2023-2학기 겜마루 Direct3D11 스터디

숭실대학교 겜마루 27기 컴퓨터학부 22학번 김민규

mkkim0612@naver.com

스터디 목적

- Direct3D의 다소 높은 학습 진입장벽을 완화하기 위함.
  - 삼각형 하나 그리는 데 코드 300줄 필요
  - CPU와 GPU의 실행 흐름과 메모리를 직접 관리해야 함
  - 로우 레벨 API
  - 엄청 좋은 자료X
  - 혼자 배우면 삽질하기 쉬움
  - 숭실대 포함 대부분의 학교에선 안 가르쳐 (경희대, 한국공대 등의 게임 특성화 학과에선 다룸) 배울 곳이 적음

스터디 운영 계획

- 장소 : 학교 빈 강의실 대여 -> 오프라인 희망자가 없어 디스코드 온라인으로 대체
- 시간 : 평일 저녁 대면으로 3회
  - 길게 잡으면 이탈자가 많을 것으로 예상해 짧게 끝내기

- 다룰 내용 : 수학과 기타 이론 최대한 배제, Direct3D 개념과 사용법에 집중
- 1회 : 배경 설명과 기초, 빈 화면 띄우기, 삼각형 그리기
- 2회 : 동적 버퍼, 텍스쳐, 조명
- 3회 : 미정

스터디 운영 계획

- 운영 방식 : 강의 형식
- 구성 : 이론 설명, 코드 설명, 코드 따라치기
  - 과제가 있으면 안 하니 현장에서 전부 해결
  - 코드를 직접 따라 쳐야 감이 잡힌다 생각

- 코드 형식 : 최대한 간단하게
  - C언어스러운 C++ 코드
  - 실패 검사 / 예외 처리 최소로
  - 화면 크기 변경 대응 X (실제 게임 만들 땐 필요하지만 D3D 개념 익히는 덴 상관없다 판단)
  - 한 함수를 길게 작성
  - 전역 변수 많이 사용

- 스터디 듣기 위한 요구 지식
  - 필수 : C언어 문법 - 함수, 포인터, 구조체 (딱 이것만 알면 됨) + 멤버 함수의 개념 (유니티 한번 써본 정도면 됨)
  - 반 필수 : 기초 렌더링 지식 - 조명이 대충 뭔지, 모델이 대충 뭔지 정도만
  - 알면 좋지만 몰라도 됨 : 선형대수, 렌더링 파이프라인, 셰이더

스터디 듣기 전 각오

- 어렵다.

  - 경희대 소프트 / 한국공대 게임공학과 3학년에서 다룸.

  - 숭실대에선 Direct3D보다 쉬운 OpenGL을 컴학 4-1 / 소프트 3-2 / 글미 3-1 에 다룸.

  - 최대한 쉽게 만들려고 했다. 그래도 어렵다.

  - 필요한 기반 지식이 많아 배울 게 많다 (제대로 하려면 컴퓨터구조, 자료구조, 선형대수, 컴퓨터그래픽스 이론, GPU구조, 멀티스레딩, C++에 대한 이해가 필요)

  - 코드 길이가 길어 흐름 파악이 힘들다. (삼각형 그리는데 300줄)

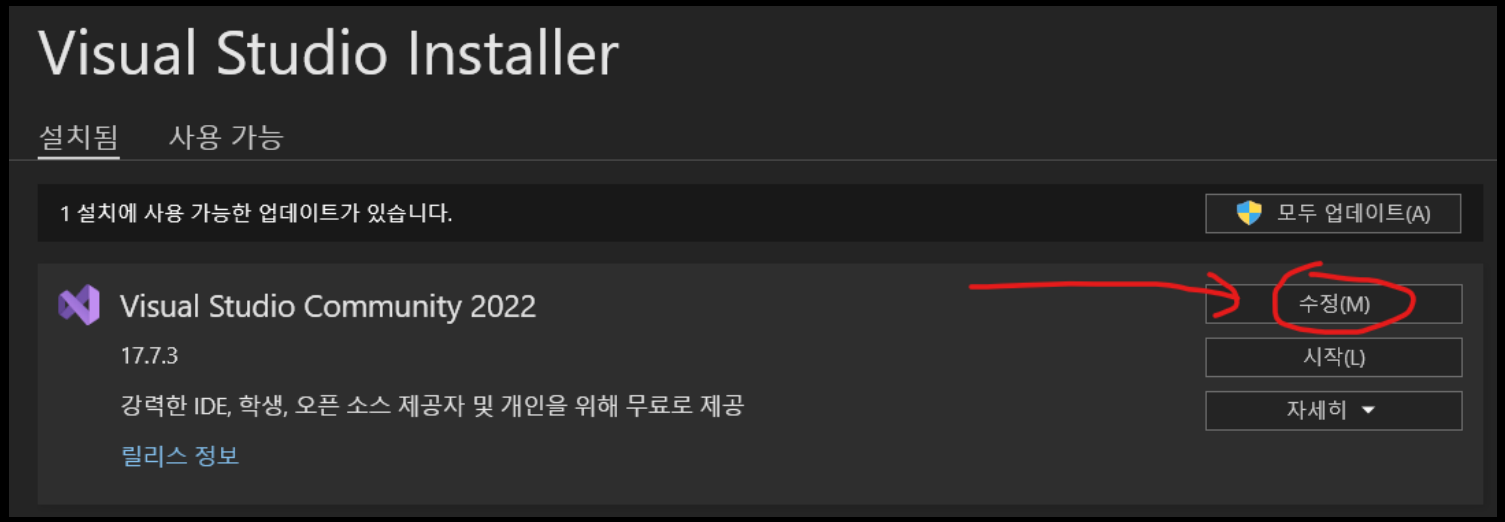
  - 그래도 1명이라도 끝까지 들으면 이득

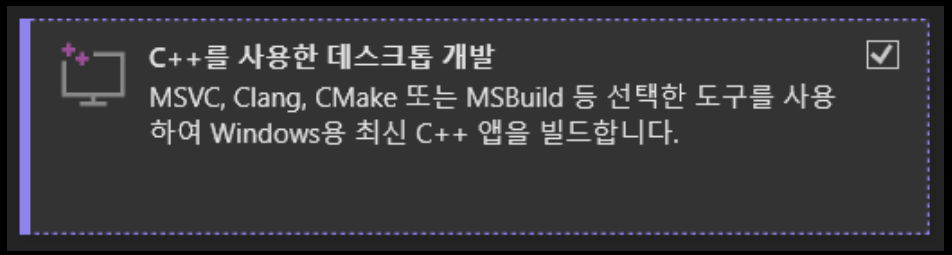
- 근데 만들고 보니 그렇게 어렵진 않을 수 있다

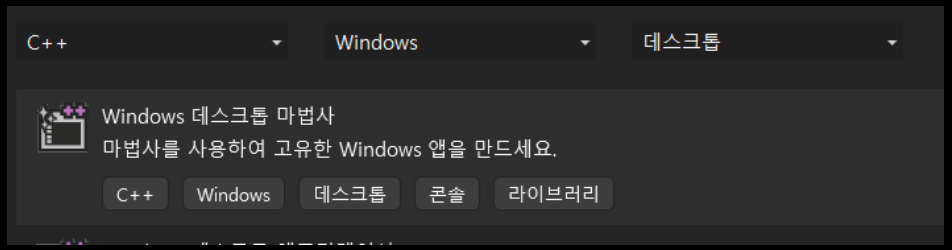
설치 준비 사항

- Windows 운영체제를 사용하는 컴퓨터 (macOS는 불가능)
- 그래픽 드라이버 최신으로 업데이트하기 (3d 게임 즐긴다면 안 해도 크게 문제없을 것)
- dxdiag 앱 실행해서 DirectX 버전 확인 (11 이상으로, 윈도우10이면 아마 12로 돼있을 것)

- Visual Studio 2022 설치 (낮은 버전도 상관없음)
- Visual Studio Installer - 수정 - C++를 사용한 데스크톱 개발







- [새 프로젝트 만들기] - [Windows 데스크톱 마법사] 존재하는지 확인

도움되는 사이트

- 공식 문서

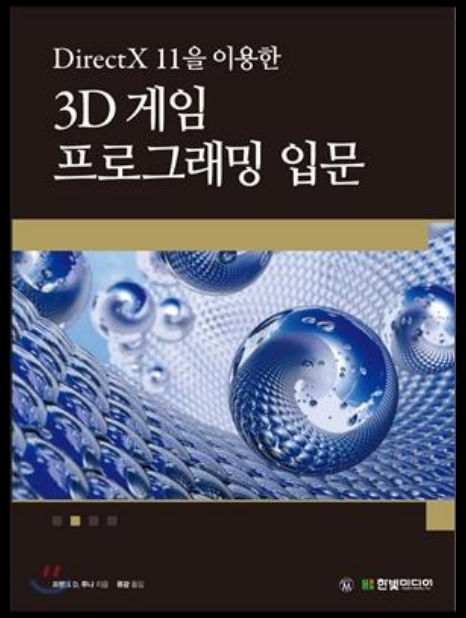
https://learn.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11

- DirectX SDK를 설치하면 Sample Browser를 통해 여러 튜토리얼 글과 샘플 코드를 볼 수 있다.

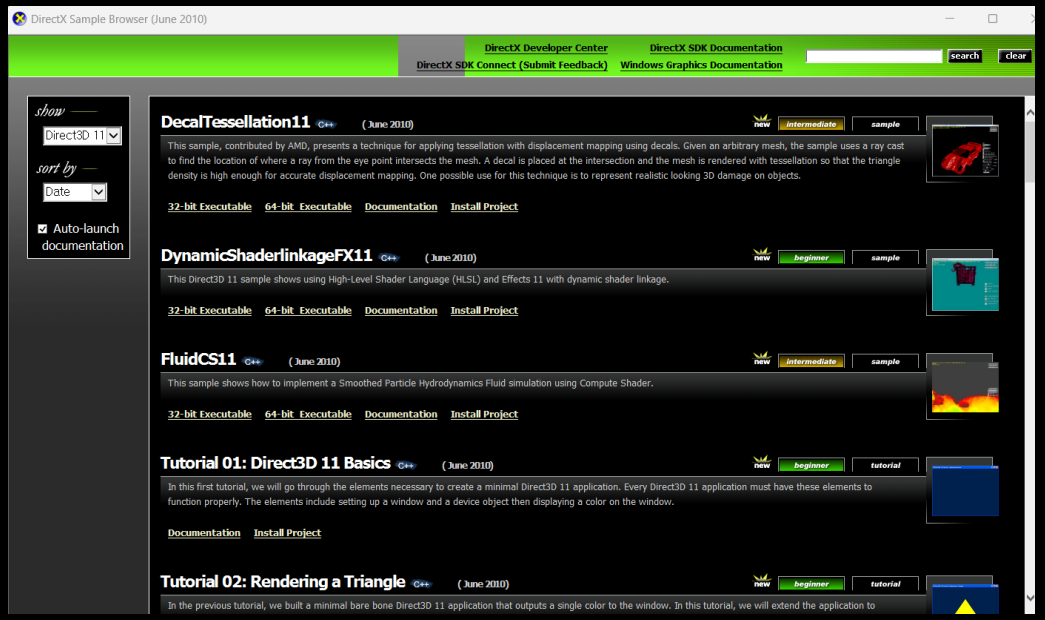
https://www.microsoft.com/en-US/download/details.aspx?id=6812

- 샘플 코드는 여기도 있다

https://github.com/walbourn/directx-sdk-samples/tree/main/Direct3D11Tutorials



-> 절판 (도서관에 존재)



Direct3D11 스터디
1회차 - hello world

- D3D에 대한 기본적 이해
- 렌더링 파이프라인의 이해
- 실습 - 빈 화면 띄우기
- 실습 - 삼각형 그리기

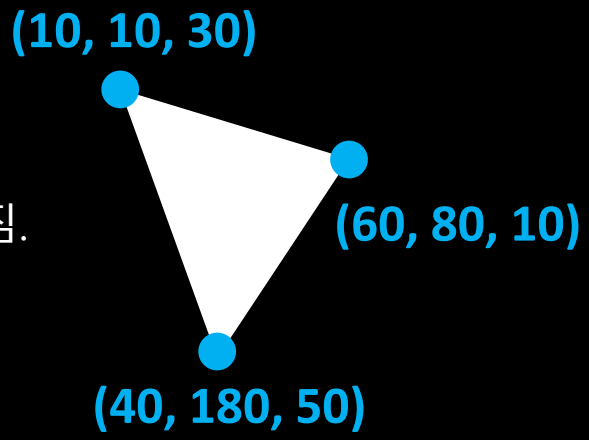
Direct3D11 스터디
Ch. 1 D3D에 대한 기본적 이해

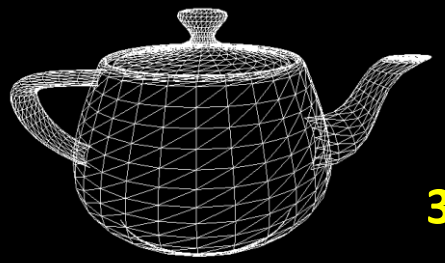
- 렌더링 파이프라인의 입력과 출력
- 렌더링 파이프라인
- GPU
- GPU 구조
- 그래픽스 라이브러리
- Direct3D
- 왜 D3D11인가?

렌더링 파이프라인의 입력과 출력

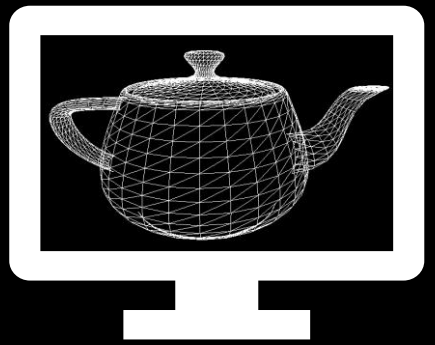
- 게임에서 어떤 것을 렌더링할까?

- 3D 게임이 가지고 있는 모델 데이터를 생각해 보자.
- 모델은 3차원 좌표를 가지는 삼각형들의 집합이다.
- 삼각형은 세 점으로 이루어지고 각 점은 3차원 좌표를 가짐.





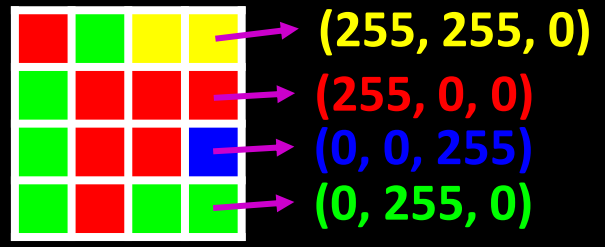
3D



2D

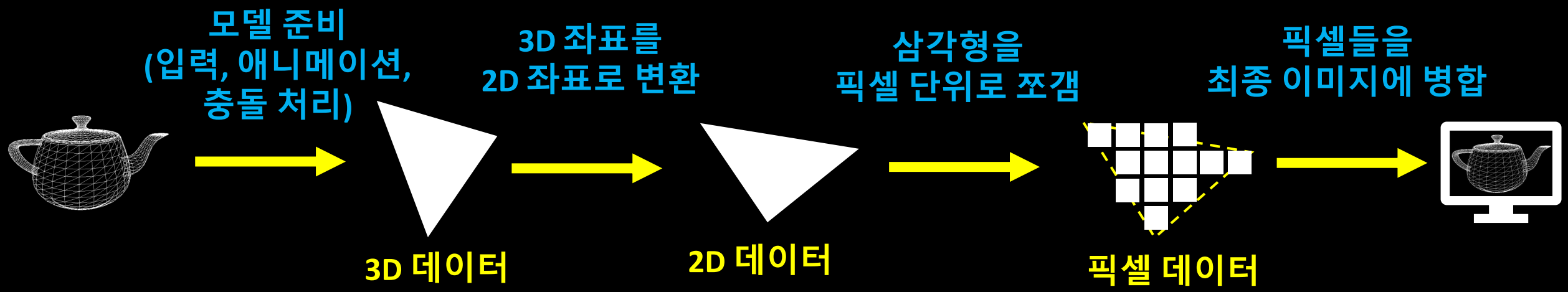
- 게임 장면이 최종 출력되는 화면은 2D 이미지이다.
- 각 픽셀이 rgb 색상을 가진다.

- 게임에서 3D 모델을 2D 이미지로 변환할 필요가 있다.
- 카메라의 방향, 캐릭터의 좌표나 애니메이션 등이 실시간으로 변하므로
- 실시간(1초에 적어도 20번 이상)으로 변환해야 한다.



렌더링 파이프라인

- 렌더링 파이프라인 : 3D 장면을 스크린의 2D 이미지로 변환하는 과정.



- 렌더링 파이프라인의 특징
  - 연산량이 매우 많다. (CPU가 감당하지 못할 정도)
  - 간단한 고정된 연산들(변환, 픽셀화, 병합 등)로 이루어져 있다

-> 렌더링 파이프라인의 고정되어 있는 연산들만을 처리하는 특별한 하드웨어가 있다면 어떨까?

-> 최초의 GPU 탄생

GPU

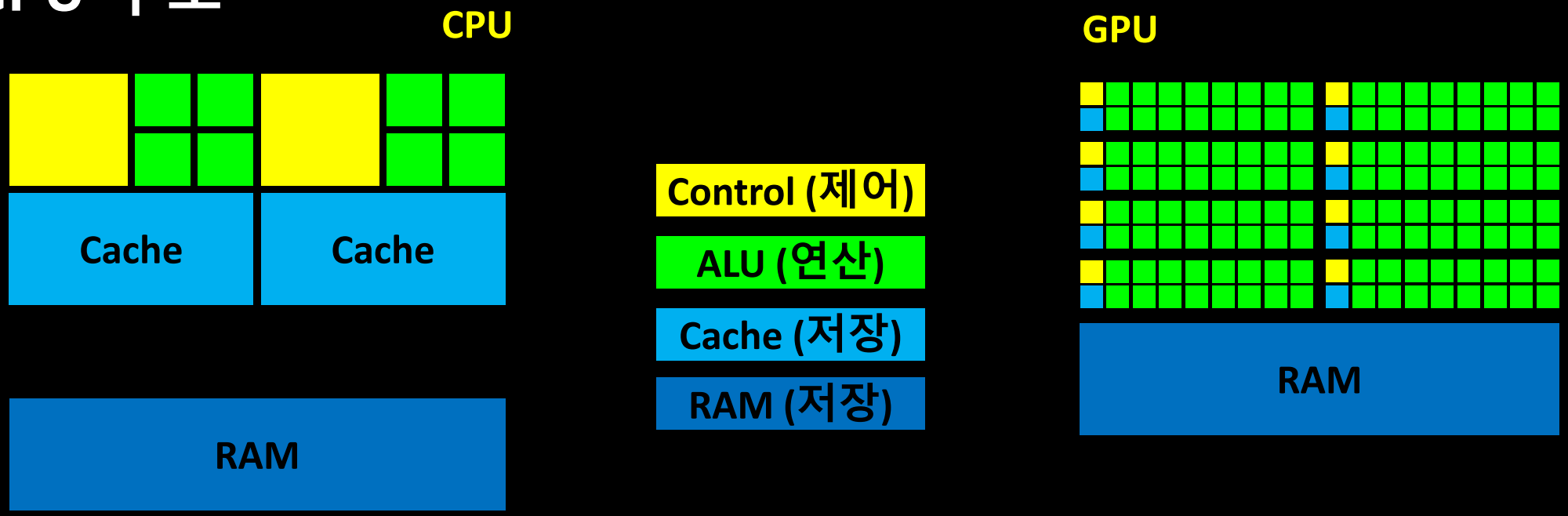
- 그래픽 카드 : 그래픽 출력을 만들어 모니터로 보내는 장치
- 그래픽 프로세싱 유닛(GPU) : 그래픽 카드의 구성 요소.
  그래픽 출력에 필요한 특정한 계산을 처리하기 위한 전자 회로.

- 초기 그래픽 카드 : Voodoo (1995, 3dfx), RIVA 128 (1997, nvidia), GeForce 256 (1999, nvidia) 등

Doom (1993)
2D로 3D를 모사

Quake (1996)
3D 그래픽 사용

GPU 구조



- CPU는 순차적으로 실행되는 단일 프로그램을 빨리 처리하고자 함.
  - 다양한 연산을 지원하고 메모리를 불규칙적으로 사용하기 위해 큰 제어부와 큰 캐시메모리를 가짐.
  - 분기 예측 / 캐시를 활용해 읽기/쓰기 속도로 인한 병목 제거

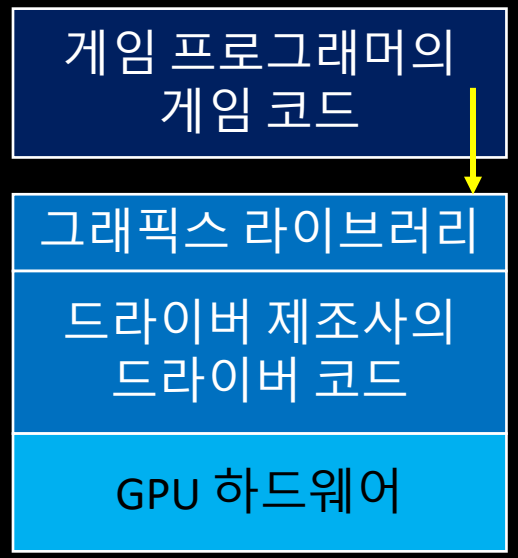
- GPU는 어느 정도 고정된 연산을 매우 많이 처리하고자 함.
  - 작은 제어부와 수많은 연산 유닛을 가짐.
  - 병렬성을 극도로 활용해 (수많은 삼각형/픽셀이 각각 따로 처리될 수 있음) 읽기/쓰기 속도로 인한 병목 제거
  - RAM이 내부에 박혀 있다. (빠른 메모리 읽기, 최신 RAM 장착)

그래픽스 라이브러리

- GPU는 특별한 구조를 가지고 있고 구조가 감춰져 있으며 자주 변한다.

-> 여러 GPU에 같은 방식으로 명령을 내리기 위한 도구가 필요하다.

-> 그래픽스 라이브러리 : GPU에 명령을 내리기 위한 도구



- 대표적 그래픽스 라이브러리 : Direct3D(Windows), OpenGL(통합), Vulkan(통합), Metal(Mac)

+ 요즘은 GPU가 범용적 목적으로 쓰이고(GPGPU) GPU에 명령을 내릴 수단은 그래픽스 라이브러리 외에도 많다.
(OpenCL, CUDA 등등)

Direct3D

- DirectX : 마이크로소프트 플랫폼에서 동작하는 멀티미디어 관련 API.
  - (DXGI, Direct2D, Direct3D, DirectWrite, DirectCompute, DirectSound, ...)
- Direct3D : DirectX의 일부. 3차원 그래픽스를 다루기 위한 API.
  - DirectX에서 Direct3D가 제일 유명해 둘이 같은 의미로 사용되기도 함.
  - Direct3D를 줄여서 D3D

왜 D3D11인가?

- D3D vs OpenGL
  - D3D는 마이크로소프트에서 관리하지만 OpenGL은 관여하는 기관이 많아 최신 경향을 늦게 따라간다.
  - OpenGL은 조금 더 수학적이고 렌더러를 소프트웨어적으로 설계하자는 느낌
  - D3D는 프로그래머가 GPU를 최대한 잘 사용할 수 있도록 해 주는 느낌
  - 사실 OpenGL도 산업에서 꽤 쓰는 것으로 알고 있다? (현업인이 아니라 잘 모름) D3D가 무조건 좋은 건 아니다

- D3D vs Vulkan : Vulkan 설계가 D3D12랑 비슷해 D3D12를 안하는 이유와 같음
- D3D vs Metal : 굳이 맥 쓸 이유가 없음

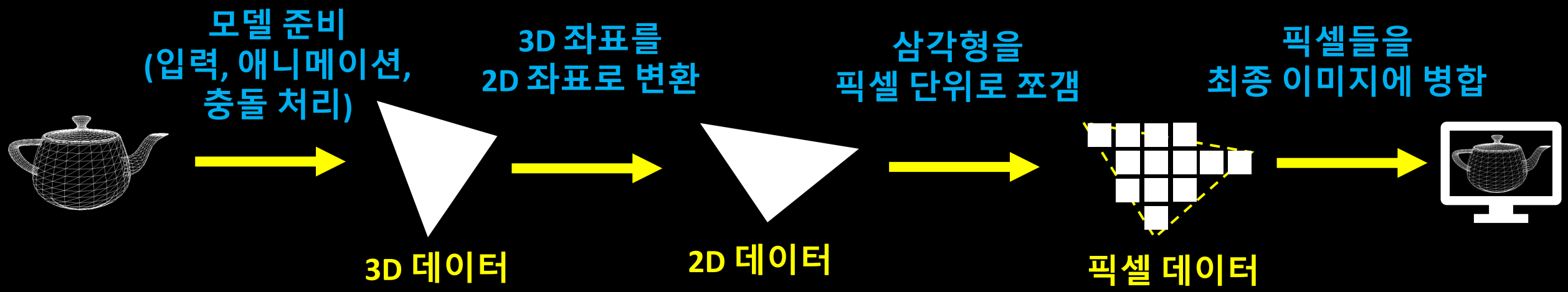
- D3D9 vs D3D10 vs D3D11 vs D3D12
  - D3D9는 너무 낡았다 (2002년, 나보다 늙음, 메이플보다 오래됨)
  - D3D10은 아무도 안 쓰는 망한 버전
  - D3D11은 지금 현역이고 아주 많이 사용되며 기능도 딱히 부족하지 않다
  - D3D12는 지나치게 어려운 반면 추가된 기능은 거의 없다 (거의 최적화 관련 변화 위주)

Direct3D11 스터디
Ch. 2 렌더링 파이프라인의 이해

- 렌더링 파이프라인
- Application
- Vertex Shader
- Rasterization
- Pixel Shader
- Depth Test
- 스왑 체인
- 필요한 자원들

렌더링 파이프라인

- 렌더링 파이프라인 : 3D 장면을 스크린의 2D 이미지로 변환하는 과정.



- 렌더링 파이프라인의 특징
  - 연산량이 매우 많다. (CPU가 감당하지 못할 정도)
  - 간단한 고정된 연산들(변환, 픽셀화, 병합 등)로 이루어져 있다

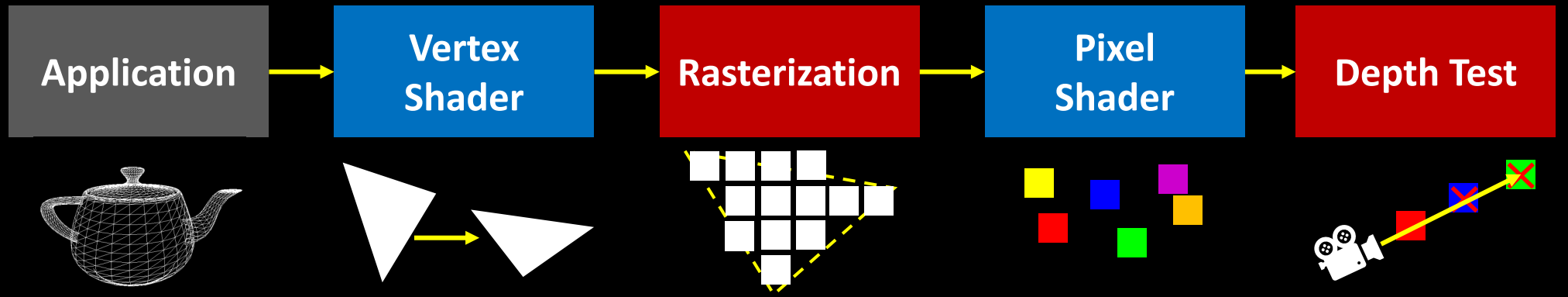
-> 렌더링 파이프라인의 고정되어 있는 연산들만을 처리하는 특별한 하드웨어가 있다면 어떨까?

-> 최초의 GPU 탄생

렌더링 파이프라인

- 렌더링 파이프라인 중 핵심적인 일부 과정만 설명.

CPU에서 처리
프로그램 가능
설정만 가능



CPU에서 입력, 애니메이션, 충돌 처리 등 계산해 모델을 GPU에 넘김

삼각형의 정점들의 좌표를 3D에서 2D로 변환하고, 그 과정에서 삼각형의 여러 값들을 계산

2D로 변환된 삼각형을 픽셀 단위로 쪼갬

픽셀들의 색을 계산함

카메라에서 보이는 픽셀만 남겨 최종 이미지에 저장

- 셰이더는 GPU에서 돌아가는 특별한 함수(C언어의 함수)라고 생각하면 된다.
- 버텍스(정점) 셰이더부터는 모두 GPU에서 수행된다.
- 더 복잡한 단계들도 많지만 생략

Application

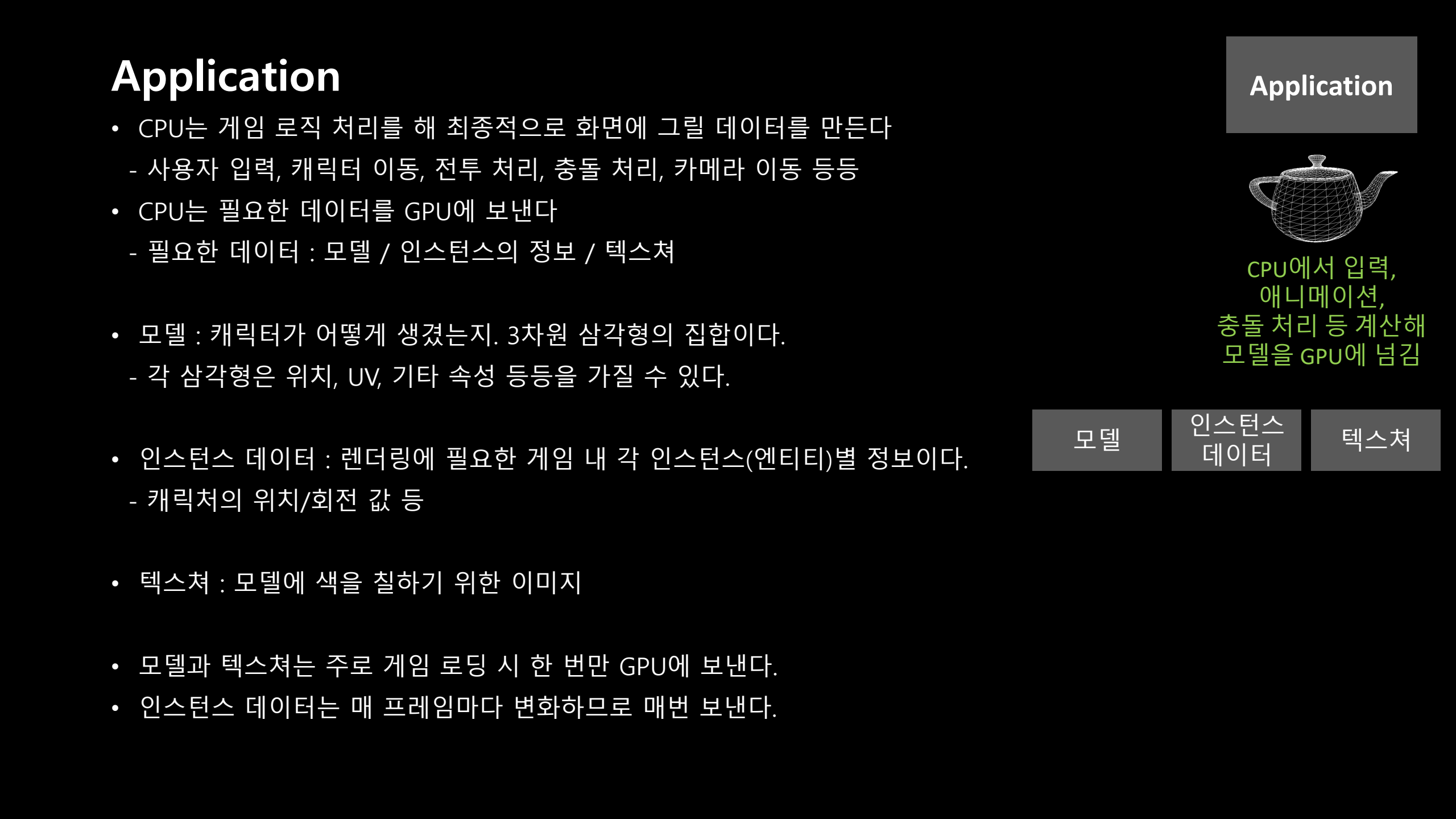
- CPU는 게임 로직 처리를 해 최종적으로 화면에 그릴 데이터를 만든다
  - 사용자 입력, 캐릭터 이동, 전투 처리, 충돌 처리, 카메라 이동 등등
- CPU는 필요한 데이터를 GPU에 보낸다
  - 필요한 데이터 : 모델 / 인스턴스의 정보 / 텍스쳐

- 모델 : 캐릭터가 어떻게 생겼는지. 3차원 삼각형의 집합이다.
  - 각 삼각형은 위치, UV, 기타 속성 등등을 가질 수 있다.

- 인스턴스 데이터 : 렌더링에 필요한 게임 내 각 인스턴스(엔티티)별 정보이다.
  - 캐릭처의 위치/회전 값 등

- 텍스쳐 : 모델에 색을 칠하기 위한 이미지

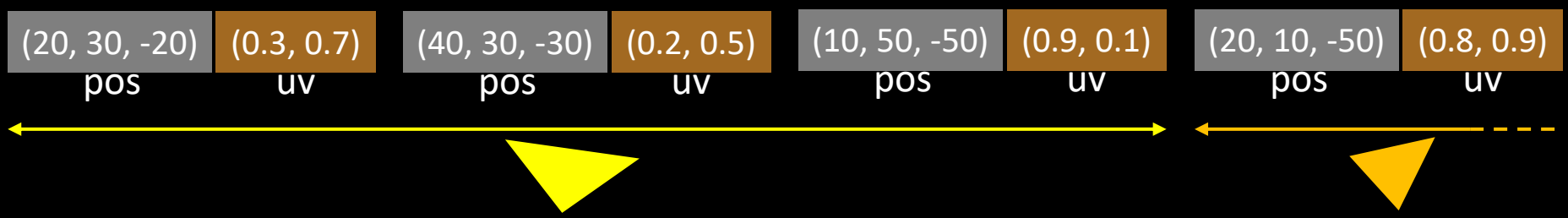
- 모델과 텍스쳐는 주로 게임 로딩 시 한 번만 GPU에 보낸다.
- 인스턴스 데이터는 매 프레임마다 변화하므로 매번 보낸다.

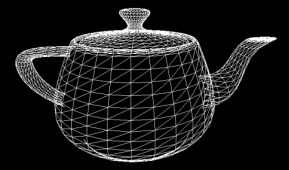


Application

- 모델은 3차원 삼각형의 집합이다. 삼각형은 정점 3개로 이루어진다. 정점은 여러 정보를 포함한다

(20, 30, -20)	(0.3, 0.7)	(40, 30, -30)	(0.2, 0.5)	(10, 50, -50)	(0.9, 0.1)	(20, 10, -50)	(0.8, 0.9)
pos	uv	pos	uv	pos	uv	pos	uv



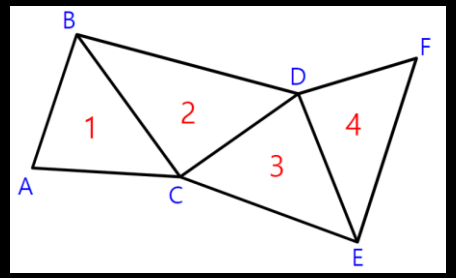


CPU에서 입력,
애니메이션,
충돌 처리 등 계산해
모델을 GPU에 넘김

Application

모델 | 인스턴스 데이터 | 텍스쳐

- 삼각형은 Strip의 형태로 삼각형당 한 정점만 보낼 수도 있다

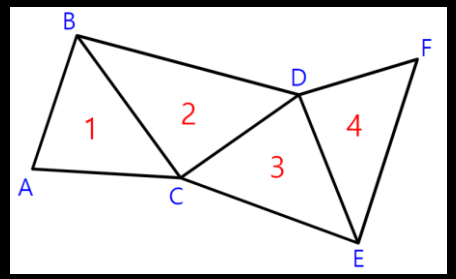


A B C B C D C D E D E F

A B C D E F

- 인덱스 데이터를 사용해 보낼 수도 있다

num	vertex	pos	uv
0	A	(20, 30, -20)	(0.9, 0.1)
1	B	(40, 30, -30)	(0.2, 0.5)
2	C	(10, 50, -50)	(0.8, 0.9)

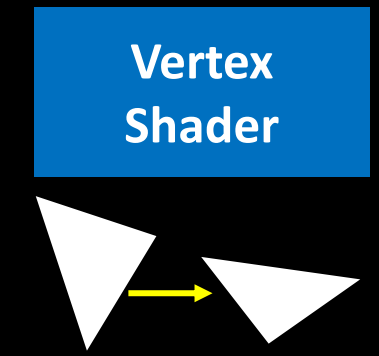


A B C B C D C D E D E F

0 1 2 1 2 3 2 3 4 3 4 5

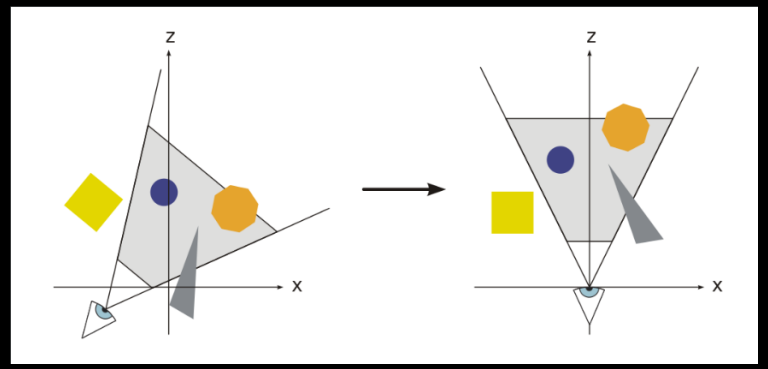
Vertex Shader

- 인스턴스 데이터의 값에 따라 모델을 알맞은 곳으로 변환한다
- 최종적으로는 삼각형들이 2D 좌표를 갖게 된다
- 좌표 변환 과정에서 색/조명/안개 등에 필요한 값들도 같이 계산한다



삼각형의 정점들의 좌표를 3D에서 2D로 변환하고,
그 과정에서 삼각형의 여러 값들을 계산

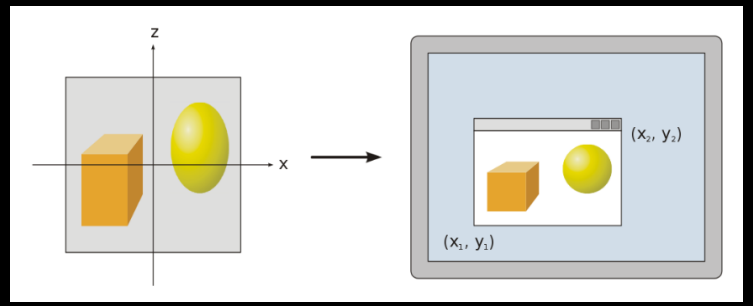
- 일단 카메라를 기준으로(카메라가 (0, 0, 0), 바라보는 곳이 +z) 이동시킨다(참고: d3d는 왼손좌표계)
  이때 z값이 카메라와의 거리가 되는데 뒤의 Depth Test에서 이 값을 사용한다



모델 | 인스턴스 데이터 | 텍스쳐

버텍스 셰이더

- 가까운 물체를 크게, 먼 물체를 작게 투영시킨다
- 화면의 해상도와 일치하는 크기의 2D 좌표로 변환한다



- 이 과정은 버텍스 셰이더를 통해 프로그래밍 가능함
- 버텍스(정점) 셰이더는 각 정점별로 적용된다

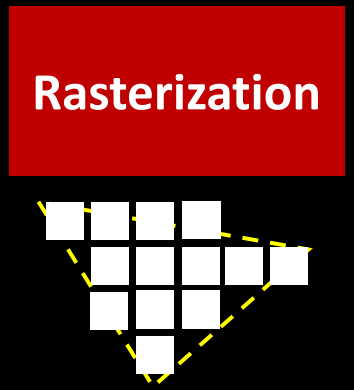
Rasterization

- 2D로 변환된 삼각형을 픽셀 단위로 쪼갠다.

- 쪼개기 전 화면 밖으로 벗어나는 부분은 제외됨 -> 화면 크기의 정보가 필요(뷰 포트)
- 또 거꾸로 뒤집혀 있는 삼각형도 제외될 수 있음

- 정점 세 개로 이루어진 삼각형은 픽셀(프래그먼트)들로 쪼개져
  파이프라인의 다음 과정으로 전달됨
- 정점에 들어있던 정보들(pos, uv, ...)이 픽셀에 전달될 때는
  픽셀의 삼각형 내 위치에 따라 보간되어 전달된다

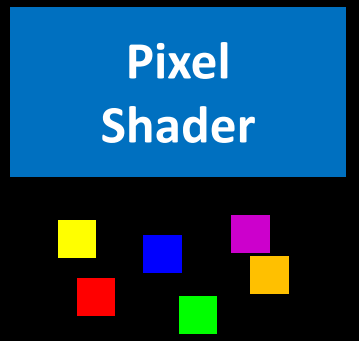
- 이 과정은 프로그래밍 불가능하며
  그래픽 카드에 이 작업을 담당하는 전용 하드웨어가 존재한다



2D로 변환된 삼각형을
픽셀 단위로 쪼갬

Pixel Shader

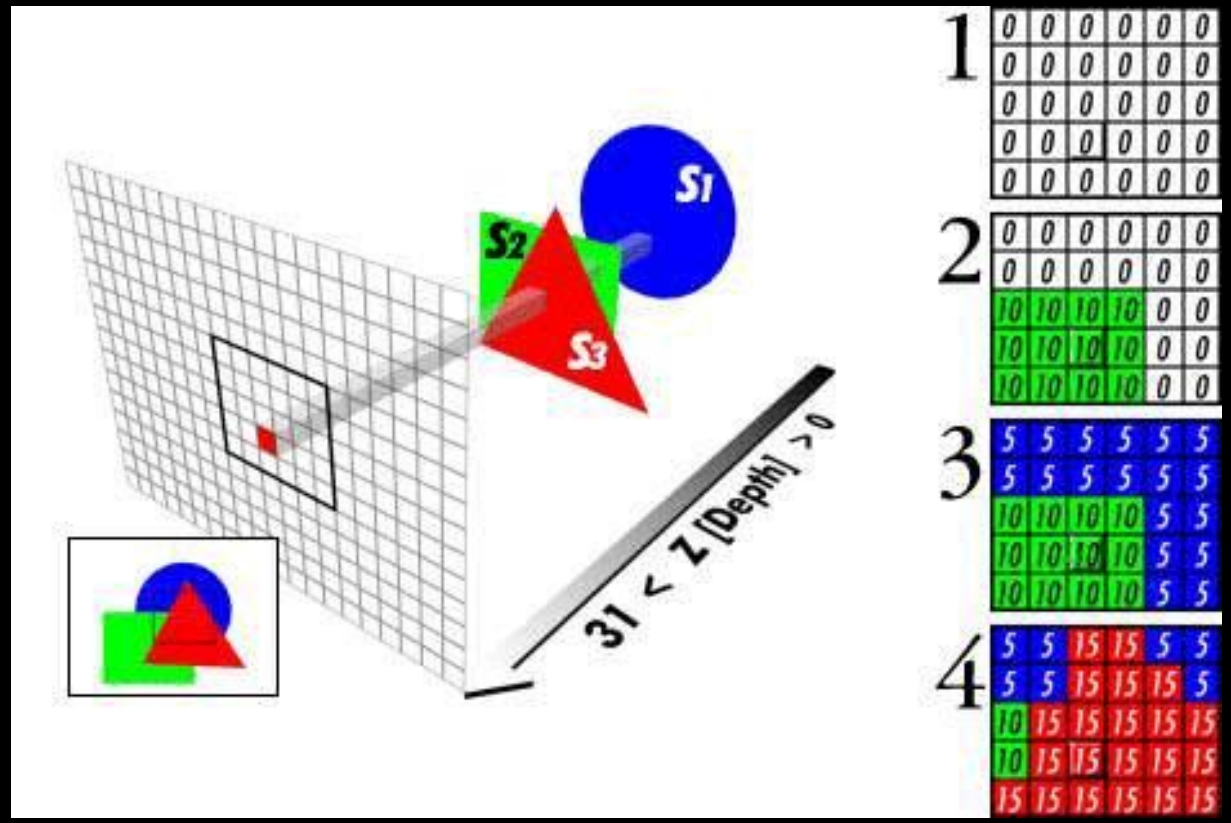
- 픽셀들에 색을 입힌다
- 픽셀 셰이더를 통해 프로그래밍 가능함
- 픽셀 셰이더는 각 픽셀별로 적용된다



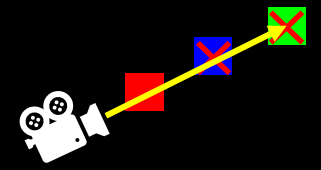
픽셀들의 색을 계산함

Depth Test

- 색을 칠한 픽셀들을 최종 화면(프레임 버퍼)에 담는다
- 두 삼각형이 겹쳐 있을 경우 어느 삼각형이 보여야 하는지 결정해야 한다
- 프레임 버퍼와 같은 크기의 z버퍼를 두고 여기에 현재 프레임 버퍼에 그려진 픽셀의 z값을 담는다
- 새 픽셀이 들어오면 픽셀의 z값과 z버퍼의 z값을 비교해
  새 픽셀이 카메라에 더 가까우면 프레임 버퍼와 z버퍼를 갱신한다



Depth Test



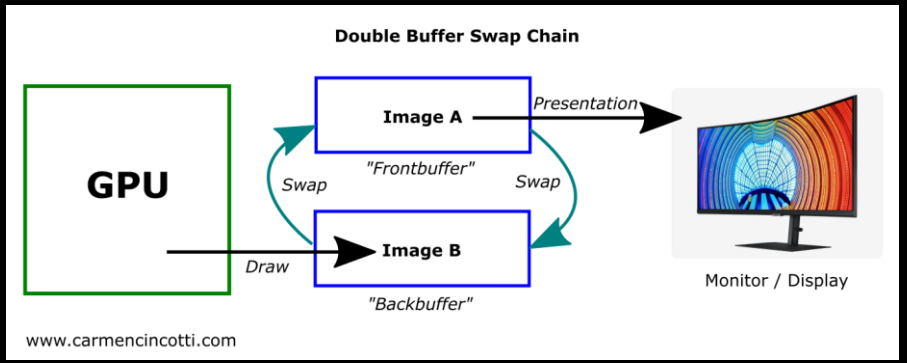
카메라에서 보이는
픽셀만 남겨
최종 이미지에 저장

모델	인스턴스 데이터	텍스쳐
버텍스 셰이더	뷰 포트	픽셀 셰이더
z-buffer	frame buffer	

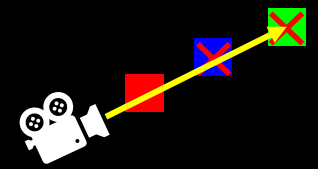
스왑 체인

- 프레임 버퍼가 하나만 존재할 경우 GPU가 프레임 버퍼에 쓸 때 모니터가 내용을 읽을 수 있다
- GPU와 모니터가 프레임 버퍼에 동시에 접근하면 이미지가 절반으로 잘린다(tearing)
- 테어링 시 반쪽엔 전 프레임의 이미지가, 나머지 반쪽엔 후 프레임의 이미지가 오게 됨

- 프레임 버퍼를 두 개 이상 사용해서 하나는 GPU가 그리고 하나는 모니터가 읽는다
- GPU가 모든 물체를 다 그리면 두 버퍼를 교환한다(swapping 혹은 flipping)
- GPU가 그리는 버퍼를 백(후면) 버퍼라고 한다. 다른 하나는 프론트(정면) 버퍼
- 이렇게 두 개 이상의 버퍼를 묶어서 스왑 체인이라고 부른다



Depth Test



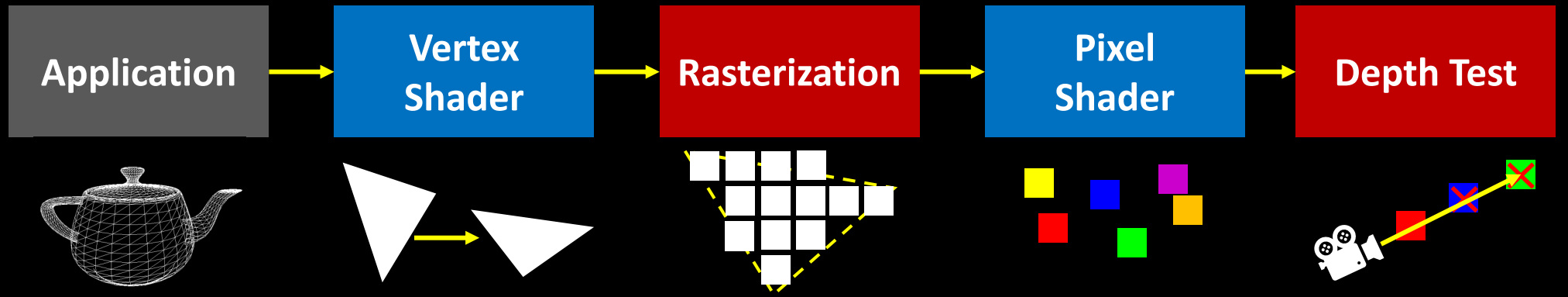
카메라에서 보이는
픽셀만 남겨
최종 이미지에 저장

모델	인스턴스 데이터	텍스쳐
버텍스 셰이더	뷰 포트	픽셀 셰이더
z-buffer	frame buffer	swap chain

렌더링 파이프라인

- 렌더링 파이프라인 중 핵심적인 일부 과정만 설명.

CPU에서 처리
프로그램 가능
설정만 가능



CPU에서 입력,
애니메이션,
충돌 처리 등 계산해
모델을 GPU에 넘김

삼각형의 정점들의
좌표를 3D에서 2D로
변환하고,
그 과정에서 삼각형의
여러 값들을 계산

2D로 변환된 삼각형을
픽셀 단위로 쪼갬

픽셀들의 색을 계산함

카메라에서 보이는
픽셀만 남겨
최종 이미지에 저장

모델	인스턴스 데이터	텍스쳐
버텍스 셰이더	뷰 포트	픽셀 셰이더
z-buffer	frame buffer	swap chain

필요한 자원들

모델	인스턴스 데이터	텍스쳐
버텍스 셰이더	뷰 포트	픽셀 셰이더
z-buffer	frame buffer	swap chain

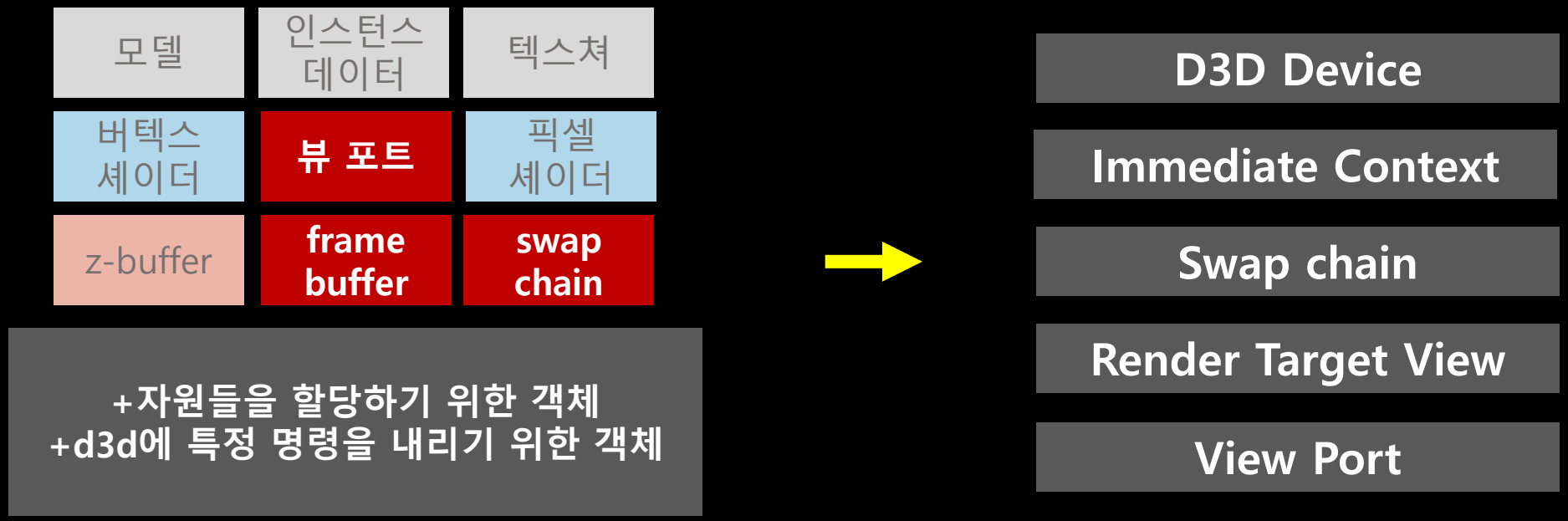
+자원들을 할당하기 위한 객체
+d3d에 특정 명령을 내리기 위한 객체

- 렌더링 파이프라인에 필요한 자원들을 할당하고 사용하는 것이 그래픽 라이브러리의 역할
- 어떤 자원이 필요한지 대략 알아봤으니 이제 필요한 모든 자원들을 구체적으로 알아보고 코드를 짜보자.

Direct3D11 스터디
Ch. 3 실습 - 빈 화면 띄우기

- 실제로 만들 자원들
- 1 - D3D Device
- 2 - Immediate Context
- 3 - Swap Chain
- 4 - Renter Target View
- 5 - View Port
- 실습

실제로 만들 자원들



- 일단 빈 화면을 띄우기 위해 필요한 것들만 만들자!

1 - D3D Device

- ID3D11Device 인터페이스
- 자원을 만들고 디스플레이 어댑터(그래픽 카드)의 능력을 열거하는 데 사용
- 모든 어플리케이션은 디바이스를 적어도 하나는 가져야 하고, 여러 개 가질 수도 있다.
- 각 디바이스는 한 개 이상의 컨텍스트를 사용할 수 있다.

2 - Immediate Context

- ID3D11DeviceContext 인터페이스
- Device Context는 디바이스의 상태를 가지고 있음
- Device Context는 파이프파인의 상태를 설정하고 렌더링 커맨드를 만드는 데 사용
- Device Context는 두 가지로 나뉨 -> Immediate, Deffered

- Immediate Context는 드라이버에 바로 렌더링함.
- 디바이스당 Immediate Context를 한 개만 가질 수 있음

- Deferred Context는 멀티 스레드 렌더링을 위해 렌더링 커맨드를 쌓다가 한번에 렌더링하기 위해 사용

- 스레딩 관련 주의사항 :

- 디바이스는 스레드 세이프
- 컨텍스트는 스레드 언세이프 (하나의 스레드로만 사용)

3 - Swap Chain

- IDXGISwapChain 인터페이스
- 출력하기 전의 렌더링된 데이터를 저장하는 버퍼들
- Direct3D로 대부분의 그래픽 계산을 하고 DXGI는 완성된 데이터를 모니터 등에 보내는 역할을 한다.

- 디바이스, 컨텍스트, 스왑 체인은 한 번에 만들 수 있다

4 - Render Target View

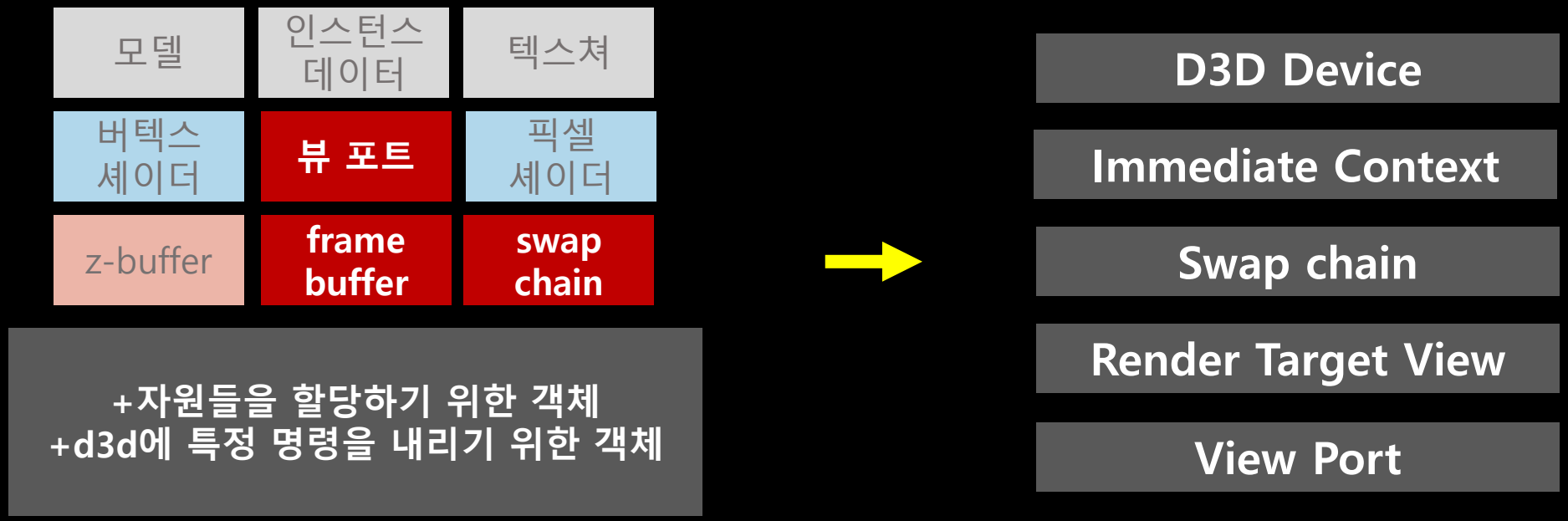
- ID3D11RenderTargetView 인터페이스
- 그래픽 파이프라인의 끝에서 픽셀들이 지정되는 버퍼에 최종적으로 그려질 수 있도록 한다

- 한 리소스가 파이프라인에서 여러 가지 용도로 쓰일 수 있다.
- 리소스가 어떤 용도로 쓰이는지 지정할 객체가 필요하다 -> Resource View

5 - View Port

- D3D11_VIEWPORT 구조체로 지정
- 화면의 크기를 정한다

실제로 만들 자원들



- 일단 빈 화면을 띄우기 위해 필요한 것들만 만들자!

실습 - 프로젝트 시작

- [Visual Studio] -> [새 프로젝트 만들기] -> [Windows 데스크톱 마법사]
- 프로젝트 생성 시 .cpp파일 하나와 그 파일에 간단한 코드가 있을 것임.

- [왼쪽 바 프로젝트] - [설정] - [구성 속성] - [링커] - [시스템] - [하위 시스템] 을 창(SUBSYSTEM:WINDOWS)로 변경

- .cpp파일에 코드 짜기 시작

실습 - win32 프로그래밍

- DirectX는 Windows 위에서 동작하므로 우선 Win32를 이용해 기본 틀을 만들어야 한다.
- 윈도우(창) 생성, 게임 루프, 이벤트 처리를 만들 것이다.

- 수많은 방식으로 만들 수 있지만 최대한 코드 양이 적고 간단한 것으로 선택

- `#include <Windows.h>` 로 사용

실습 - 이벤트 핸들러 함수

// ProjectName.cpp에 작성

```
#include <Windows.h>

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
    case WM_CLOSE:
        DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

실습 - 메인 함수

```
// ProjectName.cpp에 이어서 작성

int WINAPI WinMain(
    HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // 앞으로 이 안에다 추가 내용 작성
}
```

실습 - WNDCLASS, HWND

// ProjectName.cpp WinMain에 이어서 작성

```
WNDCLASS wc{};
wc.lpfnWndProc   = WndProc;
wc.hInstance     = hInstance;
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
wc.lpszClassName = L"WindowClass";

if (!RegisterClass(&wc))
    return FALSE;
```

```
HWND hwnd = CreateWindow(
    L"WindowClass", L"Title",
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    960, 540,
    NULL, NULL, hInstance, NULL);

if (!hwnd)
    return FALSE;

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
```

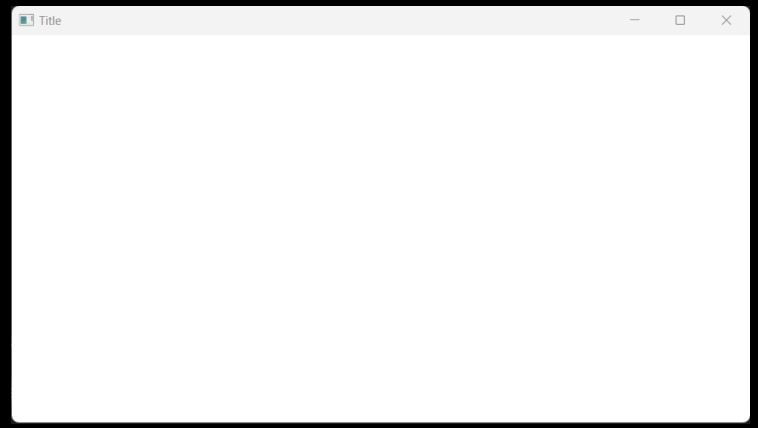
실습 - 메인 함수

// ProjectName.cpp WinMain에 이어서 작성

```
MSG msg{};
while(msg.message != WM_QUIT)
{
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // update, render
}

return msg.wParam;
```

실습 - Win32 완성

```cpp
#include <Windows.h>

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
    case WM_CLOSE:
        DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```




```cpp
int WINAPI WinMain(
    HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASS wc{};
    wc.lpfnWndProc   = WndProc;
    wc.hInstance     = hInstance;
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.lpszClassName = L"WindowClass";

    if (!RegisterClass(&wc))
        return FALSE;

    HWND hwnd = CreateWindow(
        L"WindowClass", L"Title",
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
        960, 540,
        NULL, NULL, hInstance, NULL);

    if (!hwnd)
        return FALSE;

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    MSG msg{};
    while(msg.message != WM_QUIT)
    {
        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        // update, render
    }
    return msg.wParam;
}
```

실습 - D3D cpp 파일 추가

- [소스 파일 우클릭] - [추가] - [새 항목] - [D3DDevice.cpp]

```cpp
// D3DDevice.cpp

#pragma comment (lib, "D3D11.lib")
#pragma comment (lib, "D3DCompiler.lib")
#include <Windows.h>
#include <d3d11.h>
#include <d3dcompiler.h>
#include <DirectXMath.h>

bool InitD3D(HWND hwnd)
{

}

void ClearD3D()
{

}

void Render()
{

}
```

실습 - D3D 함수 연결

```cpp
// ProjectName.cpp

#include <Windows.h>

bool InitD3D(HWND hwnd);
void ClearD3D();
void Render();
```

```cpp
if (!InitD3D(hwnd))
    return FALSE;

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);

MSG msg{};
while(msg.message != WM_QUIT)
{
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    Render();
}

ClearD3D();
```

실습 - 자원 전역 변수 선언

// D3DDevice.cpp에 작성

```
// D3DDevice.cpp

#include <Windows.h>
#include <d3d11.h>
#include <d3dcompiler.h>
#include <DirectXMath.h>

D3D_DRIVER_TYPE g_driverType = D3D_DRIVER_TYPE_HARDWARE;
D3D_FEATURE_LEVEL g_featureLevel = D3D_FEATURE_LEVEL_11_0;

ID3D11Device* g_pd3dDevice = NULL;
ID3D11DeviceContext* g_pImmediateContext = NULL;
IDXGISwapChain* g_pSwapChain = NULL;
ID3D11RenderTargetView* g_pRenderTargetView = NULL;
```

실습 - 초기화 - 준비

// D3DDevice.cpp에 InitD3D에 이어서 작성

```
// D3DDevice.cpp, function bool InitD3D(HWND hwnd)


HRESULT hr = S_OK;

RECT rc;
GetClientRect(hwnd, &rc);
UINT width = rc.right - rc.left;
UINT height = rc.bottom - rc.top;

UINT createDeviceFlags = 0;

#ifdef _DEBUG
createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

D3D_FEATURE_LEVEL featureLevel = D3D_FEATURE_LEVEL_11_0;
```

실습 - 초기화 - 디바이스/컨텍스트/스왑체인 얻기

```
DXGI_SWAP_CHAIN_DESC sd;                    // D3DDevice.cpp에 InitD3D에 이어서 작성
ZeroMemory(&sd, sizeof(sd));
sd.BufferCount = 1;
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = hwnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

hr = D3D11CreateDeviceAndSwapChain(
    NULL, g_driverType, NULL, createDeviceFlags, &featureLevel, 1,
    D3D11_SDK_VERSION, &sd,
    &g_pSwapChain, &g_pd3dDevice, &g_featureLevel, &g_pImmediateContext);

if(FAILED(hr))
    return false;
```

실습 - 초기화 - 렌더타깃뷰 얻기

// D3DDevice.cpp에 InitD3D에 이어서 작성

```
ID3D11Texture2D* pBackBuffer = NULL;
hr = g_pSwapChain->GetBuffer(
    0, __uuidof(ID3D11Texture2D), (LPVOID*)&pBackBuffer);

if(FAILED(hr))
    return false;

hr = g_pd3dDevice->CreateRenderTargetView(
    pBackBuffer, NULL, &g_pRenderTargetView);

pBackBuffer->Release();

if (FAILED(hr))
    return false;

g_pImmediateContext->OMSetRenderTargets(1, &g_pRenderTargetView, NULL);
```

실습 - 초기화 - 뷰포트 만들기

// D3DDevice.cpp에 InitD3D에 이어서 작성

```
D3D11_VIEWPORT vp;
vp.Width = (FLOAT)width;
vp.Height = (FLOAT)height;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pImmediateContext->RSSetViewports(1, &vp);

return true;
```

실습 - 화면 클리어 코드 작성

// D3DDevice.cpp에 작성

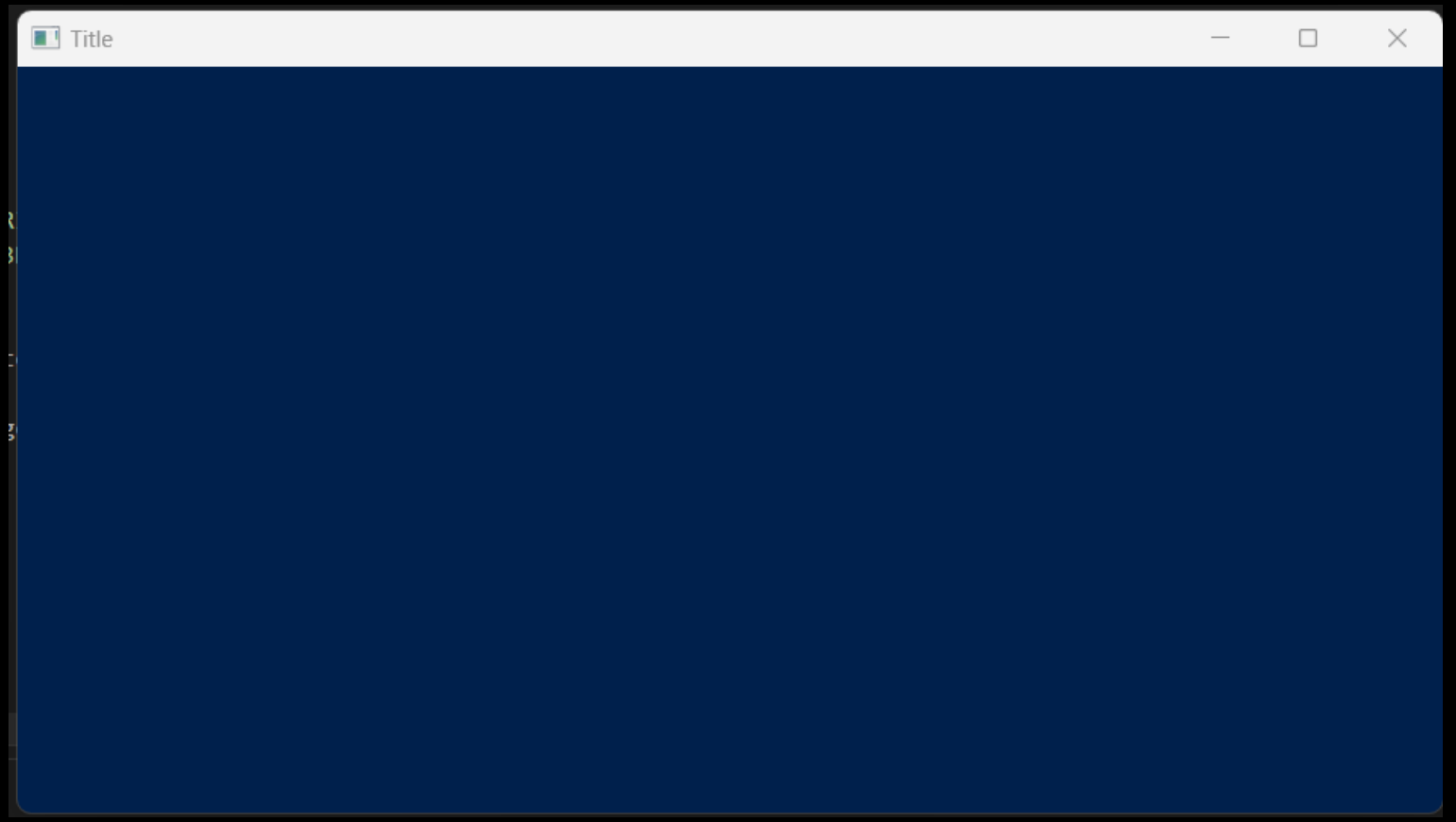
```
void Render()
{
    float clearColor[4] = { 0.0f, 0.125f, 0.3f, 1.0f };
    g_pImmediateContext->ClearRenderTargetView(g_pRenderTargetView, clearColor);
    g_pSwapChain->Present(0, 0);
}
```

실습 - 자원 해제

// D3DDevice.cpp에 작성

```
// D3DDevice.cpp

void ClearD3D()
{
    if (g_pImmediateContext != NULL) g_pImmediateContext->ClearState();

    if (g_pRenderTargetView != NULL) g_pRenderTargetView->Release();
    if (g_pSwapChain        != NULL) g_pSwapChain->Release();
    if (g_pImmediateContext != NULL) g_pImmediateContext->Release();
    if (g_pd3dDevice        != NULL) g_pd3dDevice->Release();
}
```

실습 - 실행

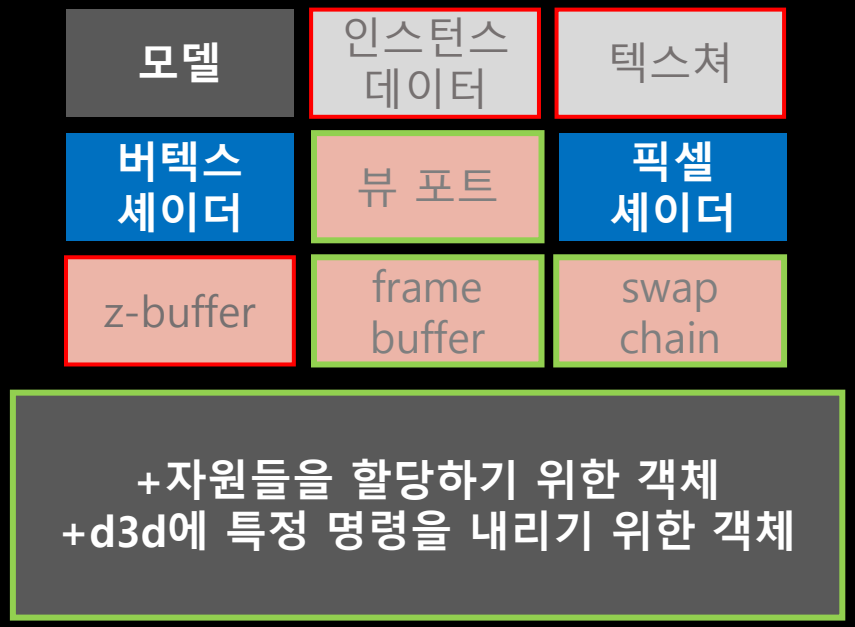


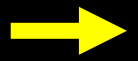
코드 다시 리뷰!

Direct3D11 스터디
Ch. 4 실습 - 삼각형 그리기

- 실제로 만들 자원들
- 1 - Vertex Shader, Pixel Shader
- 2 - Input Layout
- 3 - Vertex Buffer
- 실습

실제로 만들 자원들





Vertex Shader

Pixel Shader

Input Layout

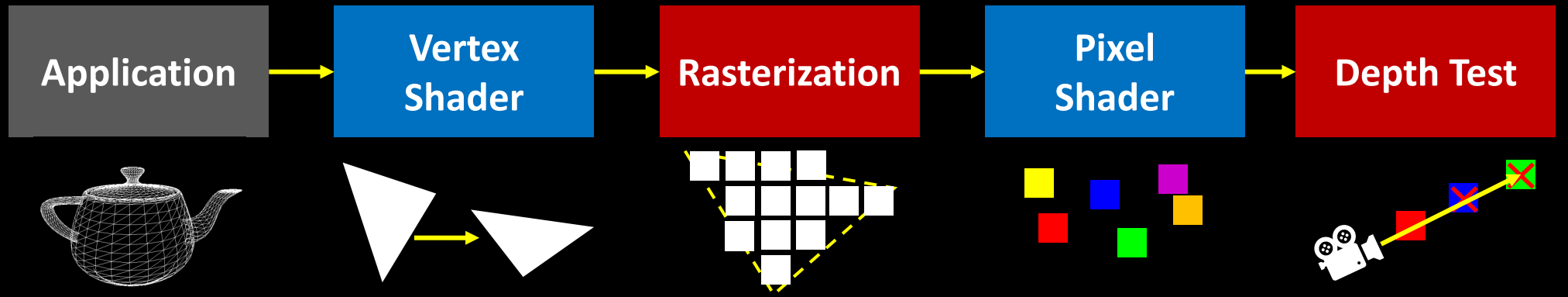
Vertex Buffer

- 삼각형을 그리는 데 추가적으로 필요한 자원들을 만들자!

1 - Vertex Shader, Pixel Shader

- 렌더링 파이프라인 중 핵심적인 일부 과정만 설명.

CPU에서 처리
프로그램 가능
설정만 가능



CPU에서 입력, 애니메이션, 충돌 처리 등 계산해 모델을 GPU에 넘김

삼각형의 정점들의 좌표를 3D에서 2D로 변환하고, 그 과정에서 삼각형의 여러 값들을 계산

2D로 변환된 삼각형을 픽셀 단위로 쪼갬

픽셀들의 색을 계산함

카메라에서 보이는 픽셀만 남겨 최종 이미지에 저장

- 셰이더는 GPU에서 돌아가는 특별한 함수(C언어의 함수)라고 생각하면 된다.
- 버텍스(정점) 셰이더부터는 모두 GPU에서 수행된다.
- 더 복잡한 단계들도 많지만 생략

- ID3DBlob이라는 크기와 포인터를 얻을 수 있는 간단한 인터페이스로 만들 것임

2 - Input Layout

- 버텍스 셰이더는 GPU에서 삼각형들을 처리하는 셰이더이다.
- 버텍스 셰이더에 입력되는 삼각형들은 CPU에서 보내주게 된다.
- CPU에서 보내주는 데이터가 어떤 의미를 가지는지(타입이 뭔지 등) 알려줄 필요가 있다.

- Input Layout을 만들 때 셰이더 정보를 같이 줘서 셰이더의 입력 정보와 일치하는지 검사한다.
- 다른 Vertex Shader의 입력 형태가 같으면 거기에도 같은 Input Layout 객체를 사용할 수 있다.

3 - Vertex Buffer

- 정점들의 정보(위치, 노멀, UV 등)를 담는 기본적인 버퍼

실습 - 자원 전역 변수 선언

// D3DDevice.cpp에 #include 아래 이어서 작성

```
// D3DDevice.cpp

ID3D11VertexShader* g_pVertexShader = NULL;
ID3D11PixelShader* g_pPixelShader = NULL;
ID3D11InputLayout* g_pInputLayout = NULL;
ID3D11Buffer* g_pVertexBuffer = NULL;
```

실습 - 자원 해제

// D3DDevice.cpp에 작성

```
// D3DDevice.cpp

void ClearD3D()
{
    if (g_pImmediateContext != NULL) g_pImmediateContext->ClearState();

    if (g_pVertexBuffer     != NULL) g_pVertexBuffer->Release();
    if (g_pInputLayout      != NULL) g_pInputLayout->Release();
    if (g_pPixelShader      != NULL) g_pPixelShader->Release();
    if (g_pVertexShader     != NULL) g_pVertexShader->Release();

    ...
}
```

실습 - 셰이더 만들기

// D3DDevice.cpp에 InitD3D에
return true; 위에 이어서 작성

```
DWORD dwShaderFlags = D3DCOMPILE_ENABLE_STRICTNESS;

#ifdef _DEBUG
dwShaderFlags |= D3DCOMPILE_DEBUG;
#endif

ID3DBlob* pErrorBlob = NULL;
```

실습 - 버텍스 셰이더 만들기

// D3DDevice.cpp에 InitD3D에
return true; 위에 이어서 작성

```
ID3DBlob* pVSBlob = NULL;
hr = D3DCompileFromFile(
    L"Shader1.fx", NULL, D3D_COMPILE_STANDARD_FILE_INCLUDE,
    "VS", "vs_4_0",
    dwShaderFlags, 0, &pVSBlob, &pErrorBlob);
if (FAILED(hr))
{
    if (pErrorBlob != NULL)
        OutputDebugStringA((char*)pErrorBlob->GetBufferPointer());
    MessageBox(NULL, L"Vertex Shader Compile Failed", L"Error", MB_OK);
    return false;
}
if (pErrorBlob != NULL)
{
    pErrorBlob->Release();
    pErrorBlob = NULL;
}
hr = g_pd3dDevice->CreateVertexShader(pVSBlob->GetBufferPointer(), pVSBlob->GetBufferSize(),
    NULL, &g_pVertexShader);
if (FAILED(hr))
{
    pVSBlob->Release();
    return false;
}
```

실습 - 픽셀 셰이더 만들기

// D3DDevice.cpp에 InitD3D에
return true; 위에 이어서 작성

```
ID3DBlob* pPSBlob = NULL;
hr = D3DCompileFromFile(
    L"Shader1.fx", NULL, D3D_COMPILE_STANDARD_FILE_INCLUDE,
    "PS", "ps_4_0",
    dwShaderFlags, 0, &pPSBlob, &pErrorBlob);
if (FAILED(hr))
{
    if (pErrorBlob != NULL)
        OutputDebugStringA((char*)pErrorBlob->GetBufferPointer());
    MessageBox(NULL, L"Pixel Shader Compile Failed", L"Error", MB_OK);
    return false;
}
if (pErrorBlob != NULL)
{
    pErrorBlob->Release();
    pErrorBlob = NULL;
}
hr = g_pd3dDevice->CreatePixelShader(pPSBlob->GetBufferPointer(), pPSBlob->GetBufferSize(),
    NULL, &g_pPixelShader);
if (FAILED(hr))
{
    pPSBlob->Release();
    return false;
}
```

실습 - 셰이더 코드 짜기

// Shader1.fx에 작성

- [솔루션 탐색기] - [리소스 파일 우클릭] - [추가] - [새 항목] - [Shader1.fx]
- [만든 파일 우클릭] - [속성] - [일반] - [빌드에서 제외] - [예]

```
float4 VS(float4 Pos : POSITION) : SV_POSITION
{
    return Pos;
}

float4 PS(float4 Pos : SV_POSITION) : SV_Target
{
    return float4(1.0f, 1.0f, 0.0f, 1.0f);
}
```

실습 - 인풋 레이아웃 만들기

// D3DDevice.cpp에 InitD3D에
return true; 위에 이어서 작성

```
D3D11_INPUT_ELEMENT_DESC layouts[1];
UINT numElements = ARRAYSIZE(layouts);
layouts[0].SemanticName = "POSITION";
layouts[0].SemanticIndex = 0;
layouts[0].Format = DXGI_FORMAT_R32G32B32_FLOAT;
layouts[0].InputSlot = 0;
layouts[0].AlignedByteOffset = 0;
layouts[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
layouts[0].InstanceDataStepRate = 0;

hr = g_pd3dDevice->CreateInputLayout(
    layouts, numElements,
    pVSBlob->GetBufferPointer(), pVSBlob->GetBufferSize(),
    &g_pInputLayout);

pVSBlob->Release();
pPSBlob->Release();
if (FAILED(hr))
    return false;

g_pImmediateContext->IASetInputLayout(g_pInputLayout);
```

실습 - 버텍스 버퍼 만들기

// D3DDevice.cpp에 InitD3D에
return true; 위에 이어서 작성

```
DirectX::XMFLOAT3 vertices[] =
{
    DirectX::XMFLOAT3(0.0f, 0.5f, 0.5f),
    DirectX::XMFLOAT3(0.5f, -0.5f, 0.5f),
    DirectX::XMFLOAT3(-0.5f, -0.5f, 0.5f),
};

D3D11_BUFFER_DESC bd;
ZeroMemory(&bd, sizeof(bd));
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(DirectX::XMFLOAT3) * 3;
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;

D3D11_SUBRESOURCE_DATA InitData;
ZeroMemory(&InitData, sizeof(InitData));
InitData.pSysMem = vertices;

hr = g_pd3dDevice->CreateBuffer(&bd, &InitData, &g_pVertexBuffer);
if (FAILED(hr))
    return false;
```

실습 - 버텍스 버퍼 만들기

// D3DDevice.cpp에 InitD3D에
return true; 위에 이어서 작성

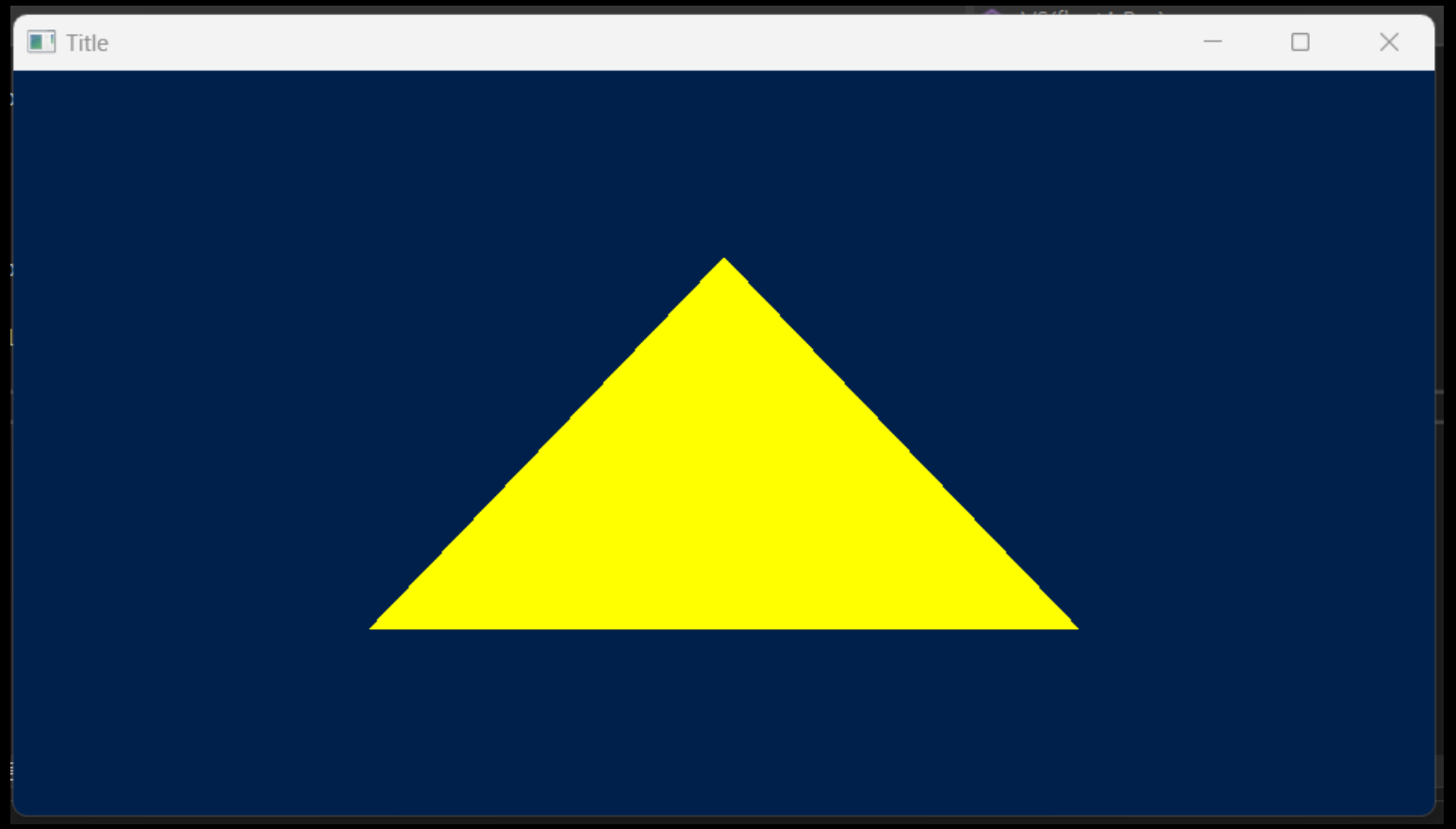
```
UINT stride = sizeof(DirectX::XMFLOAT3);
UINT offset = 0;
g_pImmediateContext->IASetVertexBuffers(
    0, 1, &g_pVertexBuffer, &stride, &offset);

g_pImmediateContext->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

실습 - 렌더 함수

// D3DDevice.cpp에 작성

```
void Render()
{
    float clearColor[4] = { 0.0f, 0.125f, 0.3f, 1.0f };
    g_pImmediateContext->ClearRenderTargetView(g_pRenderTargetView, clearColor);

    g_pImmediateContext->VSSetShader(g_pVertexShader, NULL, 0);
    g_pImmediateContext->PSSetShader(g_pPixelShader, NULL, 0);
    g_pImmediateContext->Draw(3, 0);

    g_pSwapChain->Present(0, 0);
}
```

실습 - 실행



코드 다시 리뷰!

끝

- 그래픽스 라이브러리, 렌더링 파이프라인의 개념을 익혔다
- Direct3D11을 초기화하고 화면에 삼각형을 띄워봤다
- 삼각형을 띄우는 과정에서 버퍼를 만들고 셰이더를 짰다

- 다음 내용 : 더 복잡한 버퍼 / 더 복잡한 셰이더로 3D 렌더링하기

+ 3회차 내용 선호 조사
  - 복잡한 DX11 다루기(Multiple Render Target으로 그림자 그리기)
  - D3D11을 객체지향으로 추상화해 나만의 렌더링 라이브러리 만들기
  - D3D12 해보기 (D311보다 진입장벽 높음, 삽질하기 쉬움, 언리얼에 도움?)

Direct3D11 스터디
1회차 - hello world

숭실대학교 겜마루 27기 컴퓨터학부 22학번 김민규

mkkim0612@naver.com

- D3D에 대한 기본적 이해
- 렌더링 파이프라인
- 빈 화면 띄우기
- 삼각형 그리기