**3DGP 2**

2021184002 길준형

**2025.09.01**

조명

가상적인 세상에 물체들을 비춰주는 물체

조명이 직진을 하다가 불투명한 물체에 부딪힐 경우 반사하거나, 흡수하거나, 일부 통과시키는데 우리는 모두 반사한다고 생각한다.

이때 반사된 빛은 다시 다른 물체에 부딪힐 수 있고, 이렇게 조명에서 물체로 부딪힌 빛이 눈(카메라)으로 직접적으로 온다면 직접 조명, 물체에 반사된 빛이 다른 물체에 다시 반사되어 눈(카메라)으로 들어온다면 간접 조명이라고 한다.

재질에 따라 어떤 빛은 흡수하고 반사한다.

직접조명

* 물체의 재질과 색깔에 따라 그대로 보인다.

간접조명

* 물체의 재질과 색깔이 다른 물체에 반사되었기에 재질과 색깔이 변화할 수 있다.
* 간접 반사는 무한히 일어날 수 있기에 계산도 무한하다

예전에는 애니메이션과 같은 곳에서는 실시간으로 조명처리를 할 필요가 없고 오랜 시간이 걸려도 상관이 없으므로 사용되었지만, 게임은 실시간으로 많은 처리를 하기에 큰 무리가 가므로 래스터라이제이션을 통해 조명 처리를 한다.

현실세계에서 물체가 앞뒤에 존재한다면 뒤에 있는 물체는 조명이 도달하지 않는다. 그렇다면 조명에서 광선이 나온다고 했을 때, 이 광선이 가장 먼저 닿는 물체가 있다면 더 이상 이 광선은 뒤에 있는 물체에는 도달하지 않을 거다. 우리는 이 방법을 피킹에서 해봤다.

이때 조명에서 사진을 찍는다면 2d이미지 2개(렌더타겟과 뎁스값)을 얻을 수 있다. 뎁스값을 통해 가장 먼저 닿는 위치를 알 수 있기에 조명 처리가 가능하다.

조명의 개수가 늘어나면 계산시간은 늘어나지만 3D, 공간감은 늘어난다. 물론 이렇기에 프레임 레이트를 지키지 못한다.

그렇다면 물체에 색을 입힌다면 계산시간은 줄어들지만 2D와 다를 바 없다.

//=================================================================//

**정적 물체와 정적 조명 사이의 관계**

* 벽, 천장, 바닥은 이미 고정된 물체로 항상 조명에 빛을 받고 있다. 그렇다면 이들의 값은 상수로 처리할 수 있다. 한마디로 실시간 처리할 필요가 없다는 것이다.
* 둘 중 하나라도 동적이라면 미리 계산하는 의미가 없다.

**물체와 정적 조명 사이의 거리 관계**

* 조명이 물체와 멀다면 영향이 적고, 가깝다면 영향이 크다.
* 공간을 나눠서 균일하게 비춘다고 근사하여 계산해도 상관이 없다.

**어두운 공간에서 조명을 비출 경우**

* 이 순간 조명의 영향을 계산하기보다 (비춘다 OR 안 비춘다)의 개념이다.

**조명의 세기**

* 감지할 수 있는 조명의 세기를 넘어갈 경우, 몇 개의 조명이 비춘다고 해도 가장 밝은 몇 개의 빛만 계산해도 된다.
* 약한 빛은 영향이 미비하다

//=================================================================//

RGB는 해당 색의 빛을 잘 반사한다는 것이다. 그럼 눈(카메라)에 들어올 때 빛의 색상과 물체의 재질에 반사된 색을 벡터 곱을 한 색이 들어오는 거다.

이때 빛은 RGBA에서 A는 신경 쓸 필요가 없는 데 빛은 투명하기 때문이고, 물체의 경우에는 불투명한 경우나 반투명, 투명한 경우 등 다양하게 있으므로 A를 신경 써야 된다.

**디퓨즈 반사**

* 거친 표면에서 일어난다
* 물체 표면에서 모든 방향으로 균일하게 빛을 반사한다고 가정
* 모델러는 물체를 거칠게 표현할 때 폴리곤으로 하면 너무 많은 폴리곤이 들어가지만, 이 표면에서 디퓨즈 반사가 일어난다고 하면 볼 때 거친 표면을 구현할 수 있다.
* 이때 반사되는 빛의 세기는 줄어들 수 있다. (조명 입사각 벡터와 물체표면의 법선 벡터)

**스펙큘러 반사**

* 매끄러운 표면에서 일어난다.
* 이때 빛은 한 방향으로만 반사된다.
* 반사되는 벡터 방향에 눈(카메라)가 위치한다면 밝게 보이고, 아니면 어둡게 보인다.
  + 하이라이트

**2025.09.03**

정점 조명은 삼각형이라고 가정할 때 정점 3개으로만 계산하면 된다.

* 기본적으로 상수 시간이 걸린다.
* 문제는 각 정점들은 조명정보에 의해 잘 계산되지만 나머지는 근사하게 되므로 퀄리티가 떨어진다.
* 조명이 어떤 물체에 비출 때 정점을 비춰야 조명계산을 할 수 있다.
  + 만약 그렇다면 정점을 엄청 만들어야 된다.

하지만 픽셀 조명은 각 픽셀에 대해서 모두 계산하므로 많아졌다 적어졌다를 반복한다.

* 카메라와의 거리에 따라 시간이 달라진다.

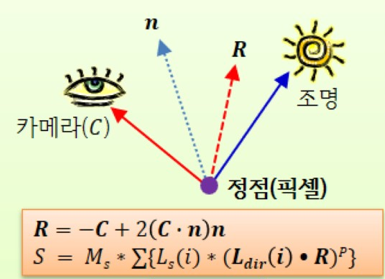
그렇다면 카메라와의 거리에 따라 조명처리를 바꿔줄 필요가 있다.

조명과 재질의 색깔을 무조건 다 같은 색으로 할 필요는 없다.

스펙큘러, 디퓨즈, 앰비언트의 색상을 모두 다른 색으로 설정해도 된다.

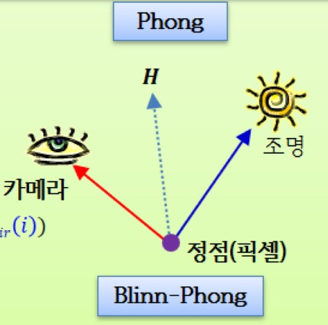


**A: 앰비언트 색상 / D: 디퓨즈 색상 / S: 스펙큘러 색상 / E: 점 자체에서 발산되는 색상**



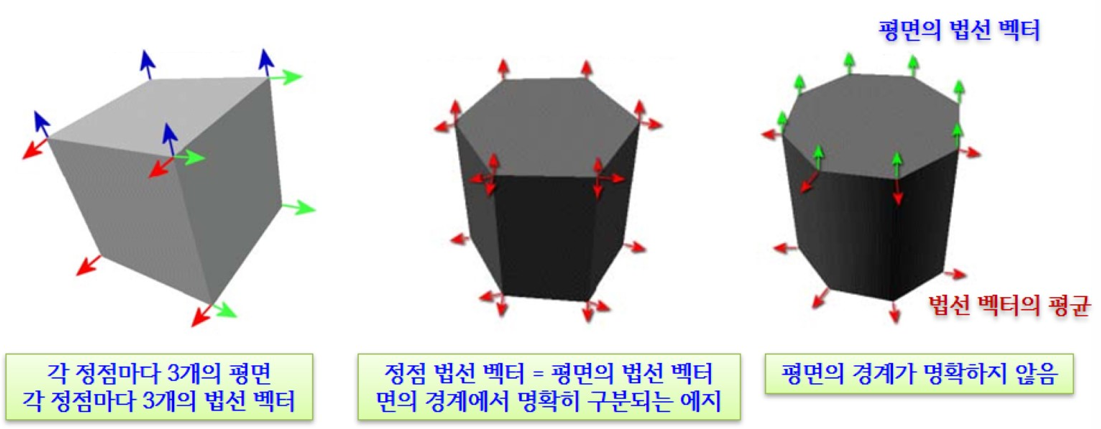
정점의 노말에 대해 카메라의 반사벡터 R을 구할 수 있는데, 이 R과 조명과의 각도가 작으면 작을수록 하이라이트가 커진다.

* 빛과 R의 값이 0도일 때 1의 값을 가지고, 90도로 갈수록 0에 가까워진다.
* 0에 가까울수록 S의 값은 기하급수적으로 작아진다.

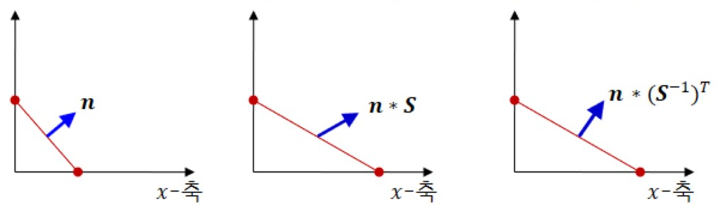
 H: 카메라와 조명의 법선의 벡터의 평균

퐁은 이 H와 정점의 노멀의 각도의 값이 작을수록 스펙큘러 반사가 강해진다고 했다.

위의 수식이든 아래 수식이든 카메라의 위치에 따라 스펙큘러 반사가 달라진다.



각 정점이 사용되는 평면에서의 법선 벡터를 구할 수 있는데, 이 정점의 법선 벡터의 평균은 사용되는 모든 평면에서의 법선 벡터들을 정규화하면 된다.



모델을 스케일링할 경우, 모델의 법선 벡터가 달라지게 되는 경우가 있다. 이 경우 스케일링변환의 역행렬의 전치를 곱해주면 된다. 만약 모두 다 같은 비율로 스케일링 될 경우에는 이럴 필요가 없지만 정규화도 진행해줘야 하는데, 이 때 계산이 많아지게 되고 프레임 레이트가 망한다.

따로 조명 별 정리하기

2025.09.08

실세계에서 물체(3D모델) 표면을 봤을 때 평평하거나, 균일하지도 않다. 보통 이런 것들을 **질감**이라고 말함. 이 질감을 시각적으로 사각형의 이미지 형태로 표현한 것을 **텍스쳐**라고 한다. 표면에 이미지를 그리는 방법을 **텍스쳐링**이라고 한다.

2D 이미지에서 각 텍스쳐의 픽셀 위치를 나타낸 것을 **텍셀**이라고 한다. 이때 텍스쳐를 읽을 때는 **샘플러**가 필요하다.

텍스쳐는 1D, 2D, 3D, 큐브의 형태일 수도 있다. 각 차원의 텍스쳐는 밉맵을 가질 수 있다. 만약 큐브를 만들 때 모두 같은 모양이라면 단 하나의 이미지만 사용하면 되겠지만, 주사위를 예를 들어보자면 총 6개의 이미지가 필요해진다. 이때 이미지들은 각각의 밉맵을 가지게 되고 이것들을 배열로써 텍스쳐를 가질 수도 있다?

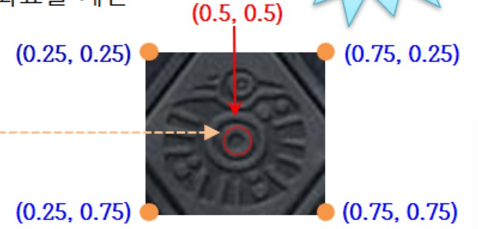
//=================================================================//

**밉맵**

****만약 하나의 텍스쳐만을 사용한다면, 거리마다 구현을 할 때 프레임마다 동적으로 샘플링해야된다. 이러면 프레임레이트가 떨어질 것이다. 그렇다면 각 거리에 맞게 미리 **2의 거듭제곱 꼴**로 텍스쳐를 만들어 둔다면 훨씬 좋을 거다. 결국 거리가 일정 이상으로 벌어지면 1X1 해상도가 되게 되어 한 픽셀로만 나타내면 된다. 이렇게 만들어 놓은 텍스쳐의 집합을 **밉맵**이라고 한다.

//=================================================================//

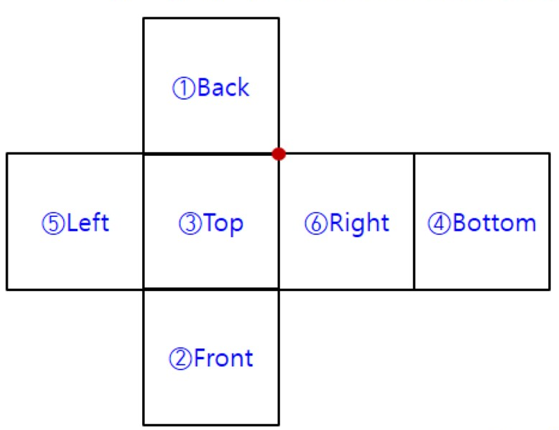
**정규화**

텍스쳐 좌표는 (0~1)로 정규화 되어있다.

만약 0~1 이 아닌 0~99로 되어 있다면 각 좌표를 사용할 때마다 코드를 맞춰줘야 하는데, 이 좌표가 바뀐다면 다시 코드를 갈아 엎어야 한다. 하지만 좌표가 정규화 되어있다면 코드를 바꿀 필요 없이 위치를 특정할 수 있다.

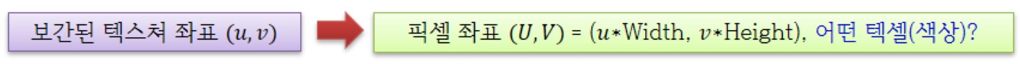
//=================================================================//

**매핑**

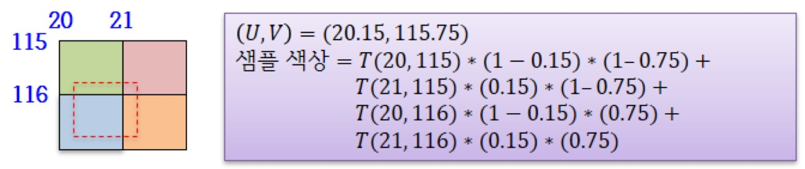
만약 큐브를 만들 때 정점을 8개만을 사용해서 구현했을 때, 이미지를 모두 같은 방향으로 나타나게 하고 싶다. Top에서 텍스쳐를 입힐 때는 빨간 점을 기준으로 (1, 0)으로 매핑해줘야하지만 Right에서는 (0, 0)으로 매핑해줘야한다. 하지만 정점을 8개만을 사용했기에 각 정점의 (u, v) 좌표 값이 똑같기에 원하는 대로 나오지 않게 된다. 그렇기에 원하는 대로 이미지를 매핑하기 위해서는 각 정점을 따로 만들어줘야 한다.

//=================================================================//

**색깔 결정**



텍셀의 좌표를 알기 위해서는 정규화된 (u, v)에 해상도를 곱하면 좌표를 구할 수 있지만, 문제는 (u, v)는 실수이고 해상도는 정수이기에 좌표가 실수로 나오게 된다.

가장 간단하게 좌표를 구하는 방법은 소수점 값을 버리거나, 반올림하는 방법이 있다. 하지만 이렇게 되면 유사한 값들이 중복되게 되어 색깔이 중복되게 된다.

이때 색상을 결정할 때 거리에 따라 밉맵의 레벨이 결정될텐데 딱 레벨 1, 레벨 2로 정해지지 않는 경우가 많다. 레벨 1.4 또는 레벨 1.7처럼 정확하게 레벨이 결정되지 않을 경우, 레벨 1에서 몇 %, 레벨 2에서 몇 %를 가져와서 섞는다면 색깔을 정할 수 있다.

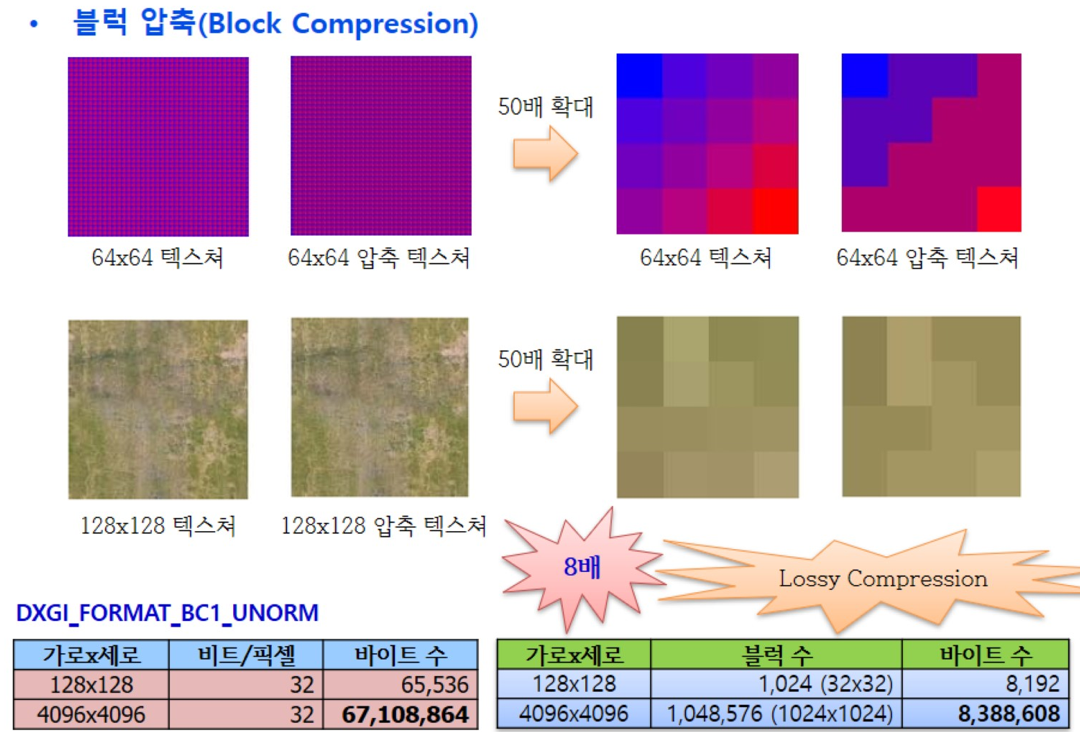
//=================================================================//

각 텍스쳐를 저장을 하다보면 정점의 데이터나 인덱스 등 많을 양의 정보를 담기에 텍스쳐의 크기가 커지게 되고 사용하기 위해서 많은 메모리가 필요하게 된다. 그렇기에 압축을 할 필요가 있다.



색깔을 나타낼 때도 색깔 벡터 2개와 실수 하나면 다양한 색깔로 나타낼 수 있다.

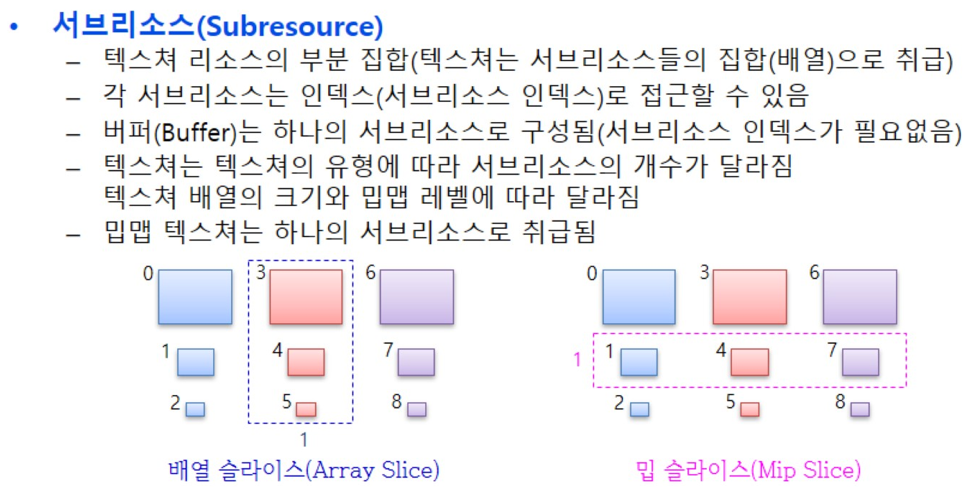
텍스쳐를 4x4 텍셀 블록으로 분할 한 뒤, 각 텍셀을 2비트로 표현한다. 각 블록은 4개가 아닌 2개의 색상을 가진다. 하지만 모든 색깔을 표현하지 못하기에 손실이 있는 압축방식이다.



텍스쳐가 투명도를 가지고 있다면 BC2 or BC3를 사용하면 되고, 아니라면 BC1을 사용하면 된다.

2025.09.10

SUBRESOIURCE 부분집합에 해당하는 데이터



텍스쳐는 반드시 루트 디스크립터 테이블로 밖에 못넘긴다.