

# 프로그래밍 면접

## 문제 ————— 목차

### 분류

#### Unreal Engine

FName, FString, FText의 차이	4
애니메이션	5
언리얼의 C++	5

#### C/C++

객체지향의 4대 특성	7
객체지향 5대 원칙	7
가상함수와 가상함수 테이블	8
RTTI	8
유니코드, 멀티바이트	9
C++ 캐스트	9
메모리 영역	10
*p++ 포인터와 증감연산 우선순위	10
push_back과 emplace_back	11
게임프로그래밍에서 c++이 사용되는 이유	11
Epsilon을 사용하는 이유	11
호출 규약 중 cdecl과 stdcall에 대해 설명	12
New와 Malloc의 차이점	12
VirtualAlloc을 써야 하는 경우	13
클래스 선언시 자동으로 선언되는 암시적 멤버 메서드 4가지	13
문자열 관리 클래스를 코딩해보기	14
문자열 코딩 문제	17

---

## 자료구조 및 STL

시간복잡도	18
큐, 스택의 차이	18
벡터와 리스트의 차이	19
리스트에서 100만개의 데이터를 검색할 때 속도 절감	20
Map과 Unordered Map, Multi Map의 차이	21
재귀 함수의 호출 순서	22
정렬알고리즘	23

---

---

---

---

---

---

## DirectX 11

렌더링 파이프라인	24
-----------	----

---

---

---

---

---

---

## 3D 그래픽

쿼터니온	25
쿼드트리, 옥트리 컬링	26
프러스텀 컬링	27
오클루전 컬링	28
백페이스 컬링	29
WVP의 Project와 Unproject	30
플레이어가 지면 위에 있는지 판별하는 방법	30
인스턴싱	31
LOD	32
알파 소팅을 해야하는 이유	33

---

Ambient, Diffuse, Specular, Emissive	34
디퍼드 렌더링	35
Ambient Occlusion	39
깊이 버퍼	40

## 운영체제 및 스레드

크리티컬 섹션에 대한 설명	41
데드락에 대한 설명과 발생 조건	42
메모리 풀 구조	44
메모리 단편화	44
운영체제의 스케줄링 방식	46
멀티 스레드가 사용되어야 하는 경우	47
동기 / 비동기 스레드가 각각 사용되는 부분	49

## 게임 프로그래밍

플래시에 비추는 몬스터 판별	50
2D평면에 10000개의 캐릭터 검출	50
다각형 내 몬스터 판별	50
지형에 여러 종류의 풀을 그릴 때 디자인 패턴	51
캐릭터에 아이템을 조합하여 생성할 때 디자인 패턴	51
Redo, Undo	51
A*의 단점	52

# FName, FText, FString

## FName

대소문자 구분 x, 변경 및 조작 불가, 빠른 속도

콘텐츠 브라우저에서 새 애셋 이름을 지을 때, 다이내믹 머티리얼 인스턴스의 파라미터를 변경할 때, 스켈레탈 메시에서 본에 접근할 때, 모두 FName 을 사용합니다. FName 은 문자열 사용에 있어서 초경량 시스템을 제공하는데, 주어진 문자열이 재사용된다 해도 데이터 테이블에 한 번만 저장되는 것입니다. FName 은 대소문자를 구분하지 않습니다. 변경도 불가능하여, 조작할 수 없습니다. 이처럼 FName 의 정적인 속성과 저장 시스템 덕에 찾거나 키로 FName 에 접근하는 속도가 빠릅니다. FName 서브시스템의 또다른 특징은 스트링에서 FName 변환이 해시 테이블을 사용해서 빠르다는 점입니다.

## FText

실시간으로 현지화되는 텍스트 처리 가능, 변경 및 조작 불가

FText는 "표시되는 문자열"을 나타냅니다. 사용자에게 표시하고자 하는 텍스트는 FText 로 처리해야 합니다. FText 클래스에는 현지화 기능이 내장되어 있어, 현지화되어 룩업 테이블에 저장된 텍스트 콘텐츠 뿐만 아니라, 숫자, 날짜, 시간, 포맷 텍스트처럼 실시간 현지화되는 텍스트도 처리할 수 있습니다. 현지화할 필요가 없는 텍스트조차도 FText 로 처리 가능합니다. 여기에는 플레이어 이름처럼 사용자가 입력한 콘텐츠나, 슬레이트로 표시되는 텍스트도 포함됩니다. FText 에는 어떠한 변환 함수도 제공되지 않는데, 표시되는 문자열에 변경을 가하는 것은 매우 불안정한 작업이기 때문입니다.

## FString

변경 및 조작 가능, 사용가능한 메서드 많음

FName 이나 FText 와는 달리, FString 은 조작이 가능한 유일한 스트링 클래스입니다. 대소문자 변환, 부분문자열 발췌, 역순 등 사용가능한 메서드는 많습니다. FString 은 검색, 변경에 다른 스트링과의 비교도 가능합니다. 그러나 바로 그것이 FString 이 다른 불변의 스트링 클래스보다 비싸지는 이유입니다.

# 애니메이션

우선 애니메이션 블루프린트는 '이벤트 그래프'와 '애님 그래프'로 이루어져 있습니다. 그 중 이벤트 그래프는 멤버 변수의 값을 변경하거나 다른 오브젝트의 값을 참조하여 애니메이션의 상태 변수들을 갱신하는 역할을 합니다. 그리고 애님 그래프는 이벤트 그래프에서 설정된 변수값으로 현재의 상태를 계산하고 그에 맞는 애니메이션을 출력하는 역할을 합니다.

이렇게 두 개로 나뉘어진 이유는 여러 가지가 있지만 기본적으로는 최적화를 위함인데요, 두 그래프는 서로 다른 스레드에서 동작하도록 설계되었습니다. 때문에 CPU의 코어가 여럿일 경우 더 효율적으로 연산될 것입니다. 그러나 병렬적으로 작동하기 위해 제약사항이 존재하는데, UObject(엔리얼의 기본 오브젝트; 액터, 컴포넌트 등)와 상호작용 하는 작업은 반드시 메인 스레드에서 동작해야 한다는 것입니다. 이런 제약 조건이 없다면 여러 스레드에서 한 값을 수정하거나, 변수를 읽는 타이밍이 어긋나는 등의 의도하지 않은 결과가 일어날 수 있습니다. 여기서 이벤트 그래프가 바로 메인 스레드에 속한 그래프이며, 애님그래프는 아닙니다. 때문에 애님그래프에서는 캐릭터 무브먼트와 같은 UObject의 값을 참조할 수 없습니다.

그러면 캐릭터의 이동 속도를 애님그래프에서 사용하려면 어떻게 해야 할까요? 바로 이동 속도 변수를 애니메이션 블루프린트에 생성한 다음, 이벤트 그래프에서 UObject의 값을 복사해 놓는 것입니다. 캐릭터의 이동 속도는 보통 매 프레임마다 바뀔 테니, 이러한 작업은 'Blueprint Update Animation' 함수에서 매 프레임(틱)마다 속도 변수를 캐릭터에게서 얻어 복사해옵니다. 그리고 애님 그래프에서는 이 복사된 값을 가져다 쓰면 안전하게 작업할 수 있습니다.

## 엔리얼의 C++

엔리얼 오브젝트는 엔리얼 엔진 내에서 객체를 쉽고, 효율적으로 관리하기 위해서 고안되었습니다. C++을 깊숙히 몰라도 엔리얼 오브젝트의 성격을 알면 마치 Java나 C#과 같은 언어처럼 사용할 수 있게 되는데요, C++ 객체가 엔리얼 오브젝트가 되면 자동으로 향상되는 기능은 다음과 같습니다.

1. **CDO(Class Default Object)** : 객체의 초기 값을 자체적으로 관리합니다.
2. **Reflection** : 객체 정보를 런타임에서 실시간 조회가 가능합니다.
3. **GC(Garbage Collection)** : 참조되지 않는 객체를 메모리에서 자동 해제할 수 있습니다.
4. **Serialization** : 객체와 속성 정보를 통으로 안전하게 보관하고 로딩합니다.
5. **Delegate** : 함수를 묶어서 효과적으로 관리하고 호출할 수 있습니다.
6. **Replication** : 네트워크 상에서 객체간에 동기화를 시킬 수 있습니다.
7. **Editor Integration** : 엔리얼 에디터 인터페이스를 통해 값을 편집할 수 있습니다.

엔리얼 오브젝트는 C++표준이 아니고, 엔리얼 엔진이 자체적으로 만들어 제공하는 프레임워크이기 때문에, 일반적인 방법으로는 만들 수 없고, 엔리얼 헤더 툴(Unreal Header Tool)이라는 프로그램의 도움을 받아야 합니다. 여러분들이 헤더 파일에서 클래스 선언을 엔리얼 오브젝트 규격에 맞게 선언해주면, 바로 컴파일하는 것이 아닌, 엔리얼 헤더 툴이 이를 파싱하고 분석합니다.

헤더 파일의 선언이 규격에 맞지 않으면 엔리얼 헤더 툴에 의해 에러가 발생되고, 규격에 맞게 선언하였다면 엔리얼 헤더 툴은 부가적인 메타 정보를 담은 소스코드를 프로젝트의 Intermediate 폴더에 생성해 줍니다. 이 작업이 모두 끝나면 이제 본격적으로 컴파일을 진행하게 됩니다

# 리플렉션

## 리플렉션

컴퓨터 언어에서 런타임에서 객체 자신에 대한 정보와 속성을 검색하고, 스스로의 구조를 수정할 수 있는 기능

C++의 다음 세대 언어인 C#이나 Java에서는 VM기반의 컴파일시 클래스 정보를 바이트 코드에 포함해 실시간으로 검색, 수정이 가능하도록 리플렉션 기능을 제공하지만 오래된 언어인 C++에선 지원하지 않음

언리얼 엔진에서는 언리얼 헤더 툴에 의해 만들어지는 UClass라는 데이터를 매개로 하여 언리얼 오브젝트의 인스턴스의 속성과 함수를 호출하는 한편 인스턴스의 값을 변경할 수 있다

언리얼 오브젝트 클래스에서 UPROPERTY와 UFUNCTION 매크로로 지정된 멤버 변수와 멤버 함수는 모두 검색이 가능합니다. 더 나아가 언리얼 오브젝트의 정보를 자세히 몰라도, 필드의 타입 정보와 이름만 알고 있으면 지정한 인스턴스 내 변수 값을 변경하거나 함수를 호출하는 것이 가능합니다.

예를 들어 우리는 모르는 서로 다른 타입의 언리얼 오브젝트 100여가지 종류가 현재 메모리에 떠 있는데, 모든 언리얼 오브젝트에 FString으로 선언된 Host라는 멤버 변수가 있고, 인자가 없는 void 타입의 RequestToken이라는 이름의 함수가 있다면, 각 언리얼 오브젝트의 클래스 헤더 정보가 없어도 런타임에서 100여가지의 각각 다른 언리얼 오브젝트 인스턴스의 Host 변수값을 읽어오고 RequestToken 함수를 호출할 수 있다는 의미가 되겠습니다.

## 객체지향의 4대 특성

---

### 추상화

대상의 특성 중 불필요한 부분을 무시하고 필요한 공통점만을 다루어, 현실의 복잡성을 극복하고 목적에 집중할 수 있도록 하는 것

### 캡슐화

객체 스스로가 자신의 상태를 책임지게 하여, 해당 객체의 역할 수행에 집중할 수 있도록 자율성을 높이는 것

### 상속성

부모 클래스의 속성과 기능을 상속받아 동일하게 사용하는 것

### 다형성

동일한 요청에 대해 서로 다른 방식으로 응답할 수 있도록 만드는 것 (오버라이딩, 오버로딩)

오버라이딩 : 상위클래스에게 상속 받은 동일한 메서드의 재정의

오버로딩 : 동일한 메서드가 매개변수의 차이에 따라 다르게 동작

## 객체지향 5대 원칙

---

### SRP 단일 책임 원칙

하나의 객체는 하나의 책임을 가져야 한다

### OCP 개방 폐쇄 원칙

기존의 코드를 변경하지 않으면서 기능을 추가 가능

### LSP 리스코프 치환 원칙

자식클래스는 최소한 부모클래스에서 가능한 행위는 수행 가능해야 한다

### ISP 인터페이스 분리 원칙

인터페이스를 클라이언트에 영향 받지 않도록 분리

### DIP 의존 역전 원칙

의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것보다는 거의 변화가 없는 것에 의존하라는 것.

# 가상함수와 가상함수 테이블

## 순수가상함수

인터페이스를 자식 클래스에게 전달하기 위해 사용하는 함수

## 가상함수

인터페이스와 함수의 선언까지 자식에게 전달하기 위해 사용하는 함수

## 가상함수테이블(vftable)

가상함수에 대해 연결되어있는 함수 포인터를 저장하는 공간

## 가상함수포인터(vfptr)

가상함수를 가진 클래스의 객체들이 갖고 있는 것을 가리킴

# RTTI (Run-Time Type Information)

## 정의

실행시간에 타입정보를 식별 정도로 풀이

실행 중에 기반 클래스의 타입 포인터의 실체를 밝혀내는데 목적을 둔다.

## 필요성

다형성을 가진 객체에 대해 정확한 타입을 알아내기 위함.

예를 들어 Base 클래스가 있고 그것을 상속받은 Child1, Child2 클래스가 존재한다고 가정.

```
Base* base = new Child1();
```

이렇게 할당하였을 경우 실행 중 base가 가리키고 있는 객체가 뭔지 알 수 없음.

하지만 RTTI를 사용하면 Child1 타입이라는 것을 정확하게 알 수 있게 됨.

## 제약사항

클래스내에 반드시 1개 이상의 가상함수를 가지고 있어야 RTTI기능이 활성화됨.

상속을 받더라도 가상함수가 없다면 객체의 타입을 알 수 없음.



# 유니코드와 멀티바이트

---

## ASCII 코드

ASCII 코드는 모든 문자 하나가 1byte를 차지. 코드 하나로 다른 문자(한글, 일어 등) 표시 불가

## 멀티바이트 문자 집합

아스키 문자 코드 + 다른 문자(2byte)들을 포함한 문자 집합. 특정 문자 집합마다 코드페이지가 존재. 종종 언어가 깨져서 나오는 문제점이 있음

## 유니코드

아스키 문자 코드 뿐만 아니라 모든 문자들을 총 망라하여 각 한 문자에 2byte씩 할당하여 만든 문자 집합. 각각의 특정 문자는 고유의 유니코드 값을 가진다.

wchar 는 모든 문자를 2바이트로 구성한다.(기존 자료형보다 메모리 2배 공간필요)  
TCHAR는 char 와 wchar 구분없이 사용 할 수 있게 해줌.

즉, C++에서 string 은 char\* // wstring은 wchar\*로 구분 되어 있겠죠?

# C++ 캐스트

---

## const\_cast

표현식의 상수성을 없애는 데 사용

## reinterpret\_cast

어떠한 포인터 타입도 어떠한 포인터 타입으로든 변환이 가능하다.

## static\_cast

형변환 능력을 가지고 있는 기본적인 캐스트 연산자.

개발자가 변수의 데이터 타입과 변환하고자 하는 대상의 데이터 타입을 모두 알고 있을 때 사용, 컴파일 시점에 타입 변환

## dynamic\_cast

기본 클래스 객체에 대한 포인터나 참조자의 타입을 파생 클래스, 혹은 형제 클래스의 타입으로 변환.

즉, dynamic\_cast는 런타임에 다형성을 이용하여 모호한 타입 캐스팅을 시도할 때 (다형성 위배), 엉뚱한 변환 결과가 넘어가지 않도록 하여, 런타임 오류가 방지하는 역할을 한다.

# 메모리 영역

---

## Code 영역

함수와 상수, 실행한 프로그램의 코드가 저장되는 공간 (정적 메모리)

## Data 영역

전역 변수와 정적(static) 변수가 저장되는 공간 (정적 메모리)

## Stack 영역

지역 변수와 매개 변수가 저장되는 공간 (동적 메모리)

## Heap 영역

동적 할당을 받아오는 공간 (동적 메모리)

# \*p++ 포인터와 증감연산 우선순위

---

## 우선순위

전위++과 \*는 연산자 우선순위가 같다.(3순위)(이연산자들은 오른쪽에서 오른쪽으로 결합)

후위++는 2순위 \*보다 우선순위

### #

\*p++ 후위형 으로 증감 연산자를 적용하면 현재 자신이 가리키는 주소에 저장된 값을 사용한 후 자신의 주소를 증가 시키고

\*++p 전위형 으로 사용하면 자신의 주소를 증가시킨후에 증가된 주소에 존재하는 값을 사용

### ##

\*++p는 포인터p 가 가지고 있는 주소 값을 1 증가시키고 증가된 주소 값에 저장된 값을 사용하겠다는 뜻.

++\*p는 포인터p 가 가지고 있는 주소 값에 저장된 값을 1증가 시키고 그 값을 사용하겠다는 뜻.

### ###

\*p++는 앞에서 설명한 것처럼 포인터p 가 가진 주소 값에 저장된 값을 사용한 후 포인터p에 저장된 주소 값을 1증가 한다는 뜻.

(\*p)++는 포인터p 가 가진 주소 값에 저장된 값을 사용한 후 그 값을 1 증가 한다는 뜻.

## push\_back과 emplace\_back

### Push\_back

#### 두 번의 생성자와 소멸자를 호출

push\_back의 인자로서 필요한 객체를 생성 후 push\_back함수 내부에서 다시 한 번 복사가 일어나고, push\_back을 빠져나갈 때 인자들이 파괴된 후 해당 객체가 파괴됨

### Emplace\_back

#### 한 번의 생성자와 소멸자를 호출

생성자로 넘길 인자만 넘겨주면 컴파일러가 알아서 할당된 생성자를 받을 수 있는 생성자를 찾아서 내부에서 생성시켜준다. 거기에 객체 복제는 물론 이동도 필요 없이 함수 내부에서 객체를 생성하여 삽입하므로 생성자와 소멸자를 한번 씩만 호출하여 훨씬 효율적이다. (c++11 가변인자 템플릿)

## 게임 플밍에서 C++ 사용 이유

### 1. 활용성

지난 수십 년간 C/C++로 작성된 많은 코드들이 남아있으며 Ms, Sony, Nintendo 등 플랫폼이 되는 기업에서 C++을 사용하고 있기 때문

### 2. 메모리 관리

C++에서는 프로그래머가 직접 메모리를 관리하며 사용 하는 것들에 대해서만 비용을 지불할 수 있기 때문

C#이나 Java등 다른 언어에서는 가비지컬렉션을 사용하여 메모리 관리가 자동으로 이루어지기 때문에 프로그래머가 직접 메모리를 관리 할 수 없다.

## Epsilon을 사용하는 이유

### 오차의 범위를 줄이기 위해 사용

float형의 10진수 값은 정확한 2진 표현이 불가능, 즉 CPU에서 정확한 표현이 불가능하다.

1보다 크면서 표현 할 수 있는 가장 작은 수와 1과의 차이가 epsilon이다.  
예를 들어, float a == 0 이라고 표현할 때 오차발생의 여지가 있기 때문에  
epsilon > a && -epsilon < a 로 사용한다.

## cdecl과 stdcall에 대해 설명

### cdecl

cdecl은 C 함수 라이브러리에서 표준으로 쓰이는 호출규약으로, 스택으로 전달한 매개변수를 함수를 호출한 위치에서 정리하는 방식이다.

### stdcall

stdcall은 스택의 정리를 함수 내부에서 처리한다. 함수를 호출하는 위치에서는 함수를 호출하기만 하고 스택 정리를 하지 않는 방식이다.

Window API들이 이 방식을 채택

cdecl 함수와 달리 stdcall 함수는 이름 뒤에 @ 기호를 붙인 뒤 정리할 스택의 크기를 명시

## New 와 Malloc

new	malloc
생성자 호출	생성자 호출 안함
연산자, 정확한 데이터 타입 리턴	함수, Void* 리턴
오류 시 Throws콜	오류 시 NULL return
free store로부터 메모리 할당	힙으로부터 할당
오버라이딩 가능	오버라이딩 불가능
컴파일러가 크기 계산	직접 크기 정할 수 있음

new는 재할당하려면 지우고 다시 할당해야 하기 때문에 재할당이 빈번하게 이루어지는 경우에는 malloc을 사용하는게 좋음(realloc으로 재할당 가능)

그 외에 c++에서는 new가 좋음

# VirtualAlloc을 써야하는 이유

메모리 단편화를 예방하기 위해 사용

사용할 메모리를 미리 예약해 놓는다

## 클래스 선언시 자동 선언되는 암시적멤버메서드

### 생성자

객체가 생성될 때 수행할 함수를 정의하는 멤버함수

### 복사생성자

자신과 같은 클래스 타입의 다른 객체에 대한 참조를 인수로 전달받아, 그 참조를 가지고 자신을 초기화 하는 방법 (암시적일 땐 얕은 복사)

- 새 객체가 같은 클래스의 객체로 초기화 될 때
- 객체가 값에 의해 함수에 전달될 때
- 객체가 값에 의해 리턴될 때
- 컴파일러가 임시 객체를 생성할 때

### 대입연산자

객체를 다른 객체에 대입하여 대입된 객체를 다른 객체로 바꾸는 역할 (암시적일 땐 얕은 복사)

### 소멸자

객체가 소멸될 때 수행할 함수를 정의하는 멤버함수

# 문자열 관리 클래스 코딩


문자열 복사, 이어 붙이기, 검색, 특정 문자열 자르기  
Float형의 값을 문자열로 변경

```
#include<iostream>
#include<string>
#pragma warning(disable: 4996)
using namespace std;

class stringManager
{
public:
    string _text1;
    string _text2;
    string _text3; //합칠 스트링
    string str_arr[1000]; //나눌 스트링
    int str_cnt = 0;
    void SSet(string text1, string text2); //셋팅
    void SPast(); //이어붙이기1
    void SPast2(); // 이어붙이기2
    void size(); //크기
    void length(); //크기2
    void search(string text4); //검색
    void SCopy(); //복사
    void Satof(float val); //형변환
    void tok(char* val); //토큰나누기
};
```

```
void stringManager::SSet(string text1, string text2)
{
    _text1 = text1;
    _text2 = text2;
}
```

```
void stringManager::SPast()
{
    _text3 = _text1 + _text2;
    cout << _text3 << endl;
}
```



```
void stringManager::SPast2()
{
    _text3=_text1.append(_text2);
    cout << _text3 << endl;

}

void stringManager::size()
{
    cout<<_text3.size() << endl;
}

void stringManager::length()
{
    cout<<_text3.length() <<endl;
}

void stringManager::search(string text4)
{
    if (_text3.find(text4) != string::npos) {
        cout << text4 << "찾음" << endl;
    }
}

void stringManager::SCopy()
{
    _text1 = _text2;
    cout << _text1 << endl;
}

void stringManager::Satof(float val)
{
    string text1 = to_string(val);

    cout << text1 << endl;
}
```

```

void stringManager::tok(char* val)
{
    char *str_buff = new char[1000];
    strcpy(str_buff, _text3.c_str());
    char *token = strtok(str_buff, val);
    while (token != nullptr) {
        str_arr[str_cnt++] = string(token);
        token = strtok(nullptr, val);
    }
    for (int n = 0; n < str_cnt; n++) {
        cout << str_arr[n] << endl;
    }
}

```

```

int main()
{
    string text1;
    string text2;
    string text4;
    char* Stok;
    char token;
    Stok = &token;

    getline(cin, text1); //문자열 입력
    getline(cin, text2);
    stringManager p = stringManager();
    p.SSet(text1, text2); //값 셋팅
    p.SPast(); //불이기1
    p.SPast2(); // 불이기2
    p.size(); //크기1
    p.length(); // 크기2
    getline(cin, text4);
    p.search(text4); //탐색
    p.SCopy(); // 문자열 복사
    float val;
    cin >> val;
    p.Satof(val);
    cin >> token;
    p.tok(Stok);
}

```



## 문자열 코딩 문제

abcdarfrqecdddajjr”의 문자열에서 처음 완성되는 “car” 단어를 찾은 후(car 문자열 사이에 어떤 문자가 들어와 있어도 상관없음) 다음 완성되는 “car” 단어 바로 전까지의 문자 개수를 찾으세요

```
#include<iostream>
using namespace std;
int main(void)
{
    string val = "abcdarfrqecdddajjr";
    string flag;
    int pos=0;
    int pos2=0;
    int exp=0;
    for (int i = 0; i < 2; i++) //처음 car와 다음 car를 찾기위한 두번 반복
    {
        pos = val.find("c", pos); //초기값 0으로 잡은 pos의 위치부터 c를 찾은후 pos에
        저장
        pos = val.find("a", pos); //c를 찾은 위치 부터 시작해서 a를 찾은후 pos 저장
        pos = val.find("r", pos); //a를 찾은 위치 부터 시작해서 r 찾은 후 pos 저장

        if (i == 0)
        { //첫번째 반복이 끝났으면 마지막의 r의 위치를 pos2에 저장
            pos2 = pos;
        }
        else {
            exp = pos - pos2-1; //두번째 찾은 car위치 값 - 첫번째 찾은 car의 값
            을 빼준뒤 -1을 해준다 -1은 바로 전까지의 문자 개수이기때문이다.
            cout << exp << endl;
        }
    }
}
```

# 시간복잡도

## 시간복잡도

알고리즘이 어떤 문제를 해결하는데 걸리는 시간 (CPU 사용량)

각 명령의 실행 시간이 특정 하드웨어 혹은 프로그래밍 언어에 따라서 그 값이 달라질 수 있기 때문에 알고리즘의 일반적인 시간복잡도는 명령어의 실제 실행시간을 제외한 실행횟수만을 고려한다.

$\Omega(N)$  - 최상 오메가

$O(N)$  - 최악 BigO

$\Theta(N)$  - 평균 세타

1 - 입력 데이터(N)에 관계 없이 항상 일정 상수시간

$\log N < N < N \log N < N^2 < N^3 < 2^n < n!$

## 공간복잡도

알고리즘이 어떤 문제를 해결하는데 필요한 공간 (RAM 사용량)

# 큐와 스택의 차이, 사용 예시

## 스택

LIFO (Last In First Out)

재귀함수 호출과정의 내부적 구현, 피보나치 수열, 운영체제의 프로세스 관리 등

## 큐

FIFO (First In First Out)

주문 시스템, 은행 전산 업무 등

# 벡터와 리스트의 차이

## 벡터와 배열의 차이

	벡터	리스트
메모리공간	미리 할당	추가 시 할당
삽입 / 삭제	맨 뒤만 빠름	중간 수정 용이
사용	데이터의 수정이 자주 없을 때	데이터의 수정이 빈번할 때

## 리스트에서 at을 구현한다면?

```
template<class T>
T LinkedList<T>::at(int index)
{
    if (index < 0 || index > this->getLength())
    {
        std::cout << "잘못된 범위입니다!" << std::endl;
        return 0;
    }

    else{
        int nodelum = -1;

        for (Node<T>* nodes = this->head; nodes != nullptr; nodes = nodes->getNextNode())
        {
            if (nodelum == index)
            {
                return nodes->getData();
            }

            ++nodelum;
        }
    }
}
```

리스트에서 100만개의 데이터를 찾는데 검색 속도를 줄일 수 있는 방법은?  
Set이나 Hash Map을 사용

## 리스트에서 100만개의 데이터를 찾는데 검색 속도를 줄일 수 있는 방법은?

Set이나 Hash\_map을 사용

### - hash table

hash\_map과 hash\_set이 사용하는 자료구조

테이블에 자료를 저장할때 키값을 해시 함수에 대입하여 버킷 번호가 나오면 그 버킷의 빈 슬롯에 자료를 저장

버킷 번호에 자료를 넣으므로 많은 자료를 저장해도 삽입, 삭제, 검색 속도가 거의 일정

### - hash\_map

장점 : 많은 자료를 저장하고 있어도 검색이 빠름

단점 : 자료가 적으면 메모리 낭비와 검색시 오버헤드 생김

검색이 빠르다고 무분별하게 사용하면 안됨. 컨테이너를 추가 삭제하는 것은 list, vector가 훨씬 빠름

적은 요소를 저장하여 검색을 하는 경우네는 vector,list를 사용하고 수천의 자료를 저장하여 검색을 하는 경우에는 hash\_map을 사용

### - hash\_set

hash\_map처럼 hash table을 자료구조로 사용하고 set처럼 key만을 저장

(hash\_set,hash\_map)과map 차이 : 정렬하지 않는다.

hashset과 hashmap의 차이

Set과 Map은 저장된 방식의 차이이다. 둘다 순서는 없다.

HashSet : {1, 2, 3, 4, 5}

HashMap : {a->1, b->2, c->2, d, ->1}

HashMap은 중복 데이터 허용, HashSet는 불허용

# Map과 Unordered Map, Multi Map

## Map과 Unordered Map의 차이

Map은 레드블랙 트리, Unordered Map은 Hash 이다

사전에 분포확률을 정확히 알 수 있어서 적절한 hashing 함수를 만들 수 있다면 unordered map이 빠르다.

사전에 분포를 알 수 없는 경우 map이 더 유리할 수 있다.  
map은 unordered map보다 메모리를 적게 먹으며 정렬된 상태로 저장된다.

## Map과 Multi Map의 차이

둘은 비슷하지만 multimap의 경우 key가 중복될 수 있는 경우에 사용한다

## Map의 레드블랙 트리

레드블랙트리 (balanced binary search tree)의 높이는  $\log n$ 에 바운드 된다  
레드블랙트리에서 search연산은  $O(\log n)$ 의 시간복잡도를 가진다

레드 블랙 트리 조건

1. Root Property : 루트노드의 색깔은 검정(Black)이다.
2. External Property : 모든 external node들은 검정(Black)이다.
3. Internal Property : 빨강(Red)노드의 자식은 검정(Black)이다.  
== No Double Red(빨간색 노드가 연속으로 나올 수 없다.)
4. Depth Property : 모든 리프노드에서 Black Depth는 같다.  
== 리프노드에서 루트노드까지 가는 경로에서 만나는 블랙노드의 개수는 같다.

조건 위배에 따라 아래 규칙이 일어난다.

Restructuring

Recoloring

Restructuring

1. 나(z)와 내 부모(v), 내 부모의 부모(Grand Parent)를 오름차순으로 정렬
2. 무조건 가운데 있는 값을 부모로 만들고 나머지 둘을 자식으로 만든다.
3. 올라간 가운데 있는 값을 검정(Black)으로 만들고 그 두자식들을 빨강(Red)로 만든다.
4. 그리고 남은 자식을 추가한다.

Recoloring

1. 현재 inset된 노드(z)의 부모와 그 형제(w)를 검정(Black)으로 하고 Grand Parent(내 부모의 부모)를 빨강(Red)로 한다.
2. Grand Parent(내 부모의 부모)가 Root node가 아니었을 시 Double Red가 다시 발생 할 수 있다.

## 자료구조 및 STL

자료구조 및 STL

자료구조 및 STL

자료구조 및 STL

## 자료구조 및 STL

자료구조 및 STL

## 자료구조 및 STL



## 자료구조 및 STL



# 정렬 알고리즘

## STL의 알고리즘에서 사용되는 정렬 알고리즘은?

algorithm.h에 있는 정렬은 퀵 정렬을 기반으로 구현되어 있다

퀵정렬의 단점인 순서대로 정렬되어 있을 때에 시간복잡도가  $O(n^2)$ 나오는 문제는 해결되어 있다

## 퀵 소트

다른 원소와의 비교만으로 정렬을 수행하는 비교 정렬에 속함

- 퀵 정렬을 하기 위해선  $O(\log n)$ 만큼의 memory를 필요로한다.
- 최악의 경우  $O(n^2)$  평균  $(n \log n)$  최선  $n(n \log n)$

다른  $(n \log n)$ 정렬 보다 빠른 이유

- 대부분의 실질적인 데이터를 정렬할 때 제곱 시간이 걸릴 확률이 거의 없도록 알고리즘을 설계하는 것이 가능하기 때문이다.

## 퀵 소트에서 $n^2$ 이 걸릴 경우

정렬할 데이터가 이미 정렬되어 있을 때  $n^2$ 이 걸린다

같은 자리의 많은 숫자가 존재한다고 할 때 효율이 좋은 정렬 알고리즘 (메모리 상관 x)

병합 정렬은 최악 최선 모두  $n \log n$  속도로 처리가 가능하다

자료구조 및 STL

# 렌더링 파이프 라인

## IA (Input Assembler) 입력 조립기

정점 정보가 들어있는 버퍼로부터 정보를 읽어온다

## VS (Vertex Shader) 정점 셰이더

IA에서 입력받은 정보를 토대로 정점을 하나씩 처리

## HS(Hull Shader) 덮개 셰이더

정점 셰이더가 넘겨준 기본 도형들을 받아 테셀레이션 계수들을 결정

## TS(Tessellator) 테셀레이터

테셀레이션 계수를 바탕으로 도형을 분할하기 위한 표본정점 결정 및 무게 중심 좌표를 산정

## DS(Domain Shader) 영역 셰이더

테셀레이션된 각 점마다 무게 중심 위치들을 출력하고 기하구조로 변환

## GS(Geometry Shader) 기하 셰이더

완성된 형태의 기본 도형들을 처리하거나 생성

## RS(Rasterize) 래스터화기

화면에 출력하기 위해 래스터 이미지로 변환, 모든 정점별 특성을 해당 픽셀 위치에 맞게 보강

주어진 기하 구조가 렌더 대상의 어떤 픽셀들을 덮는지 파악해서 그 픽셀들에 대한 단편 자료를 산출합니다. 각 단편은 모든 정점별 특성을 해당 픽셀 위치에 맞게 보강한 값들을 가지고 이렇게 생성된 단편은 픽셀 셰이더로 넘어갑니다.

## PS(Pixel Shader) 픽셀 셰이더

연결된 각 렌더 대상을 위한 색상 값을 출력

## OM(Output Merge) 출력 병합기

픽셀 셰이더 출력을 파이프라인에 연결된 깊이/스텐실 자원 및 렌더 대상 자원에 합친다.

## CS(Compute Shader) 계산 셰이더

전적으로 범용 계산에만 쓰이는 하나의 독립적인 파이프라인. 통상적인 렌더링 파이프라인과 개별적으로 수행됨.

1. 셰이더 프로그램의 호출이 특정 단계에서 입력이 처리되는 방식에 제약이 있고 스레드가 쓰이는 방식을 개발자가 직접 제어하는 것이 불가능했던 단점을 극복해주었습니다.
2. 그룹 공유 메모리 블록  
한 스레드 그룹에 속하는 모든 스레드는 그 그룹의 공유 메모리에 접근할 수 있게 하는 것으로 실행 도중에도 스레드들이 이 메모리 블록을 통해 서로 자료를 주고받을 수 있습니다.
3. 리소스에 대한 임의의 읽기 접근과 쓰기 접근 동시 가능



# 쿼터니온

## 짐벌락

오일러 각을 이용하여  $x, y, z$ 축을 회전시킬 때, 특정 축이 특정 각으로 회전한다고 가정할 경우, 해당 특정 축을 제외한 다른 두 축이 맞물려 겹치는 현상이 발생하게 되는데 이 때 다른 두 축 중 한 축이 소실되어 다른 나머지 한 축과 같이 회전하게 된다.

## 짐벌락의 예방

완전히 해결할 수는 없음

1. 오일러 각 회전의 순서를 바꾼다.

(1) 가장 자주 회전하게 될 축을 먼저 회전한다.

(2) 가장 회전을 안 하게 될 축을 2번째로 회전한다.

(3) 남은 축을 회전한다.

(이유 : 순서에 의한 짐벌락 현상은 두 번째 회전하는 축이 90도(-90도)를 회전하게 되면 1번째 축과 3번째 축이 겹치면서 발생한다)

2. 가상의 축을 기준으로 축회전을 이용한다.

(임의의 축을 놓고 회전할 때 보간의 처리(선형보간, 구면보간)의 문제로 회전 시에 뚝뚝 끊겨보이게 된다.)

3. 세축을 동시에 돌리는 쿼터니언(사원수)을 사용한다.

## 쿼터니온

쿼터니언은 '사원수'를 의미하며, 한 개의 스칼라 값과 3차원 벡터의 조합으로 이루어진다.

$q = w + xi + yj + zk$  로 표현되며, 실수부를  $s$ , 허수부를  $v$ 로 바꾸어

$q = (s, v)$ 로 표현하기도 한다.

목적 : 쿼터니언은 해밀턴이 정의하였으며, 일반적인 복소수( $a + bi$ )를 2차원 이상의 차원에서 적용할 수 있어야 한다고 생각하여 이론을 정립하였다

### 사원수의 곱

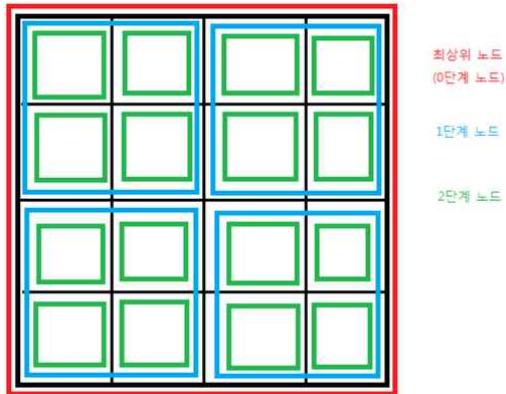
$\times$	1	$i$	$j$	$k$
1	1	$i$	$j$	$k$
$i$	$i$	-1	$k$	$-j$
$j$	$j$	$-k$	-1	$i$
$k$	$k$	$j$	$-i$	-1

# 쿼드트리, 옥트리 컬링

## 쿼드트리 컬링

공간을 재귀적인 호출로 4개의 자식노드로 분할하는 방법

1. 하나의 넓은 월드 지형을 1/4로 나눈다
2. 4개의 노드를 부모로서 1/4로 나눈다
3. 초기의 월드에는 16개의 노드 존재
4. 재귀적인 호출을 통해 각 노드간의 간격이 1보다 작거나 같을 때 까지 분할한다.



(맵의 크기는 반드시 홀수여야 한다.  $(2^n + 1)$ )

## 옥트리 컬링

자식노드를 8개씩 가지는 트리. 쿼드트리가 높이 면에 대한 분할을 하지 않는다는 점에 비해 옥트리는 높이에 대한 분할까지 시도

# 프러스텀 컬링

## 프러스텀 컬링

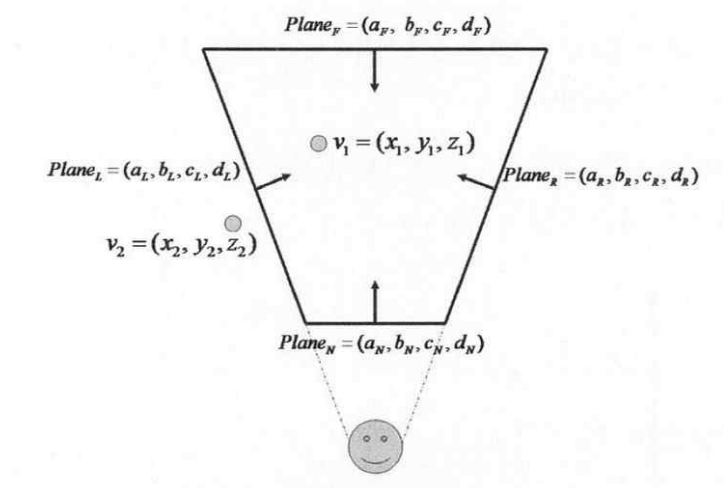
카메라의 시야 범위에 포함되는 것들만 렌더링하고, 나머지는 렌더링 하지 않는 기법

a, b, c는 평면의 방향을 나타내는 법선벡터

d는 평면과 원점간의 거리를 나타내는 값

결과값	의미
$ax + by + cz + d = 0$	점이 평면 위에 있다.
$ax + by + cz + d > 0$	점이 평면 앞에 있다.
$ax + by + cz + d < 0$	점이 평면 뒤에 있다.

6개의 평면 방정식에 점의 좌표를 대입해 모든 평면 방정식의 결과값이 양수(+)이면 절두체 내부에 포함



# 오클루전 컬링

## 오클루전 컬링

다른 오브젝트에 가려 카메라에 보이지 않는 오브젝트의 렌더링을 비활성 하는 기능

3D 컴퓨터 그래픽의 세계에서는 대부분의 경우 카메라에서 먼 오브젝트에서 먼저 그려지며, 더 가까이 있는 오브젝트가 차례차례 덮어써지게 됩니다("오버 드로우"라고 부릅니다). 오클루전 컬링은 그만큼 당연한 기능이 아닙니다. 오클루전 컬링은 절두체 컬링(Frustum Culling)과는 다릅니다. 절두체 컬링은 카메라의 표시 영역에서 벗어난 오브젝트의 렌더링을 비활성화하는 것이고, 오클루전 컬링과 같이 오버 드로우에 의해 안 보이게 되는 오브젝트 렌더링은 무효화하지 않습니다. 또한, 오클루전 컬링을 사용하면서 동시에 절두체 컬링의 혜택을 누릴 수 있습니다.

## H/W 가속을 사용하는 방식

Occluder를 그려서 z버퍼를 구성하고 이 때, pixel은 write하지 않고 zTest만 한다.

```
LPDIRECT3DQUERY Query;
pDevice->CreateQuery(D3DQUERYTYPE_OCCLUSION, &pQuery);
pQuery->Issue(D3DISSUE_BEGIN);

Render();

pQuery->Issue(D3DISSUE_END);
DWORD dwPixelCount;;
pQuery->GetData(dwPixelCount,sizeof(DWORD), D3DGEDATA_FLUSH);
(오브젝트가 1픽셀이라도 보일 때 0이 아닌 숫자를 반환해줌)
```

장점 : 구현이 쉬움

단점 : 성능이 별로임 (렌더패스를 두 번 해야 하기 때문. 쿼리문에서 한번, 그려지는 데이터면 백 버퍼에 그리는 과정 한번)

## S/W 간단한 경계를 만들어서 오클루전컬링을 하는 방식

절두체 속에 오클루전 영역을 하나 만든다.

오클루전 영역에서 카메라와 가장 가까운 오클루전 z값을 찾는다.

그리고 이 값보다 앞에 있는 오브젝트는 컬링을 적용할 필요 x

오클루전 영역을 구성하는 5개의 평면의 방정식을 구한다

5개의 평면과 오브젝트의 바운딩스피어와의 거리를 계산한다 (프러스텀 컬링과 동일)

장점 : 렌더링을 하지 않고 z값만 비교하기 때문에 속도가 빠름

단점 : 정교하지 못함

## 4. Occlusion Culling



## 백페이스 컬링

## 2. BackFace Culling

뒷면을 없애라! 백페이스 컬링!

카메라에서 보이지 않는 뒷면의 폴리곤을 제거하는 기법

CPU에서 계산하는 소프트웨어적 처리 방법과 GPU에서 계산하는 하드웨어적 처리방법이 있다.

GPU가 발전한 지금은 강 하드웨어에 맡기면 된다.

최종적으로 레스터라이즈된 삼각형의 정점이 시계 반대방향(CCW) 순서면 컬링된다.

시계방향으로 컬링하여 앞면 컬링을 할 수도 있다.

# WVP의 Project와 Unproject

## Project

3D를 2D로 변환하며 z-buffer값에 따라서 어떤 픽셀을 맨 앞에 그려줄 지를 결정하는 과정

우리가 화면으로 보는 것은 2차원이다. 즉 3차원 좌표계를 2차원 좌표계로 바꿔줘야 한다. 3차원 -> 2차원으로 변환하는 가장 간단한 방법은 한 축을 없애면 된다. 그래서 나타난게 Z-Buffer(Z버퍼)라 할 수 있다. 카메라로 부터 가까이 있는 부분부터 가장 먼 부분까지 z값을 0~1로 표현한다. 그래서 z값이 낮을수록 점들을 화면에서 제일 위에 그려준다. 이 과정이 프로젝션 변환이다.

## Unproject

화면 상의 2D좌표를 3D상의 좌표로 환산하는 과정 (마우스 피킹)

## 클리핑 평면

3D 공간에서 기하학적인 구조를 잘라내는 데 사용

뷰포트를 만들 때 사용하는 near와 far평면이 이에 해당

## 반직선과 면(삼각형)의 충돌

- 1)삼각형의 법선 벡터와 반직선의 벡터를 내적한 값이 0이면 false (평행)
- 2)반직선과 삼각형의 평면상의 교점을 구함
- 3)교점에서의 반직선의 t값이 0~1이 아니라면 false (유한직선의 범위)
- 4)교점에서의 삼각형의 w, v 값이 0 보다 작거나 w + v 값이 1보다 클 때 false (유한삼각형의 범위)
- 5)else true

## 플레이어가 지면 위에 있는지 판별하는 방법

# 인스턴싱

## 인스턴싱

같은 오브젝트가 아주 많이 그려질 때 드로우 콜을 최대한 줄일 수 있는 방법으로, 동일한 메시를 사용하는 인스턴스들을 1번의 DP Call로 렌더링하는 기술

### 1) Using Vertex Shader

#### - 장점

- (1) Buffer에 Lock을 걸지 않아도 구현이 가능
- (2) 빠른 속도를 보장

#### - 단점

- (1) 셰이더 상수 개수 제한으로 한꺼번에 수백개 이상은 그리지 못함
- (2) Skinning Shader등 셰이더 상수를 많이 사용하는 방법과 병행하기 힘들
- (3) DP콜이 Dynamic Buffer에 비해 상대적으로 많음.

### 2) Using Dynamic Buffer

#### - 장점

- (1) 버퍼 크기만큼 인스턴싱이 가능하므로 Shader보다 더 많이 가능
- (2) 스킨링 애니메이션등, Shader와 연계 되지 않음.
- (3) DP콜이 Shader보다 상대적으로 적음

#### - 단점

- (1) Lock을 걸어야 하기 때문에 부하가 상당히 큼.

### 언리얼)

폴리지(Foliage) 인스턴스드 메시 시스템은 랜드스케이프 액터나 다른 스태틱 메시 액터나 BSP 지오메트리에 스태틱 메시 세트를 빠르게 칠하거나 지울 수 있는 시스템입니다. 이러한 메시들은 하드웨어 인스턴싱을 사용하여 렌더링되는 일괄(batch) 그룹 속에 자동으로 묶이는데, 이는 단 하나의 드로우 콜에 다수의 인스턴스를 렌더링할 수 있다는 뜻입니다.

# LOD

## Level Of Detail

단계에 따라서 메시의 모델링 데이터의 정밀도를 조절하는 기술

### 정적 LOD

거리에 따른 메시의 정밀도를 미리 정해 놓고 카메라와의 거리에 따라 단계별로 바뀌가며 출력

장점) 연산이 간단하기 때문에 속도가 빠름

단점) 여러 단계의 메시를 추가로 가지고 있어야 하기 때문에 메모리 낭비가 심하고

거리에 따라서 메시의 단계가 급격히 변하기 때문에 팝핑현상이 일어날 수 있음

### 동적 LOD

카메라와 물체의 거리에 따라서 실시간으로 메시의 정밀도를 변화시키는 기법

장점) 팝핑현상이 적고 낭비되는 메모리가 없음

단점) 메시분할이나 간략화에 대한 추가 연산이 필요하기 때문에 상대적으로 속도가 느림

정적 LOD는 전처리 과정에서 미리 메시의 정밀도를 정하며,

동적 LOD는 실시간으로 메시의 정밀도를 정한다.

### Popping

어떤 오브젝트의 LOD단계가 변할 때, 갑작스럽게 모양이 바뀌는 현상

낮은 레벨에서 높은 레벨로 갈 때 사라지는 버텍스의 위치 값을 레벨간의 차이에 의한 보간을 이용하여 해결

### Crack

LOD 사용 시에 인덱스가 참조하는 정점이 줄어들어 지형에 구멍이 뚫리는 현상

주위 레벨을 체크하여 레벨 차이에 따라 삼각형을 덮어 씌워 해결

이웃노드를 분할 했는지 확인해서 덮어 씌워 해결



# 알파소팅을 해야하는 이유

## 알파소팅

불투명한 오브젝트를 모아서 모두 그리고 알파가 적용된 이펙트나 오브젝트를 맨 마지막에 그린다

알파가 없는 오브젝트들은 화면 앞의 오브젝트부터 뒤에 있는 오브젝트 순으로 렌더링하는게 빠릅니다.

이것은 화면 앞의 오브젝트를 렌더링하면 그 뒤에 가려진(!) 오브젝트는 렌더링하지 않아도 되기 때문에 속도가 빨라지는 원리입니다.

하지만 반투명, 투명 오브젝트는 위와 같이 앞에서부터 그리더라도 뒤에 있는 오브젝트가 그려져야 하기 때문에

위와 같이 처리할 수 없습니다. 그래서 반투명, 투명 오브젝트는 반대로 뒤에서부터 그려야 합니다.

그래야 앞에 반투명 오브젝트를 그리더라도, 먼저 그려진 뒤의 오브젝트가 비춰서 보이겠죠 ^^

이런 이유 때문에 게임에서는 알파가 없는 것들 따로 그리고, 알파가 있는 것들을 따로 그립니다.

## 알파소팅 문제점

피봇으로 앞뒤를 판정하기 때문에 완벽하게 앞뒤를 구분하는 것이 불가능

반투명 안에 반투명이 있을 경우 피봇축이 동일하기 때문에 먼 순으로 그리는 것이 문제가 됨  
(안에 있는 반투명이 나중에 그려져야 하지만 그러려면 피봇 위치상 밖에 위치)

## 알파소팅 문제 해결법

1. 물체를 자잘하게 쪼개서 포현
2. 알파부분을 최소화 하여 디자인
3. 알파테스팅 후 , 정렬
4. Z Write를 사용하지 않기
5. 렌더링 레이어 만들기
6. 여러 Pass를 사용하여 그림

## 알파블렌드

블렌딩 설정에 따라 색상을 혼합

## 알파테스트

설정값을 기준으로 해당되는 범위의 값을 제거

# Ambient, Diffuse, Specular, Emissive

## Ambient (주변광)

광원의 색상이 물체에 의해 반사되어 나타나는 주변 반사광의 색상

## Diffuse (확산광)

물체 표면 자체의 색상

## Specular (반사광)

광원이 반사되어 물체의 가장 밝게 빛나는 부분의 색상

## Emissive (방사광)

발광체인 물체를 표현할 때 사용

A : 빛이 비추지 못하는 영역

B : 빛이 비추어지지만 빛의 각도에 따라서 빛에 대한 영향이 연속적으로 바뀌는 영역

C : 빛이 곧바로 비추어져 빛의 색이 그대로 투영되는 영역

Ambient는 A, B, C 에 모두 영향을 미침

Diffuse는 B, C 에 영향을 미침

Specular는 C에 영향을 미침

Shininess의 값은 C 영역의 크기와 밝기를 결정

## 램버트의 법칙

디퓨즈 표면으로 들어오는 빛은 모든 방향으로 같은 강도로 반사되는 난반사로 가정한다

물체의 표면에 들어오는 빛의 세기는 표면에서 빛까지의 벡터  $l$ 과 법선벡터  $n$ 이 이루는 코사인 값에 비례

## 라이트맵

3D CG에서 빛의 방향과 세기에 따라 생성되는 빛을 받는 오브젝트의 그림자, 반사면, 면의 밝기 차이 등의 라이팅 정보를 사전에 저장하는 텍스처

플레이어가 바라보는 시야 및 씬의 환경 배치, 광원의 수와 조도 등에 따라서 화면에 출력되는 빛은 시시각각 다르게 보이게 되며 이를 시각적으로 처리하는 연산에는 빛의 입자 정보를 일일이 따져보아야 하는 상당한 부하가 수반된다.

게임 같은 경우, 빛을 계산하는 것이 콘텐츠의 전부가 아니기 때문에 라이팅에서 컴퓨팅 비용을 많이 사용하게 되면 게임플레이 코드 처리나 물리 연산 등에서 활용할 수 있는 자원이 줄어든다. 때문에 최대한 효율적으로 빛을 처리하고 다른 부분에서의 퍼포먼스도 보장하기 위하여 고안된 방법이 바로 라이트 정보를 미리 오브젝트의 텍스처에 '라이트맵'이라는 별도의 채널로 저장하여 마치 빛을 받고 있는 듯 보이게 하는 라이트맵 방식이다. 이를 흔히 업계에서는 '라이트맵을 굽는다.'고 표현한다.

라이트맵 자체는 3ds Max나 Blender 등의 모델링 툴에서 텍스처 UV와는 다른 채널에서 연애평하게 되며, 이를 언리얼 엔진이나 유니티 등 게임 엔진에서 로드하여 라이트맵으로 사용하게 된다.

실시간 동적 조명에 비해 단순 텍스처 데이터를 하나 추가로 불러와 보여주는 것이기에 계산량은 월등히 줄어들게 되는 것이 가장 큰 이점. 일부 AAA급 게임이나 다이나믹 라이팅에 특화된 콘텐츠가 아닌 이상, 사실상 관련 업계 표준의 방식이라고 해도 과언이 아니다. 또한, 미리 텍스처에 라이팅 정보가 저장되므로 사양에 관계없이 동일한 고퀄리티의 라이팅을 표현할 수 있다는 것도 큰 장점 중 하나이다.

단점으로는 빛의 정보를 미리 저장하기 위해 제작된 레벨의 라이트 배치에 맞게 프로젝트에 따라 때로는 아주 긴 시간동안 베이킹을 해주어야 한다는 점, 한 번 굽고나면 결과를 확인하고 수정한 뒤 다시 긴 시간의 빌드가 필요한 점, 실제 씬에서 라이트의 세기나 그림자 방향의 변화를 표현할 수 없기에 종종 이질감이 심할 경우가 있다는 점 등이 있다.

# 디퍼드 렌더링

## 포워드 렌더링

전통적인 렌더링 방식으로 3D공간에 존재하는 폴리곤을 픽셀화하여, 그 픽셀마다 셰이딩과 라이팅 연산을 더하는 방식으로 묘사된다. PS2 이전의 게임은 모두 포워드 렌더링 방식이라고 생각하면 된다.

### 장점

비교적 저사양 기기에서도 잘 작동하며 PC의 경우 안티앨리어싱 처리를 하드웨어에서 지원받기 때문에 거의 완벽한 처리를 할 수 있고 반투명 처리도 문제가 없다.

오랫동안 애용되어온 렌더링 방식이기에 문제가 생겨도 회피하거나 해결책이 만들어져 있는 경우가 많다.

해상도가 올라가도 요구하는 메모리가 디퍼드에 비해 적다.

### 단점

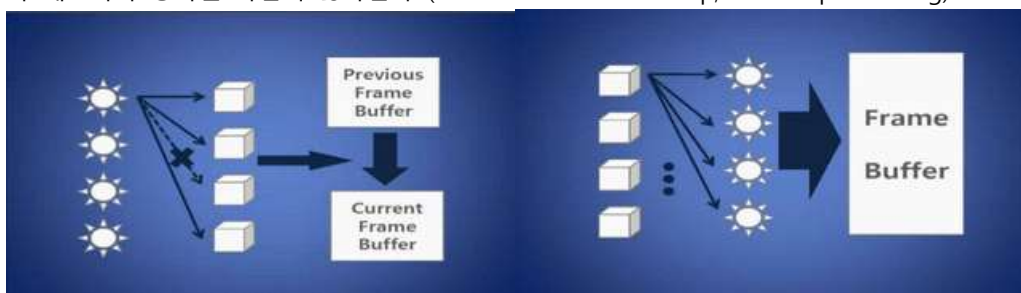
해당 라이트에 영향을 받지 않아도 라이트 연산이 되며 렌더링되지 않는 면(Culling)도 셰이딩 연산을 해야만 한다.

때문에 라이팅 연산이 느리고 여러 오브젝트로 복잡한 화면을 구성하거나 폴리곤이 많은 모델을 렌더링 걸기 불리한 방식이다.

그림자 처리가 어렵고 화면 깊이값을 이용한 포스트 이펙트는 따로 처리를 해주어야 한다.

Multi-pass 사용시 라이트에 영향을 받는 오브젝트만 연산을 하게 되지만 높은 배치 카운트를 가지게 되며

각 패스마다 중복된 작업이 많아진다. (Vertex Transform & setup, Anisotropic filtering)



## 디퍼드 렌더링

한 화면에 수많은 라이팅 효과를 넣고 싶어 만든 렌더링 기법. 실시간으로 라이팅에 반응하는 셰이더를 쓰는 게임은 기본적으로

디퍼드 렌더링을 쓰고 있다. 구현 방식은 폴리곤을 픽셀화하여 포토샵의 레이어처럼 정보를 나누어 메모리에 저장(G Buffer)한다.

여기에서 각종 셰이더와 라이팅 효과를 거쳐 화면에 보여준다.

### 장점

수많은 동적 라이팅을 실시간으로 보여줄 수 있다. 굉장히 배치가 간단해지고, 엔진에서 관리가 쉽다.

일반적인 그림자 테크닉들과 통합이 쉽다.

셰이더를 간단하게 지원하고 픽셀정보를 메모리에 저장하면서 생기는 여러가지 이점을 쓸 수 있다. 예를 들면 Depth을 이용한

특수효과 같은 것들. 오브젝트는 라이팅 계산이 없이 속성만을 렌더링하기 때문에 수많은 오브젝트를 표현할 때 유리하다.

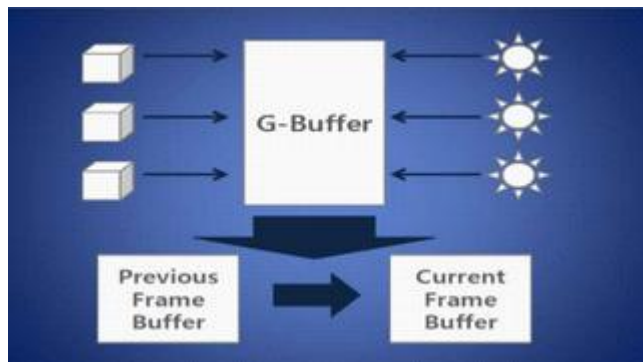
화면에 실제로 렌더링 되는 픽셀만 라이팅 계산을 수행한다.

### 단점

뒤에 그려지는 오브젝트 정보가 비디오 메모리에서 소실되기 때문에 알파가 빠지는 오브젝트를 표현할 수 없다.

요구하는 하드웨어 사양이 포워드 렌더링 방식보다 높고 해상도가 올라갈수록 요구하는 메모리가 기하급수적으로 늘어난다.

안티앨리어싱 구현이 까다롭다.



일반적인 포워드 라이팅은 광원 개수 x 메시 개수의 드로우콜 발생  
디퍼드라이팅은 광원 개수 + 메시 개수의 드로우콜

### GBuffer

: 지오메트리 버퍼, 씬의 각 점에 대한 정보를 갖는 최종 렌더링 되는 픽셀 수와 같은 크기의 렌더 타겟 이미지 집합

### 디퍼드 라이팅

1. 깊이와 노멀 정보에 대한 GBuffer를 통해 씬을 렌더링
2. 각 광원이 영향을 끼치는 픽셀을 찾아 해당 GBuffer 데이터를 읽어 들인 후 조명값을 계산해 밝기를 합산하는 렌더링 타겟에 저장
3. 마지막으로 합산된 밝기와 메시의 색을 결합해서 최종 픽셀 색상 계산

### 디퍼드 셰이딩

1. GBuffer에 렌더링 하되 후처리(포스트 프로세싱)에 필요한 정보를 모두 담는다
2. 각 광원이 영향을 끼치는 픽셀을 찾아 해당 GBuffer를 읽어 들인 후 픽셀의 조명 색상을 계산해 중첩 버퍼에 저장한다

### [디퍼드 라이팅과 다른 점]

- 후처리와 기타 정보 등, GBuffer에 저장해야할 정보량이 늘어난다.  
(요즘 GPU기준으로 개발하고 있다면 메모리가 부족하지 않기 때문에 ㄱㄷ)
- 씬을 한번만 렌더링 하면 된다

### [디퍼드 방식 단점]

- GBuffer에 저장된 값으로 한번에 계산되고 뽑아낸 픽셀을 사용하기 때문에 반투명 처리가 어렵다.
- 먼저 불투명 객체를 디퍼드로 그린 후, 반투명 객체를 포워드 방식으로 그리면 해결 가능

# Ambient Occlusion

## Ambient Occlusion

렌더링 과정 중 셰이딩의 한 방식. 각각의 표면이 광원에 얼마나 노출되어 있는지를 계산하여 그림자를 더해주는 기술이다. 일반적으로 광원과 물체를 던져놓고 지역 조명 방식[1]으로 처리를 시행하면 그림자는 있지만 실제와 미묘하게 다른데, AO는 이를 보정하고 실제에 더 가깝게 보이도록 만든다. (주위에 빛이 있으면 밝아지고 없으면 어두워진다는 개념)



## SSAO - Screen Space Ambient Occlusion

가장 많이 쓰이는 전통있는 AO. 현재 모니터에 렌더링되는 영역만을 Screen Space로 인식하여 해당 영역에만 앰비언트 오클루전을 적용함으로써 일정 수준 최적화 된 그래픽을 제공한다.

## HBAO+ - Horizon Based Ambient Occlusion+

NVIDIA가 제공하는 기술. 최근 들어 쓰이는 차세대 AO로 배틀필드4, 워쳐3, 라이즈 오브 톰 레이더 등 많은 곳에 적용되었다. SSAO보다 좀 더 높은 연산능력을 필요로 하지만 그림자의 정확성이 좀 더 높아진다.

## VXAO - Voxel Accelerated Ambient Occlusion

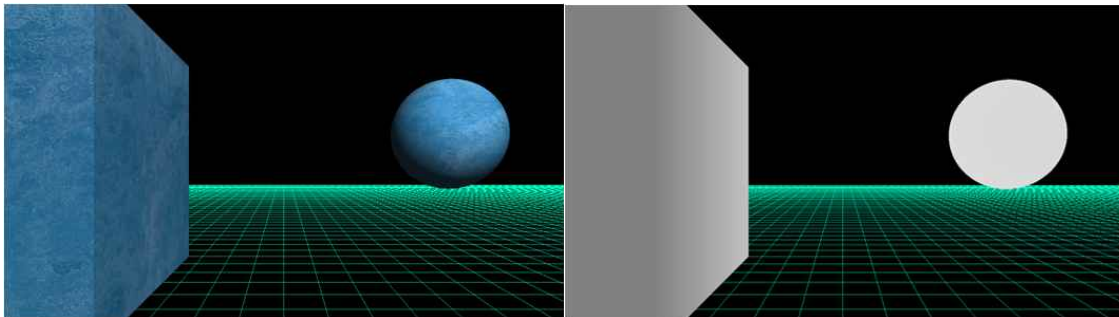
NVIDIA GameWorks에서 제공하는 기술로 라이즈 오브 톰 레이더에서 쓰였다. 말 그대로 AO를 적용할 부분을 복셀 단위로 계산하기 때문에 가장 정확성이 높은 음영을 출력할 수 있지만, 이에 따라 요구되는 GPU의 연산능력이 매우 높다.

# 깊이 버퍼

## 깊이 버퍼

3D 상에서 물체의 차폐를 결정 짓는 값으로, 주로 프러스텀에서 모든 픽셀의 깊이 값을 기록하는 데 사용한다. 0 ~ 1 사이의 값을 사용하며 같은 위치에 여러 픽셀이 자리하는 경우 깊이 값을 이용하여 어느 픽셀을 사용할지 고르게 된다.

이 때 대략 0 ~ 0.9f의 깊이 버퍼들이 장면 깊이 범위의 첫 10%에 사용되고, 0.9f ~ 1.0f가 나머지 90%에 사용된다.



## 깊이 버퍼를 이용하여 구현할 수 있는 것

### 깊이 버퍼를 이용한 그림자 매핑

- 1 패스 : 그림자 텍스처 생성(카메라를 광원 위치에 두고 투영 공간의 z값을 가지고 렌더링).
- 정점셰이더에서 투영 공간의 좌표 값을 그대로 출력하여 픽셀 셰이더에서 z좌표 값을 w 값으로 나누어 0 ~ 1의 범위의 깊이값을 얻는다.
- 2 패스 : 일반 화면 렌더.
- 깊이 값 텍스처를 오브젝트에 맵핑.
- 각 픽셀과 광원과의 거리를 구함.
- 오브젝트의 깊이 값과 깊이 버퍼의 값을 비교하여 깊이 값보다 작은 텍스처의 부분을 그림자로 표현한다.

## 해상도 문제 해결 방법

1. PSM (Perspective Shadow Map)
  - : 광원을 기준으로 가까이 있는 건 해상도 높게, 멀리 있는 건 해상도 낮게
  - \* 광원에서 멀리 있는 물체를 카메라 시점에서 가까이 보면 품질 낮음
2. LSPSM (Light Space Perspective Shadow Map)
  - : PSM 기법이 최상의 상태를 얻을 수 있도록 그림자맵 생성시의 좌표계를 조정
  - \* 조정조건은 그림자맵을 생성하는 범위를 그림자가 될 수 있는 모든 3D오브젝트를 포괄
3. Cascaded LSPSM
  - : LSPSM기법으로 깊이 그림자 기법의 약점을 많이 극복할 수 있지만, 그래도 먼 곳까지 그려지는 광대한 옥외 씬에서는 역시 1장의 그림자맵으로 차폐구조를 생성하고 있기 때문에 해상도 부족 현상이 발생. 그래서 복수의 그림자 맵을 사용하는 방법
  - \* 시점으로부터 시계를 시점 위치의 가까운 위치에서 먼 곳 까지 적당하게 분할해서, 분할한 시계 각각에 대해 LSPSM 적용

퍼센티지 클로저 필터링 : 그림자의 외곽선을 부드럽게 필터링 해주기 위해 사용.

배리언스 그림자맵 : 하드웨어 텍스처 필터링이 가능하도록 깊이 값을 저장하는 방법.



# 크리티컬 섹션에 대한 설명

## 크리티컬 섹션

### 임계 영역/구역에 대한 문제를 해결하기 위한 동기화 기법

특정 임계영역에 대한 키(크리티컬 섹션 객체)를 가져야만 임계 영역에 접근하도록 하는 것  
마이크로소프트는 동기화 기법과 임계 영역을 동일한 이름으로 사용

## 뮤텍스

### 공유되는 임계 영역을 가진 스레드들이 실행 시간이 겹치지 않게 각각 단독으로 임계 영역에 접근하게 하는 기법

mutual exclusion의 줄임말이며, 상호배제라고도 한다.  
임계영역을 보호하기 위한 개념으로, 둘 이상의 프로세스가 공유자원에 대해 동시에 읽거나 쓰는 것을 방지한다.

## 세마포어

### 프로세스 간의 신호를 주고받기 위해 사용되는 정수 값

리소스의 상태를 나타내는 카운터로, 다음 세 가지의 원자적인 연산만을 지원한다.

- 1) initialize : 세마포어 초기화 (음이 아닌 정수값으로 초기화)
- 2) decrement : 한 프로세스가 임계 영역을 점유할 시 사용
- 3) increment : 임계 영역을 점유한 프로세스가 빠져나올 시 사용

- 이진 세마포어 : 0 또는 1의 값을 가지는 세마포어, 뮤텍스와 같다 볼 수 있다.
- 카운팅/범용 세마포어 : 2 이상의 정수 값을 가질 수 있는 세마포어. 세마포어 카운터의 최대 크기만큼 스레드/프로세스가 접근 가능

## 모니터

### 모니터는 한 스레드가 임계 영역의 사용이 중지되면 사용 대기 상태의 스레드에게 알리는 역할

한 스레드가 임계 영역을 점유하고 있을 시, 임계 영역을 점유하고 싶은 다른 한 스레드는 사용 대기 상태에 놓이게 되고 점유하고 있던 스레드가 접근 권한을 사용 대기 상태의 스레드에게 줄 때 모니터가 사용된다.

## ++

- 세마포어는 뮤텍스가 될 수 있지만, 뮤텍스는 세마포어가 될 수 없다.
- 세마포어는 소유할 수 없으며, 뮤텍스는 소유할 수 있고 소유주가 그에 대한 책임을 진다.
- 뮤텍스의 경우 뮤텍스를 소유하고 있는 스레드가 이 뮤텍스를 해제할 수 있다. 하지만, 세마포어는 소유하지 않고 있는 스레드가 세마포어를 해제할 수 있다.
- 세마포어는 시스템 범위에 걸쳐있고 파일 시스템 상의 파일 형태로 존재한다. 하지만, 뮤텍스는 프로세스 범위를 가지고 프로그램이 종료될 때 자동으로 지워진다.
- 세마포어는 동기화 대상이 여러개 일 때, 뮤텍스는 동기화 대상이 오로지 하나 일 때 사용된다.

++

- 세마포어는 저수준의 어셈블리 언어에서 주로 사용되었으며, 모니터는 고수준의 자바와 같은 언어에서 사용된다.
- 세마포어는 사용자가 접근 권한을 제어해야 하지만, 모니터는 내부적으로 접근 권한을 제어(상호배타)한다. 그러나 모니터 또한 `wait()`, `notify()`, `notifyAll()`을 통해 임의적으로 접근 권한을 제어할 수 있다.

## 데드락에 대한 설명과 발생조건

### 데드락

스레드/프로세스가 자원의 접근 권한을 얻지 못해 다음 처리를 하지 못하는 상태 (교착 상태)

\* 데드락 발생 예시

- 두 개의 스레드 T1, T2와 자원 R1, 자원 R2가 있으며 두 개의 스레드 모두 자원 R1, 자원 R2가 필요할 때, 스레드 T1가 자원 R1을 얻고 스레드 T2가 자원 R2를 얻었다고 가정하면 스레드 T1은 자원 R2의 접근 권한 할당을 대기하고, 스레드 T2는 자원 R1의 접근 권한 할당을 대기하게 된다.

### 데드락의 발생조건

데드락은 아래의 4개 조건이 모두 만족하여야 발생한다.

1) 상호 배제 (Mutual exclusion)

- 자원은 한 번에 한 프로세스만이 사용할 수 있어야 한다.

2) 점유 대기 (Hold and wait)

- 최소한 하나의 자원을 점유하고 있으면서 다른 프로세스에 할당되어 사용하고 있는 자원을 추가로 점유하기 위해 대기하는 프로세스가 있어야 한다.

3) 비선점 (No preemption)

- 다른 프로세스에 할당된 자원은 사용이 끝날 때까지 강제로 빼앗을 수 없어야 한다.

4) 순환 대기 (Circular wait)

- 프로세스의 집합  $\{P_0, P_1, \dots, P_n\}$ 에서  $P_0$ 는  $P_1$ 이 점유한 자원을 대기하고  $P_1$ 은  $P_2$ 가 점유한 자원을 대기하고  $P_2 \dots P_{n-1}$ 은  $P_n$ 이 점유한 자원을 대기하며  $P_n$ 은  $P_0$ 가 점유한 자원을 요구해야 한다.

## 데드락의 처리

### 교착상태 예방 및 회피

#### - 예방(Prevention)

- : 교착 상태 발생 조건 중 하나를 제거함으로써 해결하는 방법
- 자원의 낭비가 심하다.

##### 1) 상호 배제 (Mutual exclusion) 부정

- 여러 개의 프로세스가 공유 자원을 사용할 수 있도록 한다.

##### 2) 점유 대기 (Hold and wait) 부정

- 프로세스가 실행되기 전 필요한 모든 자원을 할당한다.

##### 3) 비선점 (No preemption) 부정

- 자원을 점유하고 있는 프로세스가 다른 자원을 요구할 때 점유하고 있는 자원을 반납하고, 요구한 자원을 사용하기 위해 기다리게 한다.

##### 4) 순환 대기 (Circular wait) 부정

- 자원에 고유한 번호를 할당하고, 번호 순서대로 자원을 요구하도록 한다.

#### - 회피(Avoidance)

- : 교착 상태가 발생하면 피해나가는 방법

##### 1) 은행원 알고리즘 (Banker's Algorithm)

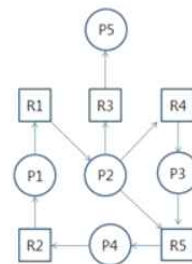
- 프로세스가 자원을 요구할 때 시스템은 자원을 할당한 후에도 안정 상태로 남아있게 되는지를 사전에 검사하여 교착 상태를 회피하는 기법
- 안정 상태에 있으면 자원을 할당하고, 그렇지 않으면 다른 프로세스들이 자원을 해지할 때까지 대기함
- 교착 상태가 되지 않도록 보장하기 위하여 교착 상태를 예방하거나 회피하는 프로토콜을 이용하는 방법

##### (2) 교착 상태 탐지 및 회복

- : 교착 상태가 되도록 허용한 다음에 회복시키는 방법

##### 1) 교착 상태 탐지(Detection)

- 자원 할당 그래프를 통해 교착 상태를 탐지할 수 있다.
- 자원 할당 그래프의 예시



##### 2) 교착 상태 회복(Recovery)

- 교착 상태를 일으킨 프로세스를 종료하거나, 할당된 자원을 해제함으로써 회복하는 것을 의미
- + 프로세스를 종료하는 방법

##### 1) 교착 상태의 프로세스를 모두 중지

##### 2) 교착 상태가 제거될 때까지 한 프로세스씩 중지

#### + 자원을 선점하는 방법

##### 1) 교착 상태의 프로세스가 점유하고 있는 자원을 선점하여 다른 프로세스에게 할당하며, 해당 프로세스를 일시 정지 시키는 방법

##### 2) 우선 순위가 낮은 프로세스, 수행된 횟수가 적은 프로세스 등을 위주로 프로세스의 자원을 선점한다.

# 메모리 풀 구조

## 메모리 풀

동일한 사이즈의 메모리 블록들을 미리 할당해 놓은 공간

### - 사용 이유

- + 메모리 풀은 단편화 없이 일정한 실행 시간으로 메모리 할당을 할 수 있도록 한다.
- + 풀 내에서 할당된 수많은 오브젝트들에 대한 메모리의 해제는 단 한번의 조작으로 가능하다.
- + 메모리 풀들은 계층구조적으로 그룹화될 수 있으며, 이러한 그룹화는 loop나 재귀함수같은 특별한 프로그래밍 구조에 적합할 수 있다.
- + 고정된 크기의 블록 메모리 풀들은 각각의 할당된 메모리 블록에 대한 메타정보 및 할당된 블록의 사이즈 같은 설명을 보관할 필요가 없다. 특히 작은 크기의 할당에서는 상당한 공간을 절약할 수 있다.
- + 메모리의 할당, 해제가 잦은 경우에 메모리 풀을 쓰면 효과적

### - 단점

- + 메모리 풀들은 메모리 풀을 사용하는 응용 프로그램에 따라 조정되어야 한다.
- + 미리 할당해놓고 사용하지 않는 순간에도 계속 할당해놓으므로 메모리 누수가 있는 방식

## 메모리 단편화

### 메모리 단편화

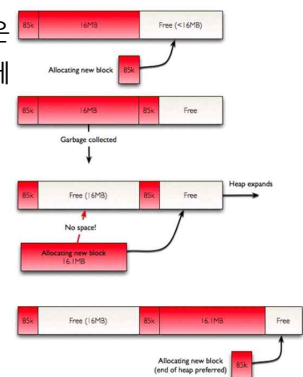
RAM에서 메모리의 공간이 작은 조각으로 나뉘어져 사용 가능한 메모리가 충분히 존재하지만 할당(사용)이 불가능한 상태

#### (1) 내부 단편화(internal fragmentation)

- 메모리를 할당할 때 프로세스가 필요한 양보다 더 큰 메모리가 할당되어서 프로세스에서 사용하는 메모리 공간이 낭비 되는 상황

#### (2) 외부 단편화(external fragmentation)

- 메모리가 할당되고 해제되는 작업이 반복될 때 할당되지 않은 작은 메모리 영역이 생기는데, 할당/사용하지 않는 메모리가 많이 생겨 총 메모리 공간은 충분하지만 실제로 할당할 수 없는 상황



++

## 페이지와 세그먼트

\* 페이지(page/paging) 기법 - 가상 메모리 사용, 외부 단편화 해결, 내부 단편화 존재

- 보조기억장치를 이용한 가상메모리를 같은 크기의 블록으로 나눈 것을 페이지라고 하고 RAM을 페이지와 같은 크기로 나눈 것을 프레임이라고 할 때,

페이징 기법이란 사용하지 않는 프레임을 페이지에 옮기고, 필요한 메모리를 페이지 단위로 프레임에 옮기는 기법.

- 페이지와 프레임을 대응시키기 위해 페이지 매핑(page mapping)과정이 필요해서 페이징 테이블(paging table)을 만든다.

- 페이징 기법을 사용하면 연속적이지 않은 공간도 활용할 수 있기 때문에 외부 단편화 문제를 해결할 수 있다.

- 단점

+ 페이지 단위에 알맞게 꼭 채워 쓰는 것이 아니므로 내부 단편화 문제는 여전히 있다.

(페이지 단위를 작게하면 내부 단편화 문제도 해결할 수 있겠지만 대신 페이지 매핑(page mapping) 과정이 많아지므로 오히려 효율이 떨어질 수 있다.)

\* 세그먼트(segmentation) 기법 - 가상메모리사용, 내부 단편화 해결, 외부 단편화 존재

- 가상메모리를 서로 크기가 다른 논리적 단위인 세그먼트로 분할해서 메모리를 할당하여 실제 메모리 주소로 변환하여 관리하는 기법

- 각 세그먼트는 연속적인 공간에 저장

- 세그먼트들의 크기가 다르기 때문에 미리 분할해 둘 수 없고 메모리에 적재될 때 빈 공간을 찾아 할당하는 기법

- 매핑을 위해 세그먼트 테이블이 필요

(세그먼트 테이블은 각 세그먼트 항목별 세그먼트 시작주소와 세그먼트의 길이 정보를 가지고 있음)

- 프로세스가 필요한 메모리 만큼 할당해주기 때문에 내부단편화는 일어나지 않으나 여전히 중간에 프로세스가 메모리를 해제하면 생기는 구멍, 즉 외부 단편화 문제는 여전히 존재한다.

# 운영체제의 스케줄링 방식

## 스케줄링

프로세스가 생성되어 실행될 때 시스템의 여러 자원을 해당 프로세스에게 할당하는 작업

### \* 스케줄링의 종류

- 장기 스케줄링
  - + 어떤 프로세스가 시스템의 자원을 차지할 수 있도록 할 것인가를 결정하여 준비상태 큐로 보내는 작업을 의미한다. 작업 스케줄링, 상위 스케줄링이라고도 하며, 작업 스케줄러에 의해 수행한다.
- 중기 스케줄링
  - + 어떤 프로세스들이 CPU를 할당 받을 것인지 결정하는 작업을 의미한다. CPU를 할당받으려는 프로세스가 많을 경우 프로세스를 일시 보류시킨 후 활성화해서 일시적으로 부하를 조절한다.
- 단기 스케줄링
  - + 프로세스가 실행되기 위해 CPU를 할당받는 시기와 특정 프로세스를 지정하는 작업을 의미한다. 프로세서 스케줄링, 하위 스케줄링이라고도 한다. 프로세서 스케줄링 및 문맥 교환은 프로세서 스케줄러에 의해 수행된다.

## 스케줄링의 기법

### 비선점 스케줄링

- + 이미 할당된 CPU를 다른 프로세스가 강제로 빼앗아 사용할 수 없는 스케줄링 기법
- + 프로세스가 CPU를 할당받으면 해당 프로세스가 완료될때까지 CPU를 사용한다.
- + 프로세스 응답 시간의 예측이 용이하며, 일괄 처리 방식에 적합하다.
- + 중요한 작업(짧은 작업)이 중요하지 않은 작업(긴 작업)을 기다리는 경우가 발생할 수 있다.
- + 비선점 스케줄링의 종류에는 FCFS, SJF, 우선순위, HRN, 기한부 등의 알고리즘이 있다.

### 선점 스케줄링

- + 하나의 프로세스가 CPU를 할당받아 실행하고 있을 때 우선순위가 높은 다른 프로세스가 CPU를 강제로 빼앗아 사용할 수 있는 스케줄링 기법
- + 우선순위가 높은 프로세스를 빠르게 처리할 수 있다.
- + 주로 빠른 응답시간을 요구하는 대화식 시분할 시스템에 사용된다.
- + 많은 오버헤드를 초래한다.
- + 선점이 가능하도록 일정 시간 배당에 대한 인터럽트용 타이머 클럭이 필요하다.
- + 선점 스케줄링의 종류에는 라운드로빈, SRT, 선점 우선순위, 다단계 큐, 다단계 피드백 큐 등의 알고리즘이 있다.

## 스케줄링의 목적

CPU나 자원을 효율적으로 사용하기 위한 기법으로 아래와 같은 목적을 가지고 있다

- (1) **공정성** : 모든 프로세스에 공정하게 할당한다.
- (2) **처리율(량)증가** : 단위 시간당 프로세스를 처리하는 비율(양)을 증가시킨다.
- (3) **CPU 이용률 증가** : 프로세스 실행 과정에서 주 기억장치를 액세스한다든지, 입출력 명령실행 등의 원인에 의해 발생할 수 있는 CPU의 낭비 시간을 줄이고, CPU가 순수하게 프로세스를 실행하는데 사용되는 시간 비율을 증가시킨다.
- (4) **우선순위 제도** : 우선순위가 높은 프로세스를 먼저 실행한다.
- (5) **오버헤드 최소화** : 오버헤드를 최소화한다.
- (6) **응답시간 최소화** : 작업을 지시하고 반응하기 시작하는 시간을 최소화한다.
- (7) **반환시간** : 프로세스를 제출한 시간부터 실행이 완료되는 시간을 최소화한다.
- (8) **반환시간 최소화** : 프로세스를 제출한 시간부터 실행이 완료될때까지 걸리는 시간을 최소화한다.
- (9) **대기시간 최소화** : 프로세스가 준비상태 큐에서 대기하는 시간을 최소화한다.
- (10) **균형있는 자원의 사용** : 메모리, 입출력장치 등의 자원을 균형있게 사용한다.
- (11) **무한 연기 회피** : 자원을 사용하기 위해 무한정 연기되는 상태를 회피한다.

## 멀티 쓰레드가 사용되어야 하는 이유

### 멀티 쓰레드

하나의 프로세스를 다수의 실행 단위로 구분하여 자원을 공유하고, 자원의 생성과 관리의 중복성을 최소화하여 수행 능력을 향상시키는 것

(하나의 프로그램이 동시에 여러 개의 작업을 수행할 수 있게 해주는 것)

### 멀티 쓰레드의 장점

- 프로세스를 이용하여 동시에 처리하던 일을 쓰레드로 구현할 경우, 메모리 공간과 시스템 자원 소모가 줄어들게 된다.
- 쓰레드 간의 통신이 필요한 경우에도 별도의 자원을 이용하는 것이 아니라 **전역 변수의 공간** 또는 동적으로 할당된 공간인 **힙(Heap) 영역**을 이용하여 데이터를 주고받을 수 있다. 그렇기 때문에 프로세스 간 통신 방법에 비해 쓰레드 간의 통신 방법이 훨씬 간단하다.
- 쓰레드의 문맥 교환은 프로세스 문맥 교환과는 달리 **캐시 메모리를 비울 필요가 없기 때문에 더 빠르다.**
- 따라서 시스템의 처리량이 향상되고, 자원 소모가 줄어들어 자연스럽게 프로그램의 응답 시간이 단축된다.
- 이러한 장점 때문에 여러 프로세스로 할 수 있는 작업들을 하나의 프로세스에서 여러 쓰레드로 나눠 수행하는 것이다.

## 멀티 쓰레드가 사용되어야 하는 경우

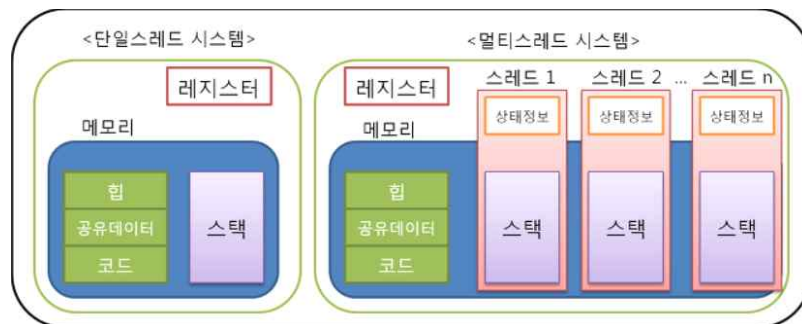
멀티 코어 CPU의 경우를 대비하여 CPU의 잉여 시간을 최대한 없애려고 할 때

## 멀티 프로세스와 멀티 쓰레드의 차이점

- 멀티 프로세스 : [데이터 영역, 힙, 스택]영역 모두를 비공유
- 멀티 쓰레드 : [ 데이터 영역, 힙, 스택 ]영역 중 스택 영역만 비공유

## 멀티 쓰레딩의 문제점

- 멀티 프로세스 기반으로 프로그래밍할 때는 프로세스 간 공유하는 자원이 없기 때문에 동일한 자원에 동시에 접근하는 일이 없었지만, 멀티 쓰레딩을 기반으로 프로그래밍할 때는 이 부분을 신경써줘야 한다.
- 서로 다른 쓰레드가 데이터와 힙 영역을 공유하기 때문에 어떤 쓰레드가 다른 쓰레드에서 사용 중인 변수나 자료 구조에 접근하여 엉뚱한 값을 읽어오거나 수정할 수 있다.
- 그렇기 때문에 멀티쓰레딩 환경에서는 동기화 작업이 필요하다.
- 동기화를 통해 작업 처리 순서를 컨트롤 하고 공유 자원에 대한 접근을 컨트롤 하는 것이다.
- 하지만 이로 인해 병목 현상이 발생하여 성능이 저하될 가능성이 높다.
- 그러므로 과도한 락(lock)으로 인한 병목 현상을 줄여야 한다.
- 공유 자원이 아닌 부분은 동기화 처리를 할 필요가 없다.
- 즉, 동기화 처리가 필요한 부분에만 synchronized 키워드를 통해 동기화하는 것이다.
- 불필요한 부분까지 동기화를 할 경우, 현재 쓰레드는 락(lock)을 획득한 쓰레드가 종료하기 전까지 대기해야한다. 그렇게 되면 전체 성능에 영향을 미치게 된다.
- 즉 동기화를 하고자 할 때는 메소드 전체를 동기화 할 것인가 아니면 특정 부분만 동기화할 것인지 고민해야 한다.



## 컨텍스트 스위칭

- Ready 상태인 A 프로세스와 Running 상태인 B 프로세스가 있고 인터럽트 요청에 의해 서로 상태가 전이된다고 가정합니다. 이 때, A 프로세스가 Ready 상태로 전이되고, B 프로세스가 Running 상태가 될 때 A 프로세스의 상태 또는 레지스터 값 등이 PCB(Process Control Block)에 저장되고, A 프로세스의 정보를 PCB에서 CPU로 적재시킵니다.

- PCB(프로세스 제어 블록: Process Control Block): PCB에는 프로세스 상태 또는 레지스터의 값 등과 같은 관련 데이터들이 저장됩니다. 프로세스 생성 시 만들어지며, 프로세스마다 고유한 PCB를 가집니다.



## 동기 / 비동기 쓰레드

---

### 멀티 쓰레드 동기화 방식

한 쓰레드가 작업 중 임계영역에 진입할 시 임계영역의 작업이 완료될 때 까지 임계영역에 진입하려는 다른 쓰레드는 작업이 중지(block)

### 동기화 쓰레드가 사용되는 부분

쓰레드간 공유되는 자원이 없을 때 다른 쓰레드에 의해 자원이 원한 결과로 도출되지 않고 훼손될 경우

### 멀티쓰레드 비동기화 방식

쓰레드들이 다른 쓰레드에 의해 중지되지 않는다

### 비동기화 쓰레드가 사용되는 부분

쓰레드간 공유되는 자원이 없을 경우

여러 쓰레드가 공유 자원에 접근하여도 훼손되지 않을 경우

## 플래시에 비추는 몬스터 판별

**Q** 플레이어가 전면에 플래시를 비추고 있다. 플래시의 조명이 부채꼴 모양이라고 할 때 플래시에 비춰지는 몬스터들을 어떻게 구할 수 있을까?

- 1) 플래시와 몬스터의 정점으로 이루어진 직선과 조명의 원뿔의 중심선의 내각이 원뿔의 내각보다 작고
- 2) 조명이 도달하는 최대 거리보다 플래시와 몬스터의 정점의 거리가 작을 때

## 2D 평면에 10000개 캐릭터

**Q** 2D 평면에서 한 화면에 10000개의 캐릭터가 있다고 하고 플레이어와 일정 거리 안에 있는 캐릭터만 검출하고 싶다. 어떤 방식으로 구현하겠는가?

**방법 1.**

프레임 업데이트 마다 각 캐릭터와 플레이어의 캐릭터의 거리를 계산하고 일정 거리 안에 들어와 있는 캐릭터를 검출한다.  $O(N)$

**방법 2.**

첫 프레임에 각 캐릭터와 플레이어의 캐릭터의 거리를 계산하고 그에 대한 거리를 캐릭터의 ID와 함께 가변 배열 안에 저장하고 쿼리 정렬을 통해 거리를 기준으로 정렬한다.  $O(N \log N)$

다음 프레임 업데이트 마다 움직인 캐릭터는 가변 배열에서 삭제한 뒤,  $O(N)$

다시 거리를 계산하고 이진 탐색을 통해 삽입될 위치를 찾는다.  $O(\log N)$

## 다각형 내 몬스터 판별

**Q** 특정 마법 사용시 다각형의 울타리에 몬스터들을 가둘 수 있다고 했을 때 울타리 내의 몬스터들을 어떻게 판별하겠는가?

몬스터를 기준으로 오른쪽으로 반직선을 구했을 때, 다각형과 교차점이 홀수면 밖, 짝수면 내부이다.

# 디자인 패턴

## 1) 지형에 여러 종류의 풀을 그릴 때

- 경량 패턴/Flyweight Pattern(게임 프로그래밍 패턴, GoF .256쪽)
- + 정의 : 공유를 통해 많은 소립 객체들을 효과적으로 지원합니다.
- + 예를 들어 한 종류의 풀은 공유하는 폴리곤(쿼드), 머티리얼(2D 텍스처)들을 공유하고 그 위치만 다르다. 즉 위치 배열들을 가지고 한 종류의 풀을 여러 군데 배치할 수 있다.
- + 여러 종류의 풀로 구현하려면 폴리곤을 공유하고 위치와 머티리얼을 다르게 해야한다.
- + DirectX, OpenGL은 경량 패턴의 한 종류인 인스턴스드 렌더링을 지원한다.

## 2) 캐릭터에 아이템을 조합하여 캐릭터 객체를 생성할 때

- 컴포넌트 패턴/Component Pattern
- + 의도 : 한 개체가 여러 분야를 서로 커플링 없이 다룰 수 있게 한다.

참고 : <https://boycoding.tistory.com/118>

# Redo, Undo

## 1) 먼저 사용자가 실행할 명령들을 커맨드 패턴으로 구현한다.

- 구현된 커맨드들은 `execute()`, `undo()`, `redo()`의 명령들을 소유하고 있다.

## 2) Undo(실행 취소)와 Redo(Undo 취소)는 명령들의 집합이며 그 명령들은 LIFO구조로 쌓이게 된다. 즉 두 개의 스택(undo, redo)으로 구현한다.

## 3) 사용자가 명령을 수행하면 undo 스택으로 명령이 쌓이고 실행 취소(undo)를 수행하게 되면 undo 스택의 top에 위치한 명령이 undo()를 수행하고 redo 스택으로 push된다.

## 4) Redo가 실행되면 redo 스택의 top에 위치한 명령의 redo()를 실행한 뒤 다시 undo 스택으로 push한다.

- 만약 undo, redo 스택이 지정한 크기를 넘어설 경우, 맨 처음 삽입된 명령을 해제한다.
- 여기서 스택으로 구현하면 모든 명령들을 pop하고 마지막에 나온 명령을 해제해야 한다.
- 즉, 이 방법을 쓰려면 리스트로 구현하거나 bottom에 위치한 원소를 탐색할 수 있는 방법을 마련하는 것이 좋을 것 같다.

## Q 타일맵 기반의 게임에서 길찾기를 수행하려고 A\*를 사용하려고 한다. A\*의 단점은?

- 최단 경로 알고리즘
- + 종류 : 다익스트라 알고리즘, 플로이드 알고리즘, A\* 알고리즘

추가) 다익스트라 알고리즘 단점

- + 목표점이 없다. : 시작점으로부터 모든 정점에 이르는 최단 거리를 구한다. 만약 하나의 목적점의 최단 거리를 구한다면 모든 정점으로부터 목적점의 거리가 구해질 때 중단하면 된다.
- + 남은 거리를 고려하지 않는다. : 모든 정점에 이르는 최단거리를 고려하여 목적점이 존재하지 않기 때문에 잔여거리를 구할 수 없다.
- + 최적 경로 보장할 수 없다.

### - A\*(A star) 알고리즘 단점

1. 경로에 따라 움직이면 현재 노드에서 다음 노드 까지 움직이고 정지 후, 다음 노드로 움직인다. 즉, 정지 후 움직이기 때문에 움직임이 부자연스럽다.
2. 노드의 크기보다 캐릭터의 크기가 일반적으로 크기 때문에 인접 노드와 현재 노드의 위치 판별이 어려울 경우가 있다. 이 경우 캐릭터는 끼이는 등의 부자연스러운 움직임이 보여진다.
3. 경로 추적 시간이 휴리스틱을 추정하는 알고리즘에 의존적이다.
4. 시작점에서 목적점까지의 경로가 존재하지 않을 경우, 모든 경로를 조사하므로 매우 비효율적이다.
5. 맵의 크기가 크면 '열린 목록'이나 '닫힌 목록'에 수백 개에서 수천 개의 노드가 들어갈 수 있기 때문에 메모리 부족 문제와 상당한 계산 시간 소요가 발생 할 수 있다.

### - A\* 알고리즘

- + 경로 채점 함수 :  $F(x) = G(x) + H(x)$

G - 시작점 A로부터 현재 사각형까지의 경로를 따라 이동하는데 소요되는 비용.

H - 현재 사각형에서 목적지 B까지의 예상 이동 비용. 사이에 벽, 물 등으로 인해 실제 거리는 알지 못한다. 그들을 무시하고 예상 거리를 산출한다. 여러 방법이 있지만, 이 포스팅에서는 대각선 이동을 생각하지 않고, 가로 또는 세로로 이동하는 비용만 계산한다.

F - 현재까지 이동하는데 걸린 비용과 예상 비용을 합친 총 비용

+ 과정

1. 시작사각형에서 검색된 인접사각형들을 열린목록에 넣습니다.

2. 다음의 과정들을 반복합니다.

ㄱ. 열린목록에서 가장 낮은 F 비용을 찾아 현재사각형으로 선택합니다.

ㄴ. 이것을 열린목록에서 꺼내 닫힌목록으로 넣습니다.

ㄷ. 선택한 사각형에 인접한 8 개의 사각형에 대해 탐색합니다.

- 만약 인접한 사각형이 갈수 없는 벽 이거나 그것이 '닫힌목록' 상에 있다면 무시, 그렇지 않은것은 다음을 계속합니다.

- 만약 이것이 '열린 목록'에 있지 않다면, 이것을 열린목록에 추가하고. 이 사각형의 부모를 현재 사각형으로 만듭니다. 사각형의 F,G,H 비용을 기록.

- 만약 이것이 이미 '열린 목록'에 있다면, G비용을 이용하여 이 사각형이 더 나은가 알아보고, 그것의 G비용이 더 작으면 그것이 더 나은 길이라는 것을 의미하므로 부모 사각형을 그 (G비용이 더 작은)사각형으로 바꿉니다, 그리고 그 사각형의 G,F비용을 다시 계산합니다.

ㄹ. 그만해야 할 때

- 길을 찾는 중 목표사각형을 '열린 목록'에 추가했을 때,

- '열린 목록'이 비어있게 될 경우.

(이때는 목표사각형을 찾는데 실패한것인데 이 경우 길이 없는 경우다.)

3. 길 저장하기.

목표 사각형으로부터 각각의 사각형의 부모사각형을 향하여 시작사각형에 도착할때까지 거슬러 올라 갑니다.

참고:

<https://itmining.tistory.com/66>

[https://cilab.sejong.ac.kr/home/lib/exe/fetch.php?media=public:paper:2013:kcc\\_2013f\\_kht.pdf](https://cilab.sejong.ac.kr/home/lib/exe/fetch.php?media=public:paper:2013:kcc_2013f_kht.pdf)

<https://withmule.tistory.com/13>