3D게임 프로그래밍01

기말고사 정리

파이프라인

※ 프리미티브 유형

▶ 입력조립기가 버텍스 쉐이더를 거쳐서 레스터 라이저로 데이터를 넘김, 정점 버퍼의 데이터를 해석하는 방법

설정하는 함수: IASetPrimitiveTopology -> 인자: D3D12_PRIMITIVE_TOPOLOGY primitivetopology

※ primitivetopology 종류

- D3D_PRIMITIVE_TOPOLOGY_UNDEFINED
- D3D_PRIMITIVE_TOPOLOGY_POINTLIST // 점들의 리스트
- D3D_PRIMITIVE_TOPOLOGY_LINELIST // 선분들의 리스트
- D3D_PRIMITIVE_TOPOLOGY_LINESTRIP // 선분들의 스트립
- D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST // 삼각형들의 리스트(대부분의 경우 사용)
- D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP // 삼각형들의 스트립

※ 스트립으로 무언가를 출력하려고 하면 -> 정점의 위치를 아래 위 아래 위 ... (반복)

※ 인접성: 각 삼각형에 인접한 이웃 삼각형 세 개에 관한 정보 포함

▶ 우리가 사용하는 대부분의 모델에는 인접성 모델이 안 들어가 있음

▶ 그래서 프리미티브 유형에 인접성이 들어간 유형이 있음

※ 인접성 들어간 primitivetopology 종류

- D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ // 기하 쉐이더에서 사용
- D3D_PRIMITIVE_TOPOLOGY_LINETRIP_ADJ
- D3D_PRIMITIVE_TOPOLOGY_TRIAMGLELIST_ADJ
- D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ

※ 인덱스 버퍼(GL 했을때랑 같은 개념)

▶ 각각의 삼각형을 위한 정점을 정점 버퍼에 넣으면 너무 많은 메모리와 시간이 듬

-> 삼각형1과 삼각형2의 두 정점은 모든 정보는 같은데 시멘틱만 달라짐

-> 즉, 버텍스쉐이더가 같은 정점에 대해 똑같은 일을 또 하기 때문에 시간이 많이 듬(낭비!)

▶ 그래서 정점 버퍼에는 '정점들만' 저장하고, 인덱스에 삼각형을 표현하기 위한 정보 저장(정점 순서)

▶ 인덱스 버퍼 사용하는데 훨씬 좋음

※ 그리기 함수(Draw)

▶ DrawInstanced(): 정점만 사용

▶ DrawIndexedInstanced(): 인덱스도 사용

※ 정점 버퍼 뷰 생성

▶ 16개까지 생성 가능(0 ~ 15)

▶ GetGPUVirtualAddress(): 버퍼 리소스의 GPU 주소 반환

▶ IASetVertexBuffers(): 정점 버퍼 연결

※ 인덱스 버퍼 뷰 생성

▶ 파이프라인의 하나의 인덱스 버퍼를 연결 가능

▶ 변수 Format: 폴리곤 개수에 따라 선택

▶ IASetIndexBuffers(): 인덱스 버퍼 연결

※ 메쉬 클래스
▶ ReleaseUploadBuffers(): 카피가 모두 일어난 업로드 버퍼를 릴리즈

※ 루트 시그니쳐
▶ 어떤 리소스들이 그래픽스 파이프라인의 쉐이더에 연결되는 가를 정의
▶ 명령 리스트들을 쉐이더 리소스에 연결
▶ 그래픽스 파이프라인의 제일 첫 데이터

※ 상수 버퍼
▶ 여러 개의 상수의 의미를 갖는 데이터들 한꺼번에 빠르게 리소스의 내용을 갱신 할 수 있는 특별한 의미의 버퍼들
▶ 모든 상수 버퍼 크기는 최소 하드웨어 할당 크기 256바이트의 배수이어야 함((nBytesOfData + 255) & ~255)
▶ 바뀔 것을 가정(기본적으로 UPLOAD 힙에 만들어야 함)
▶ GPU 입장에서 상수 버퍼는 읽을 수 있는 상태이여야 함

※ 루트 시그니쳐 파라미터
- 상수 버퍼: 매개변수가 32비트 상수들을 포함, 서술자 힙 필요 없음
- 서술자(CBV, SRV, UAV): 여러 개 넘기고 싶으면 서술자 힙을 만들어 전달
- 서술자 테이블: 서술자 힙의 영역을 포현(서술자 힙은 아님), 서술자 힙에 간접적으로 접근해야 하므로 시간 걸림
▶ D3D12_ROOT_DESCRIPTOR_RANGE: 서술자 테이블을 정의하는 구조체
▶ D3D12_ROOT_DESCRIPTOR_RANGE_TYPE: DESCRIPTOR_RANGE 타입 지정(SRV, UAV, CBV, SAMPLER)

※ SetGraphicsRootConstantBufferView(): 루트 서술자를 위한 리소스를 실제로 연결하는 함수(Set함수)

※ 루트 시그니쳐 파라미터 타입
- D3D12_ROOT_DESCRIPTOR_CBV: Constant Buffer View
- D3D12_ROOT_DESCRIPTOR_SRV: Shader Resource View
- D3D12_ROOT_DESCRIPTOR_UAV: Unordered Access View
- D3D12_ROOT_DESCRIPTOR_DESCRIPTOR_TABLE
- D3D12_ROOT_DESCRIPTOR_32BIT_CONSTANTS(상수 버퍼 한개짜리)
▶ CBV, SRV, UAV 다 쓰는게 아니라 하나만 결정해서 사용

※ 상수 버퍼 사용
① .Constants.ShaderRegister = 번호; 하고
② 스페이스는 0 박고
③. Constants.Num32BitValues = c버퍼 내의 개수;
④ SetGraphicsRoot32BitConstants()

※ 상수 버퍼 여러 개 넘기기
▶ 상수 버퍼는 갱신을 하면 무조건 전체를 갱신해야 함
▶ 상수 버퍼는 객체 당 한 개씩 하면 좋음(전체 상수 버퍼, 카메라 상수 버퍼, 플레이어 상수 버퍼....)
[0].ParameterType = D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS;
[0].Constants.Num32BitValues = 16;(float 행렬)
[0].Constants.ShaderRegister = 0; // (ex. 월드)
…
[1].ParameterType = D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS;
[1].Constants.Num32BitValues = 32;
[1].Constants.ShaderRegister = 1; // (ex. 카메라)
…

※ 상수 버퍼 뷰 생성
▶ CreateConstantBufferView(VIEW_DESC(사이즈는 256의 배수)
▶ HANDLE(상수 버퍼 뷰를 포함하는 서술자 힙의 시작))
▶ 상수 버퍼 서술자: 상수 버퍼를 파이프라인에 연결하기 위하여 서술자가 필요함

※ 서술자 힙
▶ 서술자 힙은 사용하기 전에 디바이스(파이프라인)에 연결(설정)되어야 함
▶ 같은 유형의 서술자 힙은 한 번에 하나만 설정 가능
▶ 설정하는 서술자 힙 배열의 각 원소(서술자 힙)의 유형은 서로 달라야 함
▶ SetDescriptorHeaps(서술자 힙 개수, 서술자 힙 배열/각 원소의 유형 달라야 함)

※ 루트 시그니쳐 생성
▶ 텍스쳐(쉐이더 리소스 뷰) 루트 파라미터는 반드시 서술자 테이블을 사용하여 생성
▶ 서술자 힙에서 SRV는 서로 붙어있어야 함
▶ 파라미터 1개쓰고 Range를 2개 써서도 동일하게 사용 가능
-> set한번 하면 됨

HLSL

※ HLSL

▶ 다렉에서 사용할 수 있는 쉐이더 프로그래밍 언어

▶ 미리 컴파일 될 수도 있고 실행 시에 컴파일 될 수도 있음(졸작시에는 미리 컴파일 하자)

※ 다렉 HLSL 프로그램

▶ 변수는 함수로, 함수는 문장으로 구성

▶ C C++ 공통점

- 세미콜론으로 끝남
- 문자 블록은 중괄호
- 함수와 구조체 사용
- 전처리 지원
- 매크로 정의, 조건부 컴파일, 인클루드 등

▶ 차이점

- 포인터 지원 안함
- 동적 할당 안됨
- 템플릿 지원 안함
- 재귀함수 지원 안함
- 전역변수 상수버퍼에 추가
- 유니폼 파라미터 변수 상수버퍼에 추가

※ 쉐이더 모델

▶ 쉐이더 모델에 따라 쉐이더 명령어의 성능이 추가됨

▶ 새로운 모델은 이전 버전 쉐이더 모델 포함

▶ 공통 쉐이더 코어 기반으로 설계

▶ 쉐이더 프로파일: 쉐이더를 컴파일하기 위한 쉐이더 모델

※ 쉐이더모델4

▶ 기하쉐이더

▶ 스트림 출력 객체, 템플릿 텍스쳐 객체

※ 쉐이더모델5, 5.1

▶ 구조화 버퍼, 바이트 주소 버퍼

▶ 계산 쉐이더

▶ 테셀레이션

※ 연산자

▶ 수식 연산자로 연결된 일련의 변수들과 리터럴 상수들

▶ 대부분의 연산자는 요소별 연산을 수행

▶ #define, #error, #ifdef, #include 등등 전처리 지시자 사용 가능

▶ auto, enum, friend, operator 등 예약어 사용 가능

▶ 공백문자 SPACE TAB EOL 주석 등 기본 문법 사용 가능

※ 변수
▶ 이름, 적용 범위, 메타데이터, 전역변수, 지역변수, 함수 매개변수
▶ 전역변수는 static 또는 extren 둘 중 하나
: 기호 있는 애들은 써도 되고 안써도 됨
- Storage_Class: 변수의 적용 범위와 기간에 대한 정보를 컴파일러에게 알려줌
- Type_Modifier: 변수의 데이터 형의 의미를 수정
- Type: HLSL 데이터 형(벡터와 행렬을 위한 추가 데이터 존재)
- Name: 쉐이더 변수 이름
- [Index]: 배열을 선언헐 때 배열 크기
- : Semantic: 변수의 사용 용도를 나타내는 문자열, 쉐이더 입/출력을 연결 위해 사용,
컴파일러는 무시(전역 변수, 쉐이더 매개변수 제외)
- Annotations: 전역 변수에 부착된 메타 데이터 문자열, HSLS에서는 무시
- : packoffset: 쉐이더 상수의 패킹을 지정 위해 사용
- : register: 쉐이더 변수를 특정 레지스터에 지정하기 위해 사용
- extern: 전역 변수를 쉐이더에 대한 외부 입력으로 표시, 쉐이더에서 값 변경 불가능, static 함께 사용 불가
- static: 변수가 정적임을 나타냄, 초기화 값 포함 안하면 0의 값 가짐, 정적 변수는 D3D 응용 프로그램에 안보임,
지역 변수 값 유지
- uniform: 모든 정점/픽셀의 처리를 완료할 때까지 값이 변하지 않는 변수, 전역 변수는 기본적으로 uniform 취급

※ 상수 버퍼 변수 팩킹 규칙: 팩킹 안하면 그림 안그려짐
▶ HLSL은 상수 버퍼의 변수들을 16바이트 경계를 넘지 않도록 패킹
▶ 하나의 변수가 경계에 걸치게 되면 새로운 4요소 벡터에서 시작(주소 지정)
▶ 구조체의 첫 번째 변수는 다음 4요소 벡터에서 시작(구조체 크기는 16 배수)
▶ 배열은 기본적으로 패킹되지 않고 배열의 모든 원소는 4요소 벡터로 저장
▶ 패킹된 쉐이더 상수 버퍼와 응용 프로그램의 메모리 구조가 다를 수 있음
▶ 레지스터는 float가 4개씩 들어가므로 4의 배수로 맞추는게 빈공간이 생기지 않기 때문에 좋음
-> ex) float3 ; float 4; float 1; (X) / float 4 ; float 3 ; float1 (O)

※ 레지스터 타입
- b: 상수버퍼
- c: 버퍼 오프셋
- t: 텍스쳐 또는 텍스쳐 버퍼
- s: 샘플러
- u: 무순서화 접근 뷰

※ 쉐이더 상수
▶ 상수 버퍼
- 상수 변수 사용에 최적화, CPU가 빈번히 갱신
- 쉐이더 상수를 그룹으로 묶어서 동시에 갱신
- 갱신 빈도에 따라 상수를 그룹으로 묶는게 효과적
- 하나의 상수 버퍼는 4096개의 벡터 크기를 가질 수 있음
- 파이프라인 마다 14개의 상수 버퍼 연결 가능
- 파이프라인에 연결하기 위해 뷰 필요(쉐이더에서 변수처럼 사용)
▶ 텍스쳐 버퍼
- 인덱스화된 데이터에 적합, 텍스쳐처럼 인덱스로 사용
- 파이프라인 마다 128개의 텍스쳐 버퍼를 연결할 수 있음
- 파이프라인에 연결하기 위해 뷰 필요, 텍스쳐 슬롯에 연결되어야 함

※ 리소스
▶ 쉐이더 리소스
▶ 읽기 전용: 정점 버퍼, 인덱스 버퍼, 상수 버퍼, 버퍼, 텍스트
▶ 읽기/쓰기: 무순서 접근 뷰

※ 리소스 차이점
▶ 정점버퍼/인덱스버퍼: 쉐이더에서 직접 사용할 수 없늠, 배열, 뷰가 필요(서술자 힙은 필요 없음)
▶ 상수 버퍼: 연결된 쉐이더 단계에서 전역변수처럼 사용, 구조체와 유사, 뷰가 필요
크기는 256바이트 배수, 하드웨어 레지스터를 사용, 크기 제한
▶ 버퍼: 배열, 인덱스를 사용, 구조체 배열, 크기 제한 없음, 뷰가 필요
모든 쉐이더 단계에서 사용, 여러 쉐이더 단계에 연결, 뷰가 UAV면 쓰기 가능
쉐이더 단계들 사이의 정보 전달 수단이 될 수 있음
▶ 텍스쳐: 모든 쉐이더 단계에서 사용, 뷰가 필요, 샘플러 객체를 통하여 사용할 수 있음

※ 대부분의 데이터는 상수버퍼뷰로 전달
▶ 자료형 버퍼: 하나 이상의 스칼라, 벡터 행렬을 포함하는 버퍼, Load 함수로 값을 읽음, 배열처럼 취급
ex) float4 pos = float4(1, 2, 3, 4) -> 위치, 색상으로 사용하고 배열처럼도 사용 가능
-> pos.z == pos[2]

※ Swizzling: 하나 이상의 요소를 임의의 순서로 사용
-> temp = pos.xy / pos.rg / pos.xg(이건 안됨)
▶ 배열 표현([])은 할 수 없음

※ Masking: 하나 이상의 요소에 저장 가능
-> f4D.xzyw = pos.w / pos
-> f4D.xx = pos.xy(이건 안됨)

※ 벡터를 스칼라로 사용하면 첫 번째 원소가 사용됨
ex) f4D.a = pos * 5 / f4D.a = pos.r * 5 (같음)

※ 벡터와 행렬 수식
▶ float3x3 mat3 = mat1 * mat2: 행렬의 곱이 아님(요소별 곱임)
▶ 내장 함수인 mul() 함수를 사용하자

※ 자료형
▶ 구조체 사용 가능
▶ 구조체의 멤버들
▶ centroid: 픽셀의 포함 면적에 따른 보간

※ 흐름 제어
▶ break continue do for if switch while
▶ discard: 픽셀 쉐이더에서 현재 픽셀의 결과를 출력하지 않음(버림)(문장이기 때문에 끝에 ;)

※ 반복문은 반복 조건의 계산과 점프하기 위한 오버헤드 필요
▶ a[i] = b[i] + c[i] -> a[i] = b[i] + c[i]; a[i+1] = b[i+1] + c[i+1]
-> 소요되는 시간을 줄일 수 있다면 실행 속도 향상 가능

※ 함수
▶ 리턴 데이터 타입이 있으면 [:Semantic]을 붙일 수 있음

※ 내장 함수 제공
- abs: 절대값(벡터도 넣을 수 잇음)
- clip: x가 0보다 작으면 현재 픽셀을 버림
- lerp(x, y, s): 선형 보간
- length(x): 벡터의 길이
- mul(x, y): y에다가 벡터를 넣으면 알아서 열 벡터로 바꿔서 계산해줌
- normalize(x): 벡터를 정규화

HLSL 쉐이더 최적화
① 내장함수를 사용
② 적합한 자료형 사용
③ 자료형 변환 줄이기
④ 정수 자료형 사용에 주의
⑤ 스칼라 상수를 팩킹
⑥ 불필요한 연산 줄이기

카메라

※ 원근 투영 행렬 사용 이유

▶ 화면의 종횡비가 다른 것을 보정

▶ fovy가 90도가 아닌 것을 보정

※ 카메라에서는 Z축(Look벡터)가 가장 중요함

▶ 조금만 틀어져도 내용이 바뀌기 때문

※ offset벡터가 영벡터면 현재 위치에 고정

▶ timeleg: 카메라가 회전을 할 때 3인칭 시점에서는 회전을 하더라도 항상 플레이어의 뒤통수만 보이기 때문에 지연 시간을 넣어 회전하는 시간에 딜레이를 넣어 회전하는 느낌을 줌

※ 카메라의 위치를 바꾸기보다 플레이어의 멤버 변수로 카메라의 4개의 벡터 사용(Position, Right, Up, Look)

▶ 카메라의 모드에 따라 제한 각도가 존재할 것, 몇 도 회전하고 있는지 정보가 있어야 함

-> x축 회전(pitch), y축 회전(roll), z축 회전(yaw)

▶ 현재의 속도를 나타내는 벡터 있어야 함: Velocity

▶ 중력도 있어야 함: Gravity

▶ 키보드를 막 누르면 막 이동하지 않게 제한 속도 필요

▶ 카메라가 지형에 쳐박히지 않도록 해야 함, 그것들의 정보: UpdatedContext

※ 카메라의 회전의 경우 모드에 따라 제한을 걸어둠

▶ 대부분 게임에서 실제 플레이어의 회전은 y축만 회전하고 x, z축은 카메라를 회전시킴

※ 속도 벡터 + 감속 벡터(정규화된 반대 방향 벡터 * 마찰계수) = 실제 속도 벡터

※ OnPlayerUpdateCallback(): 플레이어의 높이를 가져와서 플레이어가 지면에 쳐박히지 않도록 함

※ OnCameraUpdateCallback: 지형의 와인딩 오더가 바뀌어서 지형이 안 그려지는거 방지

※ Lean: 사람이 몸을 기울이는 행위

※ 3인칭 카메라를 y축 회전하면 플레이어를 바라보고 공전을 함

▶ 마찬가지로 카메라가 회전을 할 때 3인칭 시점에서는 회전을 하더라도 항상 플레이어의 뒤통수만 보이기 때문에 지연 시간을 넣어 회전하는 시간에 딜레이를 넣어 회전하는 느낌을 줌

▶ Update에서 늘 플레이어를 바라보게 하기 위해서 SetLookAt() 해줌

※ 절두체 컬링

▶ 투영 변환을 하면 투영 좌표계에서 정점을 클리핑 할 수 있음,

▶ 화면에 보일 수 있는 정점에 대해서만 나머지 파이프라인 처리

-> 화면 밖의 정점에 대해서 불필요한 계산 필요, 화면 밖 정점들로 인해 성능 저하 발생

▶ 투영 변환 연산을 하기 전에 정점이 카메라에 보이는 가를 판단하면?

▶ 카메라에 보이는 정점들만 렌더링 가능

-> 일부 정점에 보이지 않더라도 전체 객체는 화면에 표시될 가능성이 있음, 카메라 시야 판단을 위한 추가 계산 필요

※ 절두체 구조체

▶ 슈발 모르겠다 너무많아 유기(카메라 자료 47p)

터레인(실외 지형)

※ 실외 지형

▶ 지형의 높이를 위해서

-> y = h(x, z) 높이를 반환하는 함수를 사용

※ 높이 맵(Height Map)

▶ 각 픽셀이 지형의 높이를 나타내는 2차원 이미지

※ 함수의 조건(혹시 몰라서)

① 정의역에 있는 모든 원소가 공역에 있는 원소에 대응 되어야 함

② 정의역의 각 원소는 단 하나의 공역 원소에 대응되어야 함

※ 이미지는 함수임

※ 이미지 클래스에 이미지를 저장할 때 row 이미지로 통일

※ 하이트 맵을 받아와 z축을 맞춰주기 위해 x축 중심으로 회전

※ 절두체 컬링을 하더라도 메쉬가 너무 많으면 프레임레이트가 안나오기 때문에

-> 서브 메쉬로 나눔

▶ 각각의 서브 메쉬를 만들기 위해서 xStart,zStart, m_nWidth, m_nLength 필요

▶ 분리하는 이유: 지형 메쉬를 하나의 덩어리로 처리하면 CPU의 캐쉬는 너무 비싸기 때문에 한꺼번에 로드가 어려움

▶ 그렇다고 서브 메쉬를 너무 쪼개넣으면 Draw를 여러 번 해야 하는 문제 발생

-> 목적, 성능에 따라 하나로 그릴지, 여러 개로 그릴지 합의점을 찾아야 함

-> 이것이 일반적으로 최적화

※ 정점을 추가할 때

▶ 한 방향으로만 가면 의도치 않은 큰 삼각형이 생기기 때문에 왼쪽 방향으로 추가했으면
다음 줄로 넘어갈 때 오른쪽 방향으로 추가가 되도록 함

▶ 삼각형이 되지 않는 점(ex. 11, 11, 17)을 강제로 추가하여 올바른 삼각형이 그려지도록 함

-> 다렉 자체에서 CCW CW CCW CW CCW ... 순서로 처리함

※ 땅과 플레이어를 충돌 처리해서 서있게 함

▶ OnGetHeight(): 함수를 가져와서 현재 픽셀의 높이값을 가져옴

▶ OnGetColor(): 정점의 색상

▶ 가상 조명이 있다고 가정

▶ 너무 어둡다 싶으면 살짝 밝게 해주는 if문 존재

※ 지형의 높이와 법선 벡터

▶ 높이 맵의 픽셀 값을 사용하여 지형의 높이 계산(보간)

▶ 높이 맵의 픽셀 값을 사용하여 지형의 법선 벡터 계산

▶ 점 4개를 보간하여 픽셀의 최종 높이 결정

※ GetHeight(): 점을 구할 때 점 3개를 이용해 나머지 1개의 점을 구함

※ 만약 카메라가 있는데 카메라가 지형을 뚫고 가버리는 경우는 카메라와 지형을 보간해서 카메라의 위치를 조정

조명

※ 조명 계산

▶ 실시간으로 조명 효과 계산: Realtime Lighting

▶ 조명 효과를 미리 계산: Precomputed Lighting, Light Mapping

▶ 게임 내의 객체 중 미리 계산이 가능한 것은 미리 계산하자(상수로 취급이 된다면)

※ 빛 색

▶ 빛의 세기: 벡터의 크기

※ 재질

▶ 물체의 표면에서 빛의 반사는 재질의 성질에 따라 달리짐

▶ RGBA

▶ 빛의 반사는 빛과 재질 색상에 따라 달라짐

▶ 빛의 반사는 빛의 색상과 재질의 색상에 따라 달라짐

※ 발광 조명

▶ 물체의 표면이 스스로 빛을 발산

※ 주변 조명

▶ 물체의 표면이 빛을 반사하고 반사된 빛이 다른 물체의 표면에서 반사

▶ 모든 표면에서 같은 양의 빛을 반사하여 전체적으로 균일한 조명이 됨(간접 조명)

▶ 주변 조명만 있다면 모든 표면의 색상은 같음

▶ 조명의 위치와 방향에 상관없이 표면의 모든 점에서 같은 양의 빛 반사

※ 직접 조명

▶ 물체의 표면에 직접 들어오는 빛의 반사에 따라 표면의 색상과 밝기가 달라짐

▶ 조명의 위치와 방향에 따라 표면의 점에서 빛의 반사가 달라짐

- Diffuse Light: 확산 조명
- Diffuse Reflection: 확산 반사
- Specular Light: 스펙큘러 조명
- Specular Reflection: 스펙큘러 반사

※ 확산 조명

▶ 거칠고 매끄럽지 않은 표면에서 일어남

▶ 물체의 표면에서 모든 방향으로 균일하게 빛을 반사한다고 가정

▶ 표면에서 반사되는 빛은 카메라의 위치와 방향에 상관없이 계산될 수 있음

▶ 램버트 법칙: 빛의 반사의 양이 많은 점은 밝게, 반사가 없는 점은 어둡게 보임

※ 스펙큘러 조명

▶ 부드럽고 매끄러운 표면에서 일어남

▶ 표면이 반짝거리고 부드럽게 보이는 강한 반사를 만듬

▶ 표면이 매끄러울수록 좁은 영역에서 훨씬 강한 반사 일어남

▶ 모든 방향으로 균일하지 않으므로 보는 위치에 따라 다르게 일어남

▶ 표면에서 확산 반사와 스펙큘러 반사 모두 일어남

※ 정점 조명
▶ 각 정점에 따라 조명의 영향을 계산(빠름), 상수 시간
▶ 문제점
- 정점에 빛이 도달하지 않으면 아무리 메쉬에 빛이 도달하더라도 검정색으로 보간됨
- 계단 현상이 일어날 수 있음, 계단 현상이 생기면 정점을 더 쪼개면 되는데 그건 너무 투머치

※ 픽셀 조명
▶ 각 픽셀에 대하여 조명의 영향을 계산(느림), 상수 시간이 아님
▶ 문제점
- 카메라가 메쉬를 확대하면 픽셀의 수가 늘어나기 때문에 조명 계산량이 늘어남

※ 개쩌는 해결책!
▶ 파이프라인에서 카메라에서 먼 놈은 정점 조명으로 처리하고, 가까운 놈은 픽셀 조명으로 처리

※ 정점의 법선 벡터: 조명 계산을 위하여 각 정점은 법선 벡터를 가져야 함
▶ 같은 평면의 정점들은 다른 법선 벡터를 가질 수 있음
▶ 하나의 정점은 여러 삼각형들에 속할 수 있음
▶ 정점의 법선 벡터는 이 정점을 포함하는 모든 삼각형의 법선 벡터의 평균으로 계산

※ 조명에 의한 정점(픽셀)의 색상: I = (A + D + S) + E

※ 반사의 성질(재질)이 필요: MATERIAL
▶ 다른 게임 엔진에서는 디퓨즈, 앰비언트, 스페큘러를 한꺼번에 알비도로 처리하기도 함
▶ 조명의 성질이 필요: LIGHT

※ 정점의 법선 벡터 변환
▶ 정점이 변환되면 법선 벡터도 변환되야 함
▶ 세 축의 크기 변환이 다르면 법선 벡터 바뀜 / 같으면 법선 벡터 방향 안바뀜
▶ 접선 벡터: 법선 벡터와 수직인 벡터

※ 월드변환에 스케일링 있으면 좋지 않음(교수님 강조)
▶ 법선벡터의 변환은 역행렬의 전치행렬을 해야 됨(힘듬)
▶ 모든 픽셀에 대해서 노멀라이즈를 다시 해줘야 함

※ 점 조명(Point Lights)
▶ 모든 방향으로 빛을 발산
▶ 점과 광원까지의 거리가 m_fRange 보다 크면 조명의 영향을 받지 않음
▶ 감쇠: 점 조명은 거리에 비례하여 빛의 양이 점차적으로 줄어듬

※ 스팟 조명(Spot Light)
▶ 손전등과 비슷
▶ 한 방향으로 빛 발산
▶ m_fRange 거리까지 영향을 줌
▶ 안쪽 원의 끝에서 바깥 원 방향으로 감쇠가 일어남
▶ 거리 비례 감쇠가 일어남

※ 방향성 조명(Directional Lights)
▶ 한 방향으로 평행한 빛을 발사
▶ 방향성 조명은 거리에 따라 감쇠가 일어나지 않음

※ 재질
▶ 물체의 표면에서 빛의 반사는 재질의 성질에 따라 달라짐
▶ 빛의 반사(색상)는 빛과 재질의 색상에 따라 달라짐
▶ 빛의 반사는 빛과 색상과 재질의 색상의 벡터 곱으로 표현
▶ 정점이 조명에 어떻게 반응하는 가를 결정

※ 발광 재질
▶ 재질 자체가 빛을 발산
▶ 그러나 이 빛이 다른 점에 영향을 주지 않는 것으로 가정

※ 조명 계산 효율성
▶ 월드 좌표계
- 조명 위치, 방향, 카메라 위치: 월드 좌표계
- 정점: 모델 좌표계 -> 월드 좌표계
▶ 모델 좌표계
- 정점: 모델 좌표계
- 조명 위치, 방향, 카메라 위치: 월드 좌표계 -> 모델 좌표계
▶ 조명의 색상과 재질의 색상의 곱 미리 계산
▶ 조명 맵
- 정적인 조명과 객체들의 조명 효과 미리 계산
- 미리 계산된 조명 효과 텍스쳐로 저장해 쉐이더에서 샘플링하여 조명 미리 처리
- 조명 맵에서 샘플링하기 위한 텍스쳐 좌표를 정점에 추가적으로 설정
▶ 지연 조명 맵
- 화면에 나타나지 않는 픽셀에 대한 조명 계산은 필요 없음

프레임 계층 구조

※ 모델의 표현

▶ 모델은 단일 메쉬 이기도 하고 다중 메쉬 이기도 함

※ 다중 메쉬

▶ 여러 개의 메쉬들로 모델을 표현할 때

▶ 메쉬들을 공간적인 계층 구조 또는 프레임 계층 구조로 표현

※ 계층 구조 사용 이유

▶ 로컬 트랜스폼: 자식은 부모의 변환 행렬에 영향을 받지만, 자식의 변환 행렬엔 자식만 영향을 받음

※ 게임 오브젝트를 프레임이라고도 부름(프레임 = 행렬 + 메쉬)

▶ 그래서 하나의 모델을 구성하는 메쉬들을 모아둔 것을 프레임 계층 구조라고 부름

▶ 프레임 계층 구조는 트리로 구현

▶ 노드 = 행렬 + 메쉬 + 자식 포인터

※ 모든 트리는 이진 트리(바이너리 트리)로 바꿀 수 있음

▶ 트리의 왼쪽은 자식 노드, 오른쪽은 자신의 형제 노드로 구성

▶ 포인터를 두 개만 사용해서 프레임 계층 구조 구현 가능

- FRAME *m_pChildFrame;
- FRAME *m_pSiblingFrame;

※ 정기구학

▶ 계층 구조의 프레임 변환

▶ 프레임 Fi의 월드변환 행렬: M

▶ 객체의 월드 변환 행렬: W

▶ 루트의 경우 M0가 단위행렬일 것

▶ 모든 프레임 계층 구조로 되어있는 메쉬를 렌더링하기 위해서는 트리 구조의 루트 좌표계로 바꾸는 행렬 사용

-> M = Mk + (Mk-1 * … *(M0 * W))

※ 게임 오브젝트 구조체

- 프레임 이름
- 변환 행렬
- 메쉬
- 메쉬 수
- 자식 포인터
- 형제 포인터

※ 애니메이션은 배열의 형태로 되어있음

▶ A 애니와 B 애니 사이의 연결 동작이 부자연스러우면 A의 첫과 B의 끝 배열의 일부 원소를 제거해 부드럽게 함

※ 계층 구조 바운딩 박스

▶ 모든 자식 노드들의 바운딩 박스들을 루트의 바운드 박스로 머지함

▶ 그래서 루트의 바운드 박스가 충돌이 되면 그 메쉬가 충돌이 된 것

▶ 루트 바운드 박스가 충돌이 되면 그 객체의 노드를 따라 어떤 노드가 피킹이 된 것인지 판단