개인 학습

# 컴퓨터 과학(CS)

## 운영체제(OS)

컴퓨터 시스템의 자원들을 효율적으로 관리하고 사용자가 컴퓨터를 편리하고 효과적으로 사용할 수 있도록 환경을 제공하는 여러 프로그램의 모임  
- 컴퓨터 사용자와 컴퓨터 하드웨어 간 인터페이스로 동작하는 시스템 소프트웨어의 일종  
- 다른 응용프로그램이 유용한 작업을 할 수 있도록 환경을 제공함  
- 프로세스, 기억장치, 주변장치, 파일 등을 관리하는 기능 수행

윈도우, 리눅스, 유닉스, 맥, ios, 안드로이드 등이 있다.

기억 장치 관리  
- 보조 기억장치의 프로그램이나 데이터를 주 기억장치에 적재하는 시기, 적재 위치 등을 지정하여 한정된 주 기억장치의 공간을 효율적으로 사용하기 위한 것

|  |  |
| --- | --- |
| 반입 전략 | 프로그램이나 데이터를 언제 주 기억장치로 적재할 것인지 결정 |
| 배치 전략 | 프로그램이나 데이터를 주 기억장치의 어디에 위치시킬 것인지 결정 |
| 교체 전략 | 주 기억장치의 모든 영역이 이미 사용중인 상태에서 새 프로그램, 데이터를 주 기억장치에 배치하려고 할 때 사용중인 영역 중 어디를 교체하여 사용할 것인지 결정 |

가상 기억 장치  
- 보조 기억장치의 일부를 주 기억장치처럼 사용하는 것  
- 용량이 작은 주 기억장치를 큰 용량이 있는 것처럼 사용하는 기법  
- 주 기억장치의 이용률과 다중 프로그램의 효율을 높일 수 있음

### 페이징 기법

가장 기억장치에 보관되어 있는 프로그램과 주 기억장치의 영역을 동일 크기로 나누고 나눠진 프로그램을 동일하게 나눠진 주 기억장치의 영역에 적재하여 실행하는 방법  
- 프로세스를 일정한 작은 크기로 분할하고 주기억장치에도 동일한 크기로 영역을 나눈다  
- 하나의 프로세스는 연속적인 동작을 수행하는데 이를 분할하고 흩어 놓으면 프로세스가 정상적인 동작을 할 수 없다  
- 메모리 상에서 흩어진 프로세스를 수행하기 위해 CPU를 속이는 것

|  |  |
| --- | --- |
| 페이지 | 프로그램을 일정한 크기로 나눈 단위 |
| 페이지 프레임 | 페이지 크기로 일정하게 나눠진 주기억장치의 단위 |

- 주소 변환을 위해서 페이지의 위치 정보를 가지고 있는 페이지 맵 테이블이 필요함  
- 페이지 맵 테이블의 사용으로 비용 증가, 처리속도가 감소함  
- 내부 단편화가 발생할 수 있음

### 세그먼테이션 기법

가상 기억장치에 보관되어 있는 프로그램을 다양한 크기의 논리적인 단위로 나눈 후 주 기억장치에 적재시켜 실행하는 기법  
- 기억 공간을 페이징보다 더 절약시킬 수 있다.

|  |  |
| --- | --- |
| 세그먼트 | 프로그램을 배열이나 함수 등과 같은 논리적인 크기로 나눈 단위  - 각 세그먼트는 고유한 이름과 크기를 갖는다. |

프로세스를 물리적인 단위인 페이지 말고 논리적인 단위인 세그먼트로 자르는 방법  
- 프로세스를 세그먼트의 집합으로 생각  
- 하나의 프로세스가 동작하려면 코드, 데이터, 스택, 3가지의 세그먼트는 항상 존재해야 한다  
- 코드에서도 main, 다른 함수, 다른 루틴이 존재 가능  
- 데이터에서도 어떤 구조체, 배열 등이 더 존재할 수 있다  
-> 코드, 데이터, 스택 등을 더 작은 세그먼트로 분할 가능

물리적인 크기의 단위가 아닌 논리적인 내용의 단위로 자른다  
- 세그먼트의 크기는 일반적으로 같지 않다  
- 내부 단편화는 발생하지 않지만 외부 단편화는 발생 가능

페이지 교체 알고리즘

- 페이지 부재가 발생하면 가상 기억장치에서 필요한 페이지를 찾아 주 기억장치에 적재한다  
- 주 기억장치의 모든 페이지 프레임이 사용 중이면 어떤 페이지 프레임을 선택하여 교체할지는 결정하는 기법

|  |  |
| --- | --- |
| OPT(최적교체) | 앞으로 가장 오래 쓰지 않을 페이지를 교체 - 페이지 부재 횟수가 가장 적게 발생한다 |
| FIFO(선입선출) | 가장 먼저 들어온 가장 오래된 페이지를 교체함 - 가장 쉽고 설계가 간단함 |
| LRU(최근에 사용되지 않은 것 교체) | 최근에 가장 오랫동안 사용되지 않은 페이지를 교체  - 각 페이지마다 계수기, 스택을 둠 |
| LFU(사용 빈도) | 사용 빈도가 적은 페이지 교체 |
| NUR(LRU와 유사) | 비트를 사용함 |
| SCR(2차 기회 교체) | FIFO의 단점을 보완함 |

페이지의 크기

|  |  |
| --- | --- |
| 페이지가 작을 경우 | - 페이지 단편화 감소  - 한 페이지를 주 기억장치로 이동시키는 시간 감소  - 불필요한 내용이 주 기억장치에 적재될 확률이 적음  - 페이지 정보를 갖는 페이지 맵 테이블의 크기가 커지고 매핑 속도가 느려짐  - 디스크 접근 횟수가 많아져서 전체적인 입, 출력 시간은 증가한다 |
| 페이지가 클 경우 | - 페이지 정보를 갖는 페이지 맵 테이블의 크기가 작아지고 매핑 속도가 빨라진다  - 디스크 접근 횟수가 줄어들어 전체적인 입, 출력 효율성이 증가  - 페이지 단편화가 증가되고, 한 개의 페이지를 주 기억장치로 이동시키는 시간이 늘어남 |

지역성(Locality)  
- 프로세스가 실행되는 동안 주 기억장치를 참조할 때 일부 페이지만을 집중적으로 참조하는 성질이 있다는 이론

워킹 세트: 프로세스가 일정시간 동안 자주 참조하는 페이지들의 집합

스레싱: 프로세스의 처리 시간보다 페이지 교체에 소요되는 시간이 더 많아지는 현상  
- 다중 프로그래밍의 정도가 더 커진다면 스레싱이 발생, CPU이용률이 급감함

## 메모리

- 프로그램이 실행되기 위해서는 먼저 프로그램이 메모리에 로드되어야 한다  
- 프로그램에서 사용되는 변수들을 저장할 메모리도 필요함  
- 따라서 컴퓨터 OS는 프로그램의 실행을 위해 다양한 메모리 공간을 제공한다

메모리 공간 구조

- 코드 영역: 실행할 프로그램의 코드가 저장  
- 데이터 영역: 전역, 정적 변수들이 저장  
- 힙 영역: 사용자의 동적할당, 런타임에 크기가 결정됨  
- 스택 영역: 지역, 매개변수들, 컴파일 타임에 크기가 결정됨

코드 공간이 낮은 주소, 스택 영역이 높은 주소

코드 영역  
- 메모리의 코드 영역은 실행할 프로그램의 코드가 저장되는 영역  
- CPU는 코드 영역에 저장된 명령어를 하나씩 가져가서 처리한다

데이터 영역  
- 메모리의 데이터 영역은 프로그램의 전역 변수와 정적 변수가 저장되는 영역  
- 프로그램의 시작과 함께 할당되어 프로그램이 종료되면 소멸한다

힙 영역  
- 사용자가 직접 관리할 수 있고 관리해야 하는 메모리 영역  
- 사용자에 의해 메모리 공간이 동적으로 할당, 해제된다  
- 힙 영역은 낮은 주소에서 높은 주소 방향(힙 시작 영역에서 스택 영역을 향하는 방향)으로 할당

스택 영역  
- 함수의 호출과 관계되는 지역 변수, 매개 변수가 저장되는 메모리 영역  
- 함수의 호출과 함께 할당, 함수의 호출이 완료되면 소멸한다  
- 스택의 영역에 저장되는 함수의 호출 정보를 스택 프레임이라고 함  
- 선입 후출 방식으로 동작하여 가장 늦게 저장된 데이터가 가장 먼저 인출됨  
- 스택 영역은 메모리 주소가 높은 주소에서 낮은 주소 방향(스택 시작 방향에서 힙 영역을 향하는 방향)으로 할당됨

스택 프레임: 스택 영역에 차례대로 저장되는 함수의 호출 정보

메모리의 스택 영역은 함수의 호출과 관계되는 지역 변수와 매개 변수가 저장되는 영역  
- 스택 영역은 함수의 호출과 함께 할당되고 함수의 종료와 함께 해제됨  
- 함수가 호출되면 스택에는 함수의 매개변수, 호출이 끝난 후 돌아갈 반환 주소 값, 함수에서 선언된 지역 변수 등이 저장됨

스택 프레임으로 인해 함수의 호출이 끝난 후 함수 호출 이전 상태로 돌아감

스택 오버플로  
- 스택 내의 모든 공간이 사용되고 추가적으로 스택 프레임이 저장될 경우 해당 데이터가 스택 영역을 넘어가 저장되는 경우가 스택 오버플로  
- 스택 영역을 넘어가서도 데이터가 저장될 경우 해당 프로그램은 오동작, 보안 취약점을 가지게 된다.  
- 재귀 호출에서 재귀가 무한히 반복되는 경우 스택 오버플로가 발생함

꼬리 재귀  
- 재귀 함수는 호출이 많아질수록 메모리 오버헤드, 스택 오버플로 문제가 발생할 위험이 있음  
- 이러한 문제를 해결하기 위한 방법  
- 재귀 함수의 실행 결과가 연산에 사용되지 않고 바로 반환되게 하여 이전 함수의 상태를 유지할 필요가 없도록 하는 것  
- 꼬리 재귀를 위해서는 TCO가 지원되야 한다(C++은 지원된다)

## 프로그램, 프로세스, 스레드

프로그램: 어떠한 작업을 위해 실행할 수 있는 파일

프로세스: 컴퓨터에서 연속적으로 실행되고 있는 컴퓨터 프로그램  
- 메모리에 적재되어 실행되고 있는 프로그램의 인스턴스  
- OS로부터 시스템에 자원을 할당받는 작업의 단위  
- 동적인 개념으로는 실행된 프로그램

할당받는 시스템 자원  
- CPU 시간  
- 운영되기 위해 필요한 주소 공간  
- 코드, 데이터, 스택 힙의 구조로 된 메모리 영역

프로세스  
- 프로세스는 각각의 독립된 영역을 할당 받는다(코드, 데이터, 힙)  
- 기본적으로 프로세스 당 최소 1개의 스레드를 가지고 있음(메인 스레드)  
- 각 프로세스는 별도의 주소 공간에서 실행되며 한 프로세스는 다른 프로세스의 변수나 자료 구조에 접근할 수 없다  
- 한 프로세스가 다른 프로세스의 자원에 접근하기 위해서는 프로세스 간 통신(IPC)을 사용해야 한다(파이프, 파일, 소켓 등을 이용한 통신 방법)

스레드  
- 프로세스 내에 실행되는 여러 흐름의 단위  
- 프로세스의 특정한 수행 경로  
- 프로세스가 할당 받은 자원을 이용하는 실행의 단위

- 스레드는 프로세스 내에서 스택만 따로 할당 받는다(코드, 데이터, 힙은 공유함)  
- 스레드는 한 프로세스 내에서 동작되는 여러 실행의 흐름  
- 프로세스 내의 주소 공간이나 자원들을 같은 프로세스 내에 스레드끼리 공유하면서 실행함

같은 프로세스 내의 여러 스레드들은 같은 힙 공간을 공유함(프로세스는 다른 프로세스의 메모리에 직접 접근 불가)  
- 각각의 스레드는 별도의 레지스터와 스택을 갖지만, 힙 메모리는 서로 읽고 쓸 수 있음  
- 한 스레드가 프로세스의 자원을 변경하면 다른 이웃 스레드도 그 변경 결과를 알 수 있음

### 프로세스

- 실행중인 프로그램  
- 일반적으로 프로세서에 의해 처리되는 사용자 프로그램, 시스템 프로그램  
- 프로세서가 할당되는 실체, 디스패치가 가능한 단위  
- 프로시저(프로그램을 작게 분할한 단위, 부 프로그램)가 활동중인 것

PCB(프로세스 제어 블록)  
- 운영체제가 프로세스에 대한 중요한 정보를 저장하는 곳  
- 각 프로세스가 생성되면 고유의 PCB를 갖고 프로세스가 완료되면 PCB가 제거된다

프로세스 상태정보 전이  
- 프로세스가 시스템 내에 존재하는 동안 프로세스의 상태가 변하는 것

|  |  |
| --- | --- |
| 제출 | 작업을 처리하기 위해 사용자가 작업을 시스템에 제출한 상태 |
| 접수 | 제출된 작업이 스풀 공간인 디스크의 할당 위치에 저장된 상태 |
| 준비 | 프로세스가 프로세서를 할당받기 위해 기다리는 상태 |
| 실행 | 준비상태 큐에 있는 프로세스가 프로세서를 할당받아 실행되는 상태 |
| 대기 | 프로세스에 입,출력 처리가 필요하면 현재 실행중인 프로세스가 중단 |
| 블록 | 입, 출력 처리가 완료될 때까지 대기하는 상태 |
| 종료 | 프로세스의 실행이 끝나고 프로세스 할당이 해제된 상태 |

### 스레드

시스템의 여러 자원을 할당받아 실행하는 프로그램의 단위로 프로세스의 작업 단위로 사용된다.  
- 프로세스에는 적어도 하나의 스레드가 존재함

- 하나의 프로세스에 하나의 스레드가 존재 = 단일 스레드  
- 하나의 프로세스에 여러 스레드가 존재 = 다중 스레드

프로세스의 특성을 일부 가지고 있어 경량 프로세스라고도 함  
- 스레드 기반 프로그램에서 스레드는 독립적인 스케줄링의 최소 단위로 프로세스의 역할을 담당함

### 멀티 프로세스, 스레드

멀티 프로세스  
- 하나의 응용 프로그램을 여러 개의 프로세스로 구성, 각 프로세스가 하나의 작업을 처리하도록 하는 것

장점  
- 여러 프로세스 중 하나가 고장나도 다른 프로세스에 영향이 없음

단점  
문맥 교환의 오버헤드가 큼  
- 프로세스가 독립적인 메모리 영역을 가짐(공유 메모리가 없음)  
- 문맥 교환이 발생한다면 캐시를 초기화하고 정보를 불러와야 하며 이 비용이 큼  
IPC 문제  
- 프로세스 사이의 통신 기법인 IPC가 어렵고 복잡, 변수 공유도 불가능한 문제점이 있음

멀티 스레드  
- 하나의 응용 프로그램을 여러 개의 스레드로 구성하여 각 스레드가 하나의 작업을 처리하도록 하는 것

장점  
- 시스템 소모 자원 감소: 프로세스보다 자원 소모가 적음  
- 시스템 처리량 증가: 스레드 간 데이터 전달이 간단함 + 문맥 교환이 빠름  
- 간단한 통신 방법으로 프로그램 응답시간 감소: 스택을 제외한 코드, 데이터, 힙을 공유하여 통신 부담이 적음

단점  
- 설계가 어려움  
- 디버깅도 어려움  
- 단일 프로세스의 경우 효과가 적음  
- 다른 프로세스에서 스레드 제어 불가  
- 멀티 스레드의 경우 자원 공유 문제 발생(스레드 동기화 문제)  
- 하나의 스레드에 문제가 생기면 전체 프로세스에 문제 발생

### 스레드 동기화

실행 순서의 동기화 = 순서 엄수  
- 스레드의 실행 순서를 정의하고 반드시 이 순서에 따르도록 하는 것

메모리 접근에 대한 동기화 = 실행 순서보다는 메모리 동시 접근 문제에 중점  
- 한 순간에 하나의 스레드만 접근해야 하는 메모리 영역이 존재  
- 데이터 영역에 할당된 변수를 둘 이상의 스레드가 접근할 때 문제가 발생함  
- 메모리 접근에 있어서 동시 접근을 막는 것도 스레드 동기화

유저모드 동기화  
- 동기화 진행 과정에서 커널 코드를 사용하지 않는 방법  
- 커널모드로의 전환이 불필요하여 성능상의 이점이 존재함  
- 커널모드를 사용할 수 없어 제한되는 기능이 있음

종류: 임계 영역 기반 동기화, 인터락 함수 기반 동기화

커널모드 동기화  
- 커널에서 제공하는 동기화 기능을 활용하는 동기화 기법  
- 동기화와 관련 함수 호출 시 커널 모드의 전환이 필요하여 성능이 저하됨  
- 유저 모드에서 사용할 수 없는 기능을 사용할 수 있음

종류: 뮤텍스, 세마포어, 이름있는 뮤텍스, 이벤트 기반

\* 임계 영역  
- 둘 이상의 스레드가 접근해서 연산을 수행할 경우 문제가 발생하는 코드 블록  
- 배타적 접근이 요구되는 공유 리소스에 접근하는 코드 블록을 의미함

뮤텍스  
- 한 프로세스 안에서 여러 스레드의 임계 구역 접근을 제어하는 커널 객체

세마포어  
- 세마포어는 뮤텍스와 굉장히 유사함  
- 넓게 본다면 뮤텍스가 세마포어의 일종  
- 임계 영역에 접근할 수 있는 스레드의 개수를 조절할 수 있다

뮤텍스와 세마포어

세마포어를 뮤텍스로 만들 수는 있으나 뮤텍스로 세마포어로 만들 수는 없음  
- Count 기능의 차이로 뮤텍스가 세마포어가 될 수는 없음  
- 세마포어에서 접근 가능한 스레드 개수를 1로 만든 것을 바이너리 세마포어라고 하며 이 경우 뮤텍스와 동일한 역할을 하게 됨

세마포어는 소유가 불가능하지만 뮤텍스는 소유할 수 있다

세마포어는 시스템 범위에 걸쳐 파일 형태로 존재함

뮤텍스는 프로세스 범위에 존재하여 프로세스 종료 시 정리됨

세마포어는 동기화 개수가 1개 이상, 뮤텍스는 동기화 개수가 단 1개

이름있는 뮤텍스  
- 뮤텍스와 세마포어에 이름을 붙여줄 수 있음  
- 뮤텍스는 커널 오브젝트로 모든 프로세스들이 접근할 수 있음  
- 뮤텍스의 핸들 값은 뮤텍스를 생성한 프로세스만이 가지고 있어 다른 프로세스에게 이 핸들 값을 봐도 의미가 없음  
- 뮤텍스에 이름을 붙이면 이 이름은 OS에서 유일하게 존재하기 때문에 핸들 정보를 얻어낼 수 있음

### 문맥교환

현재 진행하는 작업(프로세스, 스레드)의 상태를 저장하고 다음 진행할 작업의 상태 값을 읽어 적용하는 과정  
- 문맥(Context): CPU가 다루는 작업의 정보

작업의 대부분의 정보는 레지스터에 저장되고 PCB로 관리된다  
- 프로세스 스택, 레디 큐에 현재 실행되고 있는 작업의 PCB 정보를 저장  
- 다음 실행할 작업의 PCB 정보를 읽어 레지스터에 적재, CPU가 이전에 진행했던 과정을 연속적으로 수행 가능

문맥 교환은 많은 자원을 소모함  
- 캐시 초기화, 메모리 매핑 초기화, 메모리 접근을 위한 커널의 상시 가동  
- 프로세스의 문맥 교환 비용이 스레드의 문맥 교환보다 자원을 많이 요구함  
- 프로세스는 각각의 메모리 영역이 존재하며 이 영역을 모두 변경해야 한다. 스레드는 스택만 독립적이기 때문에 프로세스보다 비용이 적게 들어간다.

### 프로세스 스케줄링(CPU 스케줄링)

|  |  |
| --- | --- |
| 비선점 스케줄링 | |
| FCFS | 먼저 요청한 프로세스를 순차적으로 할당해주는 기본적인 알고리즘 |
| SJF | 평균 대기 시간을 최소화하기 위해 CPU 점유시간이 가장 짧은 프로세스를 먼저 할당함  - 기아 상태: 점유 시간이 긴 프로세스가 계속 뒤로 밀림 |
| HRRN | (대기 시간 + 서비스 시간) / 서비스 시간 으로 응답률을 구함 |
| 선점 스케줄링 | |
| SRT | 짧은 시간 순서대로 수행하는 방식 |
| RR | 모든 프로세스가 같은 크기의 CPU 시간을 할당(양자시간), 그 시간 만큼만 선점하고 시간을 초과하면 다음 프로세스에 할당하는 방식 |
| MLQ | 레디 큐를 여러 개 사용하여 각각의 큐는 자신의 스케줄링 알고리즘을 수행하며 큐와 큐 사이에서도 우선순위를 부여하는 방식 |

### 교착 상태

상호 배제로 인해 일어나는 문제점으로 둘 이상의 프로세스들이 자원을 점유하며 다른 프로세스의 자원을 요구하는 상태가 무한히 지속되는 상태

|  |  |
| --- | --- |
| 발생 조건 | |
| 상호 배제 | 한 번에 한 개의 프로세스만이 공유 자원을 사용 |
| 점유 대기 | 최소 하나의 자원을 점유하며 다른 프로세스에 할당되어 사용중인 자원을 점유하기 위해 대기하는 프로세스가 있어야 함 |
| 비선점 | 다른 프로세스에 할당된 자원은 사용 종료까지 강제로 빼앗을 수 없어야 한다. |
| 환형 대기 | 공유 자원과 자원 사용을 위해 대기하는 프로세스들이 원형으로 구성되어 자신에게 할당된 자원을 점유하고 앞이나 뒤의 프로세스 자원을 요구한다. |

|  |  |
| --- | --- |
| 해결 방법 | |
| 예방 | 교착 상태 발생 조건 중 하나를 제거 (가장 자원 소모가 큼) |
| 회피 | 교착 상태가 피하면 적절히 피해나감(은행원 알고리즘) |
| 발견 | 시스템에 교착 상태가 발생했는지 점검, 교착 상태의 프로세스와 자원을 발견하는 것 |
| 회복 | 교착 상태를 일으킨 프로세스를 종료, 교착상태의 프로세스에 할당된 자원을 선점하여 프로세스의 자원을 회복하는 것 |

# 컴퓨터 그래픽스

## 렌더링 파이프라인

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 로컬  스페이스 | 월드  스페이스 | 뷰  스페이스 | 백  스페이스 컬링 | 조명 | 클리핑 | 투영 | 뷰포트 | 레스터라이즈 |

1: 로컬 스페이스  
- 다른 모델과는 상관없는, 각각의 모델을 구성하고 있는 고유 공간

2: 월드 스페이스  
- 각 로컬 스페이스의 모델들을 하나의 공간으로 구성하는 단계  
- 로컬 스페이스의 모델들을 월드 공간의 어느 위치에, 어느 크기, 회전 값을 구성할 지 그 정보들을 담고 있는 것이 월드 변환 행렬

3: 뷰 스페이스  
- 월드 공간 상의 물체들을 화면에 출력하기 위하여 가상의 카메라를 만드는 단계  
- 가상의 카메라를 만들 때 카메라의 위치, 방향, 시야각, 거리 등의 정보를 갖는 행렬로 구성된다.

4: 백스페이스 컬링  
- 폴리곤은 앞, 뒤 2면이 존재하며 뒷면은 볼 수 없음  
- 뒷면을 추려내어(컬링) 불필요한 연산을 줄일 수 있음  
- DX에서는 디폴트 값으로 시계 방향으로 구성된 폴리곤이 앞면, 반시계 방향으로 구성된 폴리곤이 뒷면이다.  
- 뷰 스페이스 기준으로 컬링이 진행됨(로컬 스페이스에서 앞면이 앞면이 아닐 수 있음)

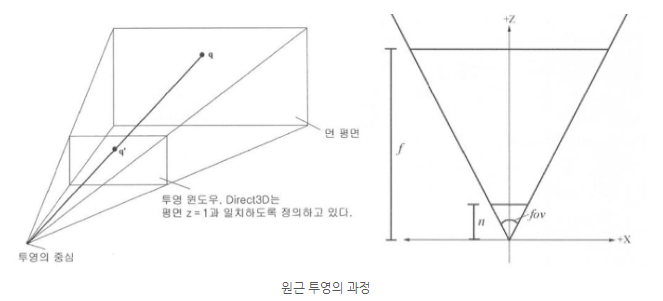
5: 조명  
- 3D 물체에 대한 사실감 표현을 위한 단계로 빛을 비추는 것

|  |  |
| --- | --- |
| Diffuse | 조명 연산을 통해 메시의 색을 나타냄 |
| Specular | 카메라와 빛의 방향에 따라 반사되는 색 |
|  |  |
| Ambient(주변색) | 광원의 위치와 무관하게 같은 모양으로 모든 점에서 반사되는 색 |
| Diffuse(확산색) | 광원이 반사될 때 나타나는 가장 주된 색 |
| Specualr(반사색) | 특정한 방향으로만 반사되는 색 (광원의 위치와 카메라의 위치로 결정됨) |
| Emissive(방출색) | 물체 표면에서 자체적으로 방출되는 색 |

7: 클리핑  
- 카메라의 시야 밖으로 벗어난 물체들에 대해서 출력을 막는 작업

|  |  |
| --- | --- |
| 내부 | 그대로 출력 |
| 외부 | 시야각을 벗어난 외부에 존재하므로 추려낸다 |
| 분리 | 시야 각 내부의 물체는 출력, 외부의 물체는 추려낸다. |

8: 투영  
- n차원에서 n-1 차원을 얻는 과정과 변환

  
- 원근 투영: 멀리 있는 물체는 작게, 가까운 물체는 크게 표현하는 원근법을 이용한 투영  
- 직교 투영: 거리에 관계없이 평행하게 투영

9: 뷰포트

윈도우의 좌표를 뷰 포트(화면)의 직사각형으로 변환하는 과정  
- 윈도우 좌표계를 이용하여 프로그램의 크기에 맞게 변화되는 과정

10: 레스터라이즈

모든 변환이 끝나면 2D로 출력되는 삼각형의 리스트가 생성된다

각 삼각형을 출력하는데 필요한 색상들의 연산을 하는 과정  
- 삼각형을 구성하는 각 정점이 갖고 있는 색상을 보간하여 삼각형의 픽셀에 색을 채운다.

## 셰이더

소프트웨어 명령의 집합  
- 그래픽 하드웨어 렌더링 효과 계산에 사용  
- GPU 프로그래밍의 가능한 렌더링 파이프라인을 프로그래밍 하는데 사용함

정점 셰이더  
- 정점의 정보값을 변화시켜서 물체를 특정 위치로 옮기거나 색을 바꾸는 작업을 수행하는 셰이더  
- 입력되는 정점은 반드시 1개, 출력도 1개 = 정점이 36개면 정점 셰이더도 36번 수행  
- 정점의 추가, 삭제는 불가능함  
- 정점의 출력 정보는 픽셀 셰이더의 입력정보로 사용됨

픽셀 셰이더  
- 정점 셰이더에서 나온 정점의 정보와 레스터라이즈 이후 나온 정보들을 이용하여 최종 픽셀 단위 계산을 수행하는 셰이더

# 언리얼 기초이론

UBT: 언리얼 빌드 프로세스를 자동화시키는 툴 모음  
-

# C++ 기초 이론

## 클래스 인스턴스 개수 제한

인스턴스를 하나로만 제한하는 싱글톤 방식, 인스턴스 수를 원하는 만큼만 생성하게 할 수 있는 pool 방식이 있다

### 기본적인 방법

객체 생성 제한 = 생성자를 private로 선언함

|  |
| --- |
| class PrintJob;  class Printer  {  public:  void submitJob(const PrintJob& job);  void reset(); void performSelfTest();  ...  friend Printer& thePrinter();  private:  Printer();  Printer(const Printer& rhs);  ...  };  Printer& thePrinter()  {  static Printer p;  return p;  } |

- 생성자를 모두 private로 선언하여 임의로 객체를 생성할 수 없게 만든다

thePrinter() 함수를 friend 함수로 선언하는 경우  
- static 객체 p가 Printer Class의 공개 정도(public, protected, private)에 관계없이 this 처럼 사용할 수 있게 해준다.  
- 문제점으로는 단순히 프로그램 상 하나만 존재하는 local static 객체만을 반환함  
- thePrinter() 함수로 객체를 얻은 경우 public 범위의 일만 할 수 있게 된다.

### 다른 방법

|  |
| --- |
| class Printer  {  public:  static Printer& thePrinter();  void submitJob(const PrintJob& job);  void reset();  void performSelfTest();  ...  private:  Printer();  Printer(const Printer& rhs);  ...  };  Printer& Printer::thePrinter()  {  static Printer p;  return p;  }  Printer::thePrinter().reset(); Printer::thePrinter().submitJob(buffer); |

- 함수 내부의 static 객체로 선언하게 되면 실제 thePrinter() 함수가 1회라도 호출되어야 static 객체가 생성된다.  
- static 객체를 class 멤버 변수로 만들 경우 초기화 시점에 대해서 명확하게 알 수 없지만 함수 static 객체의 경우 함수 호출 시 초기화된다는 것이 명확해진다.

주의 사항으로는 싱글톤 객체를 반환하는 함수를 static으로 만들어서는 안 된다  
- 함수 내부에 static 객체를 생성한 것은 객체를 프로그램 상에서 1개만 만들기 위한 것  
- 함수가 호출되는 곳마다 local static 객체를 생성하는 코드가 동작하므로 1개만 존재하게 하기 위해서 만든 것이 지역에 따라 다른 객체가 될 수 있는 문제점이 존재함

### 클래스 별로 인스턴스 수 제어

|  |
| --- |
| class Printer  {  public:  class TooMnayObjects{};  Printer();  ~Printer();  ...  private:  Printer(const Printer& rhs);  static size\_t numObjects;  ...  };  size\_t Printer::numObjects = 0;  Printer::Printer()  {  if (numObjects >= 1)  {  throw TooManyObjects();  }  ++numObjects;  }  Printer::~Printer()  {  --numObjects;  } |
| class Printer  {  ...  };  class ColocPrinter: public Printer  {  ...  };  Printer p;  ColocPrinter cp; |

ColorPrinter 객체를 생성하는 순간 exception이 발생함  
- Printer의 생성자에서 개수를 세고 예외를 던지도록 구현됨  
- 개수를 세는 것을 상위 클래스와 하위 클래스가 구분할 수 있도록 해야 함

### 인스턴스 개수만을 세는 클래스 생성

|  |
| --- |
| template<class BeingCounted>  class Counted  {  public:  class TooManyObjects{};  static size\_t objectCount()  {  return numObjects;  }  protected:  Counted();  Counted(const Counted& rhs);  ~Counted()  {  --numObjects;  }  private:  static size\_t numObjects;  static const size\_t maxObjects;  void init();  };  template<class BeingCounted>  Counted<BeingCounted>::Counted()  {  init();  }  template<class BeingCounted>  Counted<BeingCounted>::Counted(const Counted<BeingCounted>& rhs)  {  init();  }  template<class BeingCounted>  void Counted<BeingCounted>::init()  {  if (numObjects >= maxObjects)  {  throw TooManyObjects();  }  ++numObjects;  } |

- 여기서 소멸자를 virtual로 선언하면 안 된다.

|  |
| --- |
| class Printer: private Counted<Printer>  {  public:  // 유사 생성자 (pseudo-contructor)  static Printer\* makePrinter();  static Printer\* makePrinter(const Printer& rhs);  ~Printer();  void submitJob(const PrintJob& job);  void reset();  void performSelfTest();  ...  using Counted<Printer>::objectCount;  using Counted<Printer>::TooManyObjects;  private:  Printer();  Printer(const Printer& rhs);  }; |
| const size\_t Counted<Printer>::maxObjects = 10;  Printer::Printer() { }  Printer::Printer(const Printer& rhs) { // 값 복사 }  Printer::~Printer() { }  // 유사 생성자 (pseudo-contructor)  Printer\* Printer::makePrinter()  {  return new Printer();  }  Printer\* Printer::makePrinter(const Printer& rhs)  {  return new Printer(rhs);  } |

- counted 클래스를 상속받는 클래스에서 maxObject를 초기화해서 사용할 수 있게 된다.  
- 상속받아 사용하는 쪽에서 얼마든지 객체 숫자를 조절하는 것이 가능함

Counted 클래스의 일부 기능은 Printer 클래스를 사용하는 사용자에게도 공개되어야 함  
- 예를 들어 현재 객체의 수가 몇개인지를 알아내야 할 경우  
- header 파일에서 using 문을 이용해서 공개한다고 명시하면 됨

makePrinter는 유사 생성자로 단순 객체를 동적으로 만들어 반환한다  
- 실제 객체 생성자가 호출되면 counted 클래스를 상속받아 개수를 세는 로직이 동작한다  
- 동적으로 생성한 객체를 그대로 포인터로 반환하므로 자원 관리에 있어서 생각해야 한다  
- 유사 생성자를 통해 전달받은 객체는 스마트 포인터 등을 이용해 자원을 관리해야 한다.