DirectX 12 초기화

쿠재아이
김재경

COM(Component Object Model)

- 프로그래밍 언어 독립성과 하위 호환성을 가능하게 하는 기술
- C++ 클래스로 간주하고 사용해도 무방
- new, delete 키워드를 사용하지 않고 특정 함수 호출로 생성, 해제
- 참조 카운팅 기법을 사용하며 사용 횟수가 0이 되면 메모리에서 해제
- COM 객체의 수명 관리를 돕기 위해, Windows Runtime Library(WRL)는 Microsoft::WRL::ComPtr라는 클래스(스마트 포인터)를 제공
- 방대한 내용이지만 자세히 알 필요는 없는 듯

COM(Component Object Model)

```
ComPtr<ID3D12Device> d3dDevice;
D3D12CreateDevice(nullptr, D3D_FEATURE_LEVEL_11_0, IID_PPV_ARGS(&d3dDevice));
```

ID3D12Device	COM 객체
ComPtr	COM 객체의 스마트 포인터
D3D12CreateDevice	COM 객체 생성 함수

ComPtr

```
template <typename T>
class ComPtr
{
public:
    typedef T InterfaceType;

protected:
    InterfaceType *ptr_;
```

템플릿 인자 포인터 타입의 멤버변수를 가짐

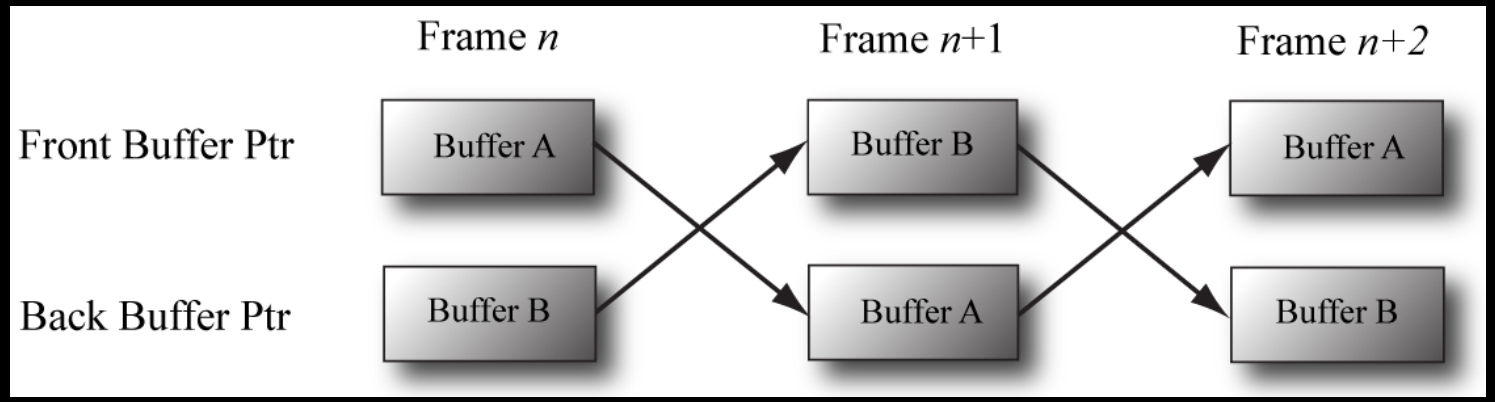
ComPtr

```
unsigned long InternalRelease() throw()
{
    unsigned long ref = 0;
    T* temp = ptr_;

    if (temp != nullptr)
    {
        ptr_ = nullptr;
        ref = temp->Release();
    }

    return ref;
}
```

소멸자에서 호출되는 함수

Swap Chain

- 프레임을 표시하는 데 사용되는 버퍼 모음(Direct3D 9)
- 더블 버퍼링을 하기 위해서는 후면 버퍼와 전면 버퍼가 필요하다
- 버퍼 내용을 변경하는 것이 아니라 버퍼의 포인터를 변경



- IDXGISwapChain이 해당 인터페이스. 전면 버퍼 텍스처와 후면 버퍼 텍스처를 담으며 버퍼 크기 변경을 위한 함수와 버퍼의 제시(presenting, 전환)를 위한 메소드 등을 제공

Swap Chain

- D3D12의 Swap Chain은 이전 버전과 다름
- automatic resource rotation을 지원하지 않음
- Automatic resource rotation enabled apps to render the same API object while the actual surface being rendered changes each frame
  (자동 리소스 순환은 앱이 렌더링 된 실제 표면이 각 프레임을 변경하는 동안 동일한 API 객체를 렌더링 할 수 있게 했습니다)
- D3D12에서는 다른 기능이 낮은 CPU 오버헤드를 가질 수 있게 한다고 함
- https://msdn.microsoft.com/en-us/library/windows/desktop/dn903945(v=vs.85).aspx

깊이 버퍼링(Z-버퍼링)

- 깊이 문제를 해결하기 위한 방법 중 하나는 먼 것부터 가까운 것 순서로 그리는 것
- 그러나 이 방법에는 두 가지 문제가 발생

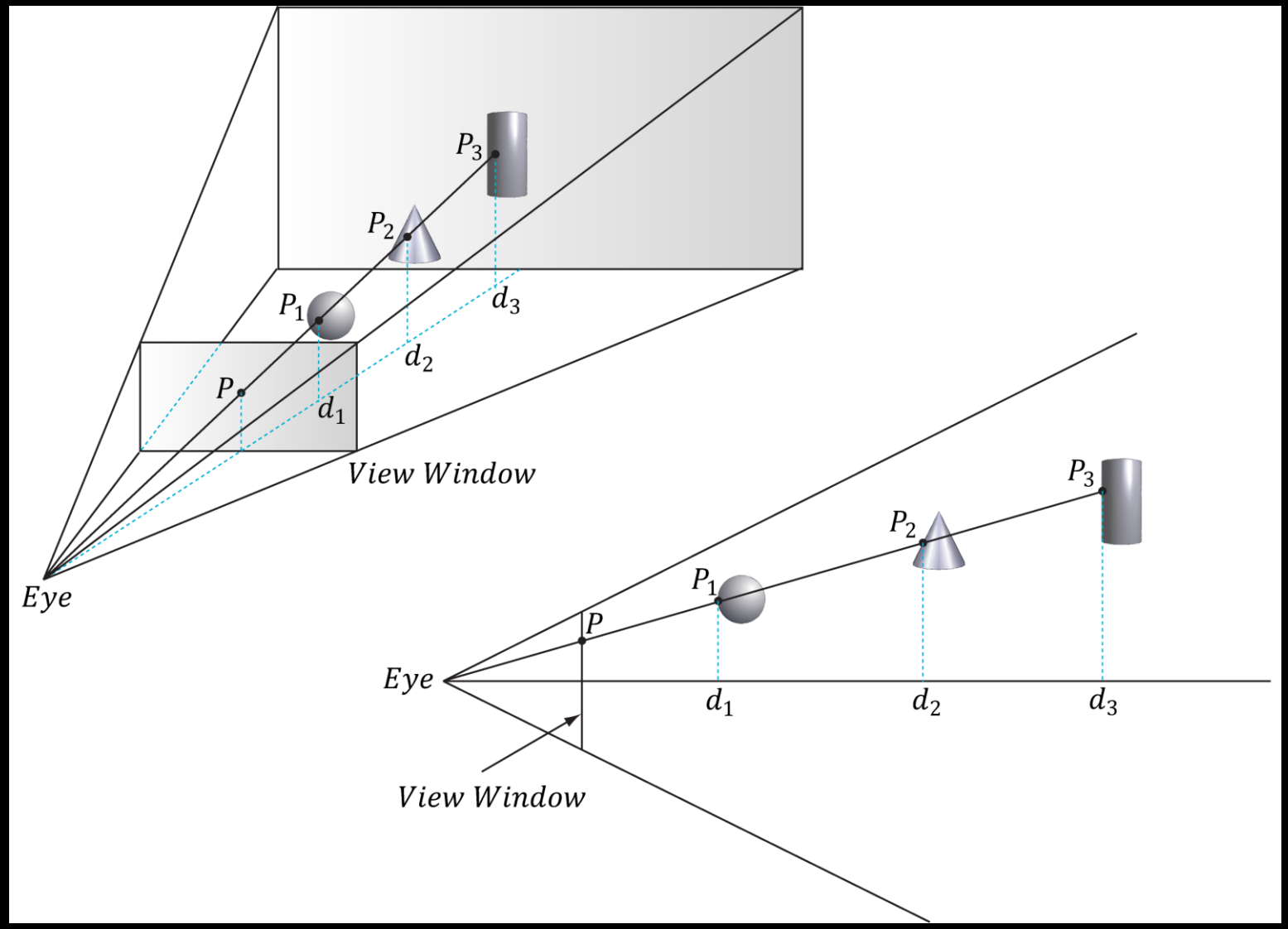
1. 물체들을 정렬해야 한다
2. 맞물린 형태의 물체들을 제대로 처리하지 못 함

- 깊이 버퍼링을 사용하면 이 문제를 해결 할 수 있다

깊이 버퍼링(Z-버퍼링)

- 한 물체가 다른 물체보다 앞에 있는지 판정하기 위한 기법
- 텍스처에 각 픽셀의 깊이 정보를 담는다
- 깊이는 0.0에서 1.0
- 깊이 버퍼의 원소들과 후면 버퍼의 픽셀들은 일대일도 대응
  (후면 버퍼가 1280 × 1024라면 깊이 버퍼도 1280 × 1024)
- 그리는 순서와 무관하게 물체들이 제대로 가려진다

깊이 버퍼링(Z-버퍼링)


P_3
P_2
P_1
P
d_1
d_2
d_3
View Window
Eye
P_3
P_2
P_1
P
Eye
d_1
d_2
d_3
View Window

깊이 버퍼링(Z-버퍼링)

연산	픽셀	깊이	설명
지우기	검은색	1.0	초기화
원기둥 그리기	P_3	d_3	깊이 갱신 O
구 그리기	P_1	d_1	깊이 갱신 O
원뿔 그리기	P_1	d_1	깊이 갱신 X

깊이 버퍼링(Z-버퍼링)

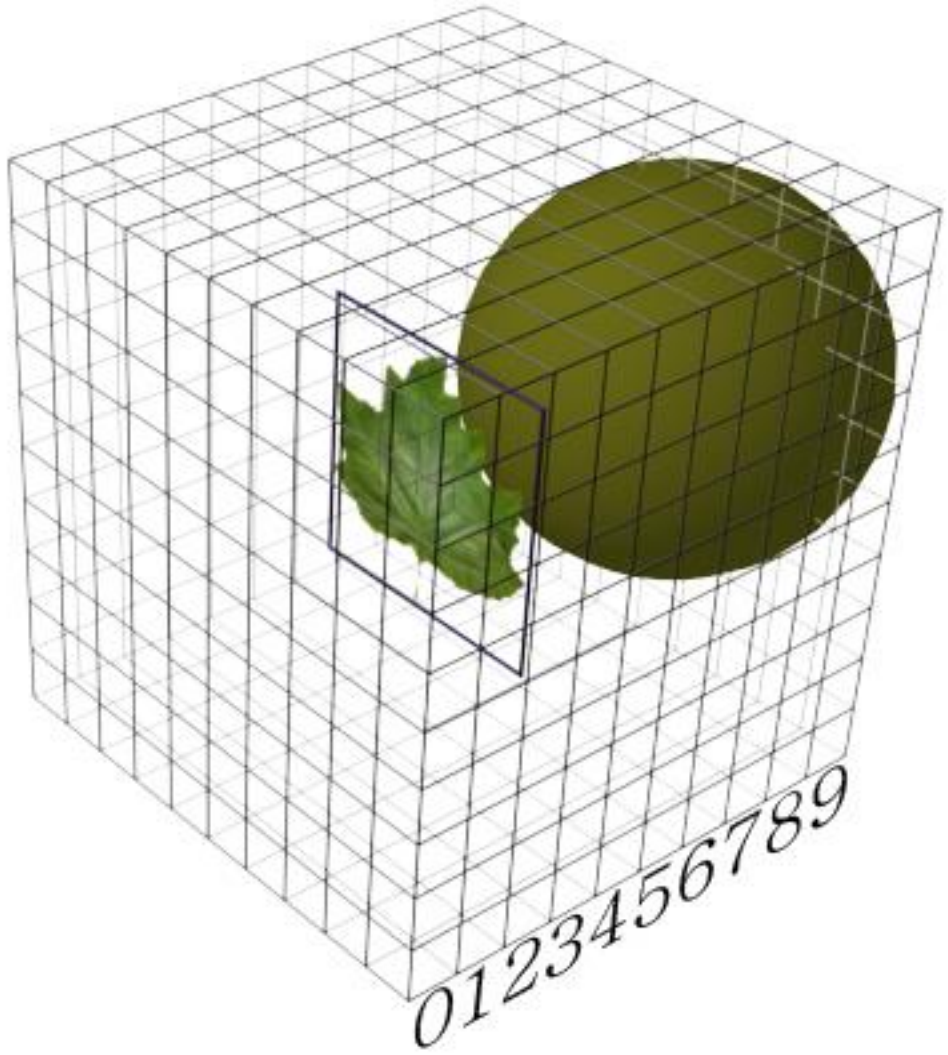
- 앞에서 보듯 깊이 버퍼링을 사용하면 정렬 유무에 상관 없이 물체를 제대로 가릴 수 있다
- 하지만 뒤에 가려질 물체를 먼저 그릴 경우 렌더링이 비효율적으로 된다
- 따라서 깊이 버퍼링을 사용 하더라도 물체를 정렬하는 것이 좋다
- 이것을 Z-order라고 부른다
- https://en.wikipedia.org/wiki/Z-order

깊이 버퍼링(Z-버퍼링)

- 하지만 깊이 버퍼링도 완전하지 않음
- 알파 블렌딩이 들어가면 문제가 생김
- 예)

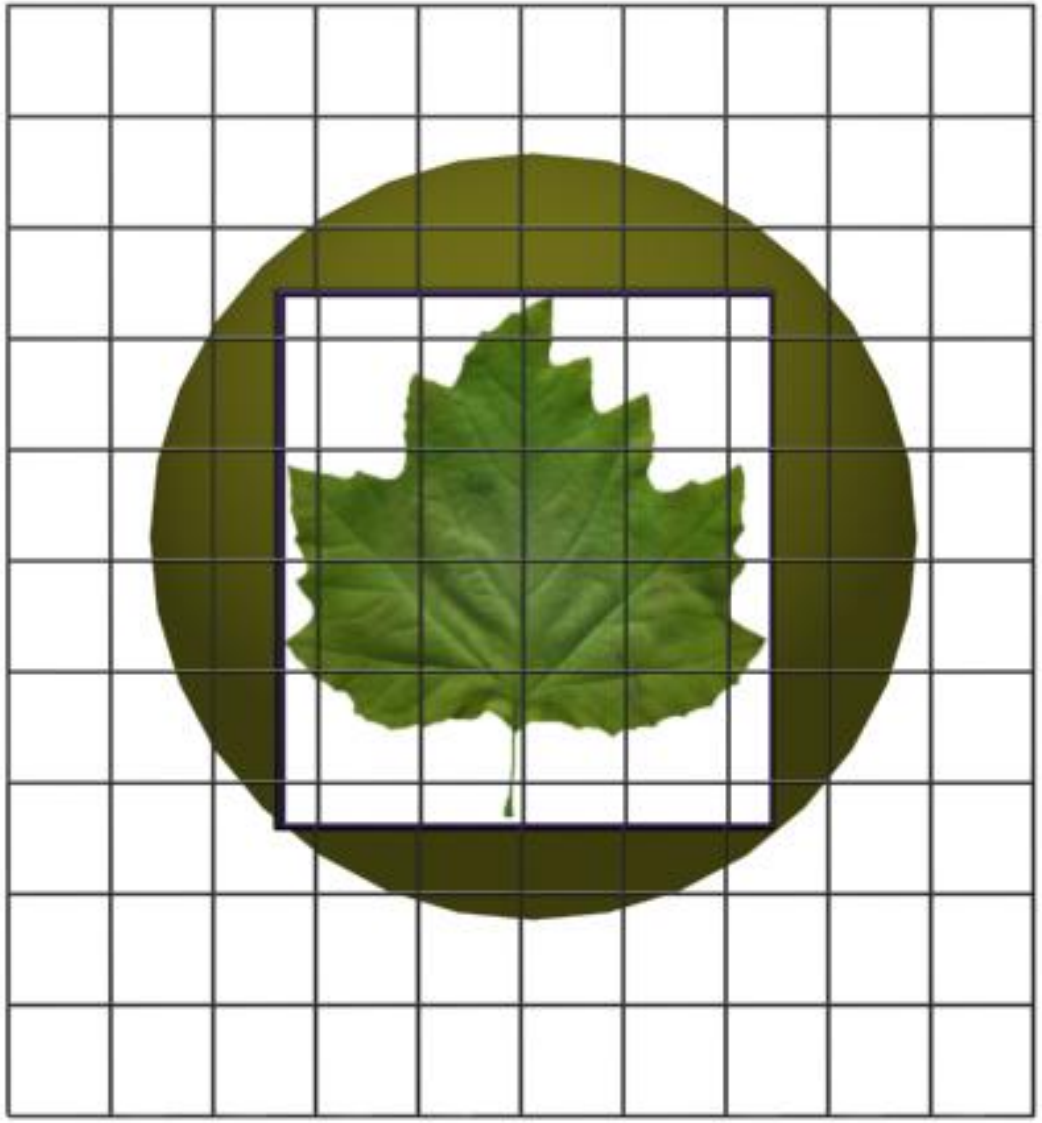
이런 나뭇잎이 있다고 하자

깊이 버퍼링(Z-버퍼링)



- 나뭇잎과 구의 위치가 사진과 같을 때
  나뭇잎이 먼저 그려지고 구가 먼저 그려진다면?

깊이 버퍼링(Z-버퍼링)



- Z-order로 인해 앞에서부터 그려진다

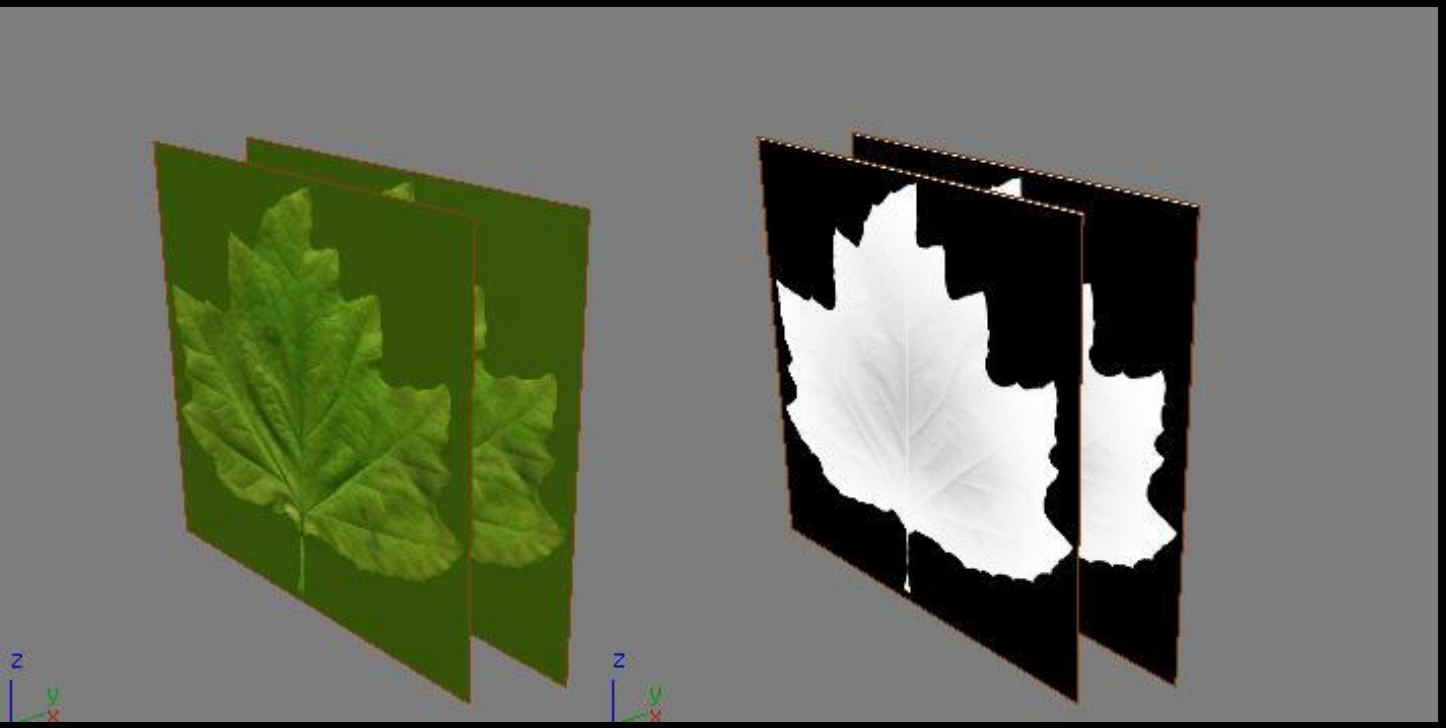
깊이 버퍼링(Z-버퍼링)

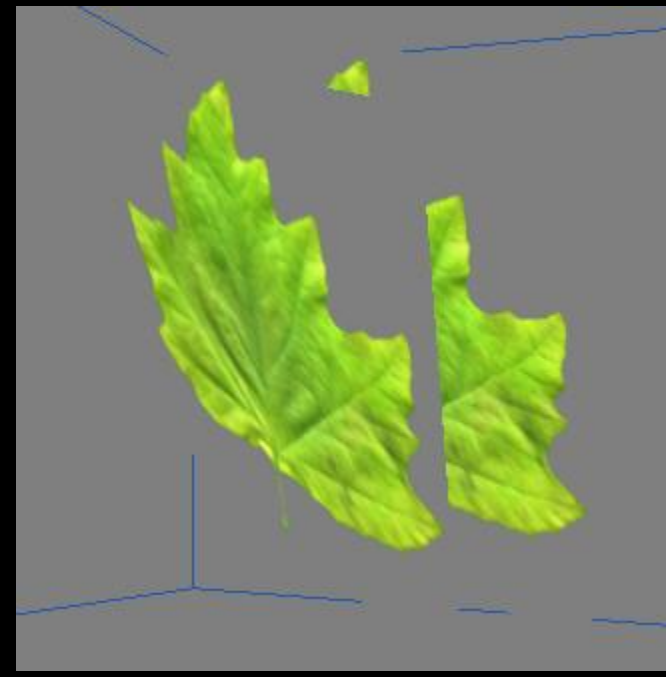
- 일반적으로는 먼저 불투명한 걸 모두 모아서 그리고 나중에 반투명한 것을 모아 그린다고 함

Q. 그럼 다 해결된건가요?

A. 아뇨 ㅋ

깊이 버퍼링(Z-버퍼링)





- 반투명끼리 그릴 때 앞의 것이 먼저 그려지면 문제 발생
- 반투명인 것들만 따로 모아서 정렬을 해야 한다

깊이 버퍼링(Z-버퍼링)

Q. 그럼 다 해결된건가요?

A. 아뇨 ㅋ

- 자세한 것은 아래 링크에…
  http://chulin28ho.egloos.com/5267860
  http://chulin28ho.egloos.com/5268685
  http://chulin28ho.egloos.com/5269434
  http://chulin28ho.egloos.com/5270691
  http://chulin28ho.egloos.com/5271687
  http://chulin28ho.egloos.com/5272883
  http://chulin28ho.egloos.com/5284164

자원과 서술자(Descriptor)

- 자원은 자원 자체에 대한 정보를 가지고 있지 않다

```
void func(int* array)
{
    for (int i = 0; i < length; i++)
        cout << array[i] << endl;
}
```

- 위와 같은 함수가 있을 때 배열의 길이를 알 수 없다. 길이 인자를 하나 더 만들어야 한다
- 즉, 배열에 대한 정보(서술)가 필요하다

자원과 서술자(Descriptor)

```
struct ARRAY_DESC
{
    int* element;
    int length;
};

void func(ARRAY_DESC arrayDesc)
{
    for (int i = 0; i < arrayDesc.length; i++)
        cout << arrayDesc.element[i] << endl;
}
```

서술자를 이용하면 자원의 여러 정보를 알 수 있다

자원과 서술자(Descriptor)

- 렌더링 파이프라인에 바인딩 되는 것은 자원이 아니라 서술자이다

- 책에서는 다음과 같은 서술자를 사용한다

1. CBV(constant buffer), SRV(shader resource), UAV(unordered access)
2. Sampler
3. RTV(render target view)
4. DSV(depth/stencil view)

- 한 자원의 여러 부분을 여러 서술자가 참조할 수 있다
  (하나의 텍스처를 RTV와 SRV가 참조할 수 있다)

- 서술자들은 응용 프로그램의 초기화 시점에서 생성해야 한다

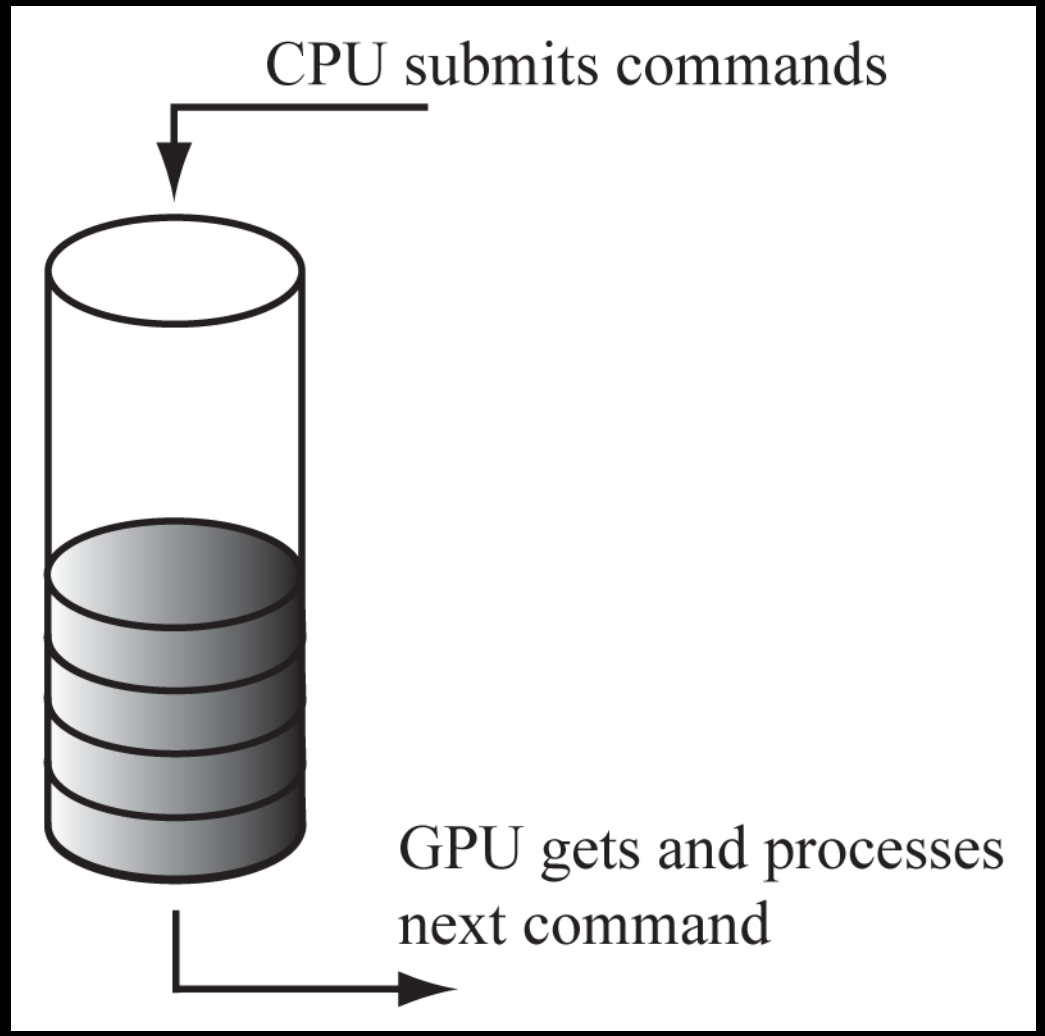
서술자 힙(Descriptor Heap)

- 서술자들의 배열. 서술자들이 저장되는 곳이다
- 같은 종류의 서술자들은 같은 서술자 힙에 저장된다
- 한 종류의 서술자에 대해 여러 개의 힙을 둘 수 있다
- CPU로만 관리 할 수 있다

DXGI(DirectX Graphics Infrastructure)

- Direct3D와 함께 쓰이는 API
- 3D에 한정되지 않는 공통적인 그래픽 기능들이 포함
- 예) IDXGISwapChain, 전체 화면 모드 전환, 지원되는 디스 플레이 모드(해상도, 갱신율 등)

Command Queue

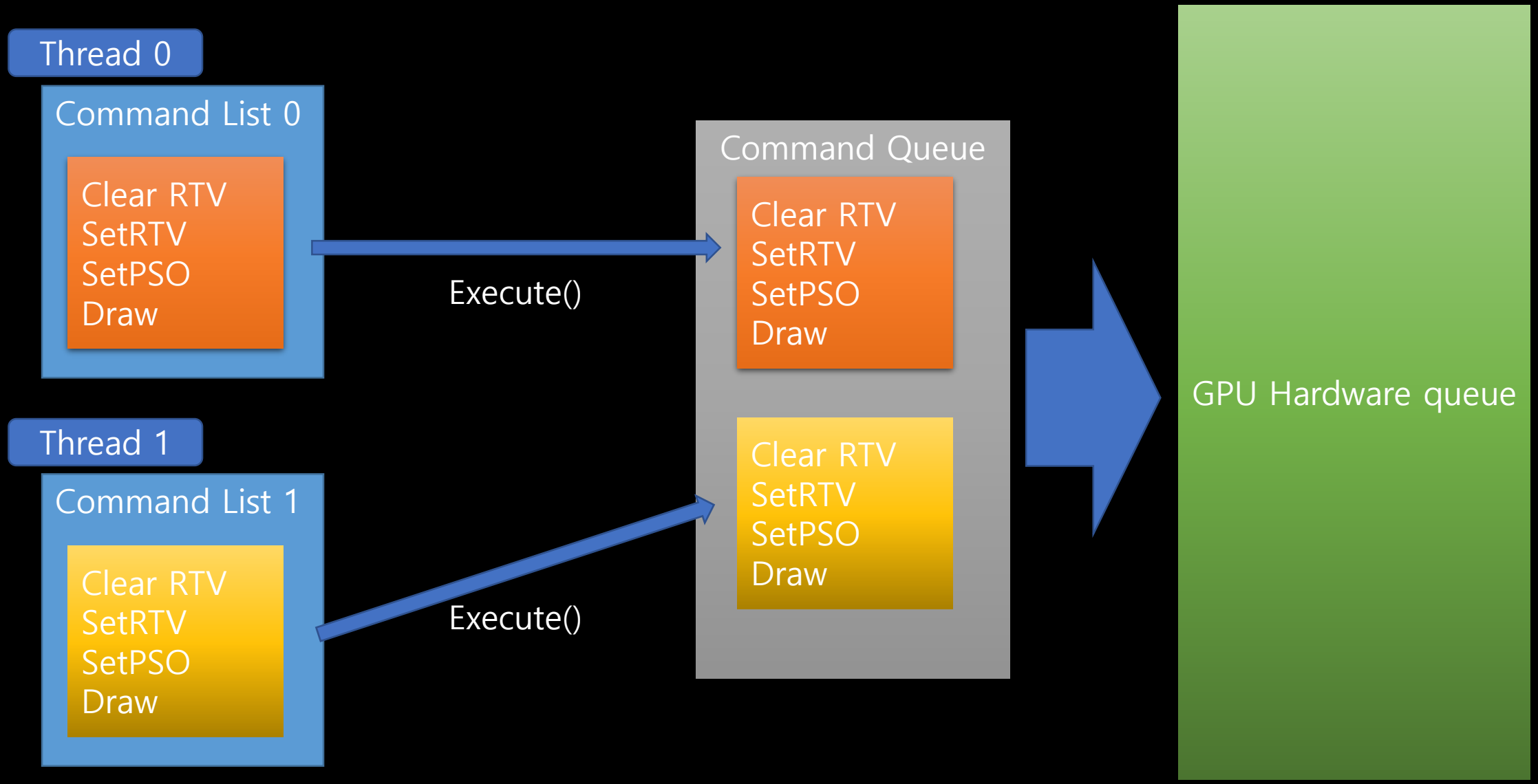


- 비동기 랜더링을 위한 디자인
- CPU는 Command List를 큐에 제출한다
- GPU는 명령을 처리할 준비가 되면 큐에서 명령어를 꺼내 실행한다
- 큐를 비우거나 꽉 채우지 않고 계속 사용하도록 하는 것이 관건

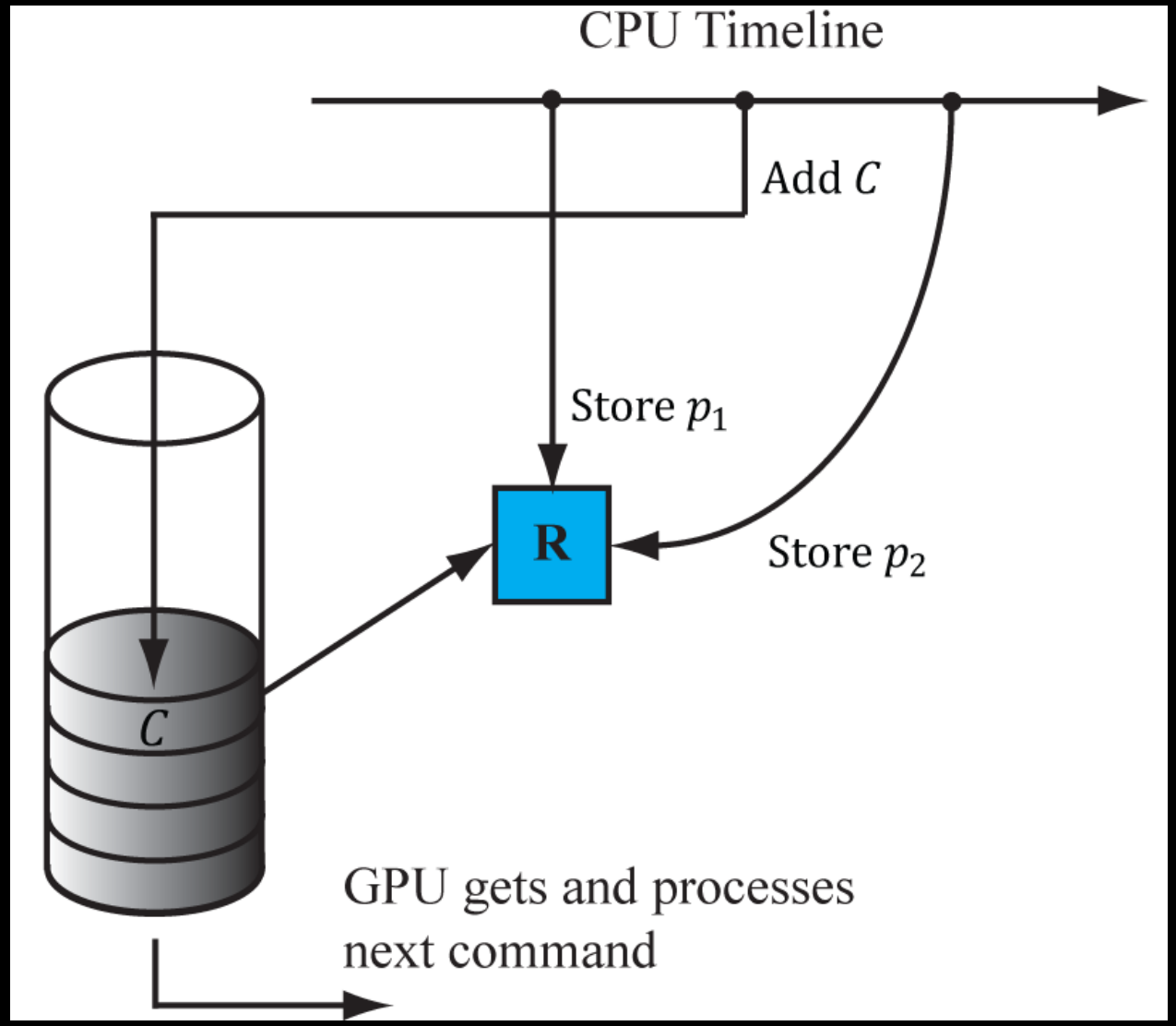
Command List, Command Allocator

- Command List - GPU가 실행하는 명령 집합
- Command Allocator - 실제로 명령이 할당되는 공간
- 1개의 Command List로도 처리는 가능하나 성능이 제대로 나오지 않음
- 멀티스레드로 여러 개의 Command List를 동시에 기록하고 각각의 스레드가 독립적으로 실행하는 것을 권장
- GPU가 Command Allocator에 담긴 모든 명령을 실행한 것이 확실해지기 전까지는 Command Allocator를 재설정하지 말아야 한다

Command List, Command Allocator

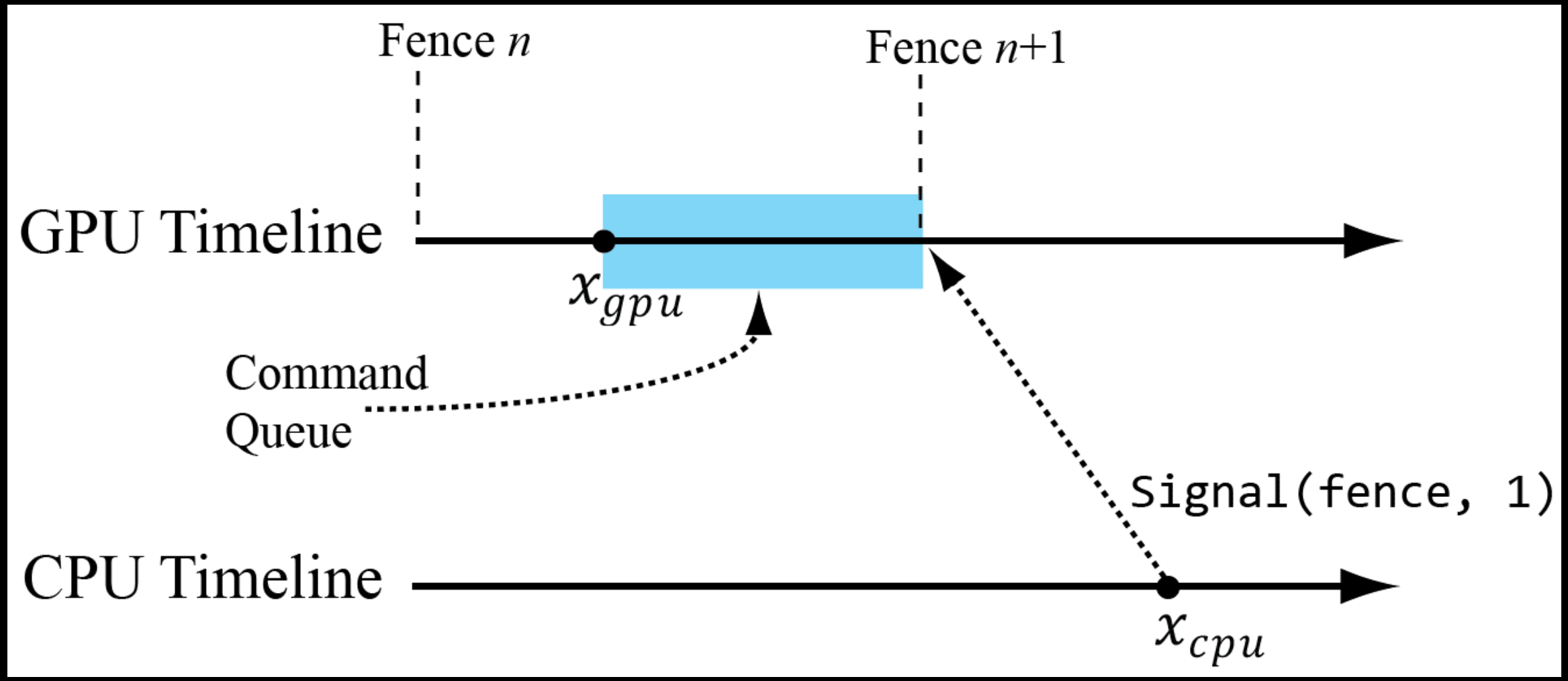


CPU/GPU 동기화



- C가 p_2를 이용하거나 R이 갱신되는 도중에 R을 사용하는 것의 의도한 행동이 아니다
- GPU가 명령을 모두 처리할 때까지 기다려야 한다
- 이 때 필요한 것이 울타리(Fence) 객체

CPU/GPU 동기화


Fence n
Fence n+1
GPU Timeline
x_{gpu}
Command
Queue
Signal(fence, 1)
CPU Timeline
x_{cpu}

자원 상태 전이

- GPU가 자원에 자료를 다 기록하지 않았거나 기록을 아예 시작하지도 않은 상태에서 자원에 자료를 읽으려 하면 문제가 생긴다. (Resource hazard)
- 이 문제를 해결하기 위해 자원들에 상태를 부여
- 임의의 상태 전이를 Direct3D에게 보고하는 것은 프로그램의 몫
- 예) 텍스처를 write 할 때에는 Render Target 상태로 설정하고 read 할 때가 되면 상태를 Shader Resource 상태로 변경(전이)한다
- GPU는 자원 위험을 피하는 데 필요한 조처를 할 수 있다(모든 쓰기 연산이 완료되길 기다린 후에 읽기를 시도하는 등)

멀티 스레드 활용

- Direct3D 12는 멀티 스레드를 효율적으로 활용할 수 있도록 설계 됨
- Command List를 멀티 스레드로 생성할 때 주의해야 할 점

1. 같은 Command List를 여러 스레드가 공유하지 않는다
2. Command Allocator 또한 여러 스레드가 공유하지 않는다
3. Command Queue는 여러 스레드가 공유할 수 있다
4. 성능상의 이유로 동시에 기록할 수 있는 Command List의 최대 개수를 반드시 초기화 시점에서 설정해야 한다

출처

- Swap Chain
- https://msdn.microsoft.com/en-us/library/windows/desktop/dn903945(v=vs.85).aspx
- https://msdn.microsoft.com/ko-kr/library/windows/desktop/bb206356(v=vs.85).aspx

- Command Queue, List, Allocator
- https://goo.gl/PrMwFN