO pDefines,
    in_opt ID3DInclude pInclude,
    in LPCSTR pEntrypoint,
    in LPCSTR pTarget,
    in UINT Flags1,
    in UINT Flags2,
    out ID3DBlob ppCode,
    out_opt ID3DBlob ppErrorMsgs
);
```

D3DCompileFromFile() 관하여 살펴보자면

pFileName: 셰이더 코드 이름,

pDefines: 셰이더 매크로를 정의하는 구조의 배열,

pInclude: 셰이더 코드 #include 에 사용되는 인터페이스 포인터,

pEntrypoint: 셰이더 함수의 이름,

pTarget: 셰이더 컴파일 때 사용하는 셰이더 모델

Flags1: 컴파일 옵션(디버그모드),

Flags2: 효과 파일을 넣는 플래그,

ppCode: 셰이더 바이트코드인 ID3DBlob 에 대한 코드,

ppErrorMsgs: 오류 발생할 때 볼 수 있는 리포트 ID3DBlob 포인터로 구성되어 있습니다.

```
typedef struct D3D12_INPUT_ELEMENT_DESC {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D12_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D12_INPUT_ELEMENT_DESC;
```

버텍스 버퍼의 내부 버텍스를 입력 어셈블러에 설명하는 입력 레이아웃을 만드는데 이는

정점을 구성하고 전달합니다. D3D12\_INPUT\_ELEMENT\_DESC()에 관해 살펴보자면

SemanticName: 매개변수의 이름(셰이더 입력 서명에 동일한 요소),

SemanticIndex: 두개이상 동일한 이름 체계를 가지고있는 경우 인덱스 번호,

Format: DXGI 포맷(예를들어 (float)x, (float)y, (float)z 을 가지고 있는 경우 4byte \*3 임으로 DXGI\_FORMAT\_R32G32B32\_FLOAT 로 정의),

InputSlot: 정점 버퍼를 입력 어셈블러에 바인딩,

AlignedByteOffset: 정점구조에서부터 속성까지 시작의 오프셋 바인드(예를 들어 정점 위치값을 xyz 만 가지고 있다면 0 로 시작 그 후 color 값을 가지고 있다면 4byte\*3 인 12 부터 시작 이런 구조는 패킷 데이터를 해독할 때 사용하는 방법),

InputSlotClass: 버텍스 구조로 사용할 것인지 인스턴스 구조로 사용할 것인지 확인합니다

InstanceDataStepRate: 다음 요소로 이동하기 위한 인스턴스 번호로 구성되어 있습니다.

#### DirectX12EnginePipeline.cpp(c++)

```
// 버텍스 버퍼 생성
{
    // 삼각형의 지오메트릭의 결정
    Vertex triangleVertices[] =
    {
        { { 0.0f, 0.25f * directX12_aspectRatio, 0.0f }, { 1.0f, 0.0f, 0.0f, 1.0f } }, // x = 0 y = 0.25 * 종횡비 맞추기
        { { 0.25f, -0.25f * directX12_aspectRatio, 0.0f }, { 0.0f, 1.0f, 0.0f, 1.0f } }, // x = 0.25 y = -0.25 * 종횡비 맞추기
        { { -0.25f, -0.25f * directX12_aspectRatio, 0.0f }, { 0.0f, 0.0f, 1.0f, 1.0f } } // x = -0.25 y = -0.25 * 종횡비 맞추기
    };

    const UINT vertexBufferSize = sizeof(triangleVertices);

    // GPU가 필요할때 마다 업로드 힘이 마샬링되기 때문에 버텍스 버퍼와 같은 정적데이터는 업로드 힘을 이용해야합니다.
    ThrowIfFailed(directX12_device->CreateCommittedResource(
        &keep(CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD)),
        D3D12_HEAP_FLAG_NONE,
        &keep(CD3DX12_RESOURCE_DESC::Buffer(vertexBufferSize)),
        D3D12_RESOURCE_STATE_GENERIC_READ,
        nullptr,
        IID_PPV_ARGS(&directX12_vertexBuffer));

    // 삼각형 데이터를 버텍스 버퍼에 복사합니다.
    UINT8* pVertexDataBegin;
    CD3DX12_RANGE readRange(0, 0); // CPU에서 이 리소스를 읽지 않는다
    ThrowIfFailed(directX12_vertexBuffer->Map(0, &readRange, reinterpret_cast<void*>(&pVertexDataBegin)));
    memcpy(pVertexDataBegin, triangleVertices, sizeof(triangleVertices));
    directX12_vertexBuffer->Unmap(0, nullptr);

    // 버텍스 버퍼 뷰어를 초기화함
    directX12_vertexBufferView.BufferLocation = directX12_vertexBuffer->GetGPUVirtualAddress();
    directX12_vertexBufferView.StrideInBytes = sizeof(Vertex);
    directX12_vertexBufferView.SizeInBytes = vertexBufferSize;
}
```

버텍스 버퍼를 사용하려면 GPU 로 가져와서 버텍스 버퍼의 리소스를 어셈블러에 바인딩을 해야합니다. GPU 로 가져오는 두가지 방법이 있습니다. 업로드 힘만 사용해서 각 프레임 마다 버텍스 버퍼를 GPU 에 업로드하는 방식입니다. 이것은 매 프레임마다 버텍스 버퍼를 램에서 비디오 메모리로 복사하기 때문에 속도가 느립니다. 따라서 일반적으로는

업로드 힙을 사용하여 버텍스 버퍼를 GPU 에 업로드한 후, 업로드 힙에서 기본 힙으로 데이터를 복사하는 방식을 채택합니다. 하지만 연습단계임으로 매프레임마다 버텍스 버퍼를 GPU 로 업로드 하는 방식으로 코드를 작성해봤습니다.

#### DirectX12EnginePipeline.cpp(c++)

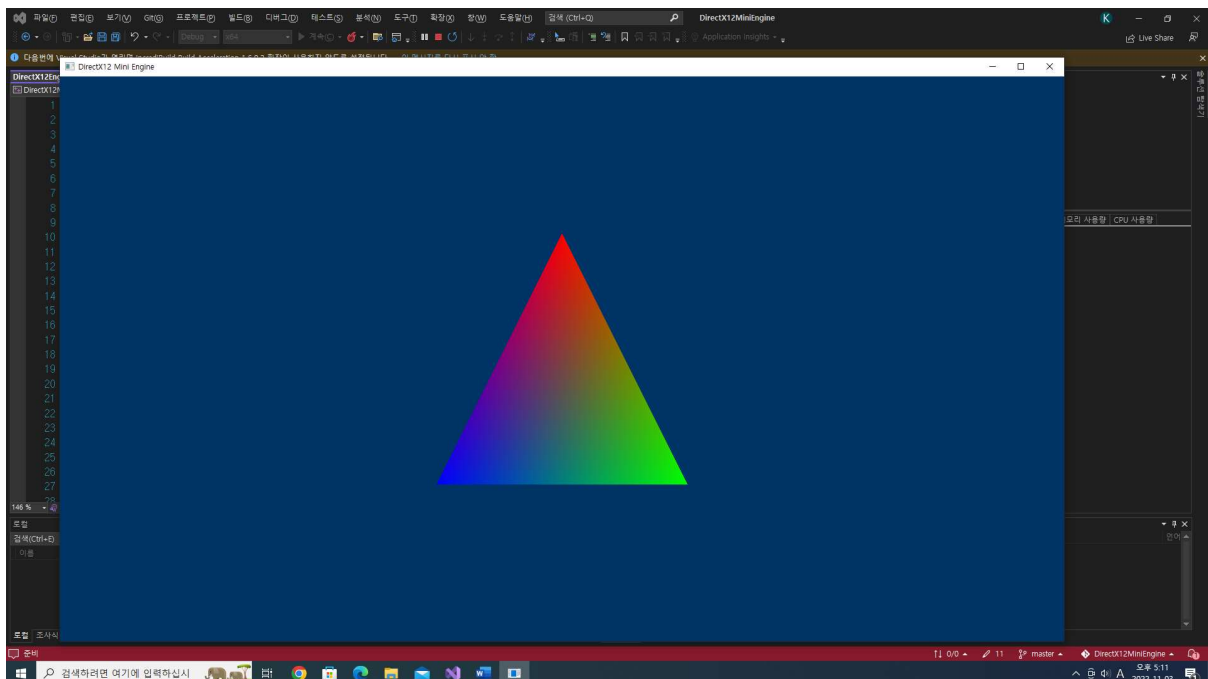
```
// 필요한 상태를 설정
directX12_commandList->SetGraphicsRootSignature(directX12_rootSignature.Get()); //그래픽스 루트 서명 설정
directX12_commandList->RSSetViewports(1, &directX12_viewport); // 뷰포트 설정
directX12_commandList->RSSetScissorRects(1, &directX12_scissorRect); // 화면 자르는 크기 설정

// 백버퍼가 렌더 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
directX12_commandList->OMSetRenderTargets(1, &rtvHandle, FALSE, nullptr); // 렌더 타겟 설정

// 명령을 기록(입력어셈블 단계)
const float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };
directX12_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr); // 렌더 타겟 뷰어 클리어
directX12_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST); //기본 유형(점 선 면)데이터 순서
directX12_commandList->IASetVertexBuffers(0, 1, &directX12_vertexBufferView); //버텍스 버퍼에 대한 CPU 핸들 설정
directX12_commandList->DrawInstanced(3, 1, 0, 0); //인덱싱 되지 않은 인스턴스 프리미티브 그리기
```

#### 4) 기본 삼각형 메쉬 결과



## 5) 코드 및 연구 - 번들

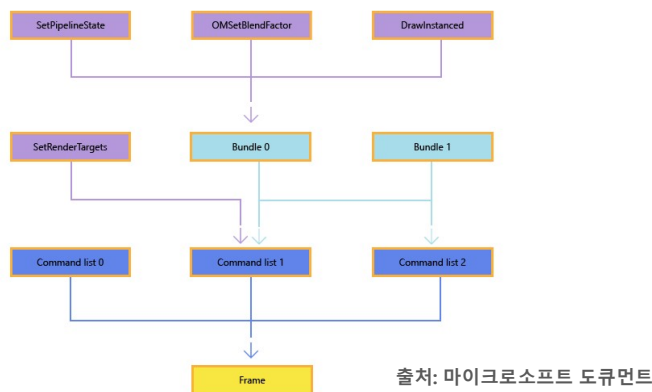
정적 삼각형을 보다 효율적으로 그리기 위해서 번들을 사용합니다.

## DirectX12EnginePipeline.h(c++)

```
ComPtr<ID3D12CommandAllocator> directX12_bundleAllocator; //번들 할당
ComPtr<ID3D12GraphicsCommandList> directX12_bundle; // 번들 만들기
```

## DirectX12EnginePipeline.cpp(c++)

```
ThrowIfFailed(directX12_device->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_BUNDLE,
IID_PPV_ARGS(&directX12_bundleAllocator))); //번들 할당 함
```



번들이란: 커맨드 리스트의 하위 레벨(수준)을 추가하여서 GPU 하드웨어의 기능을 활용하는 것입니다. 목적은 소수의 API 명령을 나중에 수행할 수 있도록 그룹화 하는 것입니다. 비용을 좀더 저렴하게 실행하기 위해서 사전 처리작업을 많이 사용합니다(유니티 에셋 번들도 같은 이야기입니다.)

## DirectX12EnginePipeline.cpp(c++)

```
// 번들을 만들고 기록
{
    ThrowIfFailed(directX12_device->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_BUNDLE,
directX12_bundleAllocator.Get(), directX12_pipelineState.Get(), IID_PPV_ARGS(&directX12_bundle)));
    directX12_bundle->SetGraphicsRootSignature(directX12_rootSignature.Get()); //그래픽스 루트 서명 설정
    //업데이트에 있는 리스트를 가져옴
    directX12_bundle->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST); //기본 유형(점 선 면)데이터 순서
    directX12_bundle->IASetVertexBuffers(0, 1, &directX12_vertexBufferView); //버텍스 버퍼에 대한 CPU 핸들 설정
    directX12_bundle->DrawInstanced(3, 1, 0, 0); //인덱싱 되지 않은 인스턴스 프리미티브 그리기
    ThrowIfFailed(directX12_bundle->Close());
}
```

OnRender()에 WaitForPreviousFrame 함수에 정의 되있는 커맨드 리스트 관련 정적 삼각형을 초기화 부분으로 가져와서 커맨드 리스트에 번들 담기

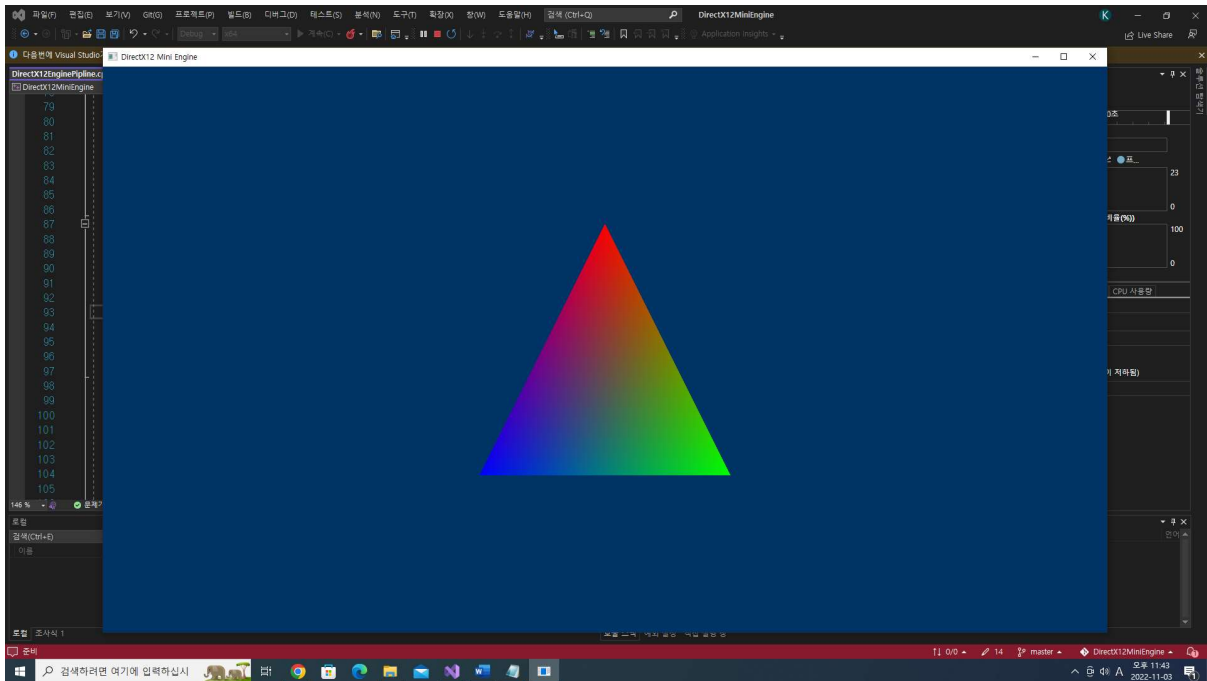
## DirectX12EnginePipeline.cpp(c++)

```
directX12_commandList->ExecuteBundle(directX12_bundle.Get()); //번들의 지정된 명령을 실행
```

Update 문에서 번들에 담았던 명령을 실행시킴



## 6) 번들 결과



## 7) 코드 및 연구 – 프레임 버퍼링

팬스와 할당자를 사용하여 GPU 에 프레임을 배정합니다.

## DirectX12EnginePipeline.h(c++)

```
ComPtr<ID3D12Resource> directX12_renderTargets[FrameCount]; //렌더타겟(프레임카운트)
ComPtr<ID3D12CommandAllocator> directX12_commandAllocator[FrameCount]; //커맨드 할당
UINT64 directX12_fenceValue[FrameCount]; // 팬스 값
void MoveToNextFrame();
void WaitForGpu();
```

## DirectX12EnginePipeline.cpp(c++)

```
//생성자
DirectX12EnginePipeline::DirectX12EnginePipeline(UINT width, UINT height, std::wstring name):DirectX12Base(width, height,
name),
directX12_frameIndex(0),
directX12_viewport(0.0f, 0.0f, static_cast<float>(width), static_cast<float>(height)),
directX12_scissorRect(0, 0, static_cast<LONG>(width), static_cast<LONG>(height)),
directX12_rtvDescriptorSize(0),
directX12_fenceValue{} //directX12_fenceValue(0)-> directX12_fenceValue{} 배열로 변경
{
}
```

## DirectX12EnginePipeline.cpp(c++)

```
//다음 프레임으로 이동
//WaitForPreviousFrame();
MoveToNextFrame();
```

## DirectX12EnginePipeline.cpp(c++)

```
//다음 프레임으로 이동
//WaitForPreviousFrame();
MoveToNextFrame();
```

## DirectX12EnginePipeline.cpp(c++)

```
// 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[i])));
    }
}
```

커멘드 할당자가 프레임 할당만큼 생겼음으로 이동

## DirectX12EnginePipeline.cpp(c++)

```
// 동기화 객체 생성후 GPU에 업로드 될때까지 기다림
{
    ThrowIfFailed(directX12_device->CreateFence(directX12_fenceValue[directX12_frameIndex], D3D12_FENCE_FLAG_NONE,
        IID_PPV_ARGS(&directX12_fence)));
    //directX12_fenceValue[directX12_frameIndex] = 1;
    directX12_fenceValue[directX12_frameIndex]++;

    // 프레임 동기화에 사용할 이벤트 핸들을 생성
    directX12_fenceEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);
    if (directX12_fenceEvent == nullptr)
    {
        ThrowIfFailed(HRESULT_FROM_WIN32(GetLastError()));
    }

    //명령 목록이 실행될때까지 기다림 루프는 완료하지만 설정이 완료될때까지 기다립니다.
    //WaitForPreviousFrame();
    WaitForGpu();
}
```

프레임이 준비될 때까지 기다림에서 GPU가 완료될 때까지 기다리는 형식으로 변경

## DirectX12EnginePipeline.cpp(c++)

```

void DirectX12EnginePipeline::PopulateCommandList()
{
    // 커맨드 리스트는 연결된 경우만 재설정 가능
    // 커맨드 리스트는 GPU처리를 완료했음으로 애플 실행
    // GPU처리를 위해 펜스 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()));

    // 필요한 상태를 설정
    DirectX12_commandList->SetGraphicsRootSignature(directX12_rootSignature->Get()); //그래픽스 루트 서명 설정
    DirectX12_commandList->RSSetViewports(1, &directX12_viewport); // 뷰포트 설정
    DirectX12_commandList->RSSetScissorRects(1, &directX12_scissorRect); // 화면 자르는 크기 설정

    // 백버퍼가 렌더 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
    DirectX12_commandList->OMSetRenderTargets(1, &rtvHandle, FALSE, nullptr); // 렌더 타겟 설정

    // 명령을 기록(입력어셈블 단계)
    const float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };
    DirectX12_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr); // 렌더 타겟 뷰어 클리어
    DirectX12_commandList->ExecuteBundle(directX12_bundle->Get()); //번들의 지정된 명령을 실행

    // 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());
}

```

## DirectX12EnginePipeline.cpp(c++)

```

// 보류중인 GPU 작업이 완료될때까지 기다림
void DirectX12EnginePipeline::WaitForGpu()
{
    // 커맨드 큐에서 시그널 신호를 예약
    ThrowIfFailed(directX12_commandQueue->Signal(directX12_fence->Get(), DirectX12_fenceValue[directX12_frameIndex]));

    // 펜스가 처리될때까지 기다림
    ThrowIfFailed(directX12_fence->SetEventOnCompletion(directX12_fenceValue[directX12_frameIndex], DirectX12_fenceEvent));
    WaitForSingleObjectEx(directX12_fenceEvent, INFINITE, FALSE);

    // 처리가 완료되면 현재 프레임의 펜스 값을증가시킴
    DirectX12_fenceValue[directX12_frameIndex]++;
}

```

## DirectX12EnginePipeline.cpp(c++)

```
// 다음 프레임 렌더링을 준비
void DirectX12EnginePipeline::MoveToNextFrame()
{
    // 대기열에서 커맨드 큐에 시그널 명령을 예약
    const UINT64 currentFenceValue = directX12_fenceValue[directX12_frameIndex]; //현재 값을 정의
    ThrowIfFailed(directX12_commandQueue->Signal(directX12_fence.Get(), currentFenceValue)); //현재 값을 대기열에 예약

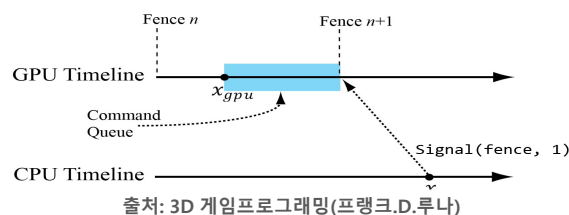
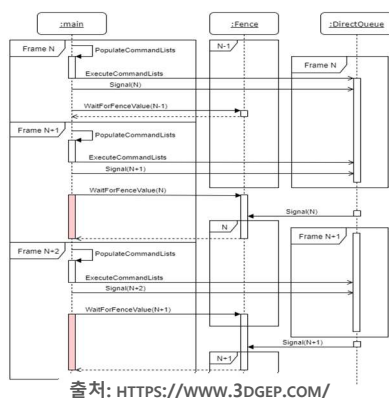
    // 프레임 인덱스를 업데이트
    directX12_frameIndex = directX12_swapChain->GetCurrentBackBufferIndex(); //백버퍼의 마지막 버퍼를 프레임인덱스로

    // 아직 프레임 렌더링할 준비가 안되어 있다면 기다리기
    if (directX12_fence->GetCompletedValue() < directX12_fenceValue[directX12_frameIndex])
    {
        ThrowIfFailed(directX12_fence->SetEventOnCompletion(directX12_fenceValue[directX12_frameIndex],
        directX12_fenceEvent));
        WaitForSingleObjectEx(directX12_fenceEvent, INFINITE, FALSE);
    }

    // 다음 프레임의 펜스값을 설정
    directX12_fenceValue[directX12_frameIndex] = currentFenceValue + 1;
}
```

Fence 를 이해하려면 GPU 와 CPU 의 동기화 부분을 먼저 이해를 해야 합니다.

렌더링 아키텍처를 보면 CPU 만으로도 이미지 렌더링을 할 수 있습니다. GPU 를 사용하는 이유는 GPU 를 사용해서 렌더링 하는 이점이 더 크기 때문입니다. GPU 는 더 많은 작업을 병렬로 처리할 수 있습니다. 또한 CPU 랑 달리 더 많은 그래픽 카드, 비디오 메모리 등 GPU 를 물리적으로 추가할 수 있습니다. 그럼 이런 많은 작업을 병렬로 처리 해주기 위해서는 동기화를 해 줘야 합니다. 그 이유는 교착상태와 기아상태 때문입니다. 멀티 코어를 사용하기 위해 멀티 쓰레드를 작성하여 동기화를 시키는 것과 같이, GPU 장치를 여러 개 사용하여 멀티 어댑터에 동기화 시켜야 합니다. DirectX12 부터는 이 부분을 좀 더 정교하게 작업할 수 있습니다.



위의 배경지식을 생각하고 밑에 시퀀스들을 보면 이해하기 더 편합니다. 결국 펜스 또는 울타리 라는 것은 프로그래머가 생각하는 GPU 의 작업 범위가 될 수 있습니다. 결국 CPU 는 GPU 의 상태를 계속 가지고 있는 것이 아니라 응답을 받아야 하기 때문입니다. (물론 엔비디아의 CUDA 프로그래밍은 다를 수 있습니다.) 펜스와 비슷한 예가 있는데, 그것은 컴퓨터와 컴퓨터가 통신할 수 있는 서버-클라이언트 방식의 패킷 시퀀스 번호입니다. 이런 식으로 좀더 쉽게 이해할 수 있습니다.

## 8) 프레임 버퍼링 결과

