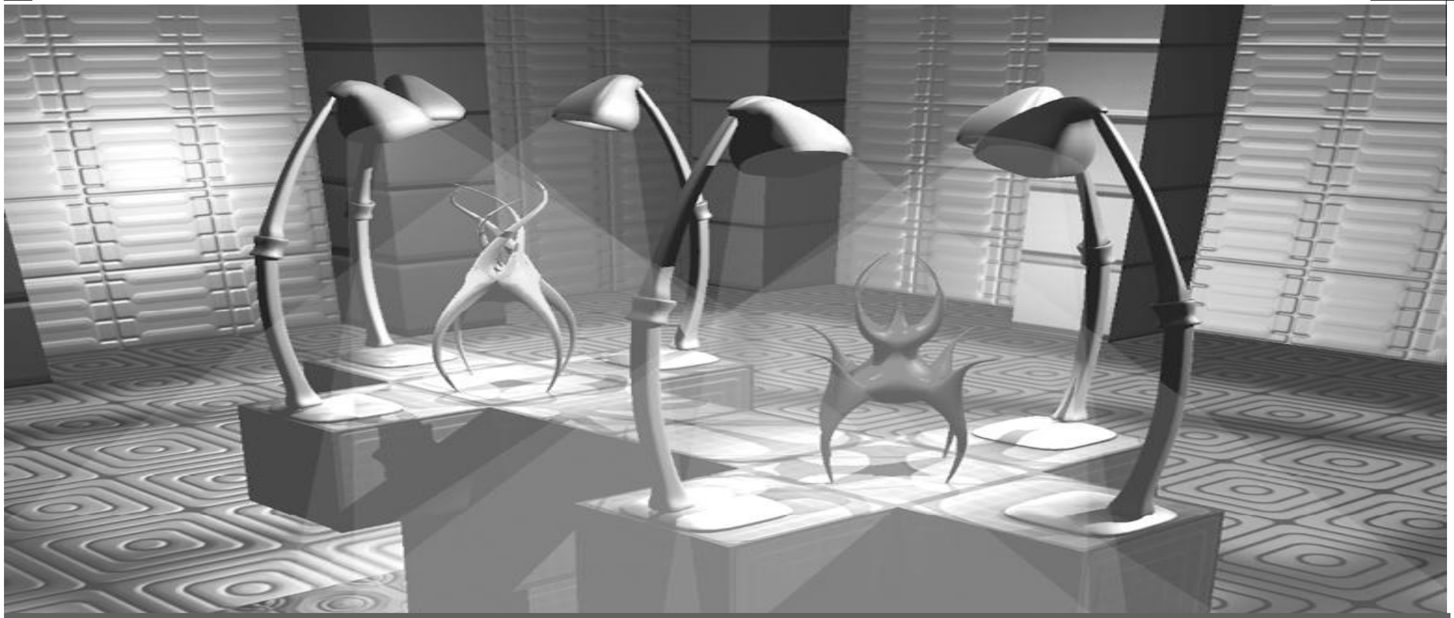


1. Dire c tX 고수준 셰이딩 언어 HLSL에 대한 소개

Craig Peeper & Jason L. Mitchell

소개

HLSL(High Level Shading Language)은 DirectX 9에서 새로이 추가된 것들 중 가장 강력하다 할 수 있다. 셰이더 작성자들은 이제 셰이더 작성 시 표준화된 고수준 언어를 이용하여 레지스터 할당이나 레지스터 제한, 명령어간의 상호 간섭 등과 같은 하드웨어적인 세부사항에 대하여 고민할 필요 없이 알고리즘에 대한 것만 생각할 수 있게 되었다. HLSL은 개발자가 하드웨어 세부사항까지도 알아야 하는 부담에서 해방시킴과 더불어, 쉬운 코드 재사용, 가독성 증가, 최적화된 컴파일러 등 여러 고급 언어의 장점들을 가지고 있다. 이 책과 또 다른 책 「ShaderX^2: DirectX 9 셰이더 프로그래밍 팁 & 트릭」에서 쓰여진 셰이더 코드들은 주로 HLSL로 작성되었기 때문에, 이 글을 읽고 나면 셰이더를 이해하고 작업하는 데 많은 도움이 될 것이다. 이 장에서는 셰이더 언어의 전체적인 개요를 살펴보고 HLSL 셰이더들을 응용프로그램에 통합시키는 방법에 대해 알아볼 것이다.

간단한 예제

HLSL의 복잡한 세부 설명에 들어가기 전에, 간단한 'procedural wood[1]'를 렌더링하기 위해 응용프로그램에 포함되어 있는 HLSL 정점 셰이더와 HLSL 픽셀 셰이더를 먼저 살펴보기로 하자. 다음은 HLSL 셰이더를 이용한 간단한 정점 셰이더이다.

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;

struct VS_PUTOUT
{
    float4 Pos     : POSITION;
    float3 Pshade  : TEXCOORD0;
};


VS_OUTPUT main (float4 vPosition : POSITION)
{
    VS_OUTPUT Out = (VS_OUTPUT) 0;
```

1) 역자 주: 나무의 결 무늬를 비트맵 대신 수학적 계산(procedure)에 의해서 생성하는 것을 말함

```
    // 절단 공간으로 변환
    Out.Pos = mul (view_proj_matrix, vPosition);

    // Pshade를 변환
    Out.Pshade = mul (texture_matrix0, vPosition);

    return Out;
}
```

이 셰이더는 처음 두 줄에서 view_proj_matrix와 texture_matrix0라는 4×4 행렬을 전역으로 선언하고 있다. 이 전역 행렬 다음에는 float4형의 Pos와 float3형의 Pshade 멤버를 가지는 VS_OUTPUT 구조체를 선언하였다.

이 셰이더의 main 함수는 입력 매개변수로 하나의 float4형 데이터를 받고, VS_OUTPUT 구조체를 반환한다. 이것은 float4형의 vPosition이 셰이더의 유일한 입력 값이고, VS_OUTPUT 구조로 출력될 것임을 알려준다. 지금 당장은 이 매개변수와 구조체 멤버 다음의 POSITION 및 TEXCOORD0 구문에 대해서는 생각하지 않아도 된다. 이것들은 '의미구조(semantic)'라 불리는데, 이들의 의미는 이 장의 후반부에서 다시 다룰 것이다.

main 함수의 실제 코드들을 살펴보면 vPosition 벡터와 view_proj_matrix 행렬을 곱하는 mul이라는 내장 함수가 보인다. 이 내장 함수는 정점 셰이더에서 벡터와 행렬 간의 곱셈 연산에 일반적으로 사용된다. 이 여기서 mul의 vPosition은 열 벡터로 사용되었는데, mul의 두 번째 매개변수로 사용되었기 때문이다. 만약, vPosition 벡터가 mul의 첫 번째 매개변수라면 vPosition은 행 벡터로 다루어져야 한다(mul 내장 함수를 비롯한 다른 내장 함수는 이 장의 후반부에서 더 다룰 것이다). 입력받은 vPosition을 절단 공간(clip space)으로 변환한 후, 3D 텍스처 좌표계를 생성하기 위해 vPosition에 texture_matrix0라는 다른 행렬을 곱해준다. 정점 셰이더에서는 정점 위치의 결정과 값으로 최소한 절단 공간으로 변환된 위치(시야/투영 행렬이 곱해진)를 출력해야 한다. 추가된 정점 셰이더에서 출력한 값은 래스터라이즈된 폴리곤을 통해서 보간되고 픽셀 셰이더의 입력 값이 된다. 이 예제의 경우, 3D Pshade는 보간자(interpolator)를 통하여 보간되어 정점 셰이더에서 픽셀 셰이더로 전달된다.

다음은 HLSL로 작성된 간단한 procedural wood 픽셀 셰이더이다. 이 픽셀 셰이더는 방금 전에 살펴본 정점 셰이더와 한 쌍으로 동작하게 작성되어 있는데, ps_2_0을 타겟으로 컴파일될 것이다.

```
float4 lightWood;   // 나무결 색상의 밝은 부분
float4 darkWood;    // 나무결 색상의 어두운 부분
float  ringFreq;    // 나무결의 촘촘한 정도

sampler PulseTrainSampler;

float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
    float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy)) * ringFreq;
    float blendFactor = tex1D(PulseTrainSampler, scaledDistFromZAxis);
    return lerp (darkWood, lightWood, blendFactor);
}
```

이 셰이더의 처음 몇 줄은 4개의 요소를 가진 2개의 부동 소수점 형식의 전역 벡터 변수와 스칼라 변수를 선언한다. 그 뒤에 PulseTrainSampler라는 샘플러를 선언한다. 샘플러에 대해서도 뒤에서 좀 더 자세히 다룰 것이므로 여기서는 일단 샘플러를 윈도우를 필터링과 텍스처 좌표계의 어드레싱 모드 같은 것들을 정의하는 상태를 비디오 메모리로 보내주기 위한 창구 같은 것이라고 알아두자.

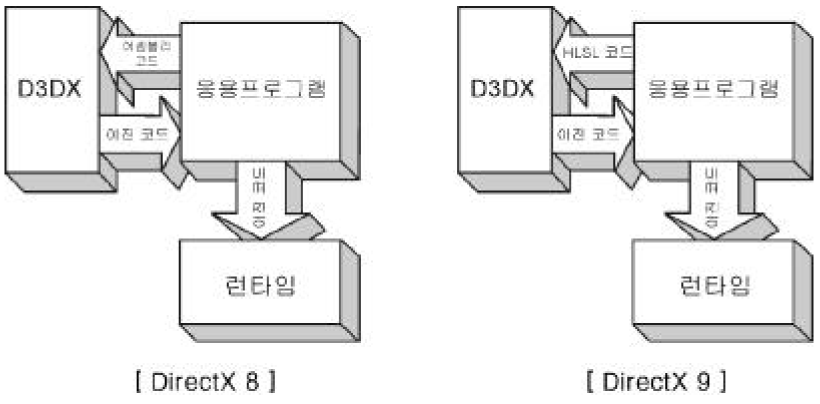
변수와 샘플러를 정의하였으니 이제 실제 셰이더의 동작 부분을 살펴보자. 여기서 Pshade라 불리는 함수의 입력 매개변수를 볼 수 있는데, 이것이 바로 위에서 나온 정점 셰이더의 결과 값이 폴리곤을 따라서 보간된 값이다. 픽셀 셰이더에서는 셰이더 공간의 z-축으로부터 데카르트 좌표계 기준 거리를 계산하고, 확대/축소한 값을 PulseTrainSampler라는 이름으로 연결된 텍스처에 접근하기 위한 1차원 텍스처 좌표 값으로 사용한다. Tex1D() 샘플링 함수에서 반환되는 스칼라 색상은 셰이더에서 전역으로 선언되어 있는 두 가지의 색(lightwood와 darkwood)을 혼합하기 위한 매개변수로 사용한다. 이 혼합의 결과로 나오는 4D 벡터가 이 픽셀 셰이더의 최종 결과 값이다. 모든 픽셀 셰이더는 최소한 4D RGBA색을 반환해야 한다. 픽셀 셰이더의 부가적인 다른 출력들에 대해서는 이 장의 뒤쪽에서 계속 알아볼 것이다.

어셈블리 언어와 컴파일 타겟

지금까지 몇몇의 HLSL 셰이더를 살펴보았다. 이제 이러한 HLSL 언어가 Direct3D, D3DX, 어셈블리 셰이더 모델 그리고 사용자의 응용프로그램과 연계되는 방식에 대해

알아보자. 셰이더라는 것은 DirectX 8.0의 Direct3D에서 처음 추가되었다. 이때에는 여러 종류의 셰이더 가상 머신들이 정의되었고, 각각은 지원되는 유명한 특정 3D 그래픽 하드웨어 제조 회사들의 그래픽 프로세서에서만 동작할 수 있었다. 이 각각의 셰이더 머신들마다 별도의 어셈블리 셰이더 언어가 디자인되었다.

DirectX 8.0과 DirectX 8.1에서 (vs_1_1과 ps_1_1 ~ ps_1_4로 이름 붙여진) 셰이더 모델로 작성된 프로그램은 어셈블리 언어로 작성되었으며 비교적 간결했다. [그림 1-1]의 왼쪽 부분에서와 같이 응용프로그램은 D3DXAssembleShader()를 통해 DX3D 라이브러리에 사람이 이해할 수 있는 어셈블리 언어 코드를 전달하고, CreatePixelShader() 또는 CreateVertexShader()를 통해 Direct3D에 전달될 이진 형태로 컴파일된 셰이더를 얻는다. 어셈블리 셰이더에 대한 더 많은 정보를 얻고자 한다면 「Direct3D ShaderX 정점& 픽셀 셰이더 팁과 트릭」(정보문화사 출간)과 같은 참고 문헌을 참조하기 바란다.



[그림 1-1] DirectX 8과 DirectX 9에서 D3D를 이용한 셰이더 어셈블리 작성법

[그림 1-1]의 오른쪽에도 나와 있듯이, DirectX9의 응용프로그램도 HLSL 셰이더를 D3DXCompileShader() API를 통해 D3DX에 전달하고, CreatePixelShader() 또는 CreateVertexShader()를 통해 direct3D에 전달될 컴파일된 셰이더의 이진 형태를 받는다는 점에서 비슷한 면이 있다. 이렇게 생성된 이진 어셈블리 코드는 국한 사용자나 개발자 시스템의 그래픽 하드웨어에 독립적이다. 즉, 생성된 이진 어셈블리 코드는 하드웨어에 관계없이 독립적으로 사용이 가능하므로, 사용자가 어디서 컴파일하거나 실행하더라도 같은 결과를 가진다. 실제로 Direct3D 실행 모듈은 오로지 이진 어셈블리 셰이더 모델만을 실행하고 HLSL에 대해서는 관여하지 않는다. 이것은 HLSL 컴파일러를 D3D

에 독립적으로 업데이트할 수 있다는 이점이 있는데, 이 책이 출판되기에 앞서 마이크로소프트는 업데이트된 HLSL과 컴파일러를 포함한 DX SDK 업데이트를 발표할 계획이다. DirectX 9에는 D3DX가 지원하는 HLSL만이 포함되어 있는 것이 아니라 가장 최신의 3D 그래픽 하드웨어의 기능들을 구현하기 위한 새로운 어셈블리 모델 또한 포함되어 있다. 응용프로그램 개발자들은 이 새로운 모델(vs_2_0, vs_3_0, ps_2_0과 ps_3_0)들을 직접 어셈블리 언어를 사용하여 작업할 수도 있지만, 셰이더의 복잡도와 작업 편의성을 감안할 때 많은 개발자들이 어셈블리보다는 HLSL을 통해 셰이더를 개발할 것으로 본다.

하드웨어에 대한 현실적 고려

HLSL을 사용해서 특정한 셰이딩 알고리즘을 표현하는 프로그램을 작성했다고 해서 이 프로그램이 특정 하드웨어만 동작한다는 뜻은 아니다. 앞서 설명한 것처럼 응용프로그램은 D3DXCompileShader() API를 가지고 HLSL 셰이더를 이진 어셈블리 코드로 컴파일하기 위해 D3DX를 호출한다. 이 API를 실행하기 위해 필요한 매개변수들 중에는 어느 셰이더 버전용으로 컴파일할 것인지를 정할 수 있는 매개변수가 있다. 따라서 만일 응용프로그램에서 실시간으로 HLSL을 컴파일한다면, 응용프로그램은 우선 실행되는 Direct3D 하드웨어가 어떤 셰이더 기능을 지원하는지를 확인한 후에 알맞은 컴파일 타겟을 선택할 수도 있다. 만일 HLSL 셰이더에 표현된 알고리즘이 선택된 컴파일 타겟에서 실행하기에 너무 복잡하다면 컴파일되지 않을 것이다. 게임의 특성 상 다양한 사양의 하드웨어들을 지원해야 하는 현실 때문에 개발자들에게 이 기능은 셰이더를 개발할 때 매우 큰 장점이 될 것이다. 실시간으로 컴파일되는 HLSL이 없다면 게임 개발자는 여전히 고성능의 그래픽 카드를 위한 버전과 구형 카드를 위한 버전을 둘 다 개발하는 소모성 작업을 계속해야 할 것이다. 그러나 제대로 작성된 HLSL은 이러한 문제를 어느 정도 해결할 수 있다.

컴파일 실패

앞서 언급했다시피, 주어진 HLSL 셰이더에서 특정한 컴파일 타겟에 컴파일이 실패했다는 것은 셰이더가 너무 복잡하여 특정 컴파일 타겟에서는 컴파일할 수 없다는 것을 의미한다.

이것은 셰이더가 너무 많은 리소스를 요구하거나 선택된 컴파일 타겟이 지원하지 않는 동적 분기와 같은 특정 기능을 사용했다는 뜻이다. 예를 들어, 어떤 HLSL 셰이더가 셰이더에서 주어진 텍스처 맵을 6번 접근할 수 있게 작성되었다고 하자. 만일 이 셰이더가 ps_1_1 컴파일 타겟에서 컴파일한다면 ps_1_1 모델은 4개의 텍스처만 지원하므로 컴파일되지 않을 것이다. 또다른 컴파일 에러의 이유 중 대부분은 선택된 컴파일 타겟의 명령어 개수가 초과될 때 나타난다. HLSL에 표현된 알고리즘이 주어진 컴파일 타겟에서 실행될 때 너무 많은 명령어를 요구하는 것이다.

여기서 강조하고 싶은 점은 컴파일 타겟을 결정하는 문제는 HLSL 문법의 사용 범위와는 무관하다는 것이다. 예를 들어, 어떤 셰이더를 루프와 서브루틴, if-else 조건문 등을 사용하여 작성하였지만 특정 타겟의 컴파일러는 이와 같은 기능을 지원하지 않는다고 하자. 이런 경우에는 컴파일러는 루프를 전개하고, 함수 호출 부분을 인라인으로 처리하여 함수 내용을 직접 코드에 풀어 써주고, if-else 구문의 두 가지 경우를 모두 실행한 뒤, if-else 구문에서 사용되었던 원래 값에 기반하여 적절한 값을 선택하는 등의 작업을 수행함으로써 해당 타겟에서 실행 가능한 코드를 만들어낼 것이다. 물론, 결과 셰이더가 너무 길거나 컴파일 타겟의 리소스 제한을 초과할 경우 컴파일은 실패하게 된다.

커맨드라인 컴파일러 - fxc

많은 개발자들은 HLSL 셰이더를 D3DX로 유저들의 컴퓨터에서 프로그램이 로딩할 때나 처음 실행될 때 컴파일하는 것보다 제품을 배포하기 전에 미리 이진 어셈블리 형태로 컴파일하는 것을 선호한다. 이렇게 하면 소스 코드를 이것저것 뜯어보기를 좋아하는 유저들로부터 숨길 수 있고, 실행되는 모든 셰이더들을 자체적으로 품질 검증 과정을 거치게 할 수 있기 때문이다.

fxc 커맨드라인 컴파일러는 오프라인으로 셰이더를 컴파일할 수 있게 하는 편리한 유틸리티로 DirectX 9 SDK에서 찾아볼 수 있다. 이 유틸리티는 커맨드라인에서 사용할 수 있을 뿐만 아니라, 특정한 컴파일 타겟을 위한 디스어셈블된 코드를 생성할 수 있는 편리한 옵션들을 제공한다. 만일 셰이더를 최적화시키기를 원하거나 가상 셰이더 머신의 능력을 보다 자세히 알고 싶다면 디스어셈블된 결과물을 살펴보기 바란다.

커맨드라인에서 사용 가능한 옵션들은 다음과 같다.

커맨드라인 옵션	설명
-T target	컴파일 타겟(기본 값: vs_2_0)
-E name	엔트리 포인트 name(기본 값: main)
-Od	최적화하지 않음
-Vd	검증하지 않음
-Zi	디버깅 정보를 삽입
-Zpr	열 기준으로 행렬 정리
-Zpc	행 기준으로 행렬 정리
-Fo file	목적 파일 출력
-Fc file	생성된 코드의 출력
-Fh file	생성된 코드를 포함한 헤더의 출력
-D id = text	매크로 정의
-nologo	저작권 메시지를 출력하지 않음

이제 여러분은 셰이더 개발에 사용되는 HLSL 컴파일러의 내용들에 대해 이해했을 것이다. 이제 언어의 실제 구조에 대해 살펴보기로 하자. 이후로도 컴파일 타겟의 의미나 기본이 되는 어셈블리 셰이더 모델의 다양한 기능들은 중요하게 다룰 것이다.

언어의 기초

여기까지 HLSL에서 정점 셰이더와 픽셀 셰이더가 어떤 것인지와 저수준 어셈블리 셰이더가 동작하는 방식에 대해 알아보았다. 이제부터는 HLSL 언어 그 자체에 대하여 보다 자세히 살펴보도록 하자.

키워드

키워드는 HLSL 언어와 미리 정의된 예약어로 사용자 프로그램 내에서 식별자로 사용할 수 없다. 키워드는 (*)로 마스크되어 있으며, 대·소문자를 구별하지 않는다.

asm*	bool	compile	const
decl*	do	double	else
extern	false	float	for
half	if	in	inline
inout	int	matrix*	out

pass*	pixelshader*	return	sampler
shared	static	string*	struct
technique*	texture*	true	typedef
uniform	vector*	vertexshader*	void
volatile	while		

아래의 키워드들은 현재는 사용되지 않지만 앞으로의 사용을 위해 예약된 키워드들이다.

auto	break	case	catch
char	class	compile	const
const_cast	continue	Default	delete
dynamic_cast	enum	explicit	friend
goto	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	short
Signed	sizeof	static_cast	switch
template	this	throw	try
typename	union	unsigned	using
virtual			

데이터 유형

HLSL은 간단한 스칼라뿐만 아니라 벡터나 행렬과 같은 보다 복잡한 데이터형도 지원한다.

스칼라 데이터형

다음과 같은 스칼라 데이터형을 지원한다.

데이터 유형	표현 방법
bool	참 혹은 거짓
int	32비트 부호 있는 정수
half	16비트 부동 소수점 실수
float	32비트 부동 소수점 실수
double	64비트 부동 소수점 실수

셰이더 어셈블리 프로그램을 짜본 사람이라면 모든 그래픽 프로세서가 현재 이러한 모든 데이터형을 지원하는 것은 아니라는 것을 알고 있을 것이다. 정수는 부동 소수점 하드웨어에서 에뮬레이션해서 사용하는데, 이는 플랫폼에서 실수 값으로 표현될 수 있는 범위를 넘어서는 정수 연산들은 정확한 결과가 나올 수 없다는 뜻이기도 하다. 또한 모든 타겟 플랫폼이 half나 double형을 지원하는 것은 아니다. 하지만 타겟 플랫폼이 지원하지 않는 경우에는 float형을 사용하여 에뮬레이션할 수는 있다.

벡터 데이터형

HLSL 셰이더에서는 벡터 값을 많이 사용한다. 벡터 데이터형을 정의하는 방법은 다음과 같이 여러 가지가 있다.

벡터	선언
vector	4차원 벡터, 각 성분은 실수형
vector<type, size>	size 크기의 벡터, 각 성분은 type 데이터형

벡터를 정의하는 가장 일반적인 방법은 다른 셰이더 제작자들이 보통 벡터를 정의하는 방식에서 볼 수 있겠지만, 아마도 데이터형 이름 뒤에 2 4 사이의 정수 값을 덧붙이는 방법이 될 것이다.

4개의 float형 데이터를 정의하기 위해서 여러분은 다음 중 아무 것이나 사용하면 된다.

```
float4 fVector0;
float  fVector1[4];
vector fVector2;
vector <float, 4> fVector3;
```

3개의 bool형의 데이터를 선언하고 싶다면 다음 중 아무 것이나 사용하면 된다.

```
bool3 bVector0;
bool  bVector1[3];
vector <bool, 3> bVector2;
```

벡터 값을 한번 정의한 후에는 배열에 인덱스 접근을 이용하거나 스위즐(swizzle)을 사용해서 직접 벡터의 성분을 명시하는 방식으로 그 성분들을 읽거나 쓸 수 있다. 스위

즐을 사용할 경우 성분의 이름은 $\{x, y, z, w\}$ 또는 $\{r, g, b, a\}$ 등의 네임스페이스를 사용하여 조합 가능하다(두 네임스페이스를 동시에 사용할 수는 없다).

예를 들면,

```
float4 pos = {3.0f, 5.0f, 2.0f, 1.0f};
float  value0 = pos[0];     // value0은 3.0f
float  value1 = pos.x;      // value1은 3.0f
float  value2 = pos.g;      // value2는 5.0f
float2 vec0   = pos.xy;     // vec0은 {3.0f, 5.0f}
float2 vec1   = pos.ry;     // 잘못된 스위즐을 사용하였으므로 오류!
```

Ps_2_0이나 더 낮은 픽셀 셰이더 모델에서는 임의의 스위즐을 기본으로 지원하지 않는다. 따라서 임의의 스위즐을 사용하는 간결한 고수준 코드를 스위즐이 지원되지 않는 타겟용으로 컴파일한다면 컴파일러는 상당히 지저분한 이진 어셈블리 코드를 생성할 것이다.

행렬 데이터형

HLSL에서 사용하고 있는 다른 일반적인 변수의 형태는 2D 배열 형태의 행렬이다. 스칼라나 벡터와 마찬가지로 행렬도 bool, int, half, float 또는 double의 데이터형을 가질 수 있다. 행렬은 어떤 크기로도 가능하지만, 셰이더에서 사용하는 일반적인 형태는 4×4의 크기를 사용한다. 앞에서 처음 보았었던 정점 셰이더의 첫 부분에 보면 2개의 4×4 float형의 행렬이 전역으로 선언된 것을 볼 수 있다.

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;
```

물론, 다른 차원의 행렬도 사용이 가능하다. 예를 들면, 다음과 같이 3×4의 크기를 가진 행렬도 사용이 가능하다.

```
float3x4              mat0;
matrix<float, 3, 4>   mat1;
```

벡터와 마찬가지로, 행렬에서 각각의 원소들은 배열과 구조체/스위즐 문법을 이용하여 접근이 가능하다. 예를 들어 다음의 배열 접근 문법은 view_proj_matrix 행렬의 최상단 좌측 원소를 참조한다.

```
float fValue = view_proj_matrix[0][0];
```

행렬의 각 원소를 접근하고 스위즐하기 위해 사용하는 문법도 있다. 0부터 시작하는 행과 열의 위치는 다음과 같이 사용할 수 있다.

```
_m00, _m01, _m02, _m03
_m10, _m11, _m12, _m13
_m20, _m21, _m22, _m23
_m30, _m31, _m32, _m33
```

1부터 시작하는 행과 열의 위치는 다음과 같이 사용할 수 있다.

```
_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44
```

행렬은 다음의 예와 같이 배열과 동일한 방법으로도 접근이 가능하다.

```
float2x2 fMat   = { 3.0f, 5.0f,     // 1행
                    2.0f, 1.0f };   // 2행

float   value0  = fMat[0];          // value0은 3.0f
float   value1  = fMat._m00;        // value1은 3.0f
float   value2  = fMat._12          // value2는 5.0f
float   value3  = fMat[1][1]        // value3은 1.0f
float2  vec0    = fMat._21_22;      // vec0은 {2.0f, 1.0f}
float2  vec1    = fMat[1];          // vec1은 {2.0f, 1.0f}
```

형 수정자(type modifier)

HLSL에는 두 개의 부가적인 형 수정자가 있다. 일반적인 const 형 수정자는 셰이더 코드 내에서 특정 변수를 수정 불가능하게 만들 때 사용한다. 이러한 변수에는 무언가를 대입하려 하면 컴파일 시 에러가 발생한다.

row_major와 col_major 형 수정자는 하드웨어가 상수를 어떤 형태로 행렬에 저장할지 결정하는 데 쓰인다. row_major 형 수정자는 행렬의 각 행이 한 개의 상수 레지스터에 저장됨을 나타내며, 마찬가지로 col_major는 각 열이 한 개의 상수 레지스터에 저장됨을 나타낸다. 형 수정자를 명시하지 않은 경우의 기본 값은 열 우선(col_major)이다.

스토리지 클래스 수정자(storage class modifier)

스토리지 클래스 수정자는 컴파일러에게 주어진 변수의 범위와 수명을 알려준다. 이 수정자는 선택적이며, 변수 형 지정 전에 쓰기만 하면 어떤 순서로도 사용이 가능하다. C에서와 마찬가지로, 변수는 static 또는 extern으로 설정할 수 있다(이 두 수정자는 동시에 사용할 수 없다). 전역으로 설정한 static 스토리지 클래스 수정자는 오직 셰이더에 의해서만 접근이 되고, API를 통한 접근은 되지 않는다. 전역이고 static이 아닌 변수들은 API를 이용한 응용프로그램으로 수정이 가능하다. C와 마찬가지로, 지역 변수로 선언된 static 수정자는 선언한 함수가 계속 불려도 이전 불렸을 때의 값이 계속 남아 있음을 나타낸다.

extern 수정자는 전역 변수에서 셰이더의 바깥쪽에서 API를 통해서 수정될 수 있다는 것을 나타내기 위해서 사용하는데, 전역으로 선언된 변수들은 이미 extern의 기능을 기본적으로 하고 있으므로 적어도 전역 변수에 대해서는 굳이 써줄 필요는 없다.

shared 수정자를 사용하면 주어진 전역 변수를 effect들 사이에서 공유할 수 있게 된다.

uniform형의 변수는 외부에서 HLSL 셰이더로 값을 지정하기 위해 사용된다(Set*ShaderConstant*() API를 통하여). 전역 변수들은 uniform으로 선언된 것과 마찬가지로 다루어진다. 이러한 변수들은 셰이더에 의해서 값이 변경될 수 있으므로 const가 될 수 없다. 예를 들어, 다음과 같이 전역 변수를 선언한다고 하자.

```
extern  float translucencyCoeff;
const   float gloss_bias;
static  float gloss_scale;
float   diffuse;
```

diffuse 변수와 translucencyCoeff 변수는 SetShaderConstant() API에 의해 설정이 되고, 셰이더에 의해서도 수정이 될 수 있다. const 함수인 gloss_bias는 Set*ShaderConstant*() API로 설정할 수 있고, 셰이더 코드 안에서는 수정할 수 없다. 마지막으

로, static 변수인 gloss_scale는 Set*ShaderConstant*() API로 설정할 수 없으며, 셰이더에서만 수정할 수 있다.

초기화

앞에서 보았던 예제와 마찬가지로 C에서와 같은 방법으로 변수를 초기화할 수 있다.

```
float2x2 fMat    = { 3.0f, 5.0f,          // 첫째열(수정자가 없으면 열 우선
                     2.0f, 1.0f };        // 2행 둘째열 임을 기억하라.)
float4   vPos    = { 3.0f, 5.0f, 2.0f, 1.0f };
float    fFactor = 0.2f;
```

벡터로 작업하기

HLSL에는 벡터를 연산할 때 꼭 알아두어야 할 사항이 몇 가지 있다. 다행히도 3D 그래픽의 셰이더를 기술할 때 쓰이는 방식 대부분은 매우 직관적이다. 예를 들면, 표준 이진 연산자는 성분별로 정의되어 있다.

```
float4 vTone = vBrightness * vExposure;
```

vBrightness와 vExposure가 모두 float4형의 데이터이라고 가정한다면 위의 연산은 다음과 같다.

```
float4 vTone;
vTone.x = vBrightness.x * vExposure.x;
vTone.y = vBrightness.y * vExposure.y;
vTone.z = vBrightness.z * vExposure.z;
vTone.w = vBrightness.w * vExposure.w;
```

주의할 점은 위의 연산은 4D 벡터인 vBrightness와 vExposure의 내적이 아니라는 것이다. 이런 형태의 행렬 변수의 곱 역시 일반적인 행렬의 곱셈과는 다르다. 벡터간의 내적과 행렬간의 곱셈은 이 장의 뒤쪽에서 다루어질 mul() 함수를 통해서 계산하여야 한다.

생성자(constructor)

HLSL 셰이더에서 볼 수 있는 또 다른 언어의 형태인 생성자는 C++와 유사하지만, 복잡한 데이터형을 다루기 위해 조금 더 개선된 것이다. 다음은 생성자의 사용 예이다.

```
float3  vPos      = float3(4.0f, 1.0f, 2.0f);
float   fDiffuse  = dot(vNormal, float3(1.0f, 0.0f, 0.0f));
float4  vPack     = float4(vPos, fDiffuse);
```

생성자는 셰이더를 원하는 형을 명시하여 일시적으로 정의할 때나(dot(vNormal, float3(1.0f, 0.0f, 0.0f)처럼 dot 함수에 들어갈 float3형을 명시적인 값으로 임시로 생성했다), 적은 크기의 데이터형들을 합쳐서 하나의 큰 크기의 데이터형으로 만들고 싶을 때(vector3형과 스칼라형을 vector4형으로 합친 float4(vPos, fDiffuse)에서처럼) 일반적으로 사용한다. 후자의 경우 float4 생성자 대신에 float3 또는 float를 사용하면 float4의 결과 값을 얻을 수 있다.

형 변환(type casting)

셰이더 코드를 쉽고 효율적으로 작성하려면 HLSL의 형 변환에 익숙해져야 한다. 유형을 정의할 때는 이미 주어진 값들과 대응하는 변수들을 취하거나 버려야 한다. 예를 들면, vResult를 초기화하고 싶을 때에는 float 0.0f로 선언해도 되고 float4(0.0f, 0.0f, 0.0f, 0.0f)로 선언해도 된다.

```
float4  vResult = 0.0f;
```

벡터나 행렬과 같은 고차원의 데이터를 저차원의 데이터형으로 전환할 때에도 비슷한 방법을 적용한다. 이런 경우에는 부가적인 데이터는 생략한다. 예를 들어, 다음과 같은 코드를 보자.

```
float3  vLight;
float   fFinal, fColor;
fFinal = vLight * fColor;
```

이 경우에는 곱셈을 하기 위해서 vLight의 첫 번째 성분 x와 fColor가 곱해진다. 이것은 fFinal = vLight.x * fColor와도 같은 의미이다.

HLSL에서 데이터형을 변환하기 위해서는 다음의 표를 익히고 있으면 좋을 것이다.

변환 유형	변환 방식
스칼라 → 스칼라	언제나 유효하다. bool형에서 정수나 부동 소수점 형태로 바뀔 때 false 값은 언제나 0이 되고, true 값은 언제나 1이 된다. 정수형이나 부동소수점형에서 bool형으로 변환할 때는 0값은 false로 되고, 0이 아니면 true가 된다. 부동소수점형을 정수형으로 바꿀 때는 소수점 이하를 버린다. 이것은 C에서와 마찬가지이다.
스칼라 → 벡터	언제나 유효하다. 벡터 값을 채우기 위해 각 원소를 스칼라 값으로 계속 복사해 넣어 변환이 이루어진다.
스칼라 → 행렬	언제나 유효하다. 스칼라 값을 행렬의 각 원소로 복사해 넣는 것으로 변환이 이루어진다.
스칼라 → 구조체	변환은 스칼라 값을 구조체를 채우기 위해 복사하는 것으로 이루어진다.
벡터 → 스칼라	언제나 유효하다. 벡터의 첫 번째 성분을 선택해서 스칼라로 변환한다
벡터 → 벡터	변환하려는 벡터는 변환 전 벡터보다 성분이 더 많으면 안 된다. 이 변환은 왼쪽의 값들을 보존하고 잘려지는 나머지 값들을 버리는 식으로 이루어진다. 이런 목적의 변환에서 열 행렬이나 행 행렬이나 숫자 구조체들은 벡터로 인식된다.
벡터 → 행렬	벡터의 크기는 행렬과 같아야 한다.
벡터 → 구조체	구조체가 벡터보다 크지 않고 구조체의 각 성분들이 모두 숫자일 경우에만 유효하다.
행렬 → 스칼라	언제나 유효하다. 맨 왼쪽의 위의 성분을 선택한다.
행렬 → 벡터	행렬의 크기는 벡터의 크기와 무조건 같아야 한다.
행렬 → 행렬	변환될 행렬은 양 차원 모두 변환 전 행렬보다 크지 않아야 한다. 변환은 왼쪽 위쪽의 값들을 보존하고 잘리는 나머지를 버리는 식으로 진행된다.
행렬 → 구조체	구조체의 크기는 행렬의 크기와 같고 구조체의 모든 성분은 숫자여야 한다.
구조체 → 스칼라	구조체는 무조건 한 개 이상의 멤버를 지녀야 한다.
구조체 → 벡터	구조체는 최소한 벡터 크기이어야 한다. 첫 번째 성분부터 벡터 크기까지의 구조체 성분은 숫자여야 한다.
구조체 → 행렬	구조체는 행렬의 크기 만큼이어야 한다. 첫 번째부터 행렬의 크기까지의 구조체 성분은 숫자여야 한다.
구조체 → 오브젝트	구조체는 최소한 한 개의 멤버를 가지고 있어야 한다. 멤버의 형은 오브젝트의 형과 반드시 일치해야 한다.
구조체 → 구조체	변환되려는 구조체보다 변환 전 구조체의 크기가 더 커지면 안 된다. 두 구조체의 대응되는 각 개별 성분들 간의 형 변환이 유효해야 한다.

구조체(structure)

셰이더의 첫 번째 예에서 보았던 것처럼, HLSL 셰이더에서는 구조체를 정의하여 편리하게 사용할 수 있다. 예를 들어, 셰이더를 작성할 때 정점 셰이더 코드에 출력 구조체를 설정하고 이를 정점 셰이더의 main 함수의 결과에 이용할 수 있다(픽셀 셰이더의 경우에는 대부분의 결과가 float4로만 나오므로 많이 쓰이지는 않는다). NPR Metallic 셰이더에서 가져온 예를 살펴보자.

```
struct VS_OUTPUT
{
   float4 Pos    : POSITION;
   float3 View   : TEXCOORD0;
   float3 Normal : TEXCOORD1;
   float3 Light1 : TEXCOORD2;
   float3 Light2 : TEXCOORD3;
   float3 Light3 : TEXCOORD4;
};
```

구조체는 HLSL에서 사용되는 일반적인 형태로 선언되었다.

샘플러(sampler)

픽셀 셰이더에서 사용하는 각각의 텍스처 맵에서 정보를 가져오기 위해서는 맵마다 sampler를 반드시 선언해주어야 한다. 앞서의 hlsl_rings() 셰이더를 상기해보자.

```
float4 lightWood;   // 나무결 색상의 밝은 부분
float4 darkWood;    // 나무결 색상의 어두운 부분
float  ringFreq;    // 나무결의 촘촘한 정도
sampler PulseTrainSampler;

float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
    float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy)) * ringFreq;
    float blendFactor = tex1D (PulseTrainSampler, scaledDistFromZAxis);
    return lerp (darkWood, lightWood, blendFactor);
}
```

이 셰이더에서는 PulseTrainSampler라는 전역 형태의 샘플러를 선언하고 이를 tex1D() 라는 내장 함수의 첫 번째 매개변수로 사용했다. HLSL의 샘플러 개념은 D3D API의 샘플러의 개념을 그대로 가져온 것으로서, 3D 그래픽 프로세서 내에서 uv 좌표에 해당하는 텍스처 텍셀의 위치를 잡는 데 필요한 텍스처 어드레싱과 필터링을 하는 곳을 의미한다. 샘플러는 주어진 셰이더에 접근하고자 하는 각각의 텍스처 맵 모두에 정의되어 있어야 하지만, 한 셰이더에서 같은 샘플러를 여러 번 사용할 수도 있다. 이러한 방법은 「ShaderX^2: DirectX 9 셰이더 프로그래밍 팁 & 트릭」에 나와있는 것과 같이 이미지 프로세싱을 하는 데 있어서 매우 일반적인 방법인데, 하나의 입력 이미지를 서로 다른 텍스처 좌표들에 여러 번 샘플링한 후에 그 결과를 필터링 커널로 전달하면 되는 것이다.

예를 들어, 다음의 셰이더는 2개의 Sobel 필터(dx, dy를 각각 필터링)를 통해 래스터라이저를 이용하여 높이 맵(Height map)을 법선 맵(Normal map)으로 변환하는 코드이다.

```
sampler InputImage;

float4 main(float2 topLeft    : TEXCOORD0, float2 left          : TEXCOORD1,
            float2 bottomLeft : TEXCOORD2, float2 top           : TEXCOORD3,
            float2 bottom     : TEXCOORD4, float2 topRight      : TEXCOORD5,
            float2 right      : TEXCOORD6, float2 bottomRight   : TEXCOORD7):
                                                                  COLOR
{
   // 8개의 각 구성을 얻어준다.
   float4 tl = tex2D(InputImage, topLeft);
   float4  l = tex2D(InputImage, left);
   float4 bl = tex2D(InputImage, bottomLeft);
   float4  t = tex2D(InputImage, top);
   float4  b = tex2D(InputImage, bottom);
   float4 tr = tex2D(InputImage, topRight);
   float4  r = tex2D(InputImage, right);
   float4 br = tex2D(InputImage, bottomRight);

   // Sobel 연산으로 dx 값을 구한다.
   //
   //           -1 0 1
   //           -2 0 2
   //           -1 0 1
```

```
    float dX = -tl.a - 2.0f*l.a - bl.a + tr.a + 2.0f*r.a + br.a;

    // dy 값을 Sobel 연산으로 구한다.
    //
    //          -1 -2 -1
    //           0  0  0
    //           1  2  1
    float dY = -tl.a - 2.0f*t.a - tr.a + bl.a + 2.0f*b.a + br.a;

    // 외적을 계산하고 다시 정규화한다.
    float4 N = float4(normalize(float3(-dX, -dY, 1)), tl.a);

    // (-1...1) 사이의 범위를 (0...1) 사이의 범위로 변환하고 반환한다.
    return N * 0.5f + 0.5f;
}
```

이 셰이더는 1개의 샘플러, Input Image만 사용했으나 tex2D() 함수를 이용하여 8번 반복 사용했다.

내장 함수

앞서 언급했던 것처럼 DirectX HLSL은 많은 내장 함수들을 가지고 있다. 수많은 내장 함수 중 수학 함수들은 코드 수를 줄이기 위해(코딩을 편하게 하기 위해) 제공되는 반면에, 위에서 언급한 tex1D()와 tex2D() 같은 종류의 것들은 데이터 샘플러를 통해 텍스처 데이터를 접근하기 위해서 제공된다.

수학 내장 함수

다음의 표에 기재된 수학 내장 함수들은 HLSL 컴파일러에 의해 어셈블리 수준의 명령어들로 바뀐다. abs()와 dot()와 같은 경우에는 하나의 어셈블리 명령으로 대응되지만, refract(), step()과 같은 대부분의 경우에는 여러 개의 어셈블리 명령어의 조합으로 대응된다. 함수들 중에는 특정 컴파일 타겟에서 지원되지 않는 것들이 있는데, ddx(), ddy(), fwidth()가 그렇다.

내장 함수	설명
abs(x)	절대값을 구한다(x의 각 성분에 대해).
acos(x)	x의 모든 성분의 아크코사인 값을 구한다. 각 성분들의 값은 [－1, 1] 사이에 있어야 한다.
all(x)	x의 모든 성분들이 0이 아닌지를 테스트한다.
any(x)	x의 성분 중 0이 아닌 것이 있는지 테스트한다.
asin(x)	x의 각 성분의 아크사인 값을 구한다. 각 성분들의 값은 [－p/2, p/2] 이내에 있어야 한다.
atan(x)	x의 아크탄젠트 값을 구한다. 반환되는 값은 [－p/2, p/2]의 범위 안에 있다.
atan2(y, x)	y/x의 아크탄젠트 값을 구한다. y와 x의 부호는 [-p, p] 범위 안에 있는 반환 값들의 사분면을 결정하는 데 쓰인다. atan2는 원점을 제외한 모든 다른 점들에 대해서도 잘 정의되어 있는데, 심지어 x가 0이고 y가 0이 아닌 상황에서도 잘 동작한다.
ceil(x)	x보다 크거나 같은 정수 값을 반환한다.
clamp(x, min, max)	[min, max] 범위를 벗어나지 않게 값을 자른다.
Clip(x)	x의 어떤 성분이라도 0부다 작으면 현재 픽셀 값을 버린다. 이것을 이용하여 x의 각 원소가 평면으로부터의 거리를 나타내게 하면 평면 클립도 구현할 수가 있다. 이것은 texkill 같은 명령어를 만들고자 할 때 사용할 수 있는 내장 함수이다.
cos(x)	x의 코사인을 구한다.
cosh(x)	x의 하이퍼볼릭 코사인(hyperbolic cosine)을 구한다.
cross(a, b)	a, b 벡터의 외적을 구한다.
D3DCOLORtoUBYTE4(x)	4D 벡터 형태인 x의 각 성분을 몇몇 하드웨어에서 지원하는 UBYTE4 스트림 형태로 바꾼다.
ddx(x)	스크린 공간 기준인 x 좌표의 편 미분계수를 구한다.
ddy(x)	스크린 공간 기준인 y 좌표의 편 미분계수를 구한다.
degrees(x)	라디안 값을 각도 값(0 360도 사이)으로 바꾼다.
determinant(m)	m행렬의 행렬식을 구한다.
distance(a, b)	a, b 사이의 거리를 구한다.
dot(a, b)	a, b 벡터의 내적을 구한다.
exp(x)	e의 x 제곱 값을 구한다.
exp2(a)	2의 x 제곱 값이다(각 원소마다).
faceforward(n, i, ng)	－n * sign(dot(i, ng)) 값을 구한다.
floor(x)	x보다 같거나 작은 것 중 가장 큰 정수 값을 구한다.

내장 함수	설명
fmod(a, b)	a/b의 부동 소수점형(실수형) 나머지를 구한다. a = i * b + f에서 I는 정수형이고, f는 x와 같은 부호를 가지고 있고, f의 절대값은 b보다는 작다.
frac(x)	X의 소숫점 부분을 구한다
frexp(x, out exp)	x의 가수와 지수를 계산한다. frexp는 exp의 결과 매개변수에 저장된 가수와 지수를 계산한다. 만일 x가 0일 때는 가수와 지수의 계산 결과를 0으로 계산한다.
fwidth(x)	abs(ddx(x))+abs(ddy(x))를 반환한다.
isfinite(x)	x가 유한이면 true, 그렇지 않으면 false를 반환한다.
isinf(x)	x가 +INF이거나 –INF이면 true를, 그렇지 않으면 false를 반환한다.
isnan(x)	x가 NAN이거나 QNAN이면 true를, 그렇지 않으면 false를 반환한다. CNAN:Net a numbor-잘못된 실수 값
ldexp(x, exp)	* 2^{exp} 값을 구한다.
len(v)	벡터 길이
length(v)	벡터 v의 길이를 구한다.
lerp(a, b, s)	s가 0이면 a를, 1이면 b를 반환하는 a와 b의 선형 보간 값인 a + s(b – a)의 결과 값을 계산한다.
log(x)	지수가 e인 x의 로그 값을 구한다. x가 음수면 indefinite 값을 반환한다. x가 0이면 +INF를 반환한다.
log10(x)	지수가 10인 X의 로그 값을 구한다. x가 음수이면 indefinite를, 0이면 +INF를 돌려준다.
log2(x)	지수가 2인 x의 로그 값을 구한다. x가 음수이면 indefinite를, 0이면 +INF를 돌려준다.
max(a, b)	a와 b 중 큰 것을 선택한다.
min(a, b)	a와 b 중 작은 것을 선택한다.
modf(x, out ip)	x 값을 x와 같은 부호를 같은 소수부와 정수부로 나눈다. 부호가 있는 소수 부분은 반환되고, 정수부는 out ip 값에 저장된다.
mul(a, b)	a와 b의 행렬 곱셈 연산을 수행한다. a가 벡터이면 행 벡터로 인식된다. b가 벡터면 열 벡터로 인식된다. a의 열과 b의 행은 반드시 같아야 한다. 결과 값은 a행×b열인 행렬이다.
normalize(v)	v/length(v)로 정규화된 벡터 v를 구한다. v의 길이가 0이면 결과는 정의되지 않는다..
pow(x, y)	xy 값을 반환한다.
radians(x)	x를 각도에서 호도(radian) 값으로 바꾼다.
reflect(i, n)	반사 벡터 v를 돌려준다. I는 입사 광선, n은 표면의 법선 벡터이라면 결과 값은 v = i – 2 * dot(i, n) * n이다.

내장 함수	설명
refract(i, n, eta)	굴절 벡터 v를 구한다(입사 광선은 I이고, 표면의 법선은 n이고, eta는 굴절 색인). eta로 주어진 n과 I 사이의 각도가 너무 크다면 refract는 (0, 0, 0)을 반환한다.
round(x)	x에 가장 가까운 정수를 찾는다.
rsqrt(x)	1 / sqrt(x)를 구한다.
saturate(x)	x의 값을 [0, 1] 사이가 되게 자른다.
sign(x)	x의 부호를 구한다. x < 0이면 -1을 돌려주고, x=0이면 0을, x > 0이면 1을 돌려준다.
sin(x)	x의 사인 값을 구한다.
sincos(x, out s, out c)	x의 사인과 코사인 값을 구한다. sin(x)는 출력 매개변수 s에 저장되고, cos(x)는 출력 매개변수 c에 저장된다.
sinh(x)	x의 하이퍼볼릭 사인을 구한다.
smoothstep(min, max, x)	x < min이라면 0을 반환하고, x > max이면 1을 반환한다. [min, max] 사이에서는 0과 1사이의 smooth Hermite 보간 값을 반환한다.
sqrt(x)	제곱근(성분별).
step(a, x)	(x = a) ? 1 : 0 값을 반환한다.
tan(x)	x의 탄젠트 값을 구한다.
tanh(x)	x에 대한 하이퍼볼릭 탄젠트 값을 구한다.
transpose(m)	행렬 m의 전치 행렬을 구한다. 입력 값이 $m_{rows} \times m_{columns}$ 차원이면 출력은 $m_{columns} \times m_{rows}$ 차원의 행렬이 될 것이다.

텍스처 샘플링 내장 함수

텍스처 데이터를 셰이더로 샘플링할 때 사용하는 16개의 텍스처 샘플링 내장 함수가 있다. 4가지 종류의 텍스처(1D, 2D, 3D, 입방체 맵)와 4가지 종류의 대상(일반적인, 순간 변화율 적용된, 투영된, 보정된)이 조합되어 16개가 된다.

내장 함수	설명
tex1D(s, t)	1차원 텍스처 룩업. s는 샘플러이다. t는 스칼라 값이다.
tex1D(s, t, ddx, ddy)	(LOD를 결정하기 위해) 순간 변화율 값을 이용한 1D 텍스처 룩업. s는 샘플러이다. t, ddx, and ddy는 스칼라이다.
tex1Dproj(s, t)	1차원 투영 텍스처 룩업. s는 샘플러이다. t는 4D 벡터이다. 텍스처 룩업을 참조하기 전에 t는 t.w로 나누어진다.

내장 함수	설명
tex1Dbias(s, t)	보정(bias)된 1차원 텍스처 룩업. s는 샘플러이다. t는 4D 벡터이다. 밉(mip) 단계는 텍스처 룩업을 참조하기 전에 t.w에 의해 조정된다.
tex2D(s, t)	2D 텍스처 룩업. s는 샘플러이다. t는 2차원 텍스처 좌표이다.
tex2D(s, t, ddx, ddy)	(LOD를 결정하기 위해) 순간 변화율을 이용한 텍스처 룩업. s는 샘플러이다. t, ddx와 ddy는 2차원 벡터들이다.
tex2Dproj(s, t)	2차원 투영 텍스처 룩업. s는 샘플러이다. t는 4차원 벡터이다. t는 룩업을 참조하기 전에(t.w가 1이 될 수 있게) t.w로 나누어진다.
tex2Dbias(s, t)	보정된 2차원 텍스처 룩업. s는 샘플러이다. t는 4차원 벡터이다. 밉(mip) 단계는 텍스처 룩업을 참조하기 전에 t.w에 의해 조정된다.
tex3D(s, t)	3차원 볼륨 텍스처 룩업. s는 샘플러이다. t는 3차원 텍스처 좌표이다.
tex3D(s, t, ddx, ddy)	순간 변화율(derivatives)이 적용된 3차원 볼륨 텍스처 룩업. s는 샘플러이다. t, ddx, ddy는 3차원 벡터이다.
tex3Dproj(s, t)	3차원 투영 볼륨 텍스처의 값을 룩업. s는 샘플러이다. t는 4차원 벡터이다. t는 텍스처를 참조하기 전에 t.w 값으로 나누어진다.
tex3Dbias(s, t)	보정된 3차원 텍스처 값을 룩업. s는 샘플러이다. t는 4차원 벡터이다. 밉(mip) 단계는 텍스처를 참조하기 전에 t.w에 의해 조정된다.
texCUBE(s, t)	입방체 텍스처 맵의 값을 룩업. s는 샘플러이다. t는 3차원 텍스처 좌표이다.
texCUBE(s, t, ddx, ddy)	순간 변화율을 적용시켜 입방체 텍스처 맵을 룩업. s는 샘플러이다. t, ddx, ddy는 3차원 벡터이다.
texCUBEproj(s, t)	투영 입방체 맵 룩업. s는 샘플러이고, t는 4D 벡터이다. t는 맵 룩업에서 가져오기 전에 t.w로 나누어진다.
texCUBEbias(s, t)	보정된 입방체 맵 텍스처 값을 룩업. s는 샘플러이다. t는 4차원 벡터이다. 밉(mip) 레벨은 텍스처 값을 참조하기 전에 t.w의 값으로 조정된다.

텍스처를 샘플링할 때 tex1D(), tex2D(), tex3D()와 texCUBE() 내장 함수가 일반적으로 널리 사용된다. ddx 또는 ddy 매개변수를 연산해야 하는 텍스처는 ddx(), ddy()와 같은 수학 내장 함수로 미리 계산된 값을 인자로 받아서 텍스처 LOD(level of detail)를 계산한다. 이는 픽셀 셰이더를 작성할 때 유용하지만, ps_2_0 또는 그 보다 낮은 컴파일 타겟에서는 지원하지 않는다.

Tex*proj () 내장 함수는 텍스처를 투영(projective)으로 읽어 들이고자 할 때 사용한다. 이 때 텍스처를 샘플링할 좌표의 각 성분은 그 성분들 중 마지막 성분으로 나눗셈을 한 이후에 쓰이게 되므로 투영 효과를 나타낼 수 있다. 이 중에서 가장 많이 사용되

는 것은 tex2Dproj()로써, 투사식 그림자 맵(Perspective Shadow map)을 구현하거나 유사한 효과를 내기 위해 사용된다.

Tex*bias 내장 함수는 텍스처 샘플링에 보정 값을 주기 위해 사용하며, 보정 값은 픽셀 단위로 계산할 수 있다. 이것을 이용하면 색상 번짐(over-blurred) 텍스처 효과를 낼 수 있다. 예를 들어, 「ShaderX^2: DirectX 9 셰이더 프로그래밍 팁 & 트릭」에 언급되어 있는 것과 같이 Radeon 9700의 Animusic Pipedream 데모에 등장하는 공들에는 모션 블러 효과가 적용되어 있는데, 바로 texCUBEbias() 내장 함수를 통해 큐빅 환경 맵을 접근하는 것이다.

```
...
    // 확장 값으로 반사를 블러시킨다.
   float3 vCubeLookup = vReflection + i.Pos/fEnvMapRadius;
   float4 cReflection = texCUBEbias(tCubeEnv, float4(vCubeLookup,
           fBlur * fTextureBlur)) * vReflectionColor;
...
```

위의 코드에서 texCUBEbias()에서 두 번째 매개변수 t로 float4(vCubeLookup.x, vCubeLookup.y, vCubeLookup.z, fBlur*fTextureBlur)가 쓰였는데, 이 fBlur*fTextureBlur가 t.w 값으로 texCUBEbias에서 보정 값으로 사용된다(texCUBE() 함수에서 t.w는 텍스처 값을 참조하기 전 밉 단계를 조정하기 위해 쓰인다).

이제 셰이더 언어의 구조에 대해서 어느 정도 알아보았으니 DirectX 9의 HLSL 셰이더에서 어떻게 데이터를 입력하고 출력하는지에 대하여 간단하게 알아보도록 하자.

셰이더 입력

정점 셰이더와 픽셀 셰이더는 varying과 uniform의 두 가지 입력 데이터형을 가진다. varying 입력은 각 셰이더가 실행될 때마다 달리 적용되는 데이터이다. 정점 셰이더의 경우 varying 데이터(위치, 법선 등)는 정점 스트림에서 들어온다. uniform 데이터(개체의 색, 세계 변환 등)는 셰이더가 실행될 때마다 동일하게 적용되는 상수이다. 어셈블리 모델로 보면 정점과 픽셀 셰이더에서 uniform 데이터는 상수 레지스터에 들어있는 값으로 볼 수 있고, varying 데이터는 v레지스터나 t레지스터에 들어있는 값으로 볼 수 있다.

uniform 입력

uniform 데이터는 HLSL에서 2가지 형태로 구분된다. 가장 일반적인 방법은 전역 변수로 정점 또는 픽셀 셰이더 안에서 사용하는 방법이다. 셰이더 안에서 전역 변수를 사용하려면 셰이더에 필요한 uniform 변수의 목록에 변수를 추가하면 된다. 다른 방법으로는 맨 위 단계 셰이더 함수의 입력 매개변수를 uniform으로 설정하는 것이 있다. 이렇게 설정하면 이 함수를 사용한 셰이더가 uniform 변수의 목록에 변수를 추가한다. 다음의 코드에서 이 두 가지 방법이 사용되었다.

```
// 전역 uniform 변수를 정의한다.
// UniformGlobal이라는 이름으로 상수 테이블에 표시된다.
float4 UniformGlobal;

// uniform이면 매개변수를 선언한다.
// '$UniformParam'이라는 이름으로 상수 테이블에 표시된다.
float4 main( uniform float4 UniformParam ) : POSITION
{
   return UniformGlobal * UniformParam;
}
```

셰이더에 의해서 사용된 uniform 변수는 상수 테이블을 통하여 응용프로그램에 전달된다. 상수 테이블은 셰이더의 실행에 앞서 반드시 상수 레지스터에 입력되어야 하는 셰이더가 사용할 uniform 변수의 사용 여부를 정의한 심볼 테이블이다.

☑ **Note**

uniform 입력 함수 매개변수는 전역 변수와는 달리 $를 앞에 붙인 형태로 상수 테이블에 나타난다. $는 같은 이름을 가진 지역 변수(local variable)와 전역 변수(global variable)를 구별하기 위해서 사용하는 것이다.

상수 테이블은 셰이더가 사용한 모든 uniform 변수의 상수 레지스터의 위치를 저장하고 있다. 이 테이블은 (각 테이블 입력 값마다) 변수의 형에 관한 정보와 특별히 명시한 경우에 한하여 기본 값을 담고 있다. 다음은 상수 테이블이 출력되었을 경우의 예이다. 상수 테이블은 컴파일러에 의해 만들어진 후 이진 형태로 저장된다. 실행 시 테이블의 내용을 해석하기 위한 API들은 "D3DX Effect를 사용하지 않고 엔진에 적용하기" 섹션에서 다루어질 것이다.

다음은 셰이더 예제를 fxc.exe를 이용하여 텍스트 형태로 출력한 것이다.

```
//
//   Generated by Microsoft (R) D3DX9 Shader Compiler
//
//   Source: hemisphere.fx
//   Flags: /E:VS /T:vs_1_1
//

// Registers:
//
//      Name             Reg    Size
//      ------------     -----  ----
//      Projection       c0        4
//      WorldView        c4        3
//      DirFromLight     c7        1
//      DirFromSky       c8        1
//      $bHemi           c18       1
//      $bDiff           c19       1
//      $bSpec           c20       1
//
//
// Default values:
//
//      DirFromLight
//          c7   = { 0.577, -0.577, 0.577, 0 };
//
//      DirFromSky
//          c8   = { 0, -1, 0, 0 };
```

varying 입력

varing 데이터는 최상위 함수의 입력 매개변수에 의해 설정된다. 모든 최상위 수준 셰이더의 입력은 의미구조를 사용하거나 셰이더의 실행을 위한 상수 값임을 알려주기 위해서 위하여 uniform 키워드를 사용하여 정의되어야 한다. 만약 최상위 셰이더에서 입력 의미구조로 표시되지 않았거나 uniform 키워드가 아닌 경우 셰이더 컴파일은 실패할 것이다.

입력 의미구조는 주어진 셰이더의 입력과 그래픽 파이프라인의 전 단계의 출력을 연결하기 위해 사용된 이름이다. 예를 들어, POSITION0은 정점 셰이더에서 입력 데이터를 연결된 정점 버퍼에서 가져오기 위하여 사용된다.

픽셀과 정점 셰이더는 각각의 셰이더 유닛으로 공급되는 그래픽 파이프라인이 다르기 때문에 다른 종류의 입력 의미구조들을 가진다. 정점 셰이더의 입력 의미구조는 정점 버퍼에서 읽어들인 각 정점의 정보를 정점 셰이더에서 쓸 수 있게 정점의 구조를 알려주는 역할을 한다(위치나 법선 벡터, 텍스처 좌표, 색상 값, 탄젠트 값, 종법선 등이 그렇다). 이런 입력 의미구조들은 정점 버퍼에 있는 정점의 구조를 기술하기 위해 사용된 D3DDECLUSAGE 열거형과 UsageIndex의 조합과 1대 1로 연결된다.

픽셀 셰이더 입력 의미구조는 래스터화(rasterization) 유닛에 의해 각 픽셀에 제공된 정보에 기술한다. 이 데이터는 현재의 정점과 각각의 정점 셰이더의 출력의 보간으로 생성된다. 이러한 기본 픽셀 셰이더 입력 의미구조는 입력 색과 텍스처 좌표계 정보를 입력 매개변수로 저장한다.

입력 의미구조는 두 가지 방법으로 셰이더 입력과 연결될 수 있다. 첫 번째 방법은 콜론(:)을 추가한 뒤 입력 의미구조의 이름으로 입력 매개변수 선언을 하는 방법이다. 두 번째 방법은 입력 의미구조를 각각의 입력 구조체의 요소로 입력 구조체를 정의하는 것이다. 이러한 형태의 방법은 모두 이 장에서뿐만 아니라, 이 책에서 사용된 모든 예제에서 사용되었다.

다음은 입력 의미구조의 예이다.

```
// 의미구조 바인딩으로 입력 구조체를 선언한다.
struct InStruct
{
    float4 Pos1 : POSITION1
};

// 위치 값을 저장하기 위한 Pos 변수를 선언한다.
float4 main( float4 Pos : POSITION0, InStruct In ) : POSITION
{
    return Pos * In.Pos1;
}
```

```
// 보간된 COLOR0 값을 담기 위한 Col 변수를 선언한다.
float4 mainPS( float4 Col : COLOR0 ) : COLOR
{
    return Col;
}
```

다음은 정점 셰이더 입력 의미구조들이다.

의미구조	설명
POSITIONn	위치
BLENDWEIGHTn	블랜드 가중치
BLENDINDICESn	블랜드 인덱스들
NORMALn	법선 벡터
PSIZEn	포인트 크기(포인트 스프라이트에서 사용된다.)
COLORn	색상
TEXCOORDn	텍스처 좌표
TANGENTn	접선
BINORMALn	종법선
TESSFACTORn	분할 매개변수

다음은 픽셀 셰이더 입력 의미구조들이다.

의미구조	설명
COLORn	색상
TEXCOORDn	텍스처 좌표

n은 PSIZE0, DIFFUSE1 등과 같이 정수가 들어갈 자리를 의미한다.

셰이더 출력

정점과 픽셀 셰이더의 입력과 출력은 파이프라인 방식으로 처리된다. 이것은 한 단계의 출력 결과 값이 다음 단계의 입력 값으로 사용된다는 의미이다. 그 예로, 정점 셰이더의 출력 결과는 픽셀 셰이더의 입력 데이터를 생성하기 위한 래스터라이저의 보

간을 위한 결과와 연결되어 있다. 픽셀 셰이더 결과는 각각의 렌더링 타겟에 쓰여진 알파 블렌딩 유닛 또는 깊이 버퍼에 쓰여진 깊이 값에 쓰여진다.

정점 셰이더 출력물은 픽셀 셰이더와 래스터라이저를 연결하기 위해 사용된다. POSITION은 정점 셰이더의 래스터라이저로 들어가서 보간된 결과 값을 만들어내지만 픽셀 셰이더에서는 등장하지 않는다. TEXCOORDn과 COLORn은 픽셀 셰이더에서 보간에 의해 생성된다.

픽셀 셰이더의 출력은 해당 렌더링 대상의 출력 색상을 정의한다. 픽셀 셰이더에서 출력한 컬러 값은 출력될 렌더 타겟이 수정되는 방법을 결정하는 알파 블랜드 스테이지와 연결된다. DEPTH 출력 의미구조는 현재 래스터 위치의 깊이 값을 바꾸는 데 사용한다.

Note

DEPTH와 다중 렌더링 타겟은 (MRT로 알려진) 몇몇 셰이더 모델에서만 지원된다.

출력 의미구조를 위한 문법은 입력 의미구조의 것과 다르지 않다. 의미구조들은 출력 매개변수로 직접 선언되거나 함수의 출력 매개변수 또는 반환 값으로 사용될 구조체에서 정의할 수 있다.

다음은 정점 셰이더 출력 의미구조들이다.

의미구조	설명
POSITION	위치
PSIZE	포인트 크기
FOG	정점 포그
COLORn	색상(예: COLOR0)
TEXCOORDn	텍스처 좌표(예: TEXCOORD0)

다음은 픽셀 셰이더 출력 의미구조들이다.

의미구조	설명
COLORn	색상 렌더 타겟 n
DEPTH	깊이 값

n은 바꿀 수 있는 정수이다(예: TEXCOORD3, COLOR0).

다음의 코드들은 HLSL에서 어떻게 다양한 방법으로 데이터가 출력되는지 보여준다.

```
// 의미구조 바인딩으로 출력 구조체를 선언한다.
struct OutStruct
{
float2 Tex2 : TEXCOORD2
};

// TEXCOORD0 데이터를 담고있는 Tex0 출력 매개변수들
float4 main(out float2 Tex0 : TEXCOORD0, out OutStruct Out ) : POSITION
{
    Tex0 = float2(1.0, 0.0);
    Out.Tex2 = float2(0.1, 0.2);
    return float4(0.5, 0.5, 0.5, 1);
}
// 보간된 COLOR0 값을 담기 위한 Col 변수를 선언
float4 mainPS( out float4 Col1 : COLOR1) : COLOR
{
    // 출력 매개변수를 사용하여 렌더 타겟1에 값을 쓴다.
    Col1 = float4(0.0, 0.0, 0.0, 0.0);

    // return형으로 선언된 값을 사용하여 렌더 타겟0에 쓴다.
    return float4(1.0, 0.9722, 0.3333334, 0);
}

struct PS_OUT
{
    float4 Color: COLOR;
    float  Depth: DEPTH;
};


//
// 픽셀 셰이더에서 출력하는 세 가지 다른 방법들
//

 PS_OUT PSFunc1() { ... }

void PSFunc2(out float4 Color : COLOR,
             out float Depth : DEPTH)
```

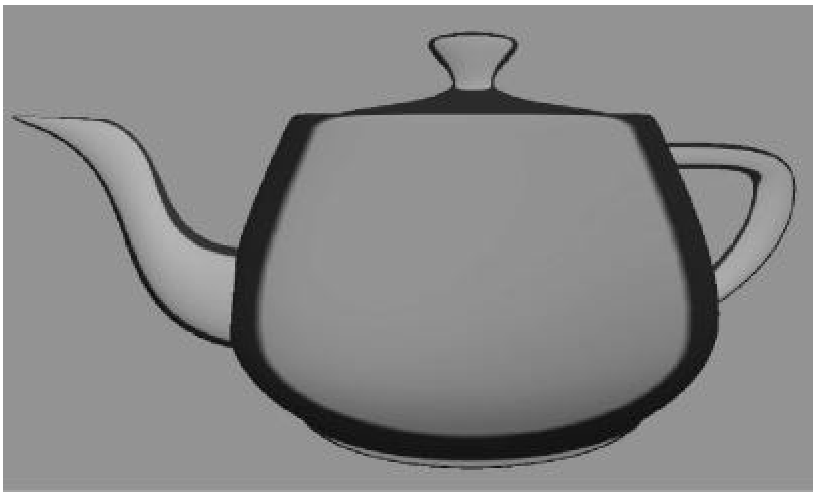
```
{
   ...
}

void PSFunc3(out  PS_OUT Out )
{
   ...
}
```

셰이더의 적용 예

여기까지 언어 그 자체와 입력과 출력에 관한 그래픽스 파이프라인이 어떻게 연결되는지에 대해서 알아보았다. 이제 NPR Metallic이라는 셰이더 예제를 살펴보자. 이 예제는 셀 애니메이션에서 보여지는 렌더링과 매우 유사하게 보인다. [그림 1-2]를 보라.

이러한 효과는 이 책의 후반부에 있는 ATI Developer Relations 사이트(http://www.ati.com/developer)를 통해 알려진 '렌더몽키를 사용한 셰이더 개발'에서 다루고 있다.



[그림 1-2] NPR Metallic

먼저, HLSL로 작성된 NPR Metallic 정점 셰이더에 대해서 살펴보자.

```
float4x4 view_proj_matrix;

float4 view_position;
float4 light0;
float4 light1;
float4 light2;

struct VS_OUTPUT
{
    float4 Pos      : POSITION;
    float3 View     : TEXCOORD0;
    float3 Normal   : TEXCOORD1;
    float3 Light1   : TEXCOORD2;
    float3 Light2   : TEXCOORD3;
    float3 Light3   : TEXCOORD4;
};

VS_OUTPUT main(float4 inPos    : POSITION,
               float3 inNorm   : NORMAL)
{
    VS_OUTPUT Out = (VS_OUTPUT) 0;

    // 변환된 정점 위치를 출력한다.
    Out.Pos = mul(view_proj_matrix, inPos);

    Out.Normal = inNorm;

    // 시야 벡터를 계산한다.
    Out.View = normalize(view_position - inPos);

    // 현재의 정점 위치로부터 각 광원을 향하는 세 개의 벡터를 구한다.
    Out.Light1 = normalize(light0 - inPos);   // 광원 1
    Out.Light2 = normalize(light1 - inPos);   // 광원 2
    Out.Light3 = normalize(light2 - inPos);   // 광원 3

    return Out;
}
```

처음에 볼 수 있는 것은 view_proj_matrix, view_position, light0, light1, light2 행렬과 전역에서 사용되는 상수이다. 이들은 모두 외부에서 API로 설정이 가능하고, 셰이더 자체에서도 변형이 가능하다.

이러한 전역 변수들 이외에도 main 함수와 같은 결과 값을 가지는 VS_OUTPUT이라는 구조체의 선언을 볼 수 있는데, 정점 셰이더가 한 개의 4D 위치 값과 5개의 3D 텍스처 좌표 값을 반환한 것을 나타낸다.

메인 함수를 보면, 정점 셰이더가 4D 벡터를 입력 위치로, 3D 벡터를 일반적인 입력 2D 벡터를 텍스처 좌표계로 사용하는 것을 알 수 있다. 입력받은 위치를 담고있는 inPos는 mul() 함수를 사용하여 view_proj_matrix에 의해 변환되는 반면에, 법선 벡터 inNorm은 수정되지 않은 채로 출력된다.

마지막으로, 오브젝트 공간에의 정점 위치에서 광원을 향하는 3D 벡터들과 시야 위치들이 모두 계산된다. 이 3D 벡터들은 normalize()를 통해 모두 정규화된다. 이 정규화된 3D 벡터들은 3D 텍스처 좌표계 형식으로 정점 셰이더로부터 출력되는데, 이것들은 후에 폴리곤 단위로 보간될 것이다.

컴파일 타겟과 어셈블리 모델에 대한 이전의 논의들을 보충하기 위해서 이 셰이더들을 컴파일해보고, 어셈블리 출력들을 확인해보자. 먼저 위의 코드들을 NPRMetallic.vhl이라는 파일로 저장한 다음, 커맨드라인에서 다음의 명령을 실행해보자.

```
fxc -nologo -T vs_1_1 -Fc -Vd NPRMetallic.vhl
```

이 정점 셰이더는 분기 제어 같은 것이 필요가 없기 때문에 vs_1_1을 컴파일 타겟으로 설정하였다. 또한, 코드 파일 생성과 검증을 생략하도록 플래그를 설정하였다. 다음의 코드는 이러한 결과로 생성된 코드이다.

```
// Parameters:
//     float4 light0;
//     float4 light1;
//     float4 light2;
//     float4 view_position;
//     float4x4 view_proj_matrix;
//
// Registers:
//     Name                Reg    Size
//     ----------------    -----  ----
//     view_proj_matrixc0  4
//     view_position    c4  1
//     light1   c5  1
//     light2   c6  1
//     light0   c7  1

    vs_1_1
    dcl_position v0
    dcl_normal v1
    mul r0, v0.x, c0
    mad r2, v0.y, c1, r0
    mad r4, v0.z, c2, r2
    mad oPos, v0.w, c3, r4
    add r1, -v0, c4
    dp4 r1.w, r1, r1
    rsq r1.w, r1.w
    mul oT0.xyz, r1, r1.w
    add r8, -v0, c7
    dp4 r8.w, r8, r8
    rsq r8.w, r8.w
    mul oT2.xyz, r8, r8.w
    add r3, -v0, c5
    add r10, -v0, c6
    dp4 r3.w, r3, r3
    rsq r3.w, r3.w
    mul oT3.xyz, r3, r3.w
    dp4 r10.w, r10, r10
    rsq r10.w, r10.w
    mul oT4.xyz, r10, r10.w
    mov oT1.xyz, v1
```

코드 파일의 첫 부분에서 이 정점 셰이더의 매개변수를 볼 수 있다. 여기서 이 셰이더가 주어진 응용프로그램에서 제대로 동작하기 위한 API로 설정해주어야 하는 전역 변수가 있다. 다음으로 21개의 어셈블리 명령으로 구성된 셰이더 코드가 있다. 이 코드들을 모두 이해할 필요는 없지만, 셰이더의 main 함수의 입력이 되는 POSITION과 NORMAL의 결과 함수인 dcl_position과 dcl_normal 구문에 대해서는 명확한 이해가 필요하다. 또한 최종 결과가 저장되는 oPos, oT0, oT1, oT2, oT3과 oT4 레지스터에 대해서도 이해를 해야 한다. 이는 각각의 멤버에 부여되는 구조체를 이루는 함수를 만들어 낸다. 꼭 필요하진 않지만, 개발 시 좀 더 최적화된 어셈블리 코드를 생성하는 HLSL을 작성하고자 한다면 fxc의 사용법을 이해하는 것이 좋다.

이제 정점들을 절단 공간으로 변환시키고 폴리곤 사이에서 보간될 출력 값들을 정의했다. 이제 이 출력 값들을 보간하여 사용할 픽셀 셰이더를 보도록 하자.

다음은 NPR Metallic 픽셀 셰이더이다.

```
float4 Material;

sampler Outline;

float4 main(float3 View:   TEXCOORD0,
            float3 Normal: TEXCOORD1,
            float3 Light1: TEXCOORD2,
            float3 Light2: TEXCOORD3,
            float3 Light3: TEXCOORD4 ) : COLOR
{
   // 법선 벡터의 정규화
   float3 norm = normalize (Normal);

   float4 outline = tex1D(Outline, 1 - dot (norm, normalize(View)));

   float lighting =(dot (normalize (Light1), norm) * 0.5 + 0.5) +
                   (dot (normalize (Light2), norm) * 0.5 + 0.5) +
                   (dot (normalize (Light3), norm) * 0.5 + 0.5);

   return outline * Material * lighting;
}
```

이전과 같이 셰이더에서 사용되는 몇몇 변수들이 전역 변수로 선언되었다. 이번에는 렌더링을 하기 위해 재질의 값을 정의하기 위한 4D 벡터인 Material, 그리고 대상의 외형을 위한 특별한 텍스처를 접근하기 위해 사용하는 샘플러 Outline 등이 전역 변수로 사용된다. 정점 셰이더에서 계산된 다섯 개의 3D 텍스처 좌표계는 픽셀 셰이더의 main 함수에서 입력으로 사용되고, 시야 벡터, 법선 벡터, 3개의 빛 벡터를 정의한다

텍스처 좌표계가 폴리곤들 사이에서 선형으로 보간되므로, 특정 픽셀에서 정규화되지 않은 값들이 생길 수 있다. 그래서 셰이더는 보간된 법선 벡터를 normalize()를 사용하여 다시 정규화한다. 외곽선 텍스처는 시야 벡터와 법선 벡터와의 내적 계산을 통해 샘플링된다. 광원은 스케일 및 조정 과정을 거친 법선 벡터와 광원의 정규화된 벡터의 내적들을 합산하여 계산한다.

픽셀 셰이더의 마지막 줄은 outline * Material * lighting의 값을 반환한다. 처음의 두 벡터의 경우에는 4D 벡터이며, 나머지 하나는 스칼라이다. 이전에 변환 유형에 대하여 이야기하였듯이, 스칼라와 벡터의 곱은 모든 성분의 값이 해당 스칼라와 같은 벡터와의 곱을 말한다. 이는 다음의 두 표현이 동일하다는 것을 의미한다.

```
return outline * Material * lighting;
return outline * Material * float4(lighting, lighting, lighting, lighting);
```

따라서 이 수행 결과는 모든 채널에 스칼라 lighting을 곱한 것이며, 최종 결과는 [그림 1-2]와 같다.

이제 NPR Metallic 정점 셰이더에서 했던 것처럼 다음의 명령어를 이용하여 코드 파일을 생성해보자.

```
fxc -nologo -T ps_2_0 -Fc -Vd NPRMetallic.phl
```

ps_2_0을 컴파일할 때와 동일한 옵션을 사용하여 컴파일한다. 그 결과로 나온 셰이더는 다음과 같다.

```
// 매개변수들
//     float4 Material;
//     sampler Outline;
//
// 레지스터들
//     Name Reg Size
//     ----------- ----- ----
//     Material c0  1
//     Outline  s0  1

    ps_2_0
    def c1, 1, 0, 0, 0.5
    dcl t0.xyz
    dcl t1.xyz
    dcl t2.xyz
    dcl t3.xyz
    dcl t4.xyz
    dcl_2d s0
```

```
dp3 r0.w, t1, t1
rsq r2.w, r0.w
mul r9.xyz, r2.w, t1
dp3 r9.w, t0, t0
rsq r9.w, r9.w
mul r4.xyz, r9.w, t0
dp3 r9.w, r9, r4
add r11.xy, -r9.w, c1.x
texld r6, r11, s0
dp3 r9.w, t2, t2
rsq r9.w, r9.w
mul r1.xyz, r9.w, t2
dp3 r9.w, r1, r9
mad r9.w, r9.w, c1.w, c1.w
dp3 r8.w, t3, t3
rsq r10.w, r8.w
mul r5.xyz, r10.w, t3
dp3 r0.w, r5, r9
mad r9.w, r0.w, c1.w, r9.w
add r9.w, r9.w, c1.w
dp3 r2.w, t4, t4
rsq r11.w, r2.w
mul r1.xyz, r11.w, t4
dp3 r8.w, r1, r9
mad r10.w, r8.w, c1.w, r9.w
add r5.w, r10.w, c1.w
mul r6, r6, r5.w
mul r0, r6, c0
mov oC0, r0
```

앞에서와 같이 변수(이 경우에는 상수 Material, 샘플러 Outline)가 파일의 제일 앞부분에 기술되어 있다. 이것은 셰이더를 제대로 적용하기 위하여 반드시 응용프로그램에서 API를 통하여 설정되어야 한다

ps_2_0 명령어 다음으로 특정 상수를 저장하고 있는 def 명령어를 볼 수 있다. 이 def 명령어는 이후의 ALU 연산에서 사용될 상수를 정의하는 실제 어셈블리 명령어 스트림에서 볼 수 있는 실행 비용이 들지 않는 명령어이다. 이런 종류의 상수 정의는 보통 HLSL 셰이더에서 직접 숫자를 명시한 리터럴 값의 결과로 생성되는데 다음의 NPR Metallic 픽셀 셰이더에서도 볼 수 있다.

```
...
1 - dot (norm, normalize(View)
...
dot (normalize (Light1), norm) * 0.5 + 0.5
...
```

상수 정의를 한 다음 dcl t*n*.xyz의 형식으로 다섯 개의 3D 텍스처 좌표를 정의하였다. 이것들은 HLSL 셰이더에서 main 함수의 입력 의미구조로 정의된 부분이 어셈블리어로 번역된 결과물이다. 텍스처 좌표계를 정의한 다음에 dc1_2d s0 명령어로 샘플러를 선언하였다. 이 명령어는 샘플러0이 2D 텍스처와 연결되어야 한다는 것을 의미한다. HLSL 셰이더에서는 주로 tex1D 내장 함수가 사용되었기 때문에 이상해 보인다. 이러한 불일치가 존재하는 것은 Direct3D API 또는 셰이더 어셈블리 언어에는 1D 텍스처 같은 것이 존재하지 않기 때문이다. tex1D()는 HLSL 컴파일러에게 '텍스처 좌표계의 한 성분만을 사용하도록 사용하겠다'는 것을 알려주는 역할을 할 뿐이다.

여기까지 HLSL과 어셈블리 사이의 관계에 대해서 어느 정도 익숙해졌을 것이라 생각된다. 다음에는 최고의 HLSL을 만들기 위한 최적화 방안에 대해서 이야기하여 보자.

최적화

DirectX HLSL 컴파일러는 훌륭한 최적화기를 내장하고 있지만, HLSL 작성자들이 개발 주기를 조금이라도 더 줄이기 위해 할 수 있는 것들은 있다. 하지만 이것들은 장기간에 걸친 학문적인 과제로, HLSL을 사용한 구 1.*x* 모델과 큰 차이를 보일는지는 알 수 없다.

셰이더를 빠르게 최적화시키기 위해서 기억해두어야 할 점은, 컴파일러는 컴파일러에게 요구한 대로 구현해주기 위해 존재한다는 것이다. 다시 말해서 여러분이 어떤 수학적인 연산을 한다거나 출력 성분의 특정 값을 얻기 위해서 셰이더를 짠다면, 단지 지시한 연산만을 수행해야 한다. 컴파일러는 사용하지 않는 코드들을 잘 찾아내서 제거할 수는 있지만, 셰이더가 넘겨주는 값이 어떻게 쓰일지 까지는 스스로 알 수 없다.

예를 들어 픽셀 셰이더가 두 번째 텍스처 좌표를 사용하지 않는다고 하자. 정점 셰이더는 그 값들은 계산하지 않을 수도 있겠지만 HLSL 컴파일러가 정점 셰이더를 컴파

일할 때 이런 사항들을 알 방법은 전혀 없다. 또한 nx1 함수를 통해 샘플러에서 텍스처 룩업을 할 수 있겠지만, tex2D() 함수를 쓰고 있었다면 HLSL 컴파일러는 두 번째 텍스처 좌표 값이 전혀 사용되지 않는다 하더라도 두 번째 텍스처 좌표를 계산할 것이다. 컴파일러는 퀄리티와 성능 간의 조율에 대해서는 고려하지 않으며, 요구하는 작업만을 정확하게 수행하는 어셈블리 프로그램을 만들기 위하여 디자인되었다.

또 한 가지 셰이더의 성능 향상을 위해 매우 중요한 것은, 값이 바뀌어야 할 곳만 계산하여 바꾸는지를 확인하는 것이다. 만약 픽셀 당 계산을 정점 당 계산으로 대체할 수 있다면 그렇게 하라. 이 최적화 방법은 종종 이런 종류의 연산들에서 최대의 효과를 거두어들인다. 이와 같은 최적화 방법은 uniform인 값(셰이더의 실행이 끝날 때까지 변하지 않는 연산)들에도 같이 적용된다. 이것의 예로 주변 색을 해당 오브젝트의 재질에 설정된 주변광으로 미리 곱해 두어서 그 결과물을 사용해서 픽셀이나 정점마다 변하지도 않는 값을 계속 계산하지 않게 하는 것을 들 수 있다.

다음 섹션에서는 지금까지 살펴본 언어의 특징들이 어셈블리 언어에서 적용되는 방식을 살펴볼 것이다. 정점이나 픽셀 셰이더를 짜는 방법을 이해하는 데에는 필요하지 않지만, 어셈블리 모델의 기본적인 효율성을 이해하는 데에는 도움이 될 것이다. 작고 효율적인 셰이더를 만들고 싶다면 어셈블리의 주요한 특징들을 이해하는 것은 필수이다.

행렬 데이터형의 사용법

HLSL이 C 표준과 크게 다른 점은 벡터와 행렬의 데이터형에 있다. 이러한 데이터형은 코드의 작성을 쉽게 할 수 있도록 하며 내장 함수들이 정확하게 동작할 수 있도록 할 뿐만 아니라, 이러한 데이터형을 정확히 사용해야 더 좋은 코드를 생성할 수 있다. 벡터 데이터형의 사용은 컴파일러가 벡터에 관련된 명령을 쉽게 처리할 수 있도록 해준다. 컴파일러는 가능한 한 자동적으로 벡터화된 스칼라 연산을 수행하지만(앞서 언급했듯이 벡터 a와 스칼라 b의 곱 a×b는 내부적으로 다음과 같이 바뀌게 된다. a×float4(b,b,b,b)), 일반적으로 최고의 성능을 내기 위해서는 HLSL 코드를 벡터형으로 작성하는 것이 좋다.

셰이더를 행렬 대신에 벡터의 배열로 적용하고자 한다면, 행렬을 행렬형 데이터로 저장하는 것을 추천한다. 행렬형의 데이터를 사용하면 컴파일러는 행렬이 사용되는 형태에 따라 내장된 행렬을 각각의 행과 열의 순서로 저장한다. 이러한 최적화는 각각

의 픽셀 또는 정점 셰이더를 생성하는 데 매우 유용하다. 앞서 언급했던 것처럼 입력 행렬의 경우에, 컴파일러는 기본적으로 행을 기준으로 하여 행렬을 저장한다.

정수 데이터형의 사용법

HLSL을 쓸 때 int 데이터형을 제대로 이해하고 사용하는 것이 중요하다. int 데이터형을 사용되지 않아야 할 곳에서 사용하면 컴파일 시 쓸모 없는 어셈블리 명령어들이 매우 쉽게 생성될 수 있기 때문이다. int 데이터형은 상대적인 주소 값을 효율적으로 참조하기 위해서 생겨났다. float 데이터형을 버림 없이 상대 주소를 참조하기 위해서 사용하면 정확하지 못한 결과가 나올 수 있다. 예를 들어 2.5를 float4×4 행렬을 참조하기 위한 인덱스로 사용한다면 2로 버림하여 행렬 2를 참조하지 않고 행렬 2의 절반에서 행렬 3의 절반까지의 값이 해당 행렬 값으로 참조될 것이다. 이 문제를 수정하기 위해서 모든 floats들은 각 원소의 크기에 곱해지기 전에 반올림되어야 한다. 이것은 매우 비용이 많이 드는 연산인데 C 언어에서 사용되는 것과 동일한 결과를 나타내는 라운딩 규칙은 기존의 명령어들로는 쉽게 구현되지 않기 때문이다.

이런 예기치 않은 결과를 내지 않기 위해서 정수 값이라는 것을 명시하려는 목적으로 int 데이터형이 추가되었다. 입력 데이터를 부동 소수점 데이터로 다루지 않는 이상 사용할 모든 입력 데이터는 int로 정의해야 한다. 예를 들어 정점 스트림으로부터 읽어온 행렬 팔레트 인덱스들은 int로 표시되어야 한다. 입력 값을 int로 정의하면, 정수형 값으로 사용하기 위한 라운딩 연산이 필요하지 않으며 그로 인한 비용도 거의 들지 않는다. 여기서 입력 값을 int로 정의해두지 않는다면 셰이더는 기대하던 연산 작업을 수행하지 않을 것이다. 만약 float를 int로 변환하거나 float를 주소를 참조하기 위해서 사용한다면 생략 연산이 일어날 것이다. int형이 아닌 중간 값들을 int로 변환하면 생략 연산을 하게 되어 실행에 부하를 줄 것이다.

다음은 int 인덱스와 float 인덱스로 생성한 것을 비교한 것이다.

```
OutPos = mul(Pos, WorldArray[Index]);

// float형으로 선언된 인덱스          // int형으로 선언된 인덱스
frc r0.w, r1.w                       mul r0.w, c60.x, r1.w
frc r0.w, r1.w                       mul r0.w, c60.x, r1.w
```

```
add r2.w, -r0.w, r1.w              mova a0.x, r0.w
mul r9.w, r2.w, c61.x              m4x4 oPos, v0, c0[a0.x]
mova a0.x, r9.w
m4x4 oPos, v0, c0[a0.x]
```

조건 분기와 성능

대부분의 기존 정점 및 픽셀 셰이더 하드웨어들은 조건 분기를 지원하지 않는다. 그 하드웨어들은 셰이더가 순차적으로 실행되고 각각의 명령어를 한 번에 하나씩 실행하도록 설계된 것이다. 새로운 하드웨어들은 제한적으로 다음 몇몇 조건 분기들을 지원한다(정적 분기(static branching), 조건 명령(predicated instructions), 정적 루프(static looping), 동적 분기(dynamic branching), 동적 루프(dynamic looping)). HLSL이 다양한 조건 분기를 지원할 수 있도록 컴파일될 수 있지만, 제한된 모델에서만 실행할 수 있다는 것을 염두에 두어야 한다. 앞서 이야기했던 것과 같이, HLSL은 컴파일 타겟에 대해서 어떤 문법적인 제약도 두지 않지만, 컴파일 타겟이 사용하지 않는 명령어를 수행할 경우에는 실행 에러가 발생할 수 있다.

루프는 셰이더에서 거의 볼 수 없다. 특정 하드웨어의 경우에는 정적이나 동적 루프를 지원하지만, 대부분은 순차적인 실행만을 지원한다. 루프를 지원하지 않는 모델의 경우에는 모든 루프는 전개된다. 루프는 느린 연산이 될 수도 있겠지만, 적은 노력으로 훌륭한 코드를 만드는 데 쓰일 수도 있다. Ps_1_1 셰이더에도 루프를 풀어서 사용하는 좋은 예가 DirectX 9 SDK의 DepthOfField 샘플이다. 고성능의 셰이더를 작성하기 위하여 컴파일러가 루프를 펼쳐주는 작업을 하는 데 사용될 수 있지만, 명령어 수 제한에 걸린다거나 반복 수가 너무 많아 성능이 저하될 수 있다는 것을 염두에 두어야 한다.

If 구문을 사용하는 것은 대부분의 어셈블리 수준의 셰이더 모델에서 많은 연산을 필요로 한다. 분기를 지원하지 않는 모델의 경우에는 if 구문의 양쪽 모두가 실행되며, 각각의 출력 결과에 따라서 선택되어 실행된다. CPU 프로그래밍 관점에서 보았을 때, 이러한 형태의 실행은 HLSL 작성자가 원하는 실행 형태가 아니다. CPU에서 사용하는 일반적인 최적화 과정은 분기문을 지원하지 않는 셰이더 모델에서는 적용되지 않는데, 이것은 복잡한 경로와 간단한 경로 모두가 실행되기 때문이다. 어떤 셰이더 모델에서는 "predicated~instructions", "static~if~blocks" 그리고 "dynamic~if~blocks"와 같은 서로 다른 수준의 분기문을 지원하기도 한다.

vs_1_1을 사용한 예,

```
if (Value > 0)
    Position = Value1;
else
    Position = Value2;
```

생성된 어셈블리 코드,

```
// Value>0인 lerp 값을 구한다.
mov r1.w, c2.x
slt r0.w, c3.x, r1.w
// Value1과 Value2 사이를 lerp한다.
mov r7, -c1
add r2, r7, c0
mad oPos, r0.w, r2, c1
```

대부분의 하드웨어 셰이딩 모델에서 지원되고 있는 정적 분기는 셰이더 상수로 지정된 참/거짓 값 여부에 따라서 해당 코드 블록의 실행을 제어하는 역할을 한다. 이 방식을 이용하여 렌더링하려는 오브젝트의 유형에 따라서 무거운 코드의 실행 여부를 쉽게 결정할 수 있다. draw 함수를 호출하기 전에 참/거짓 상수 값 플래그를 설정하여 셰이더가 지원하는 다양한 기능들을 켜고 끌 수 있는데, 이것은 해당 코드 부분을 완전히 건너뛸 수 있게 해준다. 단, 분기는 draw 함수를 호출하기 이전에 지정해두어야 하며 정점이나 픽셀 차원에서는 분기를 결정할 수 없다. 하지만 양쪽 모드를 실행하는 방식을 사용하면 픽셀이나 정점 수준에서 동적으로 분기할 수 있다.

가장 보편적인 분기문은 능동적 분기이다. 능동적 분기는 표준화된 CPU와 가장 근사한 모델에서 지원한다. 여기에서의 성능 비용은 분기문으로 소모된 것과 선택된 경로의 명령을 실행하기 위해 소모된 것을 합치면 된다. 이 실행 비용은 대부분의 사람들이 CPU 사이드 코드에서 최적화한 비용과 비교할 수 있다. 이 경우의 문제점은, 대부분의 하드웨어에서는 지원되지 않고 정점 셰이더에서만 사용 가능하다는 것이다. 셰이더를 최적화하는 것은 CPU에서 수행되는 코드를 최적화하는 것과 매우 유사하다.

입력 데이터형 선언의 중요성

셰이더에서 입력 값의 유형은 일반적으로 기대하는 것과는 다르게 사용될 것이다. 정점 버퍼에서 정점 셰이더로 데이터가 어떻게 이동하는지, 정점 셰이더 출력 값이 어떻게 픽셀 셰이더의 입력 레지스터로 들어가는지 등은 Direct3D 스펙에 잘 정의되어 있다. 셰이더에 입력되는 값은 언제나 4개의 float 벡터로 확장(예로, 정점 버퍼의 위치 값은 보통 x, y, z 세 개이다.)되어 레지스터에 저장되는데, 이 사실을 아는 것이 셰이더 데이터 로드에 관한 스펙을 잘 아는 것보다 최적화 방법을 찾는 데 좀 더 도움이 된다.

셰이더 작성자들은 흔히 정점 데이터가 셰이더의 입력 레지스터로 로드될 때 자동으로 데이터에 없는 값들이 float4형에 맞게 채워진다는 것을 이용하여 최적화를 한다. 예를 들어, 정점 셰이더의 w는 정점 버퍼에 w 값이 없으면, 1.0으로 설정된다. y와 z 성분 역시 정점 버퍼에 아무런 값도 정의되어 있지 않다면, 0.0으로 설정된다. 이는 정점 셰이더에서 위치를 다룰 때 매우 유용하다. 세계 행렬을 곱해줄 때 w 성분이 1.0으로 되어야 한다는 사실은 매우 일반적이고, 정점 버퍼는 보통 x, y, z 값만을 가지고 있다. 위치 입력 매개변수가 float3으로 정의되어 있다면 w 성분에 1.0을 넣어줄 추가적인 명령어가 필요하다. 매개변수가 float4로 정의되어 있다면 하드웨어가 입력 레지스터를 저장하면서 자동으로 1.0으로 설정할 것이다. 컴파일러는 정점 버퍼에 어떤 데이터가 있는지 필요한 데이터를 알 수 없으므로 이러한 형태의 최적화를 자동으로 미리 해주지 않는다.

또 다른 최적화 과정은 셰이더에서 모든 입력 매개변수의 데이터형을 확실히 선언해주는 것이다. 예를 들어 들어오는 데이터가 정수형이고, 이 데이터가 메모리 주소 연산을 목적으로 사용된다면, 이를 int형으로 설정해야 float형으로 자동 변환되는 일이 없을 것이다. int형으로 입력 데이터형을 선언할 때는 반드시 int형 데이터로 입력 값이 들어가야 한다. 그렇지 않다면 컴파일러가 입력 데이터는 정수형일 것이라는 가정 하에 들어간 최적화 과정 때문에 실행이 제대로 되지 않을 것이다.

정밀도 문제(log p, exp p, lit)

셰이더를 작성할 때는 정확한 결과와 최적의 성능을 얻기 위해서 정밀도에 대한 이해가 필요하다. 대부분의 셰이더 컴파일 타겟의 경우에는 정확도가 내부에서 고정되어 있고, 정확한 결과를 얻기 위해서 이러한 내용들이 고려되어야 한다. 예를 들어, ps_1_x 모델

은 상대적인 낮은 정밀도의 고정 소수점의 레지스터를 가지고 있다. 반영 하이라이트를 위해 낮은 수의 제곱만 해도 금새 띠들이 여럿 생긴 것과 같은 색 계단 현상이 생겨버리곤 한다.

vs_1_1이나 vs_2_0 같은 다른 셰이더의 경우에는 저정밀도 버전의 명령어가 따로 있다. logp, expp와 lit은 log, exp와 pow 함수의 저정밀도 버전이다. 어떤 하드웨어에서는 저정밀도와 고정밀도 함수의 수행 시간의 차가 크게 나는 것도 있다. log와 exp를 수행하기 위해서는 10개의 명령어가 필요하지만, logp, expp는 단 한 개의 명령어만 있으면 된다. vs_1_1과 같은 특정 컴파일 타겟으로 컴파일 시 이 사실을 이용하면 많은 수의 어셈블리 코드를 줄일 수 있다. 이러한 저정밀도의 명령어를 사용하려면 half라는 저정밀도 데이터형으로 캐스팅하여 저장하면 된다. 이 저정밀도로 결과를 받으려면 컴파일러는 가능한 저정밀도 버전의 명령어를 사용하여 계산해야 할 것이다. 어떤 픽셀 셰이더 하드웨어는 다른 함수의 경우에도 저정밀도로 계산했을 경우에 이득을 보는 경우가 있다.

다음은 log와 logp의 비교 예이다.

```
float LogValue = log(Value);          float LogValue = (half)log(Value)

// vs_1_1에서                          // vs_1_1에서
// 10개의 명령어로써 계산된다.          // 1개의 명령어로써 계산된다.
log r0, c0;                           logp r0, c0
```

ps_1_x 컴파일 타겟의 사용

이전의 픽셀 셰이더 모델(ps_1_1, ps_1_2, ps1_3, ps_1_4)은 비교적 자유롭게 프로그래밍할 수 있지만 제약도 가지고 있다. HLSL을 이용하게 되면 ps_1_x 컴파일 타겟은 효과적으로 컴파일이 가능하지만, 이는 고성능의 셰이더와 스스로 작성한 셰이더가 있어야만 가능하다. 명령어 개수는 사람들이 대면하게 될 첫 번째 장벽이 되지만, 이로 인하여 ps_1_x 컴파일 타겟의 다른 제한점은 무시할 수 있다.

Ps_1_x 컴파일 타겟에 대하여 기억해야 할 첫 번째 점은 하드웨어가 임의의 스위즐들을 지원하지 않는다는 것이다. 이런 제한은 스위즐을 쓰게 되면 항상 추가적인 명령어들이 따라붙게 된다는 것을 의미한다. 이런 명령어들이 추가되다보면 해당 컴파일

타겟이 지원할 수 있는 최대 명령어 수를 쉽게 초과해버리곤 한다. Ps_1_1부터 ps_1_3 타겟은 임의의 쓰기 마스크나 복제 스위즐(.r, .g, .b, .a)을 지원하지 않으므로 역시 같은 상황이 일어날 수 있다. Ps_1_4 컴파일 타겟은 복제 스위즐과 임의의 쓰기 마스크를 지원한다. 이런 제한이 있더라도 재미있고 복잡한 셰이더 쓰는 것은 의외로 쉽다. 이것들은 ps_1_x 컴파일 타겟들을 대상으로 HLSL 코드를 작성할 때의 참고 사항일 뿐이다. Ps_1_x 타겟이 임의의 스위즐 지원이 안되더라도 source나 dest 수정자를 써서 0부터 1까지로 값을 자른다거나 source의 보수를 취한다거나 부호를 바꾸고 보정을 하는 등의 작업을 추가적인 부담 없이 수행할 수 있다. 이런 수정자들은 적은 수의 명령어들로 셰이더를 생성해야 할 때 매우 유용하다. 컴파일러는 자동으로 사용할 수 있는 모든 수정자들을 사용할 것이다. 하지만 셰이더 작성자들이 특정 연산을 구현할 때 이런 수정자들을 염두에 두고 작업하면 컴파일러가 최적화하는 데 많은 도움이 될 수 있다. 실제로 몇몇 내장 함수들이 이런 종류의 셰이더를 좀 더 쉽게 작성하기 위해 HLSL에 추가되었다. 예를 들어 saturate() 내장 함수를 쓰면 실행 비용이 들지 않는 _sat 수정자가 픽셀 셰이더에서 생성될 것이다.

_ bx2 수정자

_bx2 수정자를 생성하기 위하여 HLSL 컴파일러에서 사용되는 많은 종류의 HLSL 코드가 있다. 다음의 main 함수들은 _bx2 수정자를 생성한다.

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, Tex*2 - 1);
}

float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    float3 val = Tex*2;
    val = val -1;
    return dot(Col,val);
}

float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, (Tex -.5f)*2);
}
```

이 모든 함수들은 ps_1_x 타겟에서 같은 어셈블리 셰이더를 생성한다.

```
ps_1_1
texcoord t0
dp3 r0, v0, t0_bx2
```

Tex*2 - 1 버전은 ps_2_0 타겟에서 더욱 최적화된 코드를 생성한다는 것을 기억하라.

_bias 수정자

다음의 코드는 _bias 수정자를 생성한다.

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, (Tex - .5f));
}
```

이 main 함수는 다음의 어셈블리 셰이더를 생성한다.

```
ps_1_1
texcoord t0
dp3 r0, v0, t0_bias
```

_bias는 소스가 0 1 사이의 값을 가지지 않으면, ps_1_1, ps_1_2, ps_1_3에서 실행되지 않는다. 실행되지 않는다면 이미 포화되어 있음을 의미한다.

_x2 수정자(ps_1_4 only)

다음의 코드는 _x2 수정자를 생성한다.

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, Tex*2);
}
```

이 HLSL 코드는 다음의 어셈블리 셰이더 코드를 생성한다.

```
ps_1_4
texcrd r0.xyz, t0
dp3 r0, v0, r0_x2
```

_ x2, _ x4, _ x8, _ d2, _ d4 그리고 _ d8 destination write 수정자

destination write 수정자들이 ps_1_x 모델에서 지원되며, 컴파일러는 HLSL 코드를 컴파일할 때 이들을 사용하여 어셈 명령어들을 생성할 수 있다. 명령어의 결과를 두 배(_x2)나 네 배(_x4) 혹은 절반으로 나누는(_d2) 수정자들은 ps_1_1부터 ps_1_3까지의 모델에서 지원되고 ps_1_4 모델은 다음 6개의 수정자들을 모두 지원한다(_x2, _x4, _x8, _d2, _d4, _d8 등). 다음의 코드에서 N이 2, 4, 8, 0.5, 0.25, 0.125일 때 일반적인 곱셈 명령어 대신 해당 수정자를 사용하는 명령어가 생성될 것이다.

```
static const float N = 2;

float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return (Col[0] + Col[1] )*N;
}
```

위의 HLSL 코드는 다음의 어셈블리 코드를 생성한다.

```
ps_1_1
add_x2 r0, v0, v1
```

complement 수정자(보수 수정자)

ps_1_x 타겟을 컴파일할 때 HLSL 코드로 complement 수정자를 생성할 수 있다. 이는 보수가 되는 값이 0 1 사이의 값일 때만 동작한다. 다음의 HLSL 코드는 실행 비용이 들지 않고 보수를 생성하게 해준다.

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return (1-Col[0]) * (Col[1]);
}
```

앞의 HLSL 코드는 다음의 어셈블리 코드를 생성한다.

```
ps_1_1
mul r0, 1-v0, v1
```

saturate 수정자(포화 수정자)

다음의 두 셰이더는 _sat 수정자를 생성한다. 이는 모든 픽셀 셰이더 컴파일 모델에서 지원된다.

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return saturate(Col[0]);
}
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return clamp(Col[0],0,1);
}
```

위의 두 HLSL 셰이더는 다음의 어셈블리 셰이더를 생성한다.

```
ps_1_1
mov_sat r0, v0
```

negate 수정자(부정 수정자)

다음의 셰이더는 모든 컴파일 타겟에서 지원되는 negate 수정자를 생성한다.

Note

PS_1_x 상수 레지스터는 직접적으로 값이 negate되지 않고, 임시 저장 영역으로 옮겨진 뒤에 negate 된다.

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return -Col[0];
}
```

이 HLSL 코드는 다음의 어셈블리 코드를 생성한다.

```
ps_1_1
mov r0, -v0
```

ps_1_x 타겟을 위한 전략

ps_1_x 컴파일 타겟에서 최적화하는 방법 중 하나는 먼저 셰이더를 ps_2_0으로 작성하는 것이다. 이는 ps_2_0이 적용되는 하드웨어에서는 쉽고 빠르게 원형을 만들 수 있기 때문인데, 셰이더가 원하는 대로 작동이 된다면 다시 이를 원하는 ps_1_x 모델로 크로스 컴파일하면 된다. Fxc.exe를 사용할 때 검증 옵션을 사용하지 않으면, ps_1_x 모델에서 제한이 없을 경우 생성될 명령어들이 얼마나 많은지를 확인할 수 있다. 셰이더가 적용되지 않는다 하더라도 효율적인 ps_1_x용 셰이더 코드를 얻고 실행하기 위해 어떤 불필요한 부분들이 있는지를 확인할 수 있다.

지금까지 HLSL 셰이더에 대해서 알아보았고, HLSL 셰이더를 응용프로그램에 적용하는 방법에 대해서도 알아보았다. HLSL은 D3DX Effect를 사용하지 않고도 적용할 수 있는데, 지금부터는 Effect를 사용하거나 사용하지 않는 두 방식에 대해서 좀 더 자세히 알아볼 것이다. 또한 렌더몽키와 같은 셰이더 개발 환경을 사용하면 어떤 응용프로그램 코드를 작성하지 않고도 HLSL을 사용하는 것이 가능한데, 렌더몽키에 대한 자세한 정보는 이 책의 "렌더몽키를 사용한 셰이더 개발" 장을 참고하기 바란다.

D3DX Effect를 사용하여 엔진에 적용하기

D3DX 라이브러리 중 D3DX Effect 프레임웍은 전문 개발자들의 많은 관심을 받고 있는데 DirectX 9 버전 D3DX Effect는 HLSL의 지원 부분이 더 추가되어 있다. D3DX Effect는 3차원 응용프로그램에서 특수 효과 등을 편하게 렌더링할 수 있게 감싸놓을 목적으로 추상화된 인터페이스이다. Effect는 렌더링 스테이트(Rendering state)뿐만 아니라 이전 버전들의 하드웨어를 지원하기 위한 부분까지 포함되어 HLSL이나 asm으로 짜놓은 셰이더들까지 사용할 수 있다. Effect는 일반적인 경우 하나의 .fx 또는 .fxl 파일로 저장되고, 그 파일 자체가 techniques라는 다양한 버전의 Effect들을 담게 된다. 오래된 하드웨어의 수준으로 돌릴 수 있는 조금 더 간단한 Effect 버전을 생성하고 싶을

수 있는 것이다. techniques를 이러한 경우에 사용한 좋은 예가 DirectX SDK 안에 있는 Water 샘플이다. 이 예제는 서로 다른 하드웨어를 지원하는 다른 버전들의 테크닉들을 사용하고 있다. 물론 더 기초적인 테크닉은 적은 수의 텍스처를 사용하고 복잡하지 않은 것을 보여주어 성능과 속도 사이의 절충점을 찾을 수 있게 해준다.

Effect 파일

Effect에 대해 다 알기는 힘들겠지만, HLSL과 함께 사용하려면 Effect 파일의 기본적인 구조는 이해해야 한다. 기본적인 Effect 파일은 다음과 같은 형태를 지니고 있다.

```
// 조명 관련 상수들
VECTOR g_Leye;
float4 GlobalAmbient = 0.5;
float Ka = 1;
float Kd = 0.8;
float Ks = 0.9;
float roughness = 0.1;
float noiseFrequency;

MATRIX matWorldViewProj;
MATRIX matWorldView;
MATRIX matITWorldView;
MATRIX matWorld;
MATRIX matTex0;

TEXTURE tVolumeNoise;
TEXTURE tMarbleSpline;

sampler NoiseSampler = sampler_state
{
   Texture = (tVolumeNoise);

   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU = Wrap;
   AddressV = Wrap;
   AddressW = Wrap;
   MaxAnisotropy = 16;
};
```

```
sampler MarbleSplineSampler = sampler_state
{
   Texture = (tMarbleSpline);

   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Clamp;
   AddressV  = Clamp;
   MaxAnisotropy = 16;
};

float3 snoise (float3 x)
{
    return 2.0f * tex3D (NoiseSampler, x) - 1.0f;
}


float4 ambient (void)
{
   return GlobalAmbient;
}


float4 soft_diffuse(float3 Neye, float3 Peye)
{
   // 정점에서 시점 공간(Leye) 기준의 조명을 향한 정규화된 벡터를 계산한다.
   float3 Leye = (g_Leye - Peye) / length(g_Leye - Peye);

   float NdotL = dot (Neye, Leye) * 0.5f + 0.5f;

   // N.L
   return float4(NdotL, NdotL, NdotL, NdotL);
}
float4 specular(float3 NNeye, float3 Peye, float k)
{

    // 정점에서 시점 공간(Leye) 기준의 조명을 향한 정규화된 벡터를 계산한다.
    float3 Leye = (g_Leye - Peye) / length(g_Leye - Peye);

    // Veye를 계산한다.
    float3 Veye = -(Peye / length(Peye));
    // half-angle를 계산한다.
    float3 Heye = (Leye + Veye) / length(Leye + Veye);

    // N.H를 계산한다.
    float NdotH = clamp(dot(NNeye, Heye), 0.0f, 1.0f);

    float NdotH_2  = NdotH    * NdotH;
    float NdotH_4  = NdotH_2  * NdotH_2;

    float NdotH_8  = NdotH_4  * NdotH_4;
    float NdotH_16 = NdotH_8  * NdotH_8;
    float NdotH_32 = NdotH_16 * NdotH_16;

    return NdotH_32 * NdotH_32;
}

float4 hlsl_bluemarble (float3 P : TEXCOORD0, float3 Peye : TEXCOORD1, float3
         Neye : TEXCOORD2) : COLOR
{
   float4 Ct;
   float4 Ci;
   float3 NNeye;
   float marble;
   float f;

   // 좀 더 많은 진동을 만들기 위해 나눈다.
   P = P/16;
   marble = -2.0f * snoise(P * noiseFrequency) + 0.75f;

   NNeye = normalize(Neye);

   // 색상 스플라인(알파에서 글로스 값)을 따르는 f를 큐빅 보간한다.
   Ct = tex1D(MarbleSplineSampler, marble);

   // 발광 색상
   Ci = Ct * (Ka * ambient() + Kd * soft_diffuse(NNeye, Peye)) + Ct.w * Ks *
            specular(NNeye, Peye, roughness);

   return Ci;
}
```

```
VERTEXSHADER asm_marble_vs =
decl {}
asm
{
   vs.1.1

   dcl_position v0
   dcl_normal    v3
   m4x4 oPos, v0, c[0]           // 위치를 절단 공간으로 변환한다.

   m4x4 r0, v0, c[17]            // 변환된 Pshade(0번 텍스처 행렬을 사용)
   mov oT0, r0

   m4x4 oT1, v0, c[4]            // 위치를 시점 공간으로 변환
   m3x3 oT2.xyz, v3, c[8]        // 법선을 시점 공간으로 변환
};


technique technique_hlsl_bluemarble
 pass P0
   {
      // 변수 이름을 하드웨어 레지스터에
      // 매핑시킨다.
      VertexShaderConstant[0]   = <matWorldViewProj>;
{
VertexShaderConstant[4]   = <matWorldView>;
      VertexShaderConstant[8]   = <matITWorldView>;
 VertexShaderconstant[12] = <matWorld>;
      VertexShaderConstant[17] = <matTex0>;
 VertexShader = <asm_marble_vs>;
PixelShader   = compile ps_2_0 hlsl_bluemarble();

     CullMode = CCW;
   }
}
```

이 예제를 아래부터 살펴보자. 이 Effect 파일의 마지막 부분은 렌더링 패스가 한 개인 technique_hlsl_bluemarble이라 불리는 technique로 정의되어 있다. 이 싱글 패스는 어셈블리 언어로 작성된 정점 셰이더 HLSL로 작성된 픽셀 셰이더를 사용한다. 이 패스의 처음 몇 줄은 다섯 개의 행렬을 정의하고 있는데, 이것들은 이 패스가 호출될 때 참조되는 상위 레벨의 Effect 변수들(여기서는 ID3DEffect::SetMatrix()를 사용해서

설정된 행렬 변수의 포인터가 될 것이다)로부터 특정 하드웨어 상수 레지스터로 들어갈 값들이다. 이러한 명시적인 매핑은 어셈블리 셰이더에서 쓰이기 위해 Effect 파일에서 이루어진다. 픽셀 셰이더에서는 HLSL로 쓰여졌기 때문에 이와 같은 명시적인 매핑은 하지 않아도 된다. 다음 줄은 이 패스에서 사용될 ams_marble_vs라 불리는 어셈블리로 작성된 정점 셰이더를 선언한다.

```
VertexShader = <asm_marble_vs>;
```

다음은 hlsl_bluemarble() 함수를 이용하여 ps_2_0 타겟에서 컴파일될 픽셀 셰이더를 정의한다.

```
PixelShader  = compile ps_2_0 hlsl_bluemarble();
```

위의 hlsl_bluemarlbe은 HLSL 픽셀 셰이더의 진입 지점이다. 이 코드를 살펴보면 tex1d() 내장 함수를 볼 수 있는데, 이 함수는 ambient()나 soft_diffuse() 같은 다른 유틸리티 함수들을 사용하며 이 유틸리티 함수들은 Effect보다 먼저 정의된다. 또한 ps_2_0 타겟용으로 컴파일할 것이므로 이들은 어셈블리 결과에 인라인되어 들어가게 된다.

위의 유틸리티 함수들을 보면, NoiseSampler와 MarbleSplineSampler와 같은 샘플러 선언이 보인다. 이것들은 Effect 파일에서 사용될 때를 제외하면 이전과 같은 방식으로 정의되는데, 이것들은 사용될 어드레싱과 필터링 샘플러 스테이트를 정의하는 괄호로 묶여진 코드 다음에 올 수도 있다. 텍스처 역시 샘플러의 선언 위에 보여진 것과 같이 Effect 파일 내에서 정의가 가능하다. Effect 파일의 제일 처음을 보면 응용프로그램 수준에서 설정이 가능한 전역 변수들의 선언을 볼 수 있다.

Effect API

지금까지 Effect 파일을 작성했고 이것을 파일로 저장해보았다. 이제 이것들을 응용프로그램 코드에서 사용해보자. 제일 먼저 해야 할 것이 D3DXCreateEffectFromFile() API를 사용하여 Effect를 작성하는 것인데, 이 과정이 성공적으로 수행되면 Effect에 필요한 변수들을 Effect API를 통하여 설정할 수 있게 된다. 예를 들어, 행렬은 SetMatrix()를 통해서 설정할 수 있다.

```
// 모든 행렬을 설정한다.
m_pEffect->SetMatrix ("matWorldViewProj", &m_matWorldViewProj);
m_pEffect->SetMatrix ("matWorldView", &m_matWorldView);
m_pEffect->SetMatrix ("matITWorldView", &m_matITWorldView);
m_pEffect->SetMatrix ("matWorld", &m_matWorld);
m_pEffect->SetMatrix ("matTex0", &m_ObjectParameters.m_matTex0);
```

float들이나 vector 또한 비슷하게 설정할 수 있다.

```
m_pEffect->SetFloat ("noiseFrequency ", &m_fNoiseFreq);
m_pEffect->SetVector("g_Leye", &g_Leye);
```

텍스처들도 마찬가지이다.

```
m_pEffect->SetTexture("tVolumeNoise",  m_pVolumeNoiseTexture);
m_pEffect->SetTexture("tMarbleSpline", m_pMarbleColorSplineTexture);
```

적당한 상수들이 설정되면, technique를 설정할 수 있고 모든 패스를 렌더링할 수 있다(이 중에는 패스가 한 개만 있다).

```
m_pEffect->SetTechnique(m_pEffect->GetTechniqueByName("technique_hlsl_
          bluemarble"));

m_pEffect->Begin(&cPasses, 0);
for (iPass = 0; iPass < cPasses; iPass++)
{
   m_pEffect->Pass(iPass);

   // 기하구조를 렌더한다.
}
m_pEffect->End();
```

여기서 보면 기존 방식의 응용프로그램에서는 구현해줘야 했던 귀찮은 부분들이 빠져 있음을 알 수 있다. 예를 들어, 응용프로그램이 하드웨어 상수 레지스터에 g_Leye를 저장한다거나, 노이즈 텍스처를 사용하기 위해 어떤 샘플러를 사용해야 할지를 알 필요가 없다. 이렇게 세세한 과정은 D3DX Effect 프레임워크에서 관리해준다.

D3DX Effect를 사용하지 않고 엔진에 적용하기

어떤 ISV는 크로스 플랫폼 개발 환경이나 여러 가지 문제로 인하여 D3DX에 특화된 코드를 사용하는 것을 원하지 않는 경우가 있다. HLSL 셰이더 관리를 위한 D3DX Effect가 매우 편리하기는 하지만 반드시 필요한 것은 아니다. 물론 D3DX의 편리함을 포기한다는 의미는 주어진 셰이더에서 트래킹과 상수를 설정한다거나 렌더링을 위한 샘플러 등을 직접 관리하겠다는 의미가 된다. 그럼 이것을 어떻게 하는지 살펴보기로 하자.

HLSL 코드를 컴파일 과정이 포함된 D3DX Effect를 사용하지 않기 때문에 응용프로그램에서 HLSL 컴파일러를 명시적으로 실행해주어야 한다. 이는 D3DXAssembleShader*() 루틴 대신에 D3DXCompileShader*() 루틴을 사용하는 것만을 제외하고는 어셈블리 셰이더를 사용하여 응용프로그램 코드를 작성하는 것과 매우 유사하다. 결과로 나온 asm 코드는 컴파일 과정이 없는 어셈블리 셰이더에서 했던 것처럼, 사용할 CreatePixelShader() 또는 CreateVertexShader()에 넘겨주면 된다. 다음의 코드에서 이러한 사용 방법을 볼 수 있다.

```
if (FAILED(hr = D3DXCompileShaderFromFile(g_strVHLFile, NULL, NULL, "main",
            "vs_1_1", NULL, &pCode, NULL, &m_VS_ConstantTable)))
{
   return hr;
}

if (FAILED(hr = m_pd3dDevice->CreateVertexShader((DWORD*)pCode->
            GetBufferPointer(), &m_HLSLVertexShader)))
{
   return hr;
}
```

위의 코드에서 D3DXCompileShader*() 루틴에는 D3DXAssembleShader*()에서 볼 수 없었던 몇 개의 매개변수가 있다. 이것은 컴파일할 셰이더의 이름과 컴파일 타겟(위의 "main"이나 "vs_1_1")들을 명시하기 위해서 필요하다. 또한, #defines의 값을 명시하거나, 파일을 포함하거나, 디버깅, 최적화, 검증, 행렬 데이터 순서 등을 확인하기 위한 제어 값들을 추가할 수 있다. 이 모든 입력들은 D3DXCompileShader*() 루틴으로 처음 여섯 개의 매개변수를 통해서 전해진다. 마지막 세 개의 매개변수는 컴퓨터에 의해서 채워질 버퍼(이진 어셈블리 코드, 에러 메시지, 상수 테이블)들의 포인터이다. 상수 테

이블이 주어진 HLSL 코드를 실행하기 전에 어떻게 정확한 상수 데이터를 저장해야 할지 알기 위해 응용프로그램에 의해 사용되므로, 이진 어셈블리 코드는 CreatePixelShader() 또는 CreateVertexShader()로 전달된다. Effect를 사용하지 않고 응용프로그램에 HLSL 셰이더를 적용하는데, 가장 중요한 것은 D3DXCompileShader*() 루틴에서 반환된 마지막 매개변수이다.

상수 테이블

D3DXCompileShader*() 루틴에서 반환된 상수 테이블은 고수준 상수들과 샘플러들을 특정 하드웨어 상수들과 샘플러들에 연결시키기 위해서 사용된다. 글로벌 스코프에서 정의된 정적이지 않은 변수들은 컴파일된 셰이더에서 사용할 입력 매개변수로 인식되고 셰이더가 정확하게 실행되기 위해서 반드시 적절하게 초기화되어야 한다. 상수 테이블이 바로 이런 역할을 해준다. 보통 ID3DXConstantTable 인터페이스를 직접 사용하는 것이 편한데 실제 상수 테이블의 데이터 구조를 파싱할 필요가 없기 때문이다. ID3DXConstantTable 인터페이스는 ASCII 이름으로 알려진 정의된 변수들의 handle들을 찾을 수 있는 쉬운 메소드들을 제공한다. 이 HLSL 변수들을 위한 적절한 값들은 다음의 코드에서 사용한 방법처럼 설정될 수 있다.

```
D3DXHANDLE handle;

if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "ringFreq"))
{
   m_PS_ConstantTable->SetFloat(m_pd3dDevice, handle, m_fRingFrequency);
}

if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "lightWood"))
{
   m_PS_ConstantTable->SetVector(m_pd3dDevice, handle, &lightWood);
}
```

텍스처들처럼 샘플러 스테이트도 다음의 코드처럼 맞게 설정되어야 한다.

```
if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "NoiseSampler"))
{
   m_PS_ConstantTable->GetConstantDesc(handle, &constDesc, &count);
```

```
    if (constDesc.RegisterSet == D3DXRS_SAMPLER)
    {
        m_pd3dDevice->SetTexture (constDesc.RegisterIndex,
                     m_pVolumeNoiseTexture);

        // Noise 샘플러를 위한 적절한 샘플러 스테이트를 설정한다.
        m_pd3dDevice->SetSamplerState (constDesc.RegisterIndex, …, …);

    }
}
```

렌더 스테이트나 텍스처 스테이지 스테이트, 샘플러 스테이트들은 D3DX Effect에서 지원되었던 것처럼 HLSL에서 설정할 수 없으므로 응용프로그램에 의해 직접 관리되어야 한다.

물론, 어떤 종류의 셰이더 제작 툴에서도 필요한 샘플러들이나 변수의 이름에 대한 응용프로그램 정보를 미리 알고 있을 필요는 없다. 이런 경우에 상수 테이블 안의 상수의 개수를 확인하기 위해서 ID3DXConstantTable::GetDesc() 메소드를 사용할 필요는 있다. 따라서 위의 코드에서 사용된 ID3DXConstantTable::GetConstantByName() 메소드 대신 ID3DXConstantTable::GetConstantElement () 메소드를 사용할 수 있다. D3DX Effect를 사용하지 않고 HLSL 셰이더를 응용프로그램에 적용할 경우에는 ID3DXConstantTable 인터페이스에 익숙해지는 것이 좋다.

SDK 업데이트

DX9.0을 배포하고 연이어 DirectX 9.0a 패치를 내놓은 이후에도, 마이크로소프트에서는 개발자들을 위해 정기적으로 SDK 업데이트를 배포하고 있다. 이 SDK 업데이트는 Direct3D 런타임의 수정을 포함하고 있지는 않지만, HLSL 컴파일러를 포함해 중요한 D3DX 툴 업그레이드를 포함하고 있다. 정기적으로 가장 최근 버전의 DirectX SDK의 업데이트를 받아 가장 최신의 컴파일러를 사용하여 최적화된 asm 코드를 생성할 수 있도록 하자.

결론

지금까지 DirectX 9.0의 가장 새롭고 중요한 특징 중의 하나인 Direct3D High Level Shading Language(HLSL)의 세부사항에 대해 알아보았다. 셰이더 언어 자체의 구조에 대해 소개하였고, 예제 셰이더들을 보면서 주요 개념들을 살펴보았다. 또한, 컴파일 과정과 최적의 성능을 얻기 위하여 셰이더를 어떻게 사용해야 하는지에 대해서도 알아보았다. 이 장에서 다루었던 내용들이 이 책의 뒤에서 다루어질 HLSL 셰이더를 이해할 수 있는 기초가 되고, 현재 다루고 있는 프로젝트에 HLSL 셰이더를 적용하는 계기가 되었으면 한다.

감사의 말

예제 HLSL 셰이더를 제공해준 ATI의 3D Application Research Group에 감사를 드린다. 마이크로소프트의 Dan Baker와 Loren McQuade에게도 최적화 관련 내용을 제공해주고 피드백을 해준 것에 대해 감사 드린다. 보다 명확한 개념을 가질 수 있게 가치있는 조언을 해준 Mark Wang과 Wolfgang에게도 감사의 말을 전한다.