

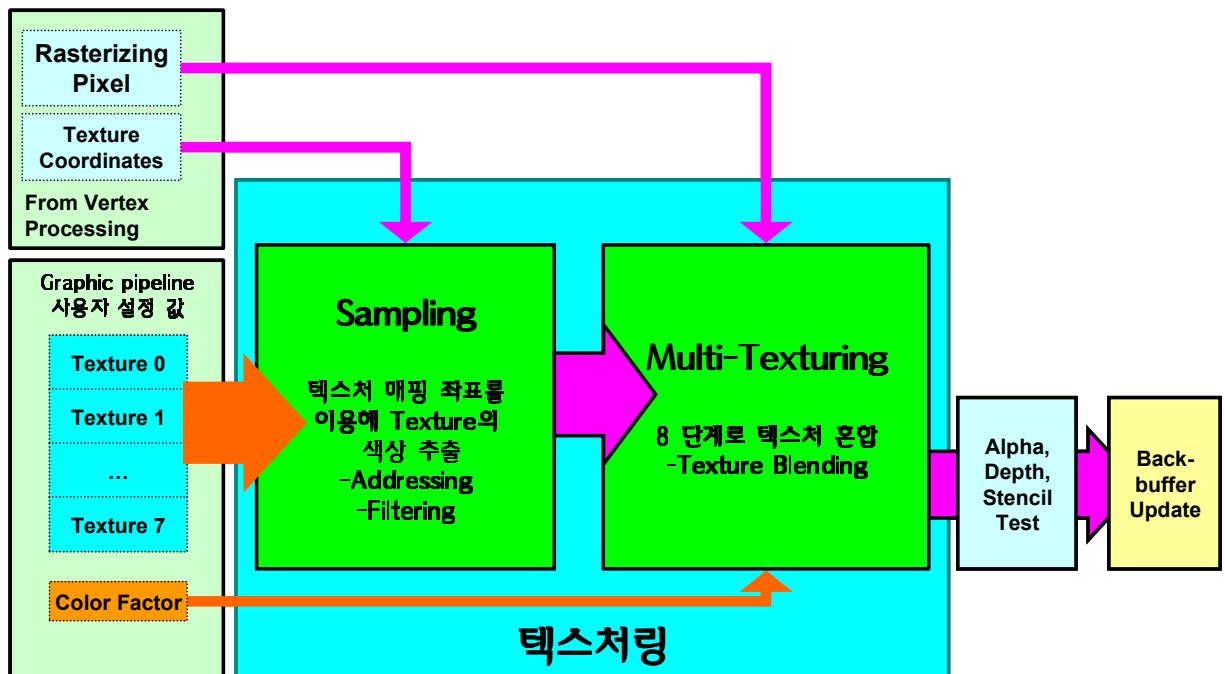
3D Game Programming Basic with Direct3D

7. Texturing

원론적으로 말한다면 폴리곤(Polygon) 만으로도 가상 세계를 전부 표현할 수 있습니다. 하지만 현실 세계의 사물을 좀 더 세밀하게 묘사 하려면 그만큼의 폴리곤이 필요하게 됩니다. 따라서 한정적인 컴퓨터로서는 어느 정도 이상 흉내 내는 것이 어려워집니다. 이런 문제를 해결하는 방법 중의 하나는 폴리곤의 구성인 삼각형에 이미지 데이터를 적용하는 것입니다. 이 방법은 이미지만 필요하므로 더 이상의 폴리곤을 만들지 않게 되어 실용적입니다.

삼각형에 적용하려는 이미지를 텍스처(Texture)라 합니다. 또한 텍스처를 삼각형에 붙일 때 "매핑(Mapping) 한다"라고 합니다. 따라서 텍스처 처리(Texturing)는 이미지 텍스처를 삼각형에 매핑하는 방법이라 할 수 있겠습니다.

D3D는 이 텍스처 처리가 파이프라인의 한 과정으로 구성되어 있습니다. 3D 기초 시간에 D3D는 크게 정점 처리과정과 픽셀 처리 과정으로 구분한다고 했습니다. 텍스처 처리는 픽셀 처리 과정의 한 부분으로 다음 그림과 같이 정점 처리 과정의 레스터 과정에서 만들어진 픽셀과 사용자가 파이프라인에 정한 텍스처들을 가지고 그림과 같이 작업을 합니다.



<파이프라인에서 텍스처 처리 위치>

그림을 보면 텍스처 처리의 첫 단계는 텍스처에서 텍스처 매핑 좌표를 이용해 색상을 추출하는 샘

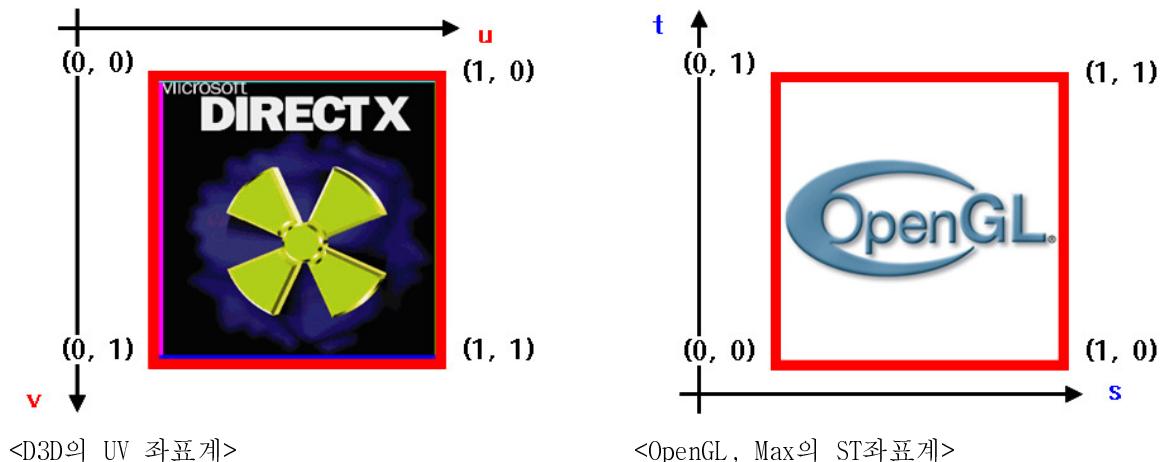
플링(Sampling) 과정을 진행 합니다. 샘플링 과정은 두 가지를 고려해야 합니다. 하나는 텍스처 매핑 좌표가 $[0,1]$ 범위 밖의 픽셀을 처리하는 어드레싱(Addressing) 방법, 다른 하나는 2차원 화면으로 전환된 삼각형의 왜곡에 따른 적절한 색상을 값을 선택하는 필터링(Filtering)입니다. 샘플링 다음 단계로 이 샘플링에서 만들어진 색상과 레스터 과정에서 만들어진 색상을 혼합(Blending)을 하는 단계가 있습니다. D3D는 최대 8단계까지 혼합 단계가 있습니다. 이 과정을 거치면 비로소 완전한 색상을 만들어냅니다.

간단히 Texturing에 대해서 알아보았습니다. 구체적인 내용을 좀 더 알아보겠습니다.

7.1 Sampling

샘플링은 어드레싱(Addressing)과 필터링 2가지가 있다고 했습니다. 먼저 어드레싱에 대해서 보겠습니다.

7.1.1. 텍스처 매핑 좌표계



텍스처 매핑에 대한 좌표계는 크게 ST-좌표계와 UV-좌표계 두 가지가 있습니다. 이 좌표계의 구분은 투영 평면을 기준으로 오른쪽 그림과 같이 이미지를 화면에 출력할 때 왼쪽 맨 아래를 $(0, 0)$, 오른쪽 맨 위를 $(1, 1)$ 로 하는 ST 좌표계와 왼쪽 그림과 같이 왼쪽 맨 위를 $(0, 0)$, 오른쪽 맨 아래를 $(1, 1)$ 하는 UV-좌표계가 있습니다.

OpenGL, MAX는 오른손 좌표계를 사용합니다. 또한 벡터와 행렬 연산 등 모든 처리 과정은 수학적인 모델을 그대로 적용하고 있습니다. 이에 따라 이미지에 대한 매핑도 오른손 법칙에 따른 ST 좌표계를 이용합니다.

D3D는 왼손 좌표계를 사용합니다. 내부의 모든 처리과정 또한 왼손 좌표계를 따라 진행이 되어 매핑 좌표도 왼손 좌표계로 구성되어 있는 UV-좌표계를 사용합니다. ST-좌표계와 UV-좌표계는 나름대로 장단점이 있습니다. ST-좌표계는 수학적인 처리가 간단한 반면 화면에 맞추기가 어려운 반면

UV-좌표계는 화면 출력에는 유리한 반면에 매핑에서 상하가 바뀌는지 신경을 써야 합니다. (저 개인적으로는 3D ST 좌표계가 편하다고 생각합니다.)

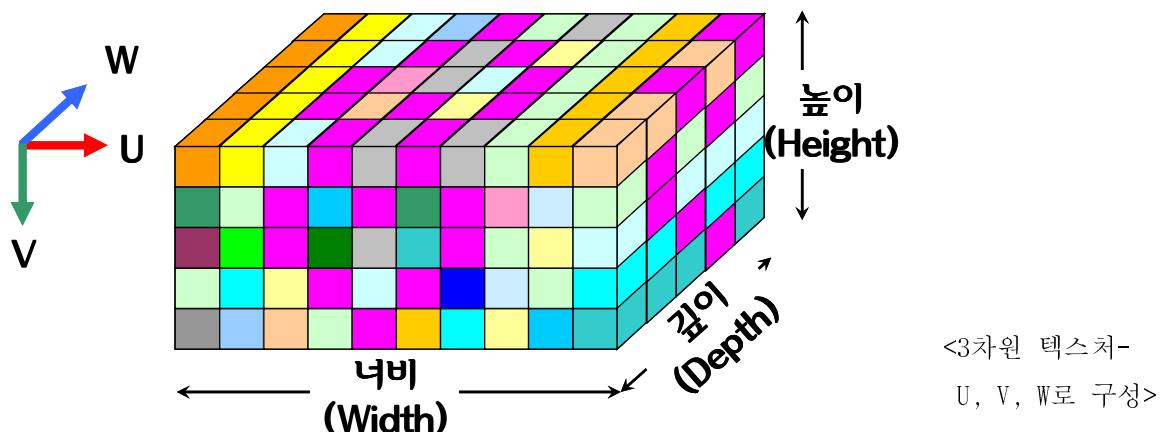
ST, UV 좌표 모두 매핑을 위한 좌표계입니다. 텍스처의 매핑 좌표의 범위는 $[0, 1]$ 로 정규화되어 있습니다. 이 정규화 값을 가지고 ST와 UV좌표계를 교환하면 다음과 같이 V는 $1-T$ 를 해야 합니다.

$$U = S, V = 1.0 - T$$

이 공식은 간단하지만 자주 이용되는 공식입니다. 특히 3D MAX 와 같은 그래픽 툴에서 렌더링 오브젝트의 매핑 좌표를 추출할 때 V값을 위와 같은 식으로 처리해야 툴에서 작업한 것과 D3D에서 실행한 결과가 동일하게 출력이 됩니다.

D3D는 2차원 텍스처뿐만 아니라 다음 그림처럼 3차원 텍스처도 지원이 됩니다. 좌표는 W를 하나 더 추가해서 U, V, W를 사용합니다. 3차원 텍스처는 마치 Adobe 사의 Photoshop의 Layer와 비슷한 모양으로 같은 크기의 2차원 픽셀을 층층이 쌓아 올린 형태입니다.

만약 3차원 텍스처를 만들고자 할 때는 dds 파일 포맷을 사용해야 합니다. 3차원 dds 파일은 SDK의 "DxTex.exe" 프로그램으로 간단히 만들 수 있습니다. 이 프로그램은 Samples 폴더에 소스 코드가 제공되니까 공부에도 많은 도움이 됩니다.



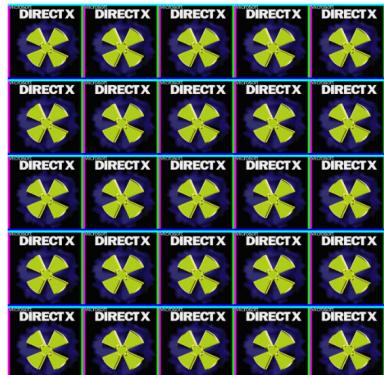
7.1.2. 어드레싱(Addressing)

앞서 텍스처의 좌표는 $[0, 1]$ 의 범위로 정규화 한다라고 했습니다. 그런데 사용자가 $[0, 1]$ 범위 밖의 텍스처 좌표를 설정하면 어떻게 될까요? 어드레싱은 정규화 되지 않은 텍스처 좌표에 대한 처리 방식입니다.

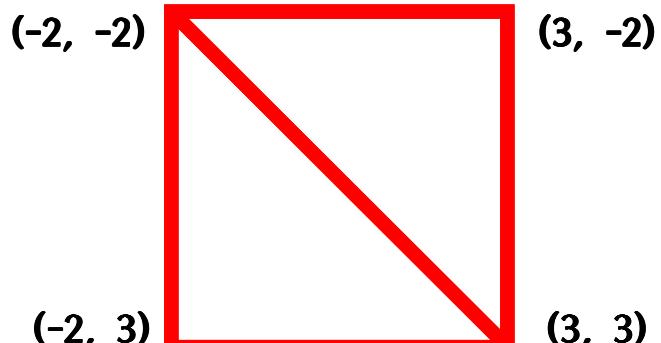
D3D는 어드레싱에 대한 지원으로 Address Mode를 "Wrap", "Mirror", "Clamp", "Border" 4종류를 선택해서 지정할 수 있습니다.

-Wrap Mode

D3D의 Default Address Mode로 다음과 같이 [0,1]밖의 텍스처 좌표는 [0,1]안의 텍스처 좌표처럼 다시 복사하는 방식입니다.

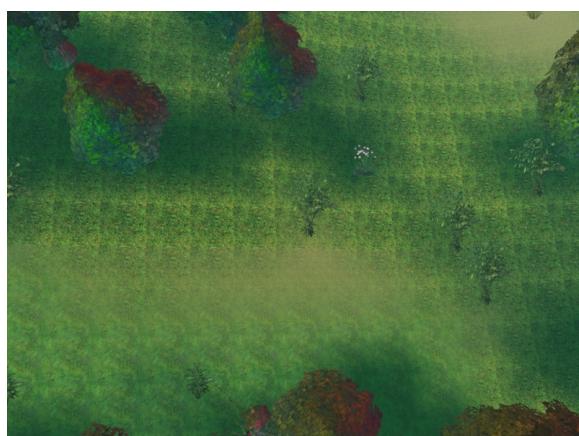


<Wrap Mode>



<매핑 좌표>

이 방식은 게임에서 실외지형을 구성에서 배경 지형 표면을 구성할 때 많이 사용합니다.



<Wrap 방식이 적용된 게임 지형>

이것은 지형이 커지면 거기에 사용되는 이미지 또한 커져야 합니다. 이미지의 크기는 제곱에 비례하므로 지형을 과도하게 늘리기란 쉬운 문제가 아닐 수 없습니다.

Wrap 방식은 이런 문제를 아주 쉽게 해결합니다.

Wrap으로 지형을 만들면 카메라가 어느 정도 거리에 있으면 배경 지형의 바둑판 무늬가 나타납니다. 이것이 게임에 지장을 주지 않도록 게임 프로그래머는 툴(Tool) 프로그램을 사용해서 조절할 수 있도록 합니다.

디바이스의 Wrap 모드의 지정은 디바이스의 SetSamplerState() 함수를 통해서 다음과 같이 합니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP);
```

SetSamplerState() 함수는 텍스처 샘플링을 담당하는 샘플러에게 작업의 형태에 대한 명령을 내리는 것과 같습니다. 함수 첫 번째 인수는 Sampler State Index입니다. 프리미티브 시간에 텍스처를 적용할 때 디바이스의 SetTexture() 함수를 사용한 적이 있습니다. 이 함수의 첫 번째 인수는 바

로 이 샘플러 인텍스입니다. 즉 위의 코드는 샘플링 할 때 Address Mode를 Wrap으로 하겠다는 것입니다.

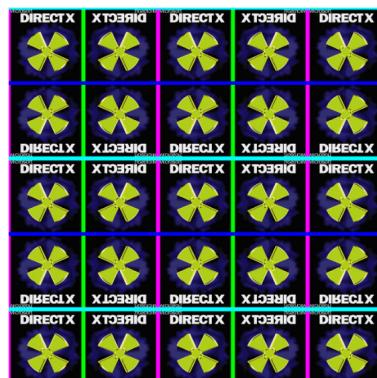
D3D는 고정 함수 그래픽 파이프라인에 총 8개의 텍스처를 설정할 수 있으므로 샘플러 또한 8개가 있다고 할 수 있겠습니다.

D3D를 처음 배울 때 알아야 할 내용이 많고, 파이프라인에 설정할 것이 많아서 헷갈리는 경우가 많으면 다음과 같이 각 샘플러에 대한 텍스처와 상태 값 설정을 몰아서 작성하는 것이 좋을 수도 있습니다.

```
m_pDev->SetTexture(0, &pTex);  
m_pDev->SetSamplerState(0, ...ADDRESS..., ...);  
m_pDev->SetSamplerState(0, ...Filter..., ...);
```

-Mirror Mode

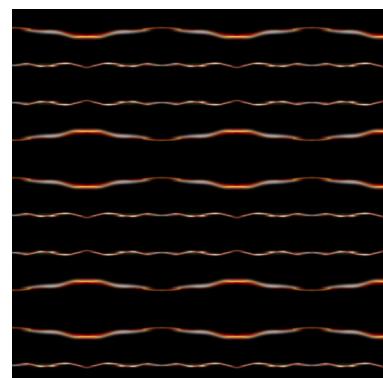
미러 모드(Mirror Mode)는 앞의 텍스처 좌표 매핑에 대해서 다음 그림과 같이 마치 거울을 복사한 것처럼 화면에 출력하는 모드입니다.



<Address Mirror Mode>



<대칭이 없는 이미지를 Mirror로 표현>



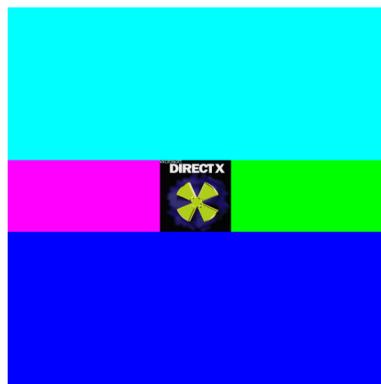
이 모드는 역시 지형에서 자주 사용이 됩니다. 또한 좌우 대칭을 표현 할 수 있어서 좌우가 연속되지 않은 이미지를 연속 모양으로 표현하고 싶을 때 사용합니다. 가장 많이 볼 수 있는 경우는 폭발, 전기 충격 등의 게임 이펙트(Game Effect)입니다.

미러 모드는 `D3DTADDRESS_MIRROR` 값을 이용해서 다음과 같이 설정합니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_MIRROR);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_MIRROR);
```

-Clamp Mode

클램프 모드(Clamp Mode)는 [0, 1] 범위 밖의 매핑 좌표는 가장 가까운 위치인 0 아니면 1에 해당하는 좌표에서 색상을 가져옵니다.



<Clamp Mode>

이 클램프 모드는 3D를 이용해서 2D게임용 스프라이트를 만들 때 이용됩니다. Wrap이나 Mirror 는 float 형 오차로 인해 경계에서 원치 않은 색상이 샘플링 될 수 있는데 이 Clamp Mode로 설정하면 가장 가까운 픽셀을 선택해서 채우게 되므로 안전합니다.

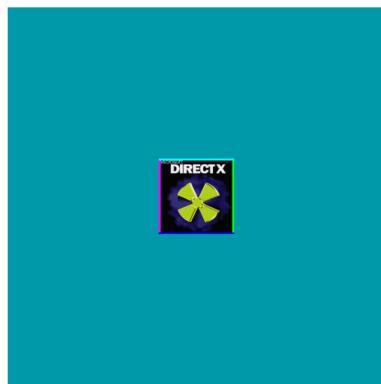
3D에서는 투영 그림자 기법에서 그림자 텍스처를 이 Clamp Mode로 설정해서 영역 밖은 그림자를 적용하지 않도록 하기도 합니다.

클램프 모드를 디바이스에 설정하는 방법은 다음과 같습니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);
```

-Border Mode

경계 모드(Border Mode)는 [0, 1] 범위 밖의 색상은 사용자가 지정한 색상으로 채웁니다.



<Border Mode- 색상 0x000099AA>

사용자는 색상을 지정하기 위해서 샘플러 상태 값 D3DSAMP_BORDERCOLOR 을 선택하고 색상을 다음과 같이 지정해서 사용합니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_BORDERCOLOR, 0x0000099AA);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_BORDER);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_BORDER);
```

-Mode 혼합

어드레싱을 설명하기 위해 U, V 좌표에 대해서 같은 모드를 적용했으나 D3D는 각 좌표에 대해서 서로 다른 모드를 연결해 사용할 수 있습니다. 예를 들어 U, V, W로 구성된 텍스처에 U는 Wrap을 V는 Clamp를 W는 Mirror Mode를 적용하고 싶다면 다음과 같이 합니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);  
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSW, D3DTADDRESS_MIRROR);
```

Address Mode에 대한 테스트는 [m3d_06_tex01_address.zip](#)의 INT CMcScene::FrameMove() 함수 부분을 참고 하기 바랍니다.

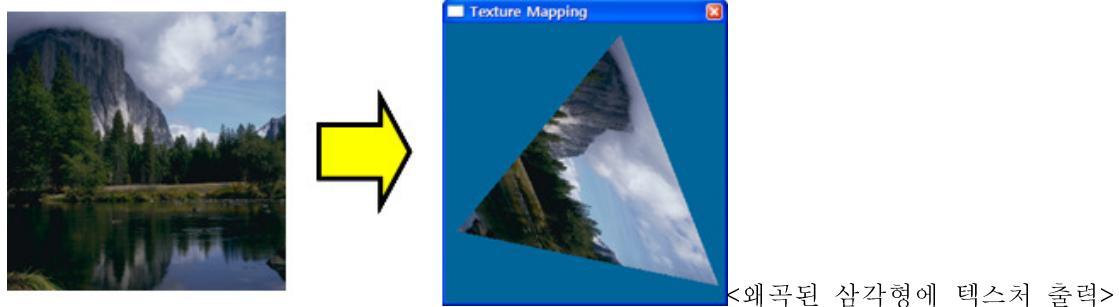
7.2 Filtering

다음 그림과 같이 텍스처의 픽셀을 화면에 1:1로 출력을 하기 위해서 특별히 더 작업을 할만한 일은 없습니다.



<화면에 텍스처의 픽셀을 1:1로 출력>

그런데 3D 렌더링 오브젝트에 매핑 된 텍스처는 정점의 위치 사이의 간격, 카메라의 움직임, 오브젝트의 움직임에 따라 화면에 많은 픽셀을 출력하거나 아니면 적게 출력 될 수 있습니다.

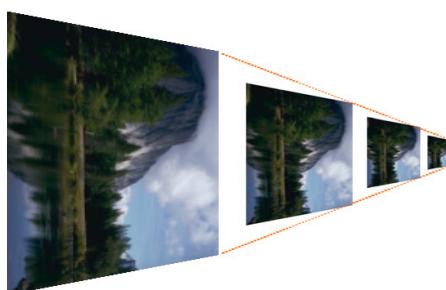


이렇게 기하 변환에 의해 발생하는 텍스처의 왜곡을 보정하기 위해서 사용되는 것이 필터링(Filtering)입니다. 이미지가 화면에서 확대되면 적은 수의 텍스처 픽셀로 화면의 픽셀을 채워야 합니다. D3D는 샘플러에게 확대(Magnification) 됐을 때 샘플러 상태 값 D3DSAMP_MAGFILTER으로 확대에 대한 필터링을 적용할 수 있습니다. 반대로 화면에서 축소되면 많은 수의 텍스처 픽셀 중에서 적절한 픽셀을 선택해야 합니다. 축소(Minification)에 대해서 샘플러 상태 값 D3DSAMP_MINFILTER으로 축소된 경우에 대한 필터링을 지정합니다.

-Mipmapping

필터링은 Anti-Aliasing 방법 중의 하나입니다. 그런데 원본 이미지에서 무조건 텍스처의 픽셀을 선택하게 되면 Anti-Aliasing 적용이 잘 되지 않는 경우가 많습니다. 예를 들어 카메라에서 멀리 떨어진 객체는 낮은 해상도의 텍스처를 적용하고, 가까이에 있는 객체는 높은 해상도의 텍스처를 적용하면 적절한 Anti-Aliasing 이 가능합니다.

이를 위해서 원본 텍스처를 레벨에 따라 각각 작은 단위로 가지고 있다가 렌더링에서 이 레벨을 선택하고 거기에 맞는 텍스처를 선택하게 하게 합니다. 이 작은 단위의 텍스처를 하위 텍스처(Sub-Texture)라 합니다. 하위 텍스처를 만드는 방법은 2의 승수로 결정합니다. 원본을 Level 0으로 하고 크기가 256x256 Level 1인 하위 텍스처는 2^{-1} 인 128x128 텍스처를 원본 텍스처의 이미지를 가지고 만듭니다. 레벨 2의 하위 텍스처는 다시 64x64로, 크기라 1이 될 때까지 다음 그림과 같이 반복을 합니다. 이렇게 구성한 텍스처를 Mip-map이라 합니다.



<Mip-map Pyramid>



<Mip-map의 구성>

Mip은 라틴어 Multum In Parvo("작은 공간에 많이"→ "작지만 많다")의 첫 글자입니다.

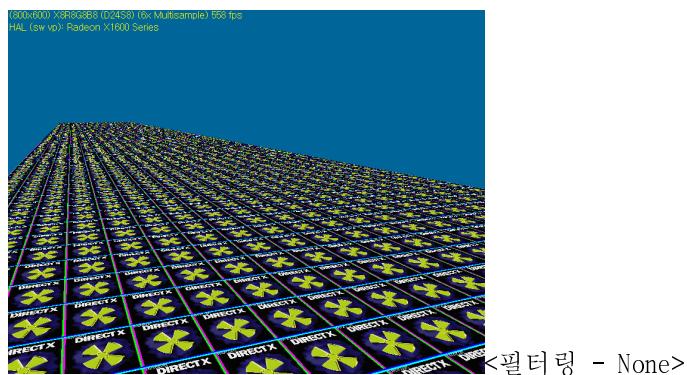
샘플러에 대한 확대, 축소, Mip 필터링 설정은 다음과 같이 합니다.

```
m_pDev->SetSamplerState(샘플러 인덱스, D3DSAMP_MAGFILTER, 필터링방법);  
m_pDev->SetSamplerState(샘플러 인덱스, D3DSAMP_MINFILTER, 필터링방법);  
m_pDev->SetSamplerState(샘플러 인덱스, D3DSAMP_MIPFILTER, 필터링방법);
```

D3D의 샘플링은 확대(Magnification), 축소(Minification), 그리고 mipmapping 적용에 대해서 필터링을 하지 않는 NONE, 근접점(Nearest Point), 쌍선형 (Bilinear Interpolation), 비등방형(AF: Anisotropic) 필터링 4 종류를 지원합니다.

- None

단어 그래도 필터링을 안 합니다. 2D 게임은 그래픽 리소스가 화면에 그대로 출력하는 경우가 많으므로 3D로 2D 게임을 제작할 때 보통 필터링을 하지 않을 때가 많습니다.

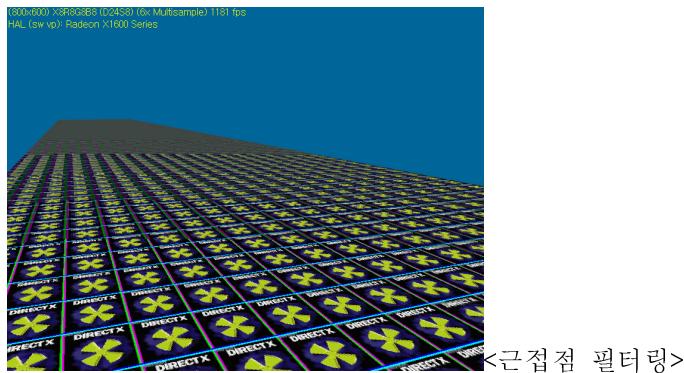


NONE 필터링을 확대, 축소, MIP에 다음과 같이 적용 합니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_NONE);  
m_pDev->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_NONE);  
m_pDev->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_NONE);
```

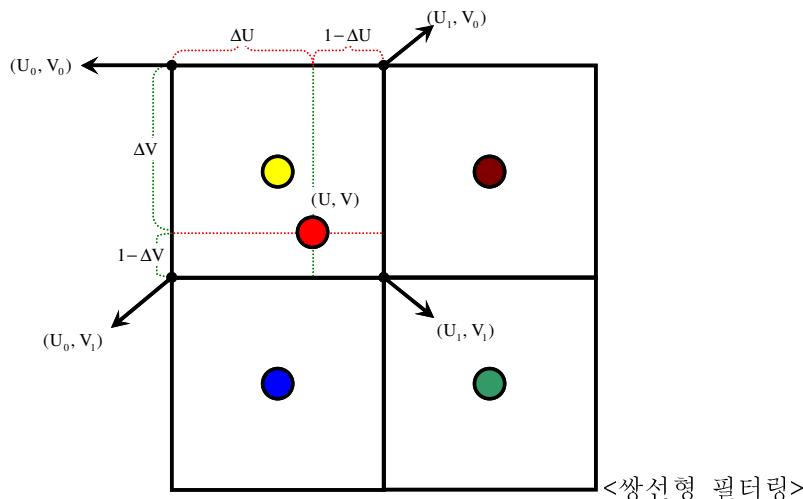
- 근접점 필터링

이 필터링은 해당 UV좌표에서 가장 가까운 픽셀을 샘플링 합니다. 따라서 구현하기 쉽고 가장 빠르게 처리됩니다.



-쌍선형 필터링(Bilinear Interpolation Filtering)

이 필터링은 해당 UV좌표에서 가장 가까운 네 픽셀을 선택하고, 각각의 픽셀에 매핑 좌표에서의 거리에 따른 가중치를 적용해서 픽셀을 결정합니다.



계산 방법은 위의 그림처럼 $[0,1]$ 사이의 텍스처 좌표를 해당 텍스처의 너비와 폭으로 만든 새로운 좌표 (U, V) 로 만듭니다. 이 때 가장 가까운 좌표 (U_0, V_0) , (U_1, V_0) , (U_0, V_1) , (U_1, V_1) 을 선택합니다.

가중치를 구해야 하는데 (U, V) 는 (U_0, V_0) 로 시작하는 픽셀 안에 있으므로 가장 큰 가중치를 적용합니다. 이 때 적용하는 가중치는 $w_{00}(U_0, V_0) = (1-\Delta U)(1-\Delta V)$ 합니다.

갈색 점에 대한 가중치는 $w_{10}(U_1, V_0) = (1-\Delta U)(\Delta V)$ 로 정합니다. 이와 비슷하게 파란 점의 가중치와 초록 점의 가중치는 각각 $w_{01}(U_0, V_1) = (\Delta U)(1-\Delta V)$, $w_{11}(U_1, V_1) = (1-\Delta U)(1-\Delta V)$ 로 정합니다.

최종 픽셀은 인접한 각각의 4개 픽셀에 이 가중치들을 곱해서 최종 색상을 결정합니다.

$$\text{Pixel} = w_{00} * \text{Pixel}(U_0, V_0) + w_{10} * \text{Pixel}(U_1, V_0) + w_{01} * \text{Pixel}(U_0, V_1) + w_{11} * \text{Pixel}(U_1, V_1)$$

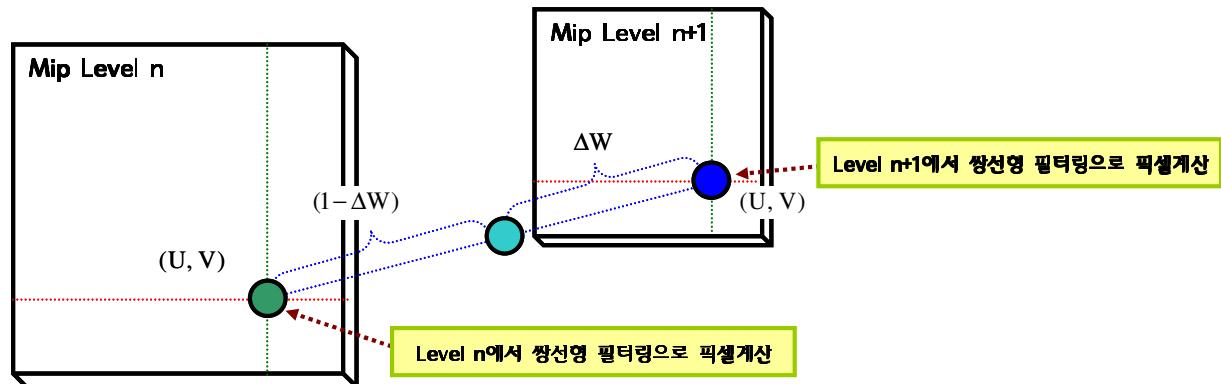
이 식은 선형(Linear)이고 U, V에 대한 연산을 동시에 진행하기 때문에 Bilinear Interpolation이라 합니다.

이 쌍선형 필터링은 렌더링 속도와 품질 면에서 우수에서 대부분 3D의 기본 필터링으로 많이 지정해서 사용합니다. 쌍선형 필터링을 지정하는 방법은 다음과 같습니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
m_pDev->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
m_pDev->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_NONE);
```

-삼중 필터링(Trilinear Filtering)

삼중 필터링은 쌍선형 필터링의 연장입니다. 다음 그림처럼 먼저 해당 레벨에 맞는 색상에 대해서 쌍선형 필터링으로 픽셀을 만듭니다. 그 다음으로 이 레벨보다 낮은 하위 텍스처에서 다시 쌍선형 필터링으로 픽셀을 만듭니다. 이 두 결과로 만든 픽셀을 선형 보간으로 최종 색상을 만듭니다.

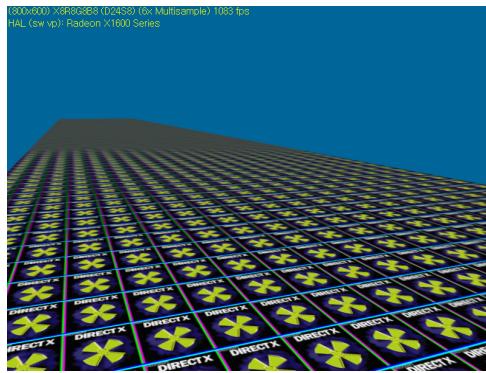


$$<\text{3중 필터링: } \text{Pixel} = (1 - \Delta w) * \text{Pixel}_{\text{level}-n} + \Delta w * \text{Pixel}_{\text{level}-n+1}>$$

이 삼중 필터링은 쌍선형 필터링만큼 빠릅니다. D3D는 mip맵 텍스처가 있으면 다음과 같이 지정하면 됩니다.

```
m_pDev->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
m_pDev->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
m_pDev->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
```

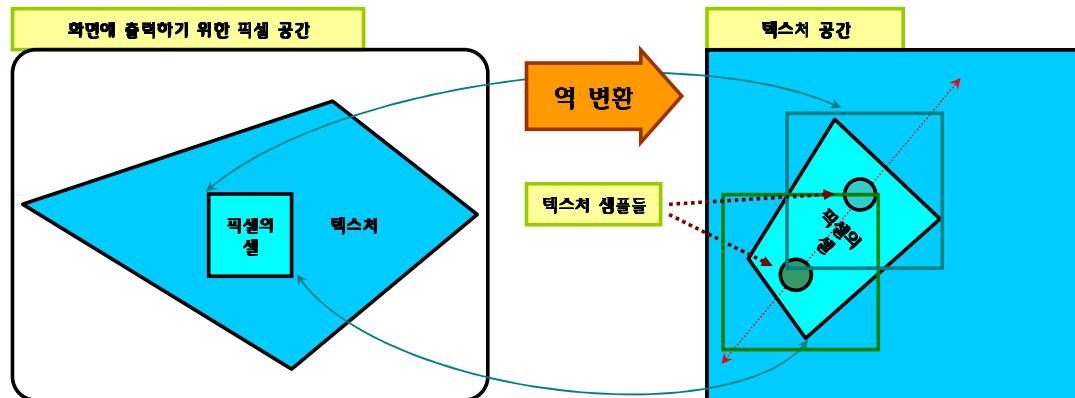
삼중 필터링이 적용된 모습은 다음과 같습니다.



-비등방(이방성) 필터링(AF: Anisotropic Filtering)

앞의 선형 필터링이나 근접점 필터링은 텍스처의 기울어진 형태를 고려하지 않고 만든 필터링입니다. 이전 그림들을 보면 해상도가 높은 영역에서는 비교적 잘 샘플링 되지만 카메라에서 면 거리에 있는 텍스처에 대한 샘플링은 그리 품질이 좋지 않습니다.

비등방 필터링은 다음 그림과 같이 화면으로 출력하기 위한 픽셀 공간에서의 픽셀 셀(Cell)을 반대로 텍스처 공간으로 역투영해서 계산하는 것입니다.



<비등방 필터링 구현 방법>

이렇게 역 변환하면 긴 사각형이 우측의 그림처럼 만들어 집니다. 이 가운데 중심 축을 비등방 (Line of Anisotropy)라 합니다. 이 축을 따라서 각각 샘플링을 하고 이 샘플링 점들을 모아서 최종 색상을 결정합니다.

앞의 그림은 2개의 픽셀을 샘플링 했지만 만약 역 변환의 결과 더 많은 픽셀을 샘플링 할 때도 있습니다. 그런데 하드웨어에서 비등방 필터링은 최근에 와서 실현된 것이 많아 자신이 사용하고 있는 그래픽 카드에서 이를 얼마나 지원되는 지 확인을 해야 합니다.

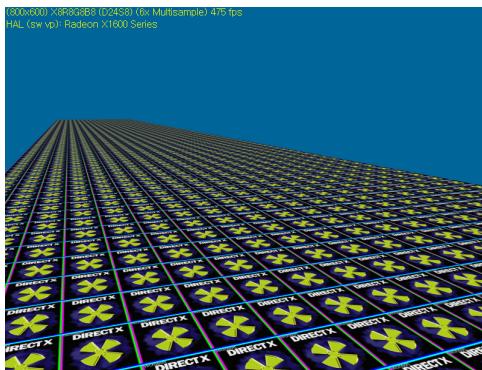
다음은 코드는 최대 비등방 값을 찾고 이것을 그래픽 파이프라인에 설정하는 코드입니다.

```

D3DCAPS9           caps;
m_pDev->GetDeviceCaps(&caps);
m_pDev->SetSamplerState(0, D3DSAMP_MAXANISOTROPY, caps.MaxAnisotropy);

m_pDev->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_ANISOTROPIC);
m_pDev->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_ANISOTROPIC);
m_pDev->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_ANISOTROPIC);

```



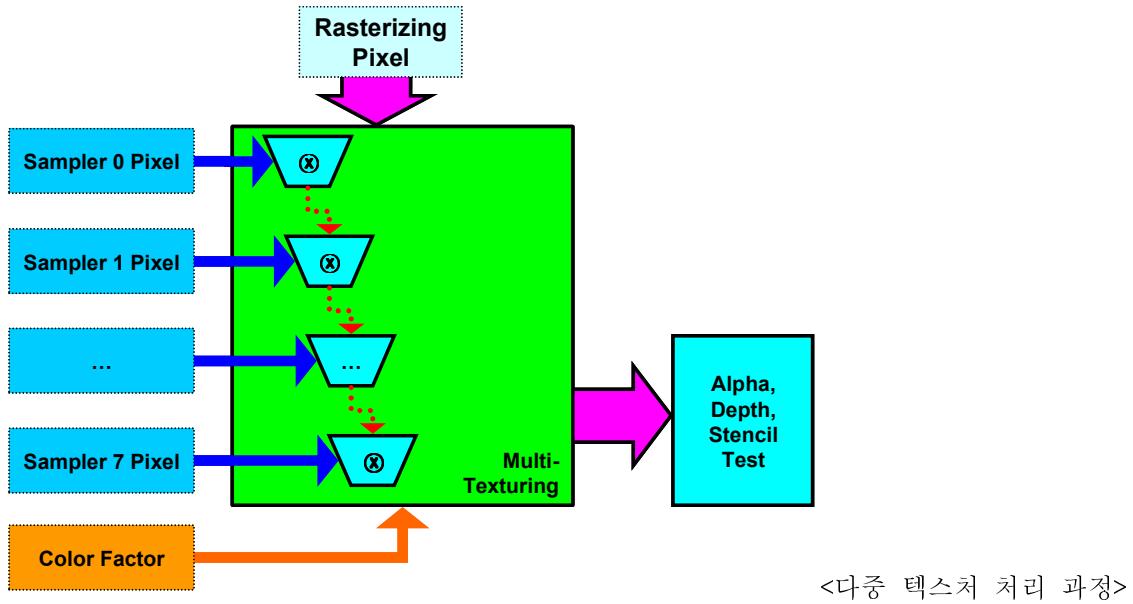
<비등방 필터링: 하드웨어에서 지원하는 최대 비등방 값 적용>

비등방 필터링을 적용한 것이 선형, 근접점 보다 품질이 아주 우수하다는 것을 알 수 있습니다. 이 방법을 권장하고 싶지만 이 필터링은 하드웨어마다 지원되는 것이 다르고 속도 또한 영향을 줍니다. 만약 원경에 좋은 Anti-Aliasing이 필요 없을 경우에는 삼중 필터링 정도가 좋습니다.

예제 [m3d_06_tex01_filtering.zip](#)를 컴파일하고 실행하면 처음에는 필터링이 적용 안된 화면이 출력 됩니다. void CMcScene::Render() 함수를 보면 숫자 키 1, 2, 3, 4에 대해서 각각 None, 근접점, 선형, 비등방 필터링을 적용한 코드를 볼 수 있습니다.

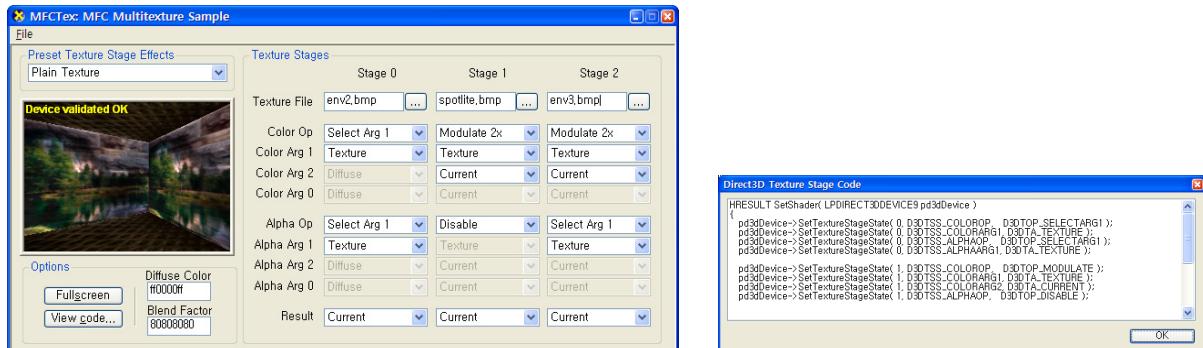
7.3 다중 텍스처 처리 (Multi-Texturing)

앞에서 파이프라인에서 텍스처 처리 과정은 크게 샘플링과 다중 텍스처 처리 과정으로 구분한다고 했습니다. 다중 텍스처 처리는 다음 그림처럼 레스터 처리 색상, 샘플러에 처리된 색상 상태 머신에 Color Factor를 가지고 총 8단계를 거쳐 후면 버퍼에 적용할 색상을 만들어 내는 것입니다.



다른 내용은 화면에 바로 출력이 되어 이해하기가 어렵지 않지만 조명과 더불어 뜻대로 잘 되지 않는 것이 이 다중 텍스처 처리입니다.

다행히도 이 다중 텍스처 처리를 연습할 수 있는 예제가 DXSDK안에 mfctex.exe파일로 있습니다. 이 프로그램을 실행하고 필요에 따라 Stage 0, Stage 1, Stage 2의 내용을 바꾸어 보면 다음과 같은 화면을 얻을 수 있습니다.



다중 텍스처의 연산은 특별한 경우에 3개의 픽셀을 사용하지만 보통 두 픽셀을 가지고 Blending하는 것입니다. 여기서 Blending은 통계적으로 어느 한쪽으로 치우침 없이 균일하게 섞는 의미가 있습니다. 초등학교 때 균일한 혼합물을 생각하면 됩니다.

D3D는 최대 8단계 0 Stage ~ 7 Stage까지 다중 텍스처를 진행 합니다. 각 단계(Stage)에 처리되는 색상을 수식으로 표현하면 다음과 같습니다.

$$\text{Pixel}_n = \text{Pixel}_{n,1} \otimes \text{Pixel}_{n,2}$$

이 것을 D3D에 맞는 프로그램을 바꾸면 다음과 같이 됩니다.

```
m_pDev->SetTextureStageState( n-Stage, D3DTSS_COLORARG1, 연산에 적용할 1번째 픽셀 지정 );
m_pDev->SetTextureStageState( n-Stage, D3DTSS_COLORARG2, 연산에 적용할 2번째 픽셀 지정 );
m_pDev->SetTextureStageState( n-Stage, D3DTSS_COLOROP, 연산 종류 );
```

마치 어셈블리어 명령어와 비슷합니다. push, push, add...

예를 들어 0 번째 단계(Stage)에서 연산에 적용할 첫 번째 픽셀을 샘플러 0번에서 가져오고, 두 번째 픽셀은 레스터 과정의 Diffuse 색상을 가져온다고 합시다. 그리고 두 픽셀을 곱해서 0단계의 픽셀로 만드는 것을 코드로 작성하면 다음과 같습니다.

```
m_pDev->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pDev->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pDev->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
```

코드에서 OP는 Operation(연산 명령), ARG는 Argument (인수)입니다. 텍스처 연산은 고정파이프라인에서 다중 텍스처 연산은 D3DTEXTUREOP이에 정의되어 있습니다. 자주 사용되는 연산은 MODULATE, ADD, SUB 연산입니다.

두 픽셀의 연산 결과는 Current에 저장이 됩니다. 이 Current를 사용하는 예를 만들어 봅시다. 1 번 단계에서 첫 번째 픽셀은 Current에서 가져오고, 두 번째 픽셀은 Sampler 1에서 가져와서 둘을 더하는 코드를 작성해 봅시다.

```
m_pDev->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );
m_pDev->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_TEXTURE );
m_pDev->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
```

여기서 D3DTA_TEXTURE는 해당 단계(Stage) 인덱스와 동일한 샘플러 인덱스에서 픽셀을 가져오게 합니다. 이것이 제대로 동작하기 위해서는 다음과 같이 작성해야 할 것입니다.

```
pDevice->SetTexture( n-Sampler, 또는 n-Stage, "Texture Pointer");
```

이렇게 Current에는 이전 단계(Stage)에서 연산한 색상이 자동으로 저장되는 공간입니다. D3D는 유연성을 갖기 위해 Current 이외에 다른 공간에도 연산을 저장할 수 있습니다. 이 공간은 "Temp" 인데 여기에 저장을 하면 Current는 아무것도 저장되어 있지 않음을 주의 해야 합니다. "Temp" 공간을 사용하는 이유는 Current 무조건 이전 단계의 결과가 저장된 것이라 Stage를 Jump 하기 어려

울 때가 많습니다. 이 때 "Temp" 를 사용하면 단계를 건너 뛰어 "Temp"에 저장된 값을 다른 단계에서도 이용할 수 있습니다.

처리의 결과를 "Temp"에 저장하고 이용하는 코드의 예는 다음과 같습니다.

```
m_pDev->SetTextureStageState( 1, D3DTSS_RESULTARG, D3DTA_TEMP );
```

```
m_pDev->SetTextureStageState( 2, D3DTSS_COLORARG0, D3DTA_CURRENT );
```

```
m_pDev->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEMP );
```

첫 번째 줄에서 D3DTSS_RESULTARG 한 다음 D3DTA_TEMP라 하면 0 번째 단계에서 처리한 Current 값을 변경하지 않고 "Temp"에 저장이 된다는 것입니다. 따라서 두 번째 줄의 D3DTA_CURRENT는 0 번째 단계의 결과를 가져오고, 세 번째 줄의 D3DTA_TEMP는 1 번째 결과를 저장한 "Temp"에서 픽셀을 가져오게 됩니다.

단계에서 처리한 결과를 "Temp"저장했다면 이것을 원래대로 환원해야 다른 프로그램에게 영향을 주지 않습니다. 렌더링이 끝난 후에 반드시 Current로 바꾸어 놓아야 합니다.

```
m_pDev->SetTextureStageState( 1, D3DTSS_RESULTARG, D3DTA_TEMP );
```

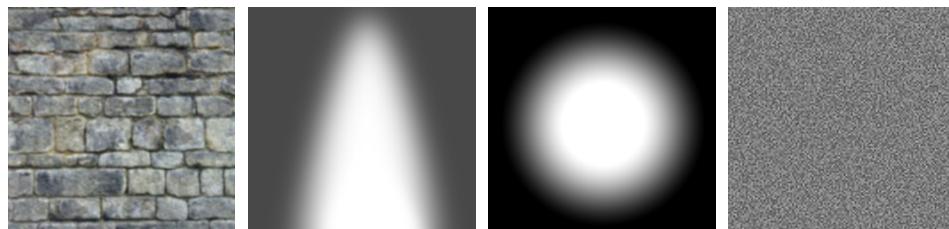
```
...
```

```
m_pDev->DrawPrimitive( ... );
```

```
m_pDev->SetTextureStageState( 1, D3DTSS_RESULTARG, D3DTA_CURRENT );
```

-멀티 텍스처의 연산(Operation)

예제 [m3d_06_tex03_multi.zip](#) 파일의 압축을 풀고 Texture 폴더를 보면 다음과 같이 5개의 텍스처가 있습니다.



<Diffuse Map, Lighting Map1, Lighting Map2, Detail Map>

첫 번째 텍스처는 디퓨즈 맵(Diffuse Map)으로 하겠습니다. 다음 두 텍스처는 라이팅 맵(Lighting Map) 이라 하겠습니다. 마지막 텍스처는 디테일 맵(Detail Map)으로 부르겠습니다.

[m3d_06_tex03_multi.zip](#)을 컴파일하고 실행한 다음 1~9번 키를 누르면 다중 텍스처 처리에 대한 예를 볼 수 있습니다.

- MODULATE 연산

D3D의 MODULATE 연산은 곱하기 연산입니다. 이 연산을 하면 전체 색상이 어두워집니다. 색상의 곱이 어두워지는 것은 이전에 말했듯이 색상을 [0, 1] 값으로 정해서 사용하기 때문입니다. MODULATE2X는 MODULATE한 다음 2배를 곱합니다. MODULATE4X는 MODULATE한 결과에 4배를 곱합니다. 예제의 키보드 "1" ~"3" 까지 실행하면 MODULATE, MODULATE2X, Moudulate4x 대한 출력을 다음 그림과 같은 화면을 볼 수 있습니다.



<MODULATE, MODULATE2X, MODULATE4X 연산>

MODULATE 방식에 대한 예는 void CMcScene::Render() 함수에 if(1 == m_nType) 안에 있는 다음과 같이 구현 되어 있습니다.

```
m_pDev->SetTexture(0, m_pTxDiff0);
m_pDev->SetTexture(1, m_pTxEnv0);

m_pDev->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );
m_pDev->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_TEXTURE );
m_pDev->SetTextureStageState( 1, D3DTSS_COLOROP,    D3DTOP_MODULATE );
m_pDev->SetTextureStageState( 2, D3DTSS_COLOROP,    D3DTOP_DISABLE );
```

SetTexture()는 셈플러에게 텍스처를 연결하는 함수입니다. 코드에서는 0 번 단계는 설정하지 않고 Default를 이용했습니다. 1번 단계에서는 0번 단계에서 처리한 Current 내용과 Arg2로 설정한 텍스처를 MODULATE하라고 되어 있습니다. 2 번 단계에서는 연산을 하지 않고 멀티 텍스처 처리를 빠져나갈 수 있도록 Disable로 정했습니다.

- ADD, SUBSTRACT 연산

ADD는 픽셀을 더하는 연산입니다. ADD, MODULATE2X, MODULATE4X는 그 결과가 1보다 클 때가 있습니다. 이런 경우에는 1로 됩니다. 반대로 SUBSTRACT 연산은 0보다 작게 될 수 있습니다. 이 때는

0이 됩니다. 예제에서 4번 키와 5번 키를 누르면 ADD와 SUBSTRACT의 차이를 확인 할 수 있습니다.



<ADD, SUBSTRACT 연산>

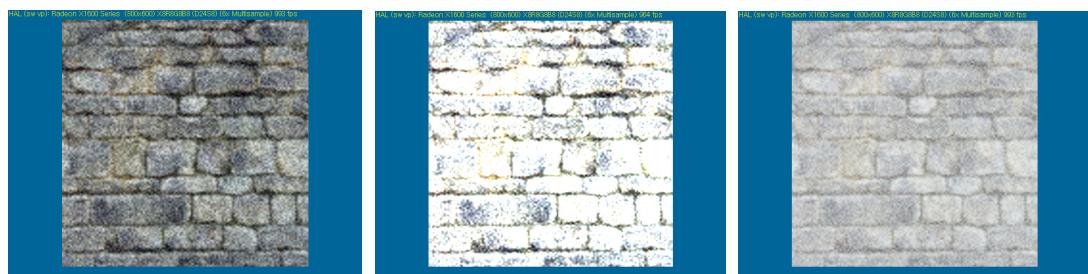
코드는 이전 코드의 COLOR OP만 D3DTOP_ADD 또는 D3DTOP_SUBTRACT로 바꾸어 주면 됩니다.

```
m_pDev->SetTextureStageState( 1, D3DTSS_COLOROP,    D3DTOP_ADD );  
또는 m_pDev->SetTextureStageState( 1, D3DTSS_COLOROP,    D3DTOP_SUBTRACT );
```

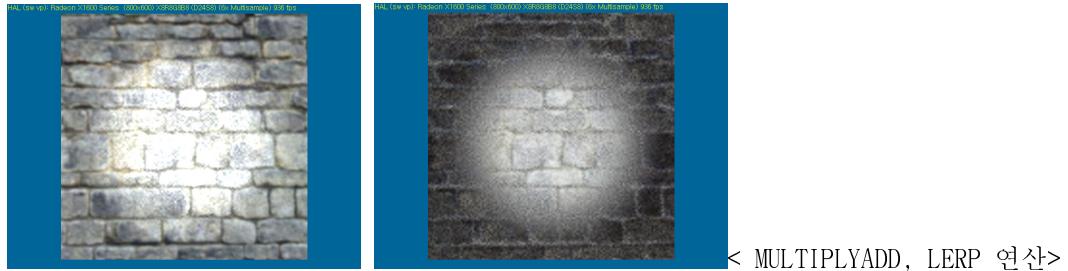
그림자의 경우에 SUBSTRACT를 사용해서 어둡게 할 수 있지만 SUBSTRACT는 그림처럼 강하게 색상이 빠져서 보통은 어두운 이미지와 MODULATE로 그림자를 많이 구현 합니다.

-ADDSIGNED, ADDSIGNED2X, ADDSMOOTH, MULTIPLYADD, LERP

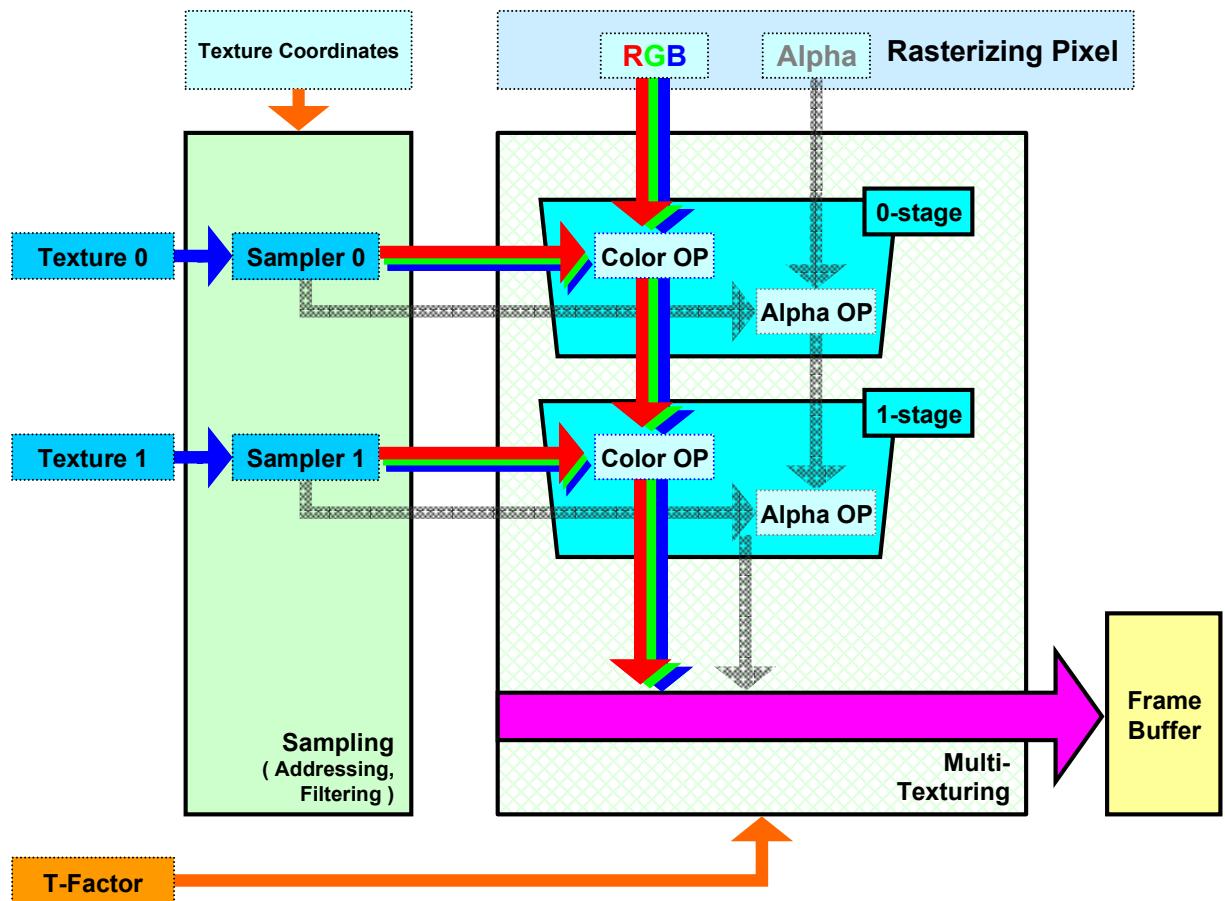
ADDSIGNED는 더하고 나서 0.5 값을 감(빼기) 합니다. ADDSIGNED2X는 ADDSIGNED 한 다음 2배를 합니다. ADDSMOOTH는 두 픽셀을 더한 값에 두 픽셀을 곱한 값을 빼 줍니다.(Arg1 + Arg2 - Arg1*Arg2), LERP은 선형 보간입니다. LERP는 세 개의 픽셀이 필요해서 ARG0도 설정해야 합니다. 계산식은 $(ARG0)*ARG1 + (1-ARG0)*ARG2 = ARG2 + (ARG1-ARG2)*ARG0$ 입니다.



<ADDSIGNED, ADDSIGNED2X, ADDSMOOTH 연산>



지금까지 예제는 전부 픽셀의 RGB만 처리했고 알파 성분에 대해서는 없었습니다. 이것이 가능했던 이유는 D3D는 파이프라인에서 RGB 값과 알파 값 처리 경로가 독립적으로 동작합니다. 다음 그림은 RGB와 알파에 대한 처리과정을 파이프라인에서 2개의 텍스처를 이용 2단계 다중 텍스처 처리를 그림으로 표현한 것입니다.



RGB 처리 과정은 RGB를 마치 위치의 x, y, z로 볼 수 있어서 픽셀에 대한 벡터 처리 과정(Vector Processing)이라 합니다. Alpha는 단일 성분 Scalar와 같아서 픽셀에 대한 스칼라 처리 과정(Scalar Processing)으로 부르기도 합니다.

벡터 처리과정에서 사용한 연산은 스칼라 처리과정에서도 동일하게 적용할 수 있습니다. 이것을 D3D 명령어로 바꾸면 COLOR 단어 대신 ALPHA로 바꾸기만 하면 됩니다. 이렇게 하면 알파 성분에 대한 다중 텍스처 처리를 할 수 있습니다.

다음 예제는 벡터 처리와 알파 처리에 대해서 동일한 연산을 적용한 예입니다.

```
m_pDev->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_CURRENT );
m_pDev->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_TEXTURE );
m_pDev->SetTextureStageState( 1, D3DTSS_COLOROP,    D3DTOP_MODULATE );
m_pDev->SetTextureStageState( 2, D3DTSS_COLOROP,    D3DTOP_DISABLE );

m_pDev->SetTextureStageState( 1, D3DTSS_ALPHAARG1, D3DTA_CURRENT );
m_pDev->SetTextureStageState( 1, D3DTSS_ALPHAARG2, D3DTA_TEXTURE );
m_pDev->SetTextureStageState( 1, D3DTSS_ALPHAOP,    D3DTOP_MODULATE );
m_pDev->SetTextureStageState( 2, D3DTSS_ALPHAOP,    D3DTOP_DISABLE );
```

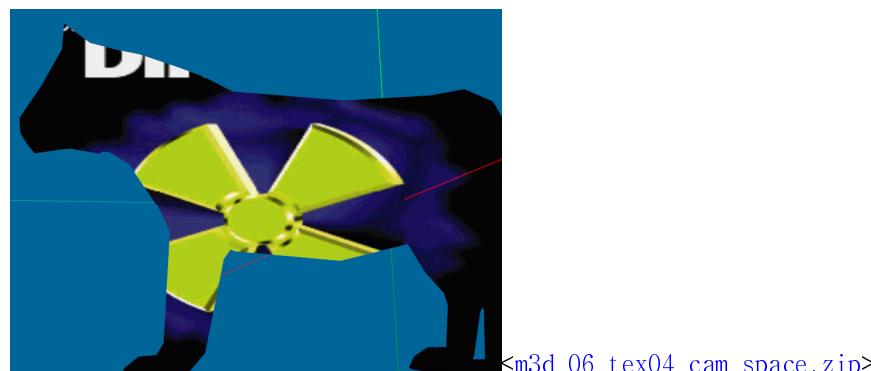
7.4 정점을 텍스처 좌표로 변환

정점의 위치, 법선 벡터를 텍스처 좌표로 사용하는 경우가 종종 있습니다. 예를 들어 거울과 같은 반사 효과를 렌더링 오브젝트에 적용하고 싶을 때 큐브 맵(Cube Map), 또는 환경 맵(Environment Map)을 만들어서 이 맵에 대한 텍스처 좌표를 오브젝트의 법선 벡터로 사용합니다.

변환된 정점의 위치를 좌표로 사용하는 대표적인 예는 투영 그림자입니다. 투영 그림자는 렌더링 오브젝트를 그림자로 사용할 텍스처에 출력하고 이 텍스처를 지형 등에 매핑 해서 그림자 효과를 만듭니다.

여기서는 간단히 정점의 위치를 텍스처 좌표로 사용하는 것만 보이겠습니다.

[m3d_06_tex04_cam_space.zip](#) 파일을 실행하고 키보드의 1번 키를 누르고 나서 카메라를 움직여도 이미지가 거의 움직이지 않는 것을 볼 수 있습니다.



void CMcScene::Render()의 if(1 == m_nType) 블록 안에 다중 텍스처 처리 함수를 사용하는 코드 중에 상태는 D3DTSS_TEXCOORDINDEX, 상태 값은 D3DTSS_TCI_CAMERASPACEPOSITION으로 설정한 코드를 볼 수 있을 것입니다.

```
m_pDev->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACEPOSITION );
```

D3DTSS_TCI_CAMERASPACEPOSITION 상태 값은 파이프라인에서 정점이 카메라 공간으로 변환(뷰 변환)된 정점 데이터를 텍스처 좌표 변환에 대한 입력 좌표로 설정하는 값입니다.

이 때 변환된 정점 데이터와 행렬을 곱한 결과를 텍스처의 좌표로 바꾸어 줍니다. 이것을 공식으로 표현하면 다음과 같습니다.

$$\text{파이프라인 최종 좌표} = \text{변환된 정점 좌표} * \text{투영 변환 행렬} * \text{텍스처 변환 행렬}$$

텍스처 변환 행렬을 파이프라인에 적용하려면 SetTransform() 함수와 SetTextureStageState() 함수에 D3DTSS_TEXTURETRANSFORMFLAGS 상태를 다음과 같이 설정합니다.

```
D3DXMATRIX mtTex;  
...  
m_pDev->SetTransform( D3DTS_TEXTURE0, &mtTex );  
m_pDev->SetTextureStageState( 0,  
    D3DTSS_TEXTURETRANSFORMFLAGS, D3DTFF_COUNT4 | D3DTFF_PROJECTED );
```

D3DTFF_COUNT4 | D3DTFF_PROJECTED 상태 값은 레스터 과정을 거치기 전에 텍스처 좌표를 마지막 값(w)으로 나누어 동차 좌표(x/w, y/w, z/w, w/w)를 만들 수 있도록 합니다.

만약 D3DTFF_COUNT3 | D3DTFF_PROJECTED 로 되어 있다면 z 값으로 (x/z, y/z, z/z) 연산을 합니다.

뷰 변환, 정규 변환을 거치면 정점의 위치는 [-1, 1] 범위를 갖는다고 했습니다.

그런데 텍스처는 [0,1]의 범위를 가집니다. 또한 D3D의 y는 아래쪽으로 증가하는 함수입니다.

따라서 최종 결과(x, y)에 U = 0.5x + 0.5, V = -0.5y + 0.5 로 해야 텍스처 좌표와 일치하게 됩니다.

이것을 행렬로 만들면
$$\begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{pmatrix}$$
 이 됩니다.

만약 투영 변환된 정점을 사용한다면 앞서 구한 행렬에 곱하면 텍스처 좌표(U, V)로 바로 바꿀 수 있지만 불행히도 D3DTSS_TCI_CAMERASPACEPOSITION는 뷰 변환된 정점을 이용한다고 했습니다. 따라서 정점을 강제로 투영 변환을 거치게 해야 하는데 이것을 파이썬 라인에서 할 수 있는 방법은 없음으로 이 행렬에 투영 행렬을 곱해서 텍스처 변환 행렬로 만들어야 합니다.

$$\text{텍스처 변환 행렬} = \text{Projection Matrix} * \begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{pmatrix}$$

Projection Matrix는 다음과 같이 디바이스에서 GetTransform() 함수를 이용해서 가져오고 이것을 최종 앞의 수식과 같이 연산을 해서 최종 텍스처 변환 행렬을 만듭니다.

```
D3DXMATRIX mtTex;
D3DXMATRIX mtPrj;

m_pDev->GetTransform( D3DTS_PROJECTION, &mtPrj );

mtTex = D3DXMATRIX(0.5f, 0, 0, 0,
                    0, -0.5f, 0, 0,
                    0, 0, 1, 0,
                    0.5f, 0.5f, 0, 1);

mtTex = mtPrj * mtTex;
```

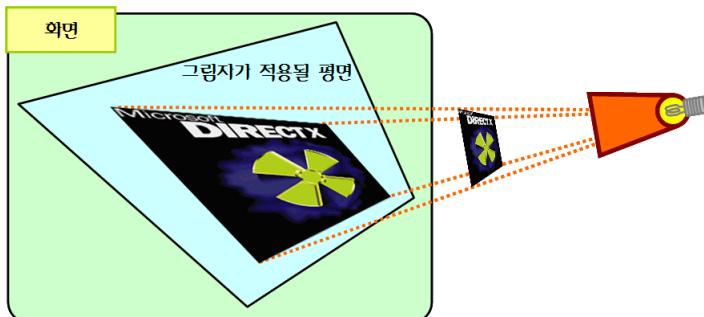
7.5 투영 텍스처 그림자

정점의 위치를 텍스처 좌표로 사용한 이전의 예는 게임에서 투영 텍스처 그림자(Texture Shadow)를 구현하는데 이용됩니다.

투영 텍스처 그림자를 만들기 위해서 여러분은 텍스처에 3D장면을 그려야 하는데 이 부분은 서피스 강의를 참고하고 여기서는 장면을 텍스처에 그렸다고 가정을 하고 이 텍스처를 그림자로 사용하는 방법을 설명하겠습니다.

투영 텍스처 그림자는 카메라의 위치와 방향 대신 조명의 위치와 방향으로 텍스처 좌표를 결정하

는 것입니다.



<원본 이미지, 투영 그림자>

이것을 구현하는 방법은 이전의 예제에서 카메라의 위치와 방향 대신에 조명의 위치와 광원의 방향으로 구성된 행렬을 만들어 적용하는 것입니다.

3D의 모든 정점은 그래픽 파이프라인의 뷰 변환을 거치게 됩니다. 그런데 우리는 카메라의 뷰 변환 대신 조명에 의한 변환 행렬이 적용되기를 원합니다. 즉, 뷰 변환을 그대로 통과되고 조명에 의한 행렬이 적용되도록 해야 하는데 쉐이더를 사용하지 않고 할 수 있는 방법은 조명 행렬에 뷰 행렬의 역행렬을 곱하는 것입니다. 이렇게 하면 파이프라인의 뷰 변환에서 "뷰 행렬 * 뷰 행렬의 역행렬 = 단위 행렬"과 같은 연산이 되어 뷰 변환을 통과한 것처럼 됩니다. 그리고 화면에 대한 투영 행렬 대신에 조명 용도로 사용할 조명의 투영 행렬을 사용한다면 더 효과적일 것입니다.

이런 내용을 공식으로 정리하면 다음과 같이 됩니다.

$$UV\text{-좌표변환 행렬} = \text{뷰 행렬 역행렬} * \text{조명 행렬} * \text{조명 투영 행렬} * \text{텍스처 변환 행렬}$$

[m3d_06_tex04_image_proj.zip](#)의 CMcScene::FrameMove() 함수는 이렇게 조명에 의한 그림자를 위한 투영 그림자 행렬을 만들고 있음을 볼 수 있습니다.

```
INT CMcScene::FrameMove()
{
    D3DXMATRIX mtTex;
    D3DXMATRIX mtView;
    D3DXMATRIX mtProj;
    D3DXMATRIX mtLgt;
    ...
    // 조명의 위치와 방향으로 조명 행렬을 만든다.
    D3DXMatrixLookAtLH(&mtLgt, &vcLgtPos, &vcLgtLook, &D3DXVECTOR3(0, 1, 0));
    ...
    // 조명의 투영 행렬을 만든다. aspect ratio는 1로 한다.
```

```

D3DXMatrixPerspectiveFovLH(&mtPrj, D3DXToRadian(45), 1, 1, 1000);

// 뷰 변환 행렬을 디바이스에서 얻어와서 뷰 행렬의 역행렬로 만든다.
m_pDev->GetTransform(D3DTS_VIEW, &mtViwI);
D3DXMatrixInverse(&mtViwI, NULL, &mtViwI);

// 텍스처 변환 행렬
mtTex = D3DXMATRIX(
    0.5F, 0.0F, 0, 0,
    0.0F, -0.5F, 0, 0,
    0.0F, 0.0F, 1, 0,
    0.5F, 0.5F, 0, 1
);

// 투영 그림자 행렬
m_mtPrjTex = mtViwI * mtLgt * mtPrj * mtTex;

```

렌더링에서는 U, V가 [0, 1] 범위 밖에서는 텍스처가 적용이 안되게 해야 하므로 다음과 같이 Address Mode를 구성합니다.

```

m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_BORDER);
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_BORDER);
m_pDev->SetSamplerState(0, D3DSAMP_ADDRESSW, D3DTADDRESS_BORDER);
m_pDev->SetSamplerState(0, D3DSAMP_BORDERCOLOR, 0x00FFFFFF);

```

m3d_06_tex04_image_proj.zip를 실행하면 그림과 같이 평면과 호랑이 메쉬에 투영 이미지가 적용되고 있음을 볼 수 있습니다.



7.6 환경 매핑(Environment Mapping)

앞에서 투영 텍스처 그림자는 정점의 위치를 텍스처 좌표로 사용한 예입니다. 그런데 정점의 위치 대신 법선 벡터를 텍스처 좌표로 사용할 때는 어떤 효과를 만들 수 있을까요? 결론부터 말씀 드리면 여러분은 반사나 굴절을 표현하는 환경 매핑(Environment Mapping)을 구현하기 위해서 정점의 법선 벡터를 텍스처 좌표로 사용해야 합니다.

환경 매핑도 그림자와 같이 실시간으로 3D 장면을 다음과 같은 텍스처 형태로 만들어야 하는데 지금은 3D 기초 시간이라 환경 매핑 텍스처가 미리 만들어졌다고 가정하고 이를 적용하는 방법을 살펴 보겠습니다.



<환경 매핑 텍스처>

환경 매핑 텍스처가 있으면 여러분은 디바이스의 SetTextureStageState() 함수로 D3DTSS_TEXTURETRANSFORMFLAGS의 값은 D3DTFF_COUNT2으로 설정하고 D3DTSS_TEXCOORDINDEX의 값은 D3DTSS_TCI_CAMERASPACENORMAL으로 정해서 그래픽 파이프라인에서 정점의 법선 벡터를 2D 텍스처 좌표로 사용하도록 합니다. 법선의 벡터도 [-1, 1] 범위 값을 가지게 되므로 텍스처 좌표가 [0, 1] 사이가 되도록 변환 행렬을 SetTransform() 함수를 통해서 지정합니다.

```
void CMcScene::Render()
...
m_pDev->SetTextureStageState(0
    , D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_CAMERASPACENORMAL);
m_pDev->SetTextureStageState(0
    , D3DTSS_TEXTURETRANSFORMFLAGS, D3DTFF_COUNT2);
```

```

D3DXMATRIX mat(
    0.5f,    0.0f,  0.0f,  0.0f,
    0.0f,   -0.5f,  0.0f,  0.0f,
    0.0f,    0.0f,  1.0f,  0.0f,
    0.5f,    0.5f,  0.0f,  1.0f
);

m_pDev->SetTransform(D3DTS_TEXTURE0, &mat);

// Object의 월드 변환
m_pDev->SetTexture(0, m_pTexSphere);
// Draw
m_pMesh->DrawSubset(0);
...
m_pDev->SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, D3DTSS_TCI_PASSTHRU);
m_pDev->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_DISABLE);

```

간단하게 환경매핑을 구현했는데 환경 매핑의 어려운 점은 적용 보다 환경 매핑 용 텍스처를 만드는 것입니다. 환경 매핑 용 텍스처 구현은 서피스 강좌를 참고 하기 바랍니다.

[m3d_06_tex04_sphere.zip](#)을 실행하면 다음과 같이 주변을 반사하는 회전하는 주전자를 볼 수 있습니다.



<환경 매핑. [m3d_06_tex04_sphere.zip](#)>

실습과제) 이전 과제 중에 32x32, 64x64 격자로 구성된 Block에서 인터넷에서 적당한 지형 이미지를 선택해서 적용해 보시오.

실습과제) 위의 과제에 디테일 텍스처를 적용해 보시오.

실습과제) 인터넷에서 빌딩 이미지를 찾아서 이 이미지로 직육면체를 만드시오. 이 직육면체를 30개 이상 생성해서 디테일 텍스처가 적용된 앞 과제에 적당히 배치하시오.