3D Game Programming Basic with Direct3D

5. 색상(Color)과 쉐이딩 모델

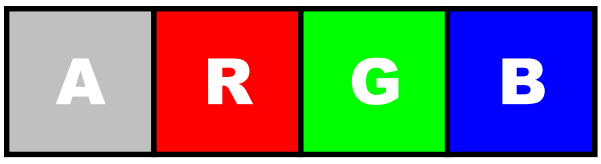
색상과 조명은 가장 쉬우면서도 또한 가장 어려운 부분입니다. 같은 색상과 조명을 사용하더라도 환경에 따라 게임의 화질(Quality)을 천국과 지옥을 오가게 만듭니다. 좀 더 화려한 색상과 조명을 만들기 위해서 많은 경험이 필요합니다. 이런 경험도 D3D가 처리하는 방식을 알고 있다면 통제가 되고 자신이 원하는 색상에 가깝게 만들 수 있습니다.

5.1 색상(Color)

쉐이딩 모델은 앞서 기초 시간에 래스터 처리에서 기하 정보를 가지고 픽셀을 채우는 방법이라 했습니다. 3D는 사람 눈으로 보는 세상을 컴퓨터에 동일하게 표현하는 것이 목적입니다. 컴퓨터의 자원은 제한이 되어 적은 자원으로 사람이 인식하는 특징만을 골라서 이론으로 만든 것이 모델입니다.
지금까지 그래픽스에서 실 세계의 사물을 삼각형의 집합(폴리곤)으로 만들고 수학적인 연산을 거쳐 2차원 화면에 표시하는 방법을 이용해서 사람의 눈으로 보고 있는 세상을 흉내 내고(Simulation) 있는 것입니다.

문제 해결 폴리곤을 이용하면 사물을 컴퓨터에 비슷하게 표현할 수 있을 알게 되었습니다. 다음으로 색을 표현하는 문제인데 32Bit True Color면 사람이 느끼는 충분한 색상을 표현할 수 있다고 합니다.

D3D는 색상을 다음 그림과 같이 Alpha 8비트, Red 8비트, Green 8비트, Blue 8비트 총 32비트로 표현합니다.



<D3D 색상>

하지만 바이트 순서는 B->G->R-A 순입니다. 이 순서를 외우기 쉽게 "배째라" 하기도 합니다. 만약 Alpha를 a, Red를 r, Green을 g, Blue를 b라 할 때 이를 32비트로 색상으로 만들면

최종 색상 = $b*2^{0} + g*2^{8} + r*2^{16} + a*2^{24}$

이 됩니다.
DXSDK는 이 식을 매크로로 다음과 같이 지원하고 있습니다.

```
#define D3DCOLOR_ARGB(a,r,g,b) ₩
    ((D3DCOLOR)((((a)&0xff)<<24)|(((r)&0xff)<<16)|(((g)&0xff)<<8)|((b)&0xff)))

#define D3DCOLOR_ARGB(a,r,g,b) ₩
    ((D3DCOLOR)((((a)&0xff)<<24)|(((r)&0xff)<<16)|(((g)&0xff)<<8)|((b)&0xff)))
#define D3DCOLOR_RGBA(r,g,b,a) D3DCOLOR_ARGB(a,r,g,b)
#define D3DCOLOR_XRGB(r,g,b)   D3DCOLOR_ARGB(0xff,r,g,b)
```

이 매크로를 사용해서 색상을 설정하는 예는 다음과 같습니다.

```
DWORD   magenta= D3DCOLOR_XRGB(     255,   0, 255);
DWORD   red    = D3DCOLOR_ARGB(255, 255,   0,   0);
DWORD   green  = D3DCOLOR_RGBA(  0, 255,   0, 255);
DWORD   blue   = D3DCOLOR_XRGB(  0,   0, 255);
```

OpenGL과 쉐이더는 색상의 순서를 B, G, R, A순으로 합니다. 또한 각각의 색상 범위를 [0, 255]가 아닌 [0.0, 1.0] float형을 사용합니다. 색상을 [0.0, 1.0] 범위로 하면 덧셈은 색상이 밝아지는 가산 연산이 되고 곱셈은 어두워지는 감산 연산을 할 수 있기 때문입니다.

D3D는 색상을 32비트로 표현을 해도 이러한 연산을 위해 D3DXCOLOR 구조체를 지원합니다. 이 구조체는 내부가 float형 4개의 r, g, b, a 16(4*4) 바이트로 구성 되어 있습니다. 또한 DWORD형 캐스팅 연산자가 있어서 32비트 색상으로 만들거나 반대로 32비트 색상을 float4형으로 바꾸어 줄 수 있습니다.

DXSDK의 d3dx9math.h파일을 보면 D3DXCOLOR 구조체가 선언을 찾을 수 있습니다.

```
typedef struct D3DXCOLOR
...
    D3DXCOLOR( DWORD argb );
    // casting
    operator DWORD () const;
    operator FLOAT* ();
    operator CONST FLOAT* () const;
```

```
...
    // binary operators
    D3DXCOLOR operator + ( CONST D3DXCOLOR& ) const;
    D3DXCOLOR operator - ( CONST D3DXCOLOR& ) const;
    D3DXCOLOR operator * ( FLOAT ) const;
    D3DXCOLOR operator / ( FLOAT ) const;
...
FLOAT r, g, b, a;
} D3DXCOLOR, *LPD3DXCOLOR;
```

자주 사용되는 부분만 남겨보았습니다. 구조체 선언의 마지막을 보면 색상에 대해서 float형이 r, g, b, a 순으로 16 Byte가 설정되어 있음을 알 수 있습니다. 생성자에 DWORD를 받아서 32비트 색상을 처리할 수 있고, 연산자 다중 정의(Overloading)으로 색상끼리 더하거나 실수 배를 해서 밝고 어둠을 쉽게 다룰 수 있도록 되어 있습니다.

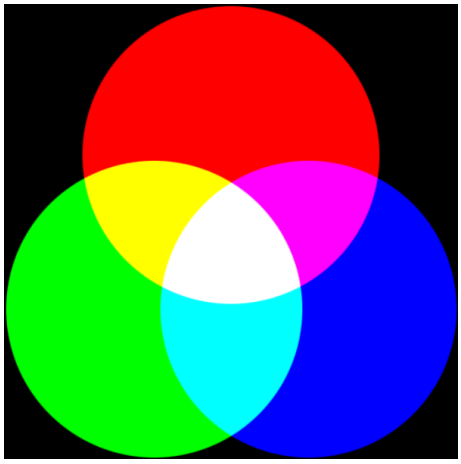
다음 예는 D3DXCOLOR 구조체를 이용한 다양한 연산입니다.

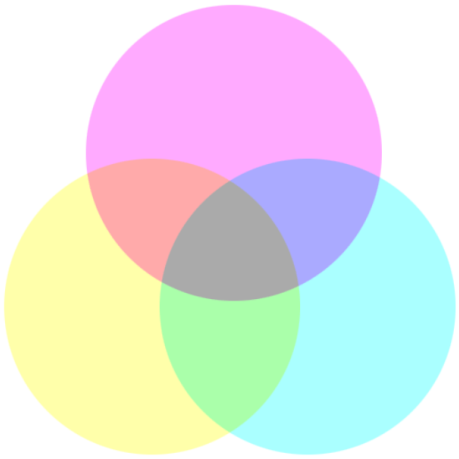
```
D3DXCOLOR       red(1, 0, 0, 1);
D3DXCOLOR       green = D3DXCOLOR(0, 1, 0, 1);
DWORD           blue  = D3DXCOLOR(0, 0, 1, 1);

D3DXCOLOR       magenta;
magenta  = red;
magenta += blue;
```

연산을 하다 보면 색상이 [0, 1] 범위를 벗어날 수 있습니다. 이런 경우에도 D3DXCOLOR 구조체는 32비트 색상을 만들 때 최종 색상이 [0, 255]로 재 조정 될 수 있도록 DWORD 캐스팅 연산자 안에 구현 되어 있습니다.

```
D3DXINLINE D3DXCOLOR::operator DWORD () const
{
    DWORD dwR = r >= 1.0f ? 0xff : r <= 0.0f ? 0x00 : (DWORD) (r * 255.0f + 0.5f);
    DWORD dwG = g >= 1.0f ? 0xff : g <= 0.0f ? 0x00 : (DWORD) (g * 255.0f + 0.5f);
    DWORD dwB = b >= 1.0f ? 0xff : b <= 0.0f ? 0x00 : (DWORD) (b * 255.0f + 0.5f);
    DWORD dwA = a >= 1.0f ? 0xff : a <= 0.0f ? 0x00 : (DWORD) (a * 255.0f + 0.5f);
    return (dwA << 24) | (dwR << 16) | (dwG << 8) | dwB;
}
```

이렇게 괜찮은 D3DXCOLOR 구조체를 색상을 처리할 때 많이 사용하기 바랍니다.





가산 연산

3D의 색상은 빛의 색상을 모델로 구성되어 있습니다. 물감은 색을 더하면 어두워지지만 빛은 더하면 그만큼 밝아 집니다.

예를 들어 Red + Blue조명을 생각해봅시다.

D3DXCOLOR 구조체를 이용하면

Red(1, 0, 0, 1) + Blue(0,0,1,1) = Magenta(1,0,1,1)

다른 예 한번 볼까요?

Red(1,0,0,1) + Green(0,1,0,1) = Yellow(1,1,0,1);

감산 연산

뺄셈

곱셈

5.2 모델(Shading Model)

사람들은 현재까지 실 세계의 사물을 삼각형의 집합(폴리곤)으로 구성하면 컴퓨터로 실제의 외형과 아주 비슷한 장면을 만들 수 있음을 알게 되었습니다. 이 방법은 앞서 배운 변환 과정에 해당하는 내용입니다.

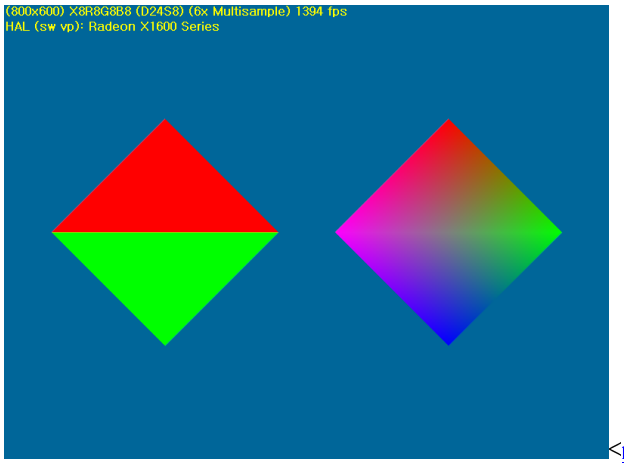
쉐이딩 모델은 앞서 기초 시간에 래스터 처리에서 기하 정보를 가지고 픽셀을 채우는 방법이라 했습니다.

이 물체의 외형에 사람의 눈이 인지하는 색상을눈이 느끼는 컴퓨터로 물체의 외형을 잘 흉내 내고(Simulation) 있는 것입니다
물체의 외
그래서 삼각형을 렌더링의 기본으로

수학적인 연산을 거쳐 화면에 ㅅ

표시하는 방법을 이용해서 사람의 눈으로 보고 있는 세상을 흉내 내고(Simulation) 있는 것입니다.
3D는 사람 눈으로 보는 세상을 컴퓨터에 동일하게 표현하는 것이 목적입니다. 컴퓨터의 자원은 제한이 되어 적은 자원으로 사람이 인식하는 특징만을 골라서 이론으로 만든 것이 모델입니다.

예제 m3d_05_color1.zip을 실행하면 다음과 같은 화면이 출력됩니다.

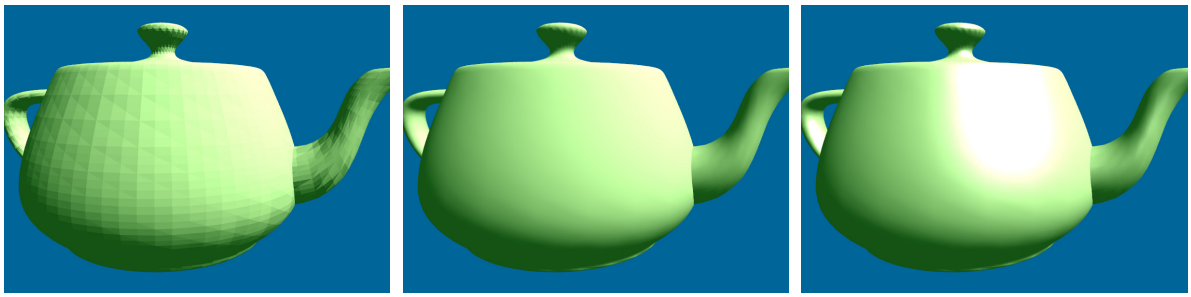


<m3d_05_color1.zip 쉐이딩 모델>

동일한 구조의 정점과 색상을 가진 두 개의 삼각형은 전혀 다른 모습으로 출력되고 있습니다.

D3D는 Flat Shading 모델, Gouraud Shading 모델, Phong Shading(Bui Tuong Phong) 모델 세 가지를 지원하고 있습니다.

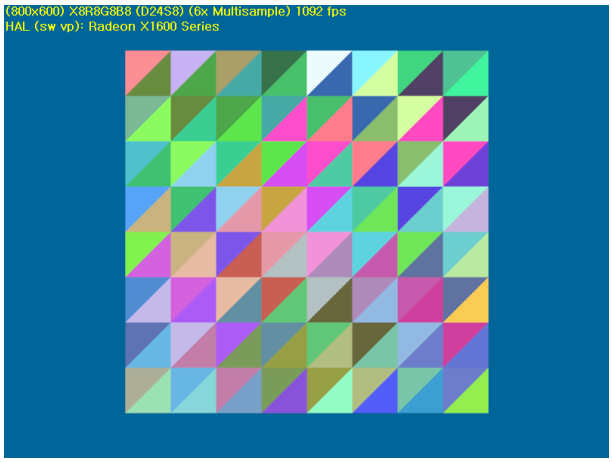
D3D는 Flat Shading 모델은 앞 그림의 왼쪽 2개의 삼각형처럼 삼각형을 시작하는 점의 색상을 가지고 삼각형 내부의 색상을 채웁니다. 이 방법은 간단해서 그래픽 하드웨어가 발전하지 못했던 초창기에서 많이 사용된 모델입니다.
구로(Gouraud) 쉐이딩 모델은 Henri Gouraud가 제안한 모델로 그림의 오른쪽 삼각형 2개와 같이 세 점의 색상을 선형(Linear)적으로 보간(Interpolation) 삼각형의 나머지 색상을 채우는 방법입니다.
구로 쉐이딩 모델은 처리 속도가 빠르고 색상 변화의 자연스러움이 잘 반영되지만 반사 되는 물체의 하이라이트(Highlight) 등은 표현이 안됩니다. Phong Shading(Bui Tuong Phong) 모델은 이러한 부분을 해결하기 위한 모델이며 특히 조명의 반사에서 많이 나오는 모델입니다.


<3D 기하 물체에 적용된 쉐이딩 모델: 왼쪽부터 Flat Shading, Gouraud Shading, Phong Shading>

D3D의 고정 기능 파이프라인은 Default가 구로 쉐이딩 모델입니다. 퐁 쉐이딩 모델은 선택 옵션이 있어도 구로 쉐이딩과 차이는 없습니다. 대신에 Specular Lighting 효과로 퐁 반사를 반영하고 있습니다.
m3d_05_color1.zip 의 McScene.cpp 파일의 void CMcScene::Render() 함수를 보면 D3D의 파이프라인의 쉐이딩 방법을 바꾸기 위해서 다음과 같이 상태 설정 함수를 이용하고 있음을 볼 수 있습니다.

```
m_pDev->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);          // Flat Shading
...
m_pDev->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);       // 구로 쉐이딩
```

실습 과제) 다음 그림과 같이 x, z 평면에 인덱스를 사용해서 정사각형이 8x8 인 지형을 만드시오.

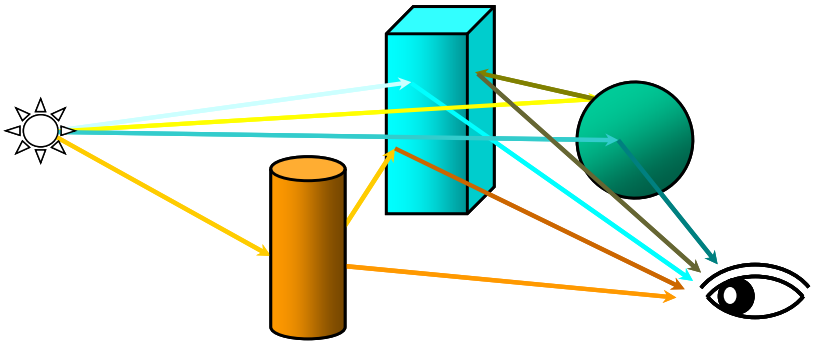


실습 과제) 위의 지형을 입력 값에 따라 4x4, 8x8, 16x16, 32x32, 64x64, 128x128을 출력할 수 있도록 프로그램을 수정하시오.

6. 조명 (Lighting) 과 재질(Material)

가장 간단한 빛 조차도 물리학의 전자기학 법칙으로 해석을 해야 합니다. 그런데 사람 눈으로 전달 되는 빛은 공기의 상태에 따라 산란이 잃어 나고 물체가 있다면 이 물체의 특성에 따라 반사되는 효과 또한 달라집니다. 여기에 물체들이 많아지기 시작하면 더욱더 이것을 정확하게 계산하기란 어렵습니다.

컴퓨터 그래픽스의 조명 연구는 이러한 복잡한 실 세계의 빛에 대한 처리 과정을 빠르게 하기 위해 좀 더 단순화 하면서도 모니터로 바라보았을 때 실 세계와 크게 다르지 않는 장면을 만들 수 있는 방법을 연구해 왔습니다.

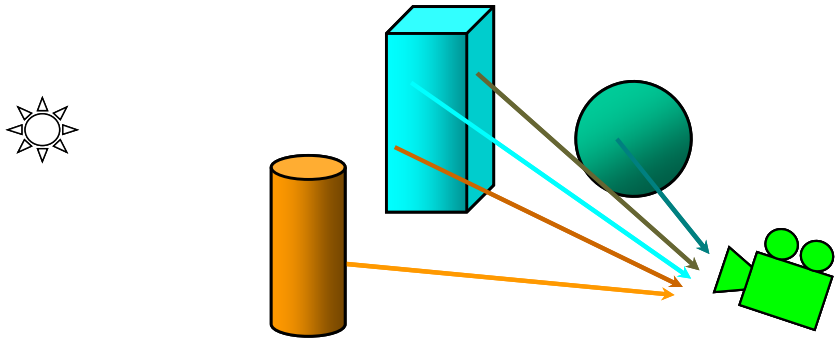


<실 세계 빛>

이것의 결과가 조명 모델입니다.

조명 모델은 다음 그림처럼 사람 눈으로 들어오는 복잡한 경로를 없애 버리고 최종 눈으로 들어오는 빛만 계산합니다. 이것은 물체들 사이의 상호 작용은 없고 물체가 스스로 빛을 발산하는 것처

럼 되어 버립니다.
또한 복잡한 전자기학 법칙은 버리고 경험적, 통계적인 내용을 도입해서 반사를 결정하는 것을 환경(Ambient) 광 효과, 분산(Diffuse) 광 효과, 정반사(Specular) 광 효과, 방사(Emissive) 광 효과, 4 종류로 압축했습니다.

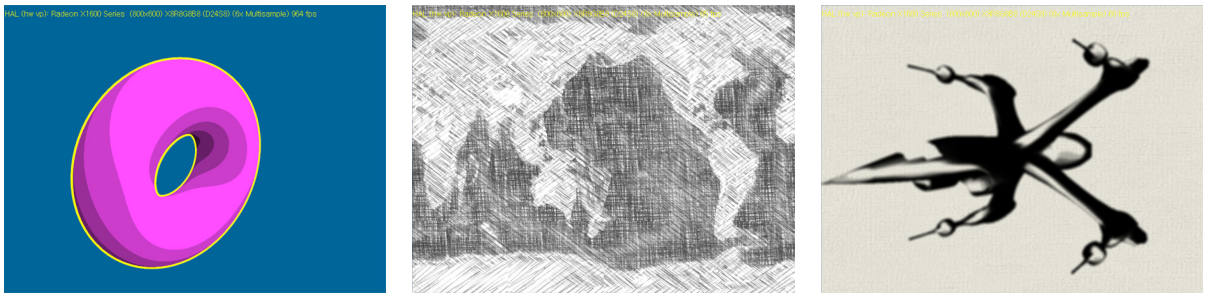

<조명 모델로 단순화된 세계>

이 4종류의 효과 또한 서로 상호작용하지 않고 단순한 덧셈으로 최종 색상을 결정하는 방법을 만들어서 컴퓨터에서 좀 더 빠른 실 세계와 비슷한 장면을 연출 할 수 있게 되었습니다.

간단히 4종류의 효과를 설명하면 환경 광 효과는 반사와는 상관 없이 물체의 전체 밝기에 대한 값으로 단순히 빛과 연산을 합니다.
분산 광 효과는 램버트(Lambert) 확산 모델을 적용한 것으로 램버트(Lambert) 확산은 빛에 대한 물체의 반사 밝기(Intensity)를 반사 면의 법선 벡터와 빛의 방향에 대한 내적으로 처리합니다.

정반사 광 효과는 하이라이트를 적용하기 위해 퐁 반사(퐁 쉐이딩)와 같은 모델을 사용합니다. 이 모델은 시선 벡터와 반사되는 빛의 방향 벡터를 내적(Dot Product)한 값에 적당한 승 수(Power: 멱)를 적용해서 반사의 밝기를 설정합니다. 한편 퐁 반사를 개량해서 하드웨어에서 빠르게 처리할 수 있도록 Blinn-Phong 반사 모델도 있습니다.
방사 광 효과는 빛과는 아무 상관 없이 물체 스스로 빛을 발산하는 효과입니다.

3D 게임에서는 이 램버트 확산을 폴리곤의 밝기로 표현하고 퐁 쉐이딩은 하이라이트 표현에 적용을 하며 이 둘은 렌더링 할 때 보통 혼합해서 처리합니다. 또한 실제 세계를 반영하지 않고 만화와 같은 비현실적인 세계를 표현하는 것을 비 실사 렌더링(NPR: Non-photorealistic Rendering)이라고 합니다.
카툰(Cartoon) 렌더링, 해칭(Hatching) 등은 게임에서 자주 사용되는 NPR인데 이 둘은 조명의 반사를 비 실사로 표현한 것입니다.



<NPR 예: 카툰, 해칭, 수묵화 표현>

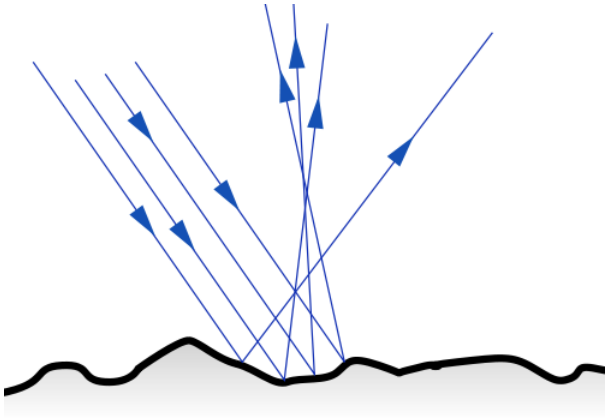
NPR의 예는 이후 쉐이더에서 살펴보고 여기서는 조명의 기초가 되는 분산 광 조명, 정반사 광 조명에 대해서 집중하겠습니다.

6.1 분산 조명(Diffuse Lighting) 효과

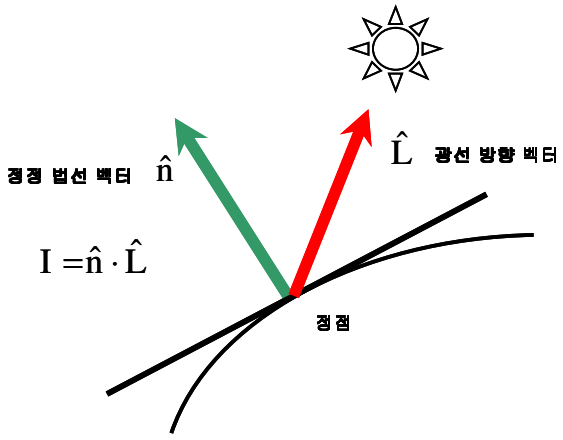
램버트 확산을 D3D는 분산 조명(또는 확산 조명, 난반사 광)으로 표현합니다. 램버트 확산은 다음 그림과 같이 물빛의 방향과 물체의 법선 벡터의 내적으로 밝기를 정합니다.

만약 실 세계의 반사를 정확하게 구현하려면 빛과 상호작용하는 모든 요소를 물리 법칙에 적용해야 합니다. 이렇게 하면 정확한 그림을 만들 수는 있겠지만 처리의 속도는 얼마나 걸릴지 예측하기 어렵고 시간 또한 많이 듭니다. 그런데 램버트 확산 법칙을 사용하면 정점의 개수와 빛의 개수만 고려하면 됩니다.

계산은 각각의 빛에 대해서 정점의 법선 벡터와 내적을 합니다. 이들을 전부 합하면 빛의 반사 세기가 됩니다. 만약 색상을 적용할 경우 빛의 색상과 오브젝트의 재질 색상을 곱합니다.



<실 세계의 반사>



< 램버트 확산: 분산 조명>

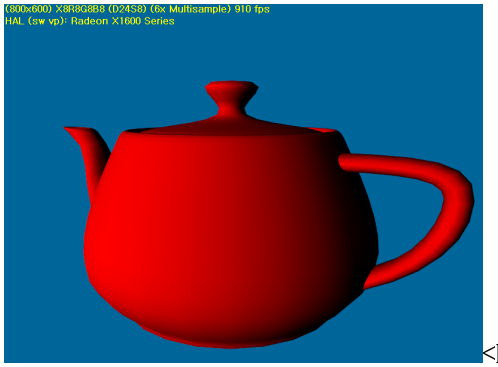
이 램버트 확산을 수식으로 표현하면 다음과 같습니다.

$$I = \sum_i \hat{n} \cdot \hat{L}_i \times (\text{Vertex Diffuse} \otimes L_i\text{'s Diffuse Color})$$

⊗은 다음과 같이 최종 성분은 각각의 성분끼리 곱한 결과를 말합니다.

$$A \otimes B = (A_1, A_2, A_3, A_4) \otimes (B_1, B_2, B_3, B_4) = (A_1 * B_1,\ A_2 * B_2,\ A_3 * B_3,\ A_4 * B_4)$$

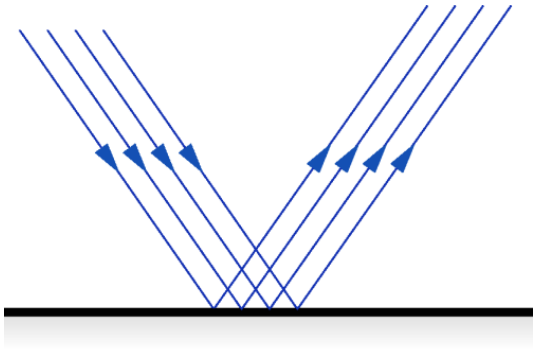
이 결과를 3D 모델에 적용하면 다음 그림과 같은 효과를 만들어 냅니다.



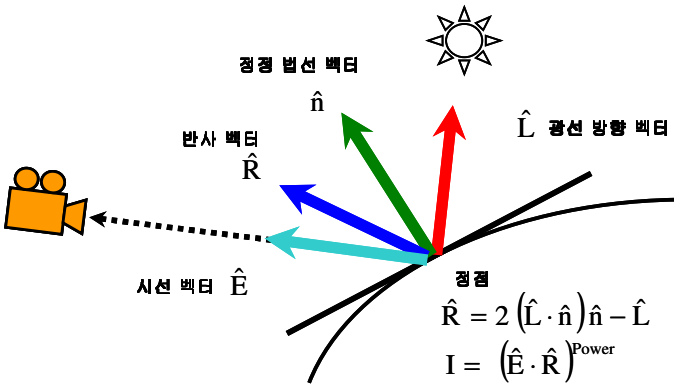
<Diffuse 조명 효과: m3d_05_lighting01_diffuse.zip>

6.2 정반사 조명(Specular Lighting) 효과

정반사 효과는 거울이나 바람이 없는 호수의 반사처럼 빛이 사람의 시선 한 방향으로 강하게 반사되는 효과 입니다. 램버트 확산은 카메라를 고려하지 않으므로 카메라의 움직임에 따른 정반사 효과를 만들어 낼 수 없습니다. 또한 내적을 사용하기 때문에 이의 결과인 cosΘ 함수의 연속으로 완만한 밝기 차이를 보여 강하게 반사하는 효과를 만들어 내지 못합니다. 퐁 반사 모델은 이런 문제들을 해결하고 또한 간결한 수식으로 처리 속도가 빠릅니다.
다음 그림은 실 세계의 정반사와 퐁 반사를 비교해 놓은 그림입니다.



<실 세계의 반사>



< 퐁 반사: 정반사 조명>

퐁 반사를 수식으로 표현하면 다음과 같습니다.

$$I = \sum_{i} \left(\hat{E} \cdot \hat{R}_i\right)^{Power} \times (Specular \otimes L_{Specular-i} Color)$$

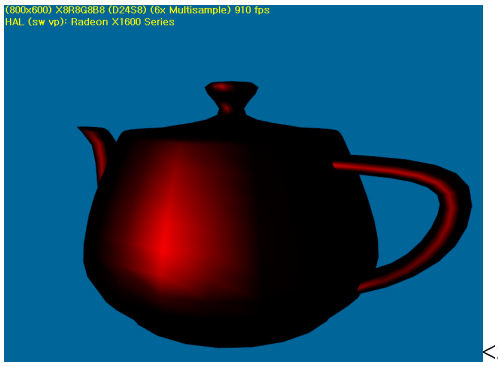
위의 수식을 보면 램버트 확산은 정점의 법선 벡터를 이용하지만 퐁 반사는 시선 벡터와 빛의 반사 벡터를 이용하며 또한 내적의 결과에 적당한 승 수(Power: 멱)를 적용해서 밝기를 결정하고 있습니다.

흥미로운 것은 시선 벡터와 반사 벡터 모두 단위벡터이면 이 둘의 내적은 cosΘ에 비례합니다. 그런데 cosΘ는 [-1, 1] 범위를 갖습니다. 따라서 cosΘ에 많은 승수를 할 수록 cosΘ가 1 근처의 값만 남고 나머지는 거의 0에 가까운 값이 됩니다.

한번 해볼까요? 0.5를 제곱하면 0.25 또 제곱하면 0.0625, 이를 또 제곱하면, 0.00390625가 됩니다. 색상은 255일 때 1.0이고 1이면 1/255,= 0.00392… 가 되어 0.5를 8승만 하면 1/255 보다 작은 값이 되어 결국 색상이 0으로 됩니다.

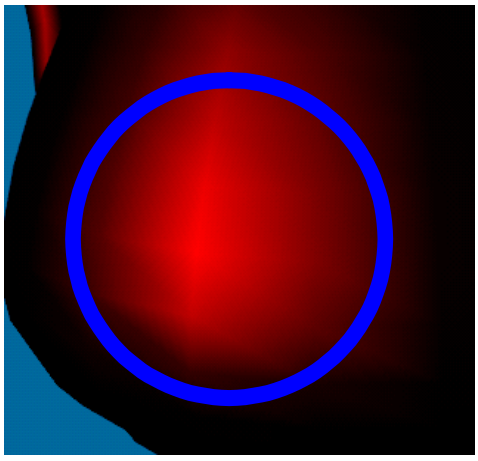
직관적으로 이를 정리하면 강하게 반사되는 빛의 영역을 좁게 만들고 싶을 경우는 승수(Power) 값을 크게 하고, 반대로 넓게 표현하려면 승수 값을 작게 하면 됩니다. 보통 승수를 10 정도에서 이보다 크게 하거나 줄여서 적당한 값을 선택합니다.

다음은 퐁 반사를 고정 기능 파이프라인에서 구현한 그림입니다.



<Specular 효과: m3d_05_lighting05_model_specular.zip>

앞서 쉐이딩 모델에서 D3D의 고정 파이프라인 퐁 반사를 반영한다고 만약 했습니다.



옆 그림은 위 그림의 일부 영역을 확대한 그림입니다. 자세히 보면 각진 형태의 색상을 볼 수 있습니다. 이것은 D3D의 조명 처리는 정점 처리 과정에서 진행 되기 때문입니다.
따라서 정점 단위로 처리하는 과정에서는 정점에만 퐁 반사를 반영하고 래스터 과정에서는 구로 쉐이딩을 적용해 선형적으로 픽셀을 보간하기 때문입니다.

전에는 이러한 문제들이 크게 부각되지는 않았습니다. 하지만 그래픽 카드의 성능이 좋아짐에 따라서 이 문제도 어느 정도 해결을 했습니다.

만약 쉐이더 버전 2.0 이상 지원되는 그래픽 카드가 있다면 퐁 반사를 정점 처리 과정이 아닌 픽셀 처리 과정에서 계산할 수 있습니다. 이것은 고정 기능 파이프라인으로는 불가능 하고 오직 픽셀 쉐이더를 통해서만 가능합니다. 이 방법은 이 책의 픽셀 쉐이더 부분에서 다시 설명하므로 여기서는 그냥 넘어 가겠습니다.

D3D는 분산 광 (Diffuse Lighting), 정반사 광(Specular Lighting) 효과 이외에 전체를 밝게 해주는 환경 광(Ambient Lighting) 효과와 스스로 빛을 내는 자체 발광 효과(Emissive)도 있습니다. Emissive 색상은 밝기에 그냥 더 해주며 이 요소는 조명에는 없고 재질에만 있습니다.
환경 광 효과는 안개가 자욱한 날을 많이 비유하는데 정점의 법선, 카메라의 시선에 상관없이 어느 방향으로든 밝기가 같은 효과를 만들어 냅니다. 보통 다음과 같은 수식으로 표현합니다.

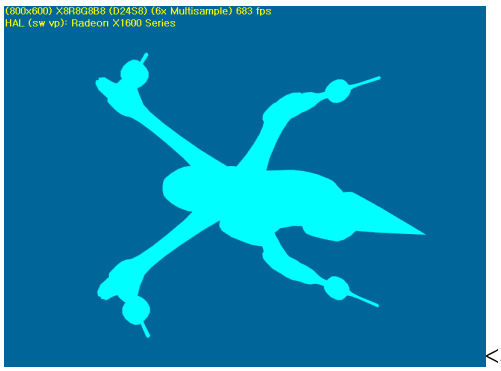
$$\text{Ambient Intensity} = C_a \otimes \left(G_a + \sum_i L_i * t_i \right)$$

여기서 G_a 는 재질(Material), 정점 Diffuse 색상, 정점 Specular 세 가지 중 하나가 되며 D3D는 재질을 기본으로 합니다. 만약 정점이 어떤 색상 정보도 없다면 자동으로 재질(Material)의 Ambient성분으로 대치 됩니다. 이것의 확인은 조명을 설명하기 위해 이전의 주전자 그림들과 다음의 X-Wing 그림들의 정점 구조체를 보면 다음과 같이 색상이 없고 위치와 법선만 있는 구조로 되어 있음을 볼 수 있습니다.

```
struct VtxN
{
        D3DXVECTOR3     p;      // 위치 벡터
        D3DXVECTOR3     n;      // 법선 벡터
        enum {FVF = (D3DFVF_XYZ|D3DFVF_NORMAL)};
};
```

G_a 성분은 Global Ambient Color 상태 머신에 설정된 값으로 다음과 같이 이 값을 변경할 수 있습니다.

```
pDevice->SetRenderState(D3DRS_AMBIENT, D3DXCOLOR(0,1,0,1)); // Green
```



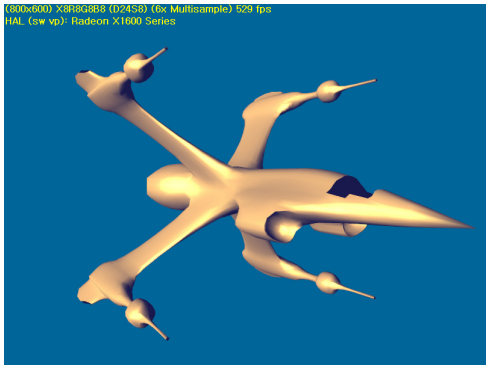
<환경 광 효과: m3d_05_lighting01_ambient.zip>

D3D는 이 4 가지 조명 효과를 더해서 다음과 같이 최종 반사의 밝기를 정합니다.

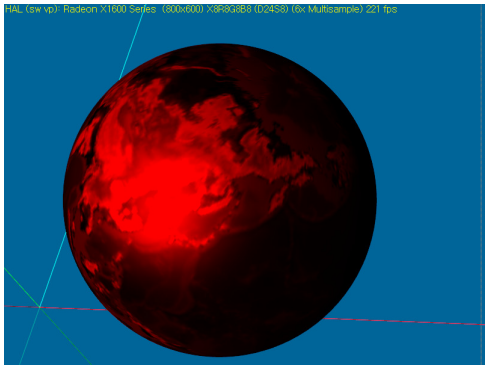
반사의 세기 = Ambient + Diffuse + Specular + Emissive

이것을 수식으로 표현하면 다음과 같습니다.

$$I = C_{Ambient} \otimes \left(G_{Ambient} + \sum_i L_{Ambient-i} * t_i \right) + \sum_i \hat{n} \cdot \hat{L}_i \times (VertexDiffuse \otimes L_{Diffuse-i})$$
$$+ \sum_i \left(\hat{E} \cdot \hat{R}_i \right)^{Power} \times (Specular \otimes L_{specular-i}) + Material_{Emissive}$$



<All Lighting Model - m3d_05_lighting01_model.zip>



<Diffuse+ Specular - m3d_05_lighting01_model2.zip>

6.3 조명(Lighting Source)의 종류와 재질(Material)

지금까지 D3D에 적용되는 조명 모델을 살펴보았습니다. 조명 모델을 컴퓨터에서 구현하기 위해서 필요한 요소는 조명(Lighting Source)과 재질(Material)입니다. 조명에 대한 소스(source), 즉 광원은 실 세계의 광원이 아닌 조명 모델에 구현하기 위한 광원으로 D3D는 광원을 점 광원(Spot Light), 방향 광원(Directional Light 또는 평행 광원), 그리고 점적 광원(Spot Light), 3 종류로 나누어서 파이프라인에 적용합니다.
D3D는 이 들 조명들을 구현하기 위해서 필요한 변수들을 D3DLIGHT9 구조체에 저장해서 사용합니다. D3DLIGHT9 구조체를 보면 다음과 같이 선언 되어 있습니다.

```
typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE    Type;           // 조명의 종류. Point, Directional, Spot Light
    D3DCOLORVALUE   Diffuse;        // 난반사 광. 반사 Object의 Diffuse와 연산
    D3DCOLORVALUE   Specular;       // 정반사 광. 반사 Object의 Specular와 연산
    D3DCOLORVALUE   Ambient;        // 주변 광. 주어진 Ambient 값과 연산
    D3DVECTOR       Position;       // 광원의 위치. 점 광원, 점적 광원에 필요
    D3DVECTOR       Direction;      // 빛의 방향. 평행광원에 필요
    float           Range;          // 빛의 도달 거리
    float           Falloff;        // 점적 광원의 감쇠 정도
    float           Attenuation0;   // 거리에 대한 감쇠.(1/Att) 점 광원, 점적 광원에 필요
    float           Attenuation1;   // 거리에 대한 감쇠. 1/(Att*거리)
    float           Attenuation2;   // 거리에 대한 감쇠. 1/(Att*거리 제곱)
    float           Theta;          // 점적 광원의 안쪽 원뿔 각(Radian)
    float           Phi;            // 점적 광원의 바깥 쪽 원뿔 각(Radian)
```

```
} D3DLIGHT9;
```

Type은 광원의 종류 지정하는 값입니다. 이 값은 점 광원은 D3DLIGHT_POINT, 방향 광원은 D3DLIGHT_ DIRECTIONAL, 점적 광원은 D3DLIGHT_SPOT로 합니다.
다음의 3종류의 변수 Diffuse, Specular, Ambient는 각간 분산 광, 정반사 광, 주변 광 효과를 지원하기 위한 값입니다. Position은 광원의 위치, Direction은 광원의 방향을 나타냅니다. Range는 광원 효과를 적용할 최대 거리로 광원과 정점의 거리가 이 값보다 크면 조명효과가 적용이 안됩니다. Falloff는 점적 광원에서 안쪽 원뿔의 감쇠율로 두 원뿔간의 세기 차이를 지정하는 값입니다. Attenuation0~2는 광원의 위치와 정점 사이의 거리에 따른 감쇠율로 방향 광원은 해당 하지 않습니다. Theta는 점적 광원에서 안쪽 원뿔의 각도이고, Phi는 바깥쪽 원뿔의 각도 입니다.

이 구조체를 이용해 조명을 설정하고 파이프라인에 연결하는 방법은 점 광원에서 다시 설명하겠습니다.

조명이 있다고 해서 무조건 반사가 되는 것이 아닙니다. D3D는 재질(Material)을 통해서만 그 반사 효과를 만들어 냅니다. 즉, 재질(Material)은 조명을 가지고 반사의 밝기를 정하는 역할을 하는 것이라 할 수 있습니다. 이 재질을 D3D는 파이프라인에서 이용할 수 있도록 D3DMATERIAL9 구조체에 다음과 같이 정의해 놓았습니다.

```
typedef struct _D3DMATERIAL9 {
    D3DCOLORVALUE   Diffuse;   // 조명의 Diffuse와 곱셈 연산
    D3DCOLORVALUE   Ambient;   // 상태 머신에 설정된 Ambient과 곱셈 연산
    D3DCOLORVALUE   Specular;  // 조명의 Specular와 곱셈 연산
    D3DCOLORVALUE   Emissive;  // 조명에 상관 없는 자체적인 발광. 그냥 더함
    float           Power;     // Specular 하이라이트 세기. 값이 크면 하이라이트는 좁아짐
} D3DMATERIAL9;
```

Diffuse, Ambient, Specular 값은 조명의 Diffuse, Ambient, Specular 값과 항상 $Mtrl\_diffuse \otimes Light\_diffuse$, $Mtrl\_ambient \otimes Light\_ambient$, $Mtrl_diffuse \otimes Light_diffuse$의 곱셈 연산을 합니다.
이 곱셈연산으로 인해서 조명과 재질에 색상이 있어도 전체가 검정색으로 출력 되는 일들이 많이 있습니다. 예를 들면 만약 조명의 Diffuse 값이 D3DXCOLOR(1, 0, 0, 1)인 붉은 색이라 하고, 재질의 Diffuse 값이 D3DXCOLOR(0, 1, 0, 1)인 초록색이라 합시다. 앞서 조명과 재질의 색상은 항상 곱셈 연산을 한다 했으므로 이 둘의 곱은 각 성분 별로 진행하면 다음과 같이 됩니다.
D3DXCOLOR(1 * 0, 0 * 1, 0 * 0, 1* 1) = D3DXCOLOR(0,0,0,1) 인 검정색이 됩니다.

또한 색상의 값이 [0, 255]가 아닌 [0.0, 1.0] 범위 값이라는 것을 명심해야 합니다. 그래서 늘

조명과 재질의 연산은 항상 원래 보다 어둡습니다.

마지막 Power는 퐁 쉐이딩에서 반사 벡터와 시선 벡터의 내적에 대한 승수 입니다. 이 값이 클수록 하이라이트는 좁아집니다.

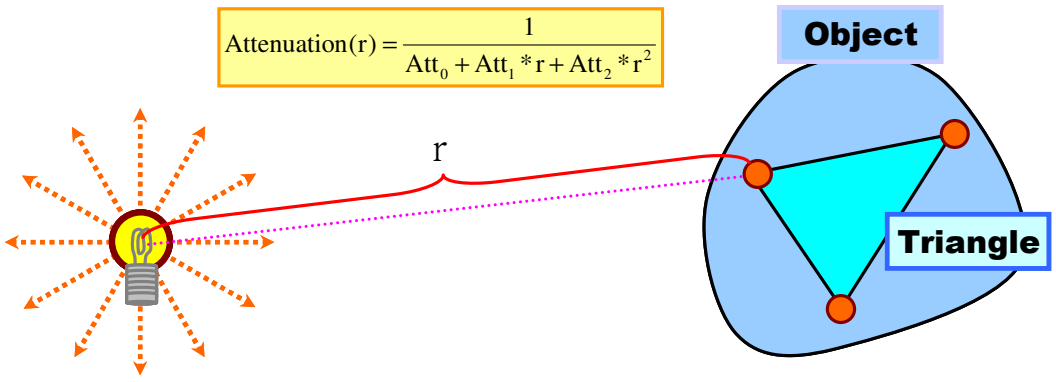
이 구조체를 설정하고 파이프라인에 연결해야 하는데 이 때 디바이스의 SetMaterial() 함수를 이용합니다. 다음 코드는 D3D의 고정 기능 파이프라인에 재질을 설정하는 예입니다.

```
D3DMATERIAL9 mtrl={0};
mtrl.Diffuse    = D3DXCOLOR(1,0,1,1);    // 초록색
mtrl.Ambient    = D3DXCOLOR(1,0,1,1);
mtrl.Specular   = D3DXCOLOR(1,0,1,1);
mtrl.Emissive   = D3DXCOLOR(1,0,1,1);
mtrl.Power      = 10.f;
pDevice->SetMaterial(&m_Mtrl);
```

간단히 조명과 재질에 대해서 알아 보았습니다. 다음으로 조명의 종류와 설정 방법을 설명하겠습니다.

6.3.1. 점 광원 (Point Light)

점 광원(sport light)은 다음 그림과 같이 커버 없는 백열 전구를 생각하면 됩니다. 점 광원 빛은 사방으로 고루 퍼지며 빛의 세기(Intensity)는 광원의 위치에서 정점까지의 거리 r의 $\frac{1}{r^2}$ 로 줄어듭니다. 이것을 Attenuation 이라 합니다. Attenuation Attenuation0~2 세 요소가 있습니다. 다음 그림은 점 광원과 Attenuation을 나타낸 그림입니다.



<점 광원과 오브젝트>

점 광원은 Range 가 있어서 정점과 광원의 위치에 대한 거리 r이 Range보다 크면 조명을 반영하지 않습니다. 이 부분을 주의해야 하며 또한 Attenuation0~1값도 적절하게 설정해야 합니다. 만약 이 값 전부가 0이면 빛의 세기가 줄어들지 않고 Range에서 갑자기 0이 되어 조명 효과가 제대로 적용되지 않은 장면을 만들 수도 있게 됩니다.

다음 예는 D3D에서 점 광원을 정의하고 D3D 파이프라인에 연결하는 코드 입니다.

```
D3DLIGHT9       Light;
D3DXCOLOR       Color(1,1,1,1);
memset(&Light, 0, sizeof(D3DLIGHT9));
Light.Type      = D3DLIGHT_POINT;
Light.Ambient   = Color * 0.3f;
Light.Diffuse   = Color;
Light.Specular  = Color*0.2f;
Light.Position  = D3DXVECTOR3 (0.0f, 0.0f, 0.0f);
Light.Range        = 2000.0f; Light. Falloff      = 1.0f;
Light.Attenuation0 = 0.5f; Light.Attenuation1 = 0.1f; Light.Attenuation2 = 0.1f;

// 라이팅 설정
m_pDev->SetLight(0, &Light);
m_pDev->LightEnable(0, TRUE);
```

D3D는 조명을 총 8개까지 연결할 수 있습니다. 파이프라인에 조명을 연결하도록 하는 함수가 디바이스의 SetLight() 함수 입니다. 이 함수의 첫 번째 인수에 조명 인덱스를 넣고, 그 다음에 적용할 조명의 주소를 인수로 전달합니다.
D3D는 또한 각 조명을 각각 On/Off가 가능 합니다. 조명의 On/Off는 LightingEnable() 함수를 통해서 하고 이 함수의 첫 번째 인수에 조명인덱스를, 두 번째 인덱스는 TRUE/FALSE를 넣어서 활성화/비활성화를 합니다.

개별적으로 각각의 조명을 설정했다면 상태 머신에서 전체 조명을 활성화하기 위해서 다음과 같이 상태 함수를 호출해야 합니다.

```
m_pDev->SetRenderState(D3DRS_LIGHTING, TRUE);
```

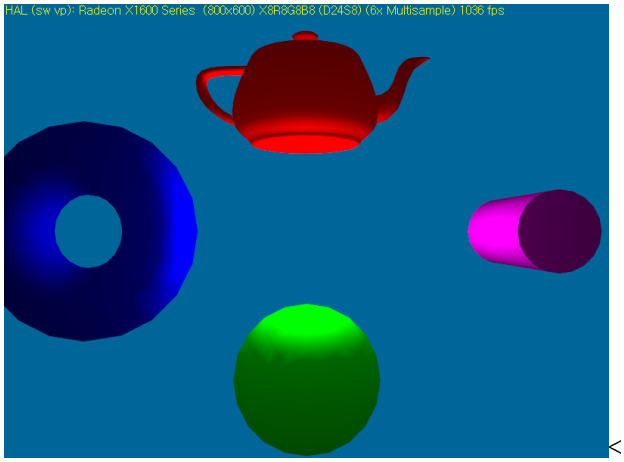
주의해야 할 것은 이렇게 조명을 활성화 했어도 각각의 조명이 비활성화(Off) 되어 있으면 조명이 적용 안됩니다.

만약 렌더링 오브젝트의 법선이 크기가 1인 단위 벡터로 정규화 되지 않았다면 다음과 같이 파이프라인에서 정규화할 수 있도록 활성화 해야 합니다.

```
m_pDev->SetRenderState(D3DRS_NORMALIZENORMALS, TRUE);
```

이것은 파이프라인에서 정규화를 하기 때문에 속도에 영향을 줍니다. 따라서 조명이 적용될 렌더링 오브젝트의 정점 법선은 미리 정규화 되어 있으면 좋습니다. D3D는 NORMALIZENORMALS 값을 디폴트로 FALSE로 되어 있습니다.

다음은 렌더링 오브젝트에 반영된 점 광원 효과를 보이는 그림입니다.



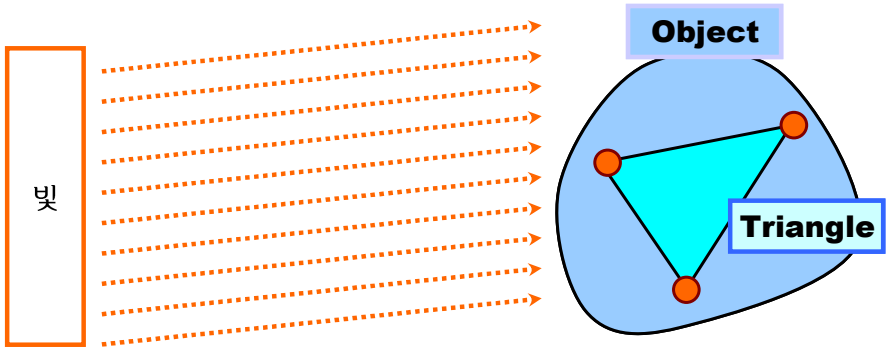
<점 광원: m3d_05_lighting02_point.zip>

이 그림에 적용된 점 광원의 위치는 (0, 0, 0)에 있습니다. 그림을 자세히 보면 중심에서 멀어질수록 점점 어두워 지고 있음을 볼 수 있습니다.

6.3.2. 방향 광원(Directional Light)

방향 광원(directional light)은 가장 일반적인 광원입니다. 만약 빛이 태양처럼 아주 멀리서 오면 평행하게 도달 하는데 이것이 방향 광원입니다.
게임에서 실내 지형의 경우 조명 효과를 극대화 하기 위해 평행 광원은 어둡게 하고 점 광원 등을 주로 사용할 때가 있지만 실외 지형의 조명은 전체적으로 이 방향 광원을 사용해서 조명을 조절합니다.
실외 지형을 적용한 게임들은 보통 태양의 위치에서 오는 주 광원 하나와 좌, 우, 위, 아래, 뒤에서 각각의 다른 색으로 약하게 보조 광원을 약하게 주 과3~4개 정도의 평행 광원을 많이 사용합니다.
방향 광원을 그림으로 나타내면 다음과 같습니다.
이 점 광원과는 다르게 광원의 위치와 거리가 필요 없고, Ambient, Diffuse, Specular 와 빛의 방

향 벡터만 있으면 구현 됩니다.

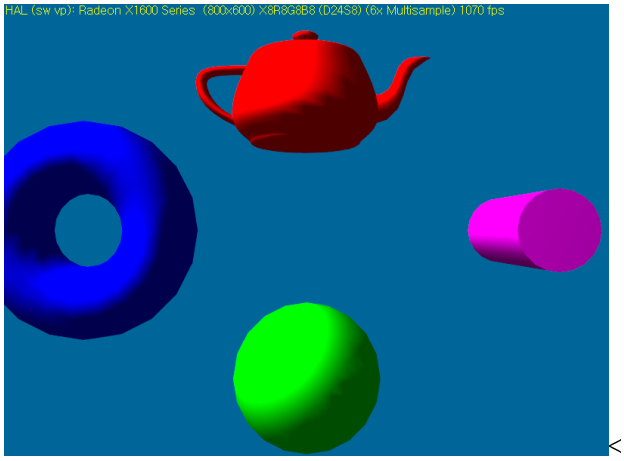


<방향 광원>

다음 코드는 방향 광원을 설정하는 예입니다.

```
D3DLIGHT9       Light;
D3DXCOLOR       Color(1,1,1,1);
D3DXVECTOR3     vcDir( 1.0f, -1.0f, 0.25f); D3DXVec3Normalize(&vcDir, &vcDir);
memset(&Light, 0, sizeof(D3DLIGHT9));
Light.Type      = D3DLIGHT_DIRECTIONAL;
Light.Ambient   = Color * 0.2f; Light.Diffuse   = Color;
Light.Specular  = Color * 0.7f; Light.Direction = vcDir;
```

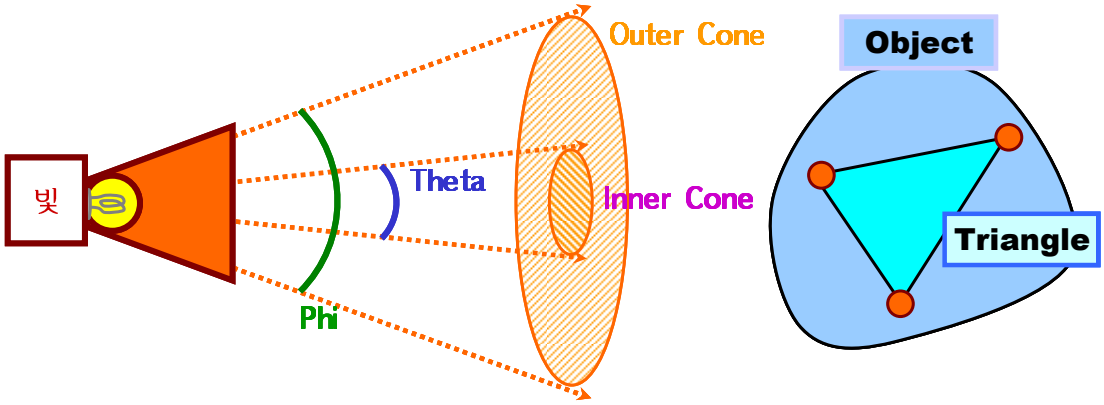
코드를 보면 방향 벡터를 D3DXVec3Normalize() 함수로 단위벡터로 만들고 있습니다. 고정 파이프라인에서는 꼭 단위벡터로 만들지 않아도 됩니다만 이후에 쉐이더를 고려한다면 습관적으로 방향 벡터는 단위 벡터로 만들어 놓는 것이 좋습니다. 방향광원의 특징은 위의 코드와 같이 광원의 종류와 색상 이외에 방향만 설정되어 있습니다. 이 방향광원에 대한 예제가 m3d_05_lighting02_directional.zip 이며 다음은 이 예제의 실행모습입니다.



<방향 광원: m3d_05_lighting02_directional.zip>

6.3.3. 점적 광원(sport light)

점적 광원은 방향 광원과 점 광원의 합성입니다. 따라서 이 광원은 광원의 위치, 반영될 최대 거리 값, 밝게 비춰 주는 안쪽에 대한 각도, 약간 덜 밝은 바깥 쪽 조명에 대한 각도가 필요합니다.

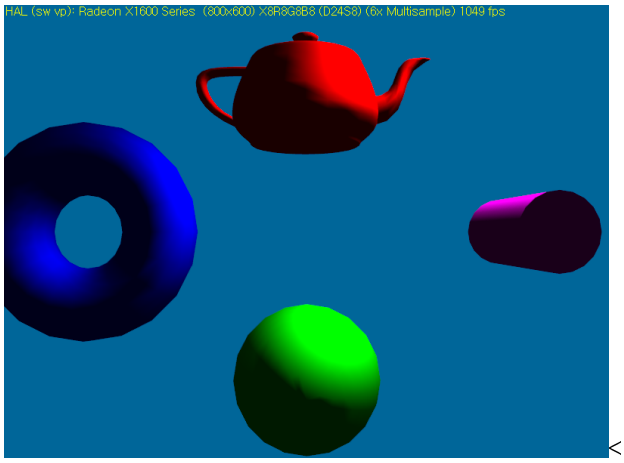


<점적 광원>

점적 광원 설정은 다음과 같이 합니다.

```
D3DLIGHT9       Light;
D3DXCOLOR       Color(1,1,1,1);
D3DXVECTOR3     vcDir(-0.5f,-1, 0); D3DXVec3Normalize(&vcDir, &vcDir);  // 빛의 방향

memset(&Light, 0, sizeof(D3DLIGHT9));
Light.Type      = D3DLIGHT_SPOT;
Light.Ambient   = Color * 0.0f;
Light.Diffuse   = Color;
Light.Specular  = Color * 0.7f;
Light.Position  = D3DXVECTOR3(5,7,0);
Light.Direction = vcDir;
Light.Range        = 5000.0f; Light.Falloff      = 1.0f;
Light.Attenuation0 = 0.1f;    Light.Attenuation1 = 0.1f; Light.Attenuation2 = 0.0f;
Light.Theta        = 0.5f;    Light.Phi          = 1.2f;
```

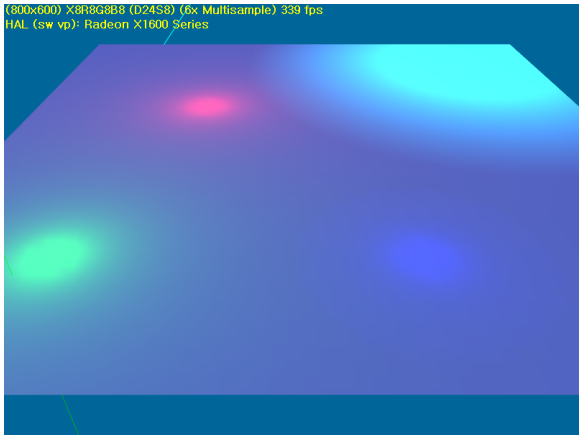
m3d_05_lighting02_spot.zip을 컴파일하고 실행하면 다음과 같이 화면이 출력 됩니다.



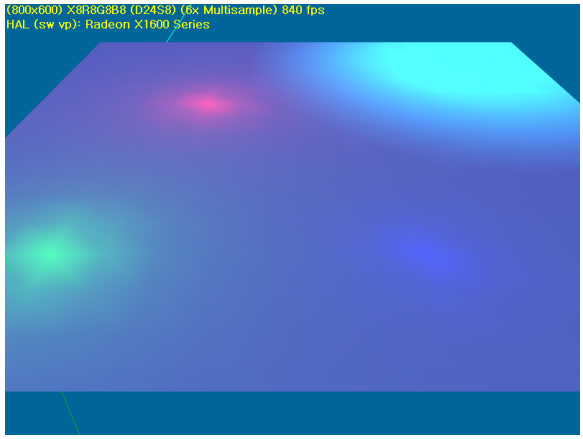
<점점 광원: m3d_05_lighting02_spot.zip>

점적 광원은 방향 광원과 점 광원의 합성입니다. 따라서 이 광원은 광원의 위치, 반영될 최대 거리 값, 밝게 비춰 주는 안쪽에 대한 각도, 약간 덜 밝은 바깥 쪽 조명에 대한 각도가 필요합니다.

앞의 3그림을 보면 방향 광원은 어느 정도 구분이 가지만 점 광원과 점적 광원은 구분이 잘 안 됩니다. 이 두 광원의 차이를 보려면 평면에 조명을 비춰보는 것이 제일 좋습니다.



<여러 조명들 64x64>



<여러 조명들 16x16>

이 그림은 m3d_05_lighting03_all.zip 예제로 점 광원 3개 점적 광원 1개, 방향 광원 1개를 평면에 출력하는 예제입니다. 앞서 고정 파이프라인에서 조명은 정점에 적용이 된다고 했습니다. 그림에서 보면 정점이 많은 왼쪽 그림은 조명을 잘 반영하고 있는 반면에 오른 쪽 그림은 약간 어색합니다.

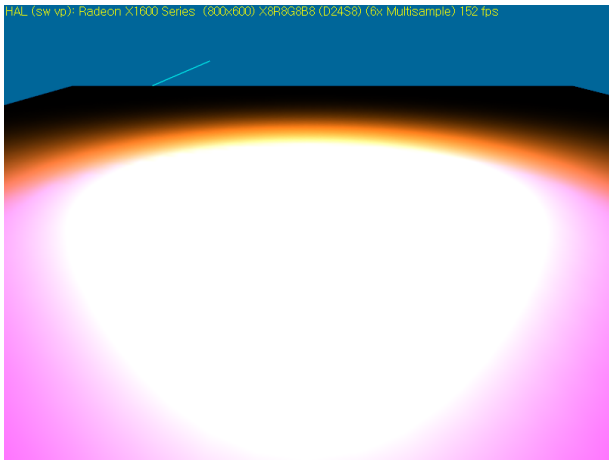
만약 여러분이 게임에서 조명효과를 잘 반영하려면 충분한 양의 폴리곤이 필요합니다. 이것이 어려운 경우에는 텍스처를 조명 대신해서 매핑 해야 합니다. 이것을 라이팅 맵(Lighting)이라 하는데 라이팅 맵은 폴리곤이 적어 조명 효과를 내기 어려울 때 텍스처에 조명을 미리 그려 넣고 폴리곤에 다시 매핑 해서 출력하는 방법입니다. 이 내용은 텍스처와 쉐이딩에서 다시 하겠습니다.

6.3.3. 조명, 재질, 색상 관계

지금까지 사용한 렌더링 오브젝트는 색상이 없고 위치와 법선 벡터만 있는 정점이었습니다. 그런데 만약 정점에 색상이 있다면 어떻게 될까요? 이런 경우 D3D는 재질의 색상을 이용하지 않고 정점의 색상을 이용합니다.

예를 들어 정점 구조체를 다음과 같이 설정한 후에 렌더링을 하면 다음 그림과 같은 모습을 만들 수 있습니다.

```
struct VtxND
{
        D3DXVECTOR3     p;              // Position
        D3DXVECTOR3     n;              // Normal
        DWORD           d;              // Diffuse
        enum { FVF = (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_DIFFUSE)};
};
```



<Diffuse가 있는 정점의 라이팅-m3d_05_lighting03_d.zip>

즉 다음과 같이 재질을 설정해도 정점에 색상이 있다면 전혀 먹히지 않는 다는 것입니다.

```
D3DMATERIAL9 mtrl={0};
mtrl.Diffuse = D3DXCOLOR(5,1,5,1);
…
m_pDev->SetMaterial( &mtrl );
```

이 때 중요한 것은 정점의 FVF 설정 값입니다. 위의 예제는 D3D의 파이프라인이 FVF 값을 해석하

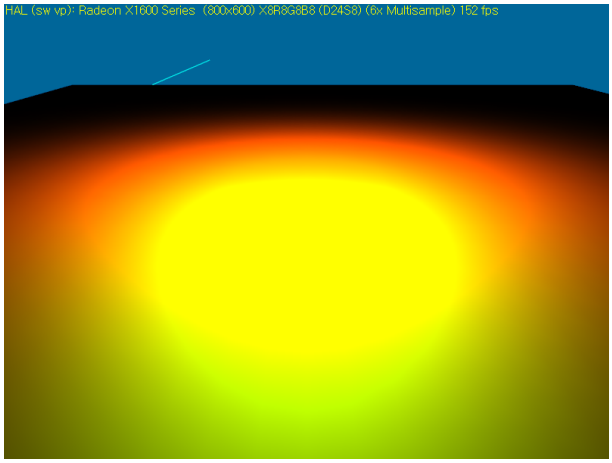
고 나서 D3DFVF_DIFFUSE 값이 있음을 알고 재질의 Diffuse 값이 아닌 정점의 Diffuse값을 가지고 계산을 합니다.

재질에는 Diffuse 있었고, 또한 정점에도 Diffuse를 사용할 수 있었습니다. 이번에는 재질에 있는 Specular 값도 정점에 적용할 수 있을까요? 예 그렇습니다. 지난 번 FVF를 배울 때 정점에도 Specular를 사용할 수 있습니다. 마찬가지로 정점에 Specular를 설정하면 D3D는 재질의 Specular 값이 아닌 정점의 Specular 값과 라이팅의 Specular 값을 가지고 계산을 합니다. 이 때 반사 벡터와 내적에 승수 해야 하는 Power는 재질의 Power 값을 그대로 사용합니다.

정점에 Specular를 사용하기 위해서 구조체를 다음과 같이 만듭니다.

```
struct VtxNS
{
        D3DXVECTOR3     p;
        D3DXVECTOR3     n;
        DWORD           s;
        enum { FVF = (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_SPECULAR)};
};
```

자료 구조는 이전의 색상을 사용할 때와 같습니다. 차이는 FVF에 D3DFVF_SPECULAR로 정하고 있다는 것입니다.
이렇게 정점에 Specular 변수를 정하고 렌더링을 하면 다음과 같은 조명 효과를 만들어 낼 수 있습니다.



<Specular가 있는 정점의 라이팅-m3d_05_lighting03_s.zip>

Diffuse 때와 마찬가지로 재질의 Specular 값을 설정해도 이 것은 적용이 안되며 오직 재질의

Power 값만 조명 효과에 적용이 됩니다.

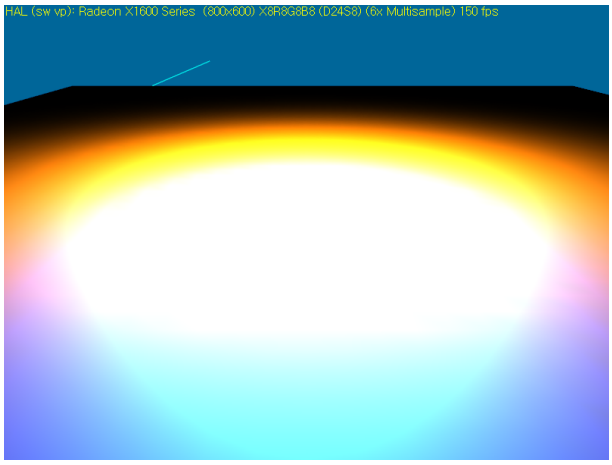
```
D3DMATERIAL9 mtrl={0};
mtrl.Power      = 10.f;

m_pDev->SetMaterial( &mtrl );
```

마지막으로 정리 하는 차원에서 정점에 Diffuse와 Specular 모두 갖고 있을 때 조명을 적용해 봅시다. 먼저 정점 구조체를 다음과 같이 만들어야 하겠습니다.

```
struct VtxNDS
{
        D3DXVECTOR3     p;
        D3DXVECTOR3     n;
        DWORD           d;              // Diffuse
        DWORD           s;              // Specuar
...
        enum { FVF = (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_DIFFUSE|D3DFVF_SPECULAR)};
};
```

Diffuse, Specular를 동시에 적용할 경우 Diffuse가 먼저 나와야 합니다.
재질 설정은 더 단순해졌습니다.

```
D3DMATERIAL9 mtrl={0};  mtrl.Power = 10.f;
m_pDev->SetMaterial( &mtrl );
```



<Diffuse, Specular가 있는 정점의 라이팅-m3d_05_lighting03_ds.zip>

이렇게 정점에 Diffuse와 Specular가 있으면 D3D는 디폴트로 재질의 Diffuse, Specular가 아닌 정점의 Diffuse, Specular와 연산하는 것을 확인해봤습니다.
앞서 색상은 float 형으로 하면 [0, 1.0] 범위를 갖는 다고 했습니다. 그런데 조명은 D3DXVECTOR 구조체와 같은 float 형 4개를 사용합니다. 만약 여러분이 이전에 보인 예제처럼 강한 조명을 사용하고 싶다면 주저 말고 1 이상의 값을 사용해 보세요. m3d_05_lighting03_ds.zip의 McScene.cpp 파일의 void CMcScene::Render() 함수를 보면 1보다 큰 값을 사용해서 좀 더 밝은 조명을 다음과 같이 설정하고 있음을 볼 수 있습니다.

```
memset( &m_light, 0, sizeof(m_light) );
m_light.Type = D3DLIGHT_SPOT;

m_light.Diffuse          = D3DXCOLOR( 3, 1.5, 0, 0);
m_light.Specular         = D3DXCOLOR( 2, 4, 16, 1);
```

지금까지 색상, 조명, 그리고 조명과 색상의 관계를 살펴 보았습니다. 조명과 색상은 경험이 많을수록 색감이 좋아집니다. 다음에 나오는 과제 중에 조명을 연습할 툴 만들기가 있는데 이것을 지금 당장 어렵더라도 나중에 시간을 들여서 꼭 만들어 보고 그것으로 게임에 적용할 조명을 연습해 봐야 합니다. 단원을 마치는 차원에서 프로그램 절차를 정리해 보겠습니다.
조명을 프로그램 하는 절차를 정리하면 다음과 같습니다.

조명 프로그래밍 절차
1. 라이팅 구조체 설정: D3DLIGHT9
2. 라이팅 설정: pDevice->SetLight(nIndex, &Light). nIndex: 0~7(총 8개까지 라이팅 설정 가능)
3. 라이팅 활성화: pDevice->LightEnable(nIndex, TRUE);
4. 렌더링 가상 머신의 라이팅 적용: pDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
5. 재질 설정: D3DMATERIAL9
6. 재질 연결: pDevice->SetMaterial(&d3dmaterial);

주의사항
1. 조명을 활성화 시켜놓고 조명 적용이 안되면 검은 색으로 표현됨.
2. 디바이스의 Clear 색상을 검정색으로 하면 조명이 적용 또는 문제를 파악하기 어려움. 이를 위해 배경화면은 검은색과 흰색은 피함
3. 조명이 적용되려면 정점에 법선 벡터가 꼭 필요
FVF = … D3DFVF_NORMAL…

4. 법선 벡터가 정규화 되지 않으면 정규화 과정을 거치도록 디바이스에 설정

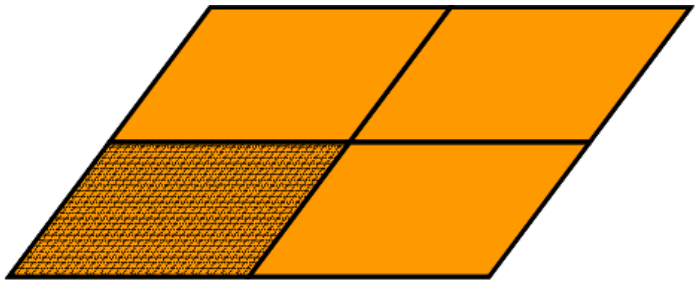
pDevice->SetRenderState(D3DRS_NORMALIZENORMALS, TRUE);
보통 안 하는 것이 속도 면에서 이득

5. 빛과 반응하는 재질 설정도 반드시 필요
pDevice->SetMaterial();

6. 전체 화면이 어둡지 않도록 약간의 밝기가 유지되도록 Ambient 값을 설정
pDevice->SetRenderState(D3DRS_AMBIENT, 0x000F0F0F);

실습과제) 8x8, 16x16, 32x32, 64x64 격자로 구성된 Block을 정점들을 UP를 이용해서 Index 방식으로 화면에 출력하시오.
Block의 Cell의 Width = 32; 높이 설정은 임의로 합니다.

실습과제) 위의 32x32 Block을 동적으로 4개를 생성해서 이들을 연속적인 지형으로 출력해 보시오.



실습과제) API 함수를 이용해서 다음과 같이 조명을 연습할 수 있는 Tool을 제작하시오.