실시간 해칭 렌더링에서 그림자 기법
Shadow Techniques in Real-time Hatching Rendering

김찬수, 김대명
호남대학교

Kim Chan-Soo, Kim Dae-Myung
Honam Univ.

요약

컴퓨터 그래픽스의 연구는 크게 사실적 렌더링 분야와 비사실적 렌더링 분야로 나눌 수 있다. 사실적 렌더링과 달리 비사실적 렌더링은 만화, 수채화, 해칭과 같은 사람이 손으로 표현한 것 같은 분위기와 특징을 갖는 영상을 표현해 보자는데 그 목적이 있다고 할 수 있다. 본 연구에서는 비사실적 렌더링 기법의 하나인 실시간 해칭 렌더링기법과 사실적 렌더링기법인 그림자 기법을 통합하여 실시간 해칭 렌더링에서 그림자를 생성하기 위한 방법을 모색하였다. 그림자 기법에는 투영 텍스쳐 그림자 기법을 이용한 그림자 기법을 해칭 렌더링에 적용하여 해칭 렌더링에서 실시간 해칭 그림자를 생성하기 위한 기법을 제시한다.

Abstract

The research of computer graphics is divided into two parts of photorealistic rendering and non-photorealistic rendering. The purpose of non-photo realistic rendering is to make image like cartoon, water-color, hatching etc. In this paper, we study for real-time hatching rendering and shadow techniques and we combine two techniques to make real-time hatching shadow. In shadow techniques we apply projected texture shadow to hatching rendering. Eventually, we introduce natural real-time hatching shadow through comparison and analysis.

I. 서 론

3차원 그래픽스에서 그림자는 임의의 3차원 영상이 렌더링 될 때, 보다 사실적인 영상을 표현해내기 위한 수단으로 사용되고 있다. 이러한 사실적 영상에 바탕을 두어 그림자 렌더링 기법 역시 연구가 진행되었고 발전해 왔기 때문에 비사실적 렌더링 기법과 같은 특수한 영상을 렌더링 하는 경우에 대해서는 그림자에 대한 구체적인 사용과 방법 그리고 구현에 대한 차별성이 제시되지 않는 것이 사실이다. 그러나 비사실적 렌더링과 같이 인간 친화적 렌더링 기법에서도 그림자는 그림자가 가지고 있는 성격의 특성상 충분히 적용될 수 있으며 필요성이 대두될 수 있다. 이러한 특성을 고려하여 그림자 기법이 비사실적 렌더링에 적용될 경우 어떤 특징과 문제점이 발생될 수 있는지에 대하여 연구의 필요성이 제시된다.

데생은 연필과 팬 같은 도구를 사용해 빛과 음영 그리고 그림자를 고려하여 사물을 그림으로 표현해 내는 가장 기초적이고 대표적인 미술 영역의 하나이다. 비사실적 렌더링의 하나인 실시간 해칭은 바로 이러한 데생기법을 3차원 그래픽스 영상에서 실시간으로 표현해 내는 렌더링 기법이다. 본 논문에서는 해칭에서 그림자 기법을 적용하고도 해칭이 갖는 특징을 그대로 표현해내면서 그림자가 접목된 그래픽 효과를 얻어내기 위해 그림자 렌더링 기법의 기술적 특성과 한계를 분석하고, 해칭 렌더링에 적용해 봄으로써 크게는 비사실적 렌더링과 사실적 렌더링의 접목이라는 성과를 얻고 구체적으로는 해칭 렌더링에서 그림자를 사용할 수 있는 효율적인 방법론을 제시하므로 연구의 목적을 달성하고자 한다.

본 연구에서는 3차원 실시간 해칭 렌더링 기법을 구현하기 위하여 DirectX 그래픽 라이브러리를 기반으로 HLSL(High Level Shader Language)를 이용한 세이딩 프로그래밍 기법을 [2, 4, 6, 7, 13, 15, 16, 17]사용하였고, 해칭 그림자를 생성하기 위하여 투영 텍스쳐 그림자 기법[3]을 사용하는 방법을 제시하였다.

II. 실시간 투영 텍스쳐 해칭 그림자

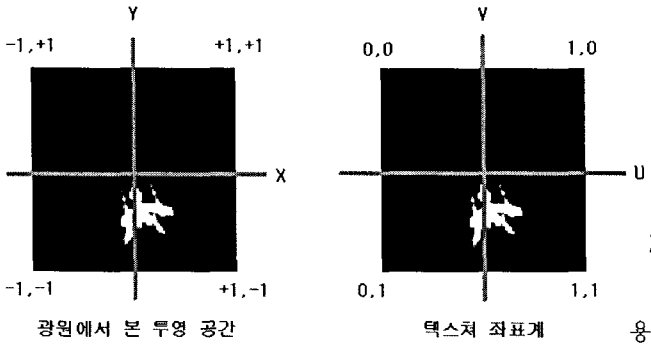
1. 투영 그림자 생성

투영 텍스쳐 그림자를 생성하기 위해서는 카메라를 광원에 위치시키고 광원에서 본 투영 공간의 텍스쳐를 생성해야 한다. 생성된 텍스쳐는 텍스쳐 좌표계인 UV좌표로 변환되고, 변환된 텍스쳐를 그림자가 맵핑될 지형 또는 오브젝트에 맵핑하여 오브젝트가 가지고 있는 색상과 투영 텍스쳐의 색상 혼합으로 그림자 효과를 표현해 낼 수 있다.

투영 텍스쳐의 기술을 바탕으로 실시간 해칭 렌더링에 그림

자를 맵핑하게 된다. 그러나 일반적인 투영 텍스쳐는 그림자가 될 부분은 흰색으로 그 외의 영역에는 검정색으로 채워진 텍스쳐이기 때문에 사실적 렌더링에서는 그림자 효과를 표현해 낼 수 있으나 해칭 렌더링에 투영 텍스쳐를 이용하여 일반적인 방법으로 텍스쳐 맵핑과 색상 혼합을 하게 되면 해칭 렌더링이 갖는 스트로크의 특징을 표현해내기 어렵게 된다. 이러한 문제를 셰이더를 이용하여 투영 텍스쳐의 그림자 영역에 대해서만 스트로크를 그려주고 이를 바탕으로 색상혼합을 하면 그림자 영역에도 해칭 스트로크를 표현해낼 수 있다.

투영 텍스쳐는 카메라를 광원의 위치에 놓고 광원에서 본 투영 공간의 텍스쳐를 생성한다. 동시에 각 정점에 대해 텍스쳐 좌표의 값을 알려준다. 광원으로부터 본 투영 공간과 텍스쳐 좌표는 상수배이며, 중심 위치가 다르다는 점 외에는 큰 차이는 없다. 그림 1은 광원에서 본 투영 공간과 이를 텍스쳐 좌표계로 변환할 경우의 좌표 차이를 보여준다.



▶▶ 그림 1. 투영 텍스쳐 좌표

광원에서 본 투영 공간은 중심이 (0,0)인데 비해, 텍스쳐 좌표계는 좌측 상단이 (0,0)이다. 이는 텍스쳐 좌표 체계인 U,V 좌표가 0부터 1까지의 크기만을 가지는 특징과 관련이 있다. 또한 투영 공간의 폭은 2이며 투영 공간과 텍스쳐 좌표계는 y 좌표계가 반대라는 점이 다르다. 이러한 차이를 정리하면 투영 공간(x, y)와 텍스쳐 좌표계(u, v)의 관계는 다음과 같다.

$$U = +0.5X + 0.5$$
$$V = -0.5Y + 0.5$$

간단한 식으로 텍스쳐 좌표로 변환을 할 수 있으나 실제 3D에서는 x, y, z그리고 동차 공간인 w를 고려한 4차원의 벡터로 계산되기 때문에 실제 U, V의 계산은 아래와 같다.

$$U = +0.5(X/W) + 0.5 = 0.5(+X+W)/W$$
$$V = -0.5(Y/W) + 0.5 = 0.5(-Y+W)/W$$

이는 결과적으로 동차 좌표인 W로 나누는 것을 제외하면 x, y와 w의 덧셈으로 이루어진 선형 방정식이 된다. 따라서 이 변환은 행렬로 계산할 수 있으며, 동차 좌표계로 계산하는 경우의 변환 행렬은 아래와 같다.

$$\begin{bmatrix} U \\ V \\ ? \\ W \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

텍스쳐 좌표계의 z 성분은 필요 없지만 4차원 벡터로 계산하면 셰이더 처리에 편하기 때문에 위와 같은 식으로 연산을 수행한다. 전체적인 변환 행렬을 정리하면 모델의 로컬 좌표계로부터 텍스쳐 좌표계로 변환하기 위한 절차는 아래와 같다.

$$\begin{bmatrix} U \\ V \\ ? \\ W \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{광원에서 본 투영변환} \end{bmatrix} \begin{bmatrix} \text{광원에서 본 뷰변환} \end{bmatrix} \begin{bmatrix} \text{월드 변환} \end{bmatrix} \begin{bmatrix} \text{로컬 정점 좌표} \end{bmatrix}$$

2. 해칭 렌더링에 일반 투영 텍스쳐 그림자 적용

렌더링 처리 과정에서 투영 텍스쳐를 이용하여 그림자를 적용하기 위해서는 먼저 그림자로 사용할 투영 텍스쳐를 생성하는 것이며, 두 번째는 생성된 텍스쳐를 그림자가 적용될 오브젝트에 붙여서 장면을 렌더링 한다. 투영 텍스쳐를 생성하는 것은 셰이더를 사용하지 않고 DirectX 만을 이용하여 생성이 가능하다. 렌더링 타겟을 변경한 후 표면(Surface)에 깊이 버퍼를 생성하여 텍스쳐를 생성할 수 있다. 여기서 표면이란 텍스쳐에 렌더링할 경우 텍스쳐 내부의 하나의 화면에 렌더링하게 되므로, 렌더링할 곳을 지정하기 위해서 표면의 인터페이스를 얻어내는 것을 말한다.

텍스쳐가 생성되면 생성된 텍스쳐를 픽셀 셰이더로 가져와 기존의 그림자가 맵핑될 오브젝트의 색상과 그림자 텍스쳐를 혼합한다. 해칭 렌더링에서 맵핑될 오브젝트의 색상은 Tonal Art Map을[1, 8, 14] 적용시키고 정점 별로 6-Way 블렌딩 처리가 끝난 해칭 렌더링의 최종 색상이 되며 이 색상과 투영 텍스쳐 그림자가 혼합 된다. 아래 그림은 해칭 렌더링이 적용되지 않고 순수하게 투영 텍스쳐만 가지고 그림자가 그리워질 오브젝트에 맵핑되어 그림자 효과를 표현한 모습을 보여준다. 그림에서 보이는 바와 같이 임의의 형상인 우주선이 바닥의 오브젝트에 맵핑되는 것을 볼 수 있으며 오브젝트의 바닥이 단면이 아닌 볼록한 지형에 따라 그림자 처리가 이루어지는 모습도 볼 수 있다.

▶▶ 그림 2. 투영 텍스쳐 그림자 적용

그림2는 해칭 렌더링을 구현하고 거기에 투영 텍스쳐 그림자를 맵핑한 모습을 보여준다. 실제 투영 텍스쳐를 해칭 렌더링에 적용하면 그림에서 보이는 것처럼 그림자를 표현할 수 있으나 해칭 렌더링의 특징인 스트로크를 표현하지는 못한다. 이러한 효과는 해칭 렌더링과 일반적인 그림자 매핑이 적절히 조화를 이루지 못하며 해칭 렌더링에 그림자 처리를 보여주기 위해서는 그림자에 적절한 해칭 기법 처리를 해주어 스트로크의 표현을 나타내 주어야 한다.

3. 투영 텍스처 해칭 그림자 생성

해칭 렌더링에서는 음영 처리뿐만 아니라 그림자의 영역도 스트로크를 이용하여 그림자의 느낌을 표현해야 한다. 위의 그림2를 보면 알 수 있지만 해칭 렌더링에 일반적인 투영 텍스쳐 그림자를 혼합하여 그림자를 표현하면 그림자가 그려진 영역에 단색의 그림자가 처리되면서 해칭 스트로크를 표현하는데 문제가 있음을 알 수 있다. 이러한 문제를 해결하고 해칭 렌더링에 그림자의 영역 역시 스트로크 느낌으로 표현해 주기 위해서는 해칭 그림자를 생성해야 한다.

해칭 렌더링에 일반적인 투영 텍스쳐 그림자를 맵핑할 경우 나타나는 문제점을 해결하고 해칭의 스트로크 느낌이 나는 해칭 그림자를 생성하기 위해서 픽셀 셰이더를 이용하여 해결할 수 있다. 앞서 설명한대로 투영 텍스쳐가 생성된 후 픽셀 셰이더로 넘어와 그림자가 맵핑될 오브젝트 색상과 혼합된다고 했다. 여기서 투영 텍스쳐가 그림자가 그리워질 부분이 흰색의 영역임을 알 수 있으므로 흰색의 영역에 대해서만 임의의 스트로크 텍스쳐 색상과 합성한 후 합성된 투영 텍스쳐 그림자를 맵핑하면 해칭의 느낌이 나는 그림자를 생성할 수 있다.

그림자로 사용할 임의의 스트로크 텍스쳐를 직접 만들 경우 해칭 렌더링에서 사용한 Tonal Art Map의 텍스쳐 스트로크 형태를 고려하여 그림자 스트로크 텍스쳐도 제작하는 것이 좋다. 그러나 직접 텍스쳐를 제작하지 않고도 해칭 렌더링에 사용된 Tonal Art Map을 재사용하여 그림자 느낌을 표현할 수도 있다. 해칭 그림자를 생성하는 원리는 실제 픽셀 셰이더에서 이루어진다. 먼저 투영 텍스쳐를 보게되면 검은 영역과 흰 영역의 색상으로 이우어져 있다. 투영 텍스쳐가 꼭 검정과 흰색의 영역으로 그림자가 그리워지는 형상과 그리워지지 않는 형상을 나눌 필요는 없지만 색상의 극과 극인 흰색과 검정색으로 나눌 경우 픽셀 처리에서 그림자의 영역과 아닌 영역을 확연히 구분할 수 있기 때문이다. 투영 텍스쳐에서 해칭 그림자를 표현하는 방법에도 투영 텍스쳐가 흰색과 검정색으로 확연히 구분되어 있음을 활용하였다. 픽셀 셰이더에서 투영 텍스쳐를 불러들인 후 흰색 부분만 판별하여 기존의 해칭 렌더링을 표현하기 위해 가지고 있는 Tonal Art Map 텍스쳐를 다시 가져와서 흰색의 부분 즉 그림자가 그리워질 부분에만 Tonal Art Map 또는 자신이 직접 제작한 스트로크 텍스쳐를 그려주면 된다.

원리는 간단하지만 빛의 변화나 그림자가 지워질 물체가 변환되었을 때 고려해야할 사항이 존재한다. 스트로크 텍스쳐를 가져와 그림자가 그리워질 흰색의 영역에만 다시 그려주면 되지만 스트로크 텍스쳐를 가져올 때의 텍스쳐 좌표를 고려해야 한다. 만약 스트로크 텍스쳐의 좌표가 특정 영역에 맵핑되도록 고정되어 있을 경우 빛의 방향이 변하거나 또는 그림자가 그리워질 물체가 변환이 일어났을 때 그림자의 영역도 동시에 변화가 일어나는데 텍스쳐 좌표를 고려하지 않으면 스트로크 텍스쳐는 고정되어 그림자는 변하지만 스트로크는 그대로 그려지면서 어색한 화면이 렌더링되게 된다. 이러한 문제를 해결하기 위해서 상황에 따라 스트로크 텍스쳐 좌표의 변환이 필요한데 투영 텍스쳐를 생성할 때 사용되는 U, V 좌표 변환을 사용할 수 있다. 이 변환은 광원에서 본 뷰 행렬과 광원에서 본 투영 행렬 그리고 투영 공간의 중심이 (0,0)임을 고려하여 텍스쳐 좌표계로의 변환 행렬을 모두 포함하고 있기 때문에 물체의 변화나 빛에 대응하여 좌표의 변환이 일어난다. 이를 활용하여 스트로크 텍스쳐 좌표를 설정해주고 그려주게 되면 변환에 따라 스트로크가 바뀌면서 해칭 그림자를 생성해 낼 수 있게 된다.

4. HLSL을 사용한 그림자 생성

실제 HLSL을 이용하여 구현하기 위해서 정점 셰이더에서 픽셀 셰이더로 넘기는 데 필요한 데이터는 다음 표1과 같다.

[표 1] 정점셰이더에서 픽셀 셰이더로 넘기는 데이터

타 입	변 수	SEMANTIC	설 명
float4	Pos	POSITION	셰이딩되는 모든 정점 좌표
float2	TexHatch1	TEXCOORD0	Tonal Art Map 1, 2, 3이 저장된 텍스쳐 좌표
float2	TexHatch2	TEXCOORD1	Tonal Art Map 4, 5, 6이 저장된 텍스쳐 좌표
float2	TexStroke	TEXCOORD2	투영 텍스쳐의 그림자 영역에 그려질 스트로크 텍스쳐 좌표
float4	TexShadow	TEXCOORD3	투영 텍스쳐 좌표
float3	Blend123	TEXCOORD4	6 Way 블렌딩에 사용할 가중치 1, 2, 3의 값
float3	Blend456	TEXCOORD5	6 Way 블렌딩에 사용할 가중치 4, 5, 6의 값

HLSL에는 타입이 float4 float3 등 매트릭스 연산의 특성상 한 번에 여러 개의 타입 정보를 묶어 놓은 타입이 존재한다. 또한 실제로 GPU 레지스트리 상에 저장된 데이터의 참조와 구분을 위해 SEMANTICS 가 존재하는데 HLSL에 대한 자세한 내용은 관련 문헌을 참고하기 바란다. 실제 HLSL 코드를 살펴보면 간단한 코드로 그림자 텍스쳐를 생성할 수 있음을 볼 수 있다. 아래 코드는 투영 텍스쳐에 스트로크 텍스쳐를 그려주는 역할을 한다.

```
if(ShadowColor.R == 1.0f && ShadowColor.G == 1.0f
&& ShadowColor.B == 1.0f)
    ShadowColor = HatchingColor;
```

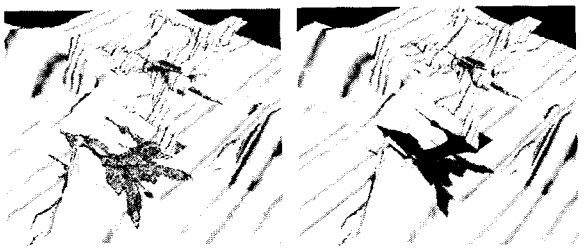
투영 텍스쳐의 그림자 컬러가 1.0f 즉 흰색인 것을 판별하여 그 영역에 대해서만 스트로크 텍스쳐를 그려주고 있다. 이렇게 하면 투영 그림자 텍스쳐의 그림자 영역인 흰색 부분은 스트로크로 채워지게 된다. 만약 Tonal Art Map의 단계별 스트로크가 텍스쳐 그레이 스케일 값을 이용하여 RGB색상에 저장되어 있을 경우에는 내적으로 색상 변환을 해주면 된다. 내적을 이용한 색상 변환은 아래와 같다.

```
ShadowColor = dot( float4(1.0f, 1.0f, 1.0f, 1.0f) -
ShadowColor, ShadowColor );
```

이렇게 얻은 최종 투영 텍스쳐의 색상에 float4(1.0f, 1.0f, 1.0f, 1.0f)에서 빼주는 작업이 들어간다. 이는 색상을 다시 뒤집어 최종 렌더링 색상에 적용하기 위함이다. 해칭 렌더링의 최종 색상과 곱하면 해칭 그림자가 적용된 실시간 해칭 렌더링을 구현할 수 있다. 상수 0.7f은 그림자의 밝기 톤을 조절하는데 사용되어 값이 커질수록 연한 그림자가 나타나며 값이 작을수록 진한 그림자가 나타나기 때문에 적절한 값을 조절하여 그림자의 농도를 결정할 수 있다.

```
return FinalColor * float4(1.0f, 1.0f, 1.0f, 1.0f) -
ShadowColor * 0.7f;
```

아래의 그림3은 일반 투영 텍스쳐를 해칭 렌더링에 그대로 적용한 화면과 투영 텍스쳐를 해칭 그림자로 변화시킨 후 해칭 렌더링에 적용한 모습을 보여준다. 일반 투영 텍스쳐를 해칭 렌더링에 적용할 경우 그림자가 단색의 어두운 영역으로 그려지면서 해칭의 스트로크와 조합되지 못하고 어색한 화면을 연출하는 것을 볼 수 있다. 반면 해칭 그림자가 적용된 렌더링에서는 그림자에 스트로크가 표현되면서 실시간 해칭 렌더링에 그림자가 적용된 화면을 볼 수 있다.



투영 텍스쳐 해칭 그림자 일반 투영 텍스쳐 그림자

▶▶ 그림 3. 투영 텍스쳐 해칭 그림자

III. 결 론

본 연구에서는 비 실사 렌더링 분야의 하나인 해칭 렌더링에서 사실적 렌더링을 위해 발전해온 그림자 렌더링 처리 기법을 해칭 렌더링에 적용하였다. 사실적 렌더링에서 적용되던 그림자 기법을 실시간 해칭 렌더링에서도 그림자 처리를 위해 사용하기 위해 해칭 렌더링의 스타일에 맞게 그림자를 그릴 수 있도록 그림자의 영역에 대해서 해칭의 느낌을 표현할 수 있는 방법을 모색하고 방안을 제시하였다. 투영 텍스쳐 그림자 기법에 Tonal Art Map과 기법을 접목하여 실시간 해칭 렌더링에서 자연스러운 해칭 그림자를 표현할 수 있는 방안을 연구 모색 하였다.

투영 텍스쳐 그림자 기법과 연동하여 실시간 해칭 렌더링에서 사용되는 Tonal Art Map을 재사용하거나 또는 그림자 처리에 필요한 스트로크 텍스쳐를 직접 생성하여 그림자의 영역에 대해서만 스트로크를 표현하는 방안을 DirectX 그래픽 라이브러리 기반에서 HLSL을 이용하여 구현해 보았다. 실제 일반적인 그림자 기법을 해칭 렌더링에 적용한 경우와 해칭 그림자를 생성한 후 해칭 렌더링에 적용한 화면을 비교한 후 일반적인 그림자 처리 기법을 사용했을 때 나타나는 해칭 렌더

링 상의 부자연스러운 화면을 그림자 처리 부분에 대해서도 해칭 스트로크 처리가 가능하도록 하여 그림자 처리가 더해진 자연스러운 해칭 화면을 렌더링할 수 있는 결과를 보여주었다.

기본적으로 투영 텍스쳐 그림자는 그림자 처리 기법의 특성상 자기 그림자를 그릴 수 없다. 자기 그림자를 그리지 않고 물체를 지면에만 그림자처리 하기를 원한다면 해칭 렌더링에서도 투영 텍스쳐를 사용하여 그림자의 표현이 가능하다.

또한 투영 텍스쳐 그림자 처리에서는 빛과 물체의 변화에 따라 실시간 해칭 그림자 처리가 이루어지도록 하였지만 투영 텍스쳐 그림자의 색상을 판별해야 하기 때문에 높은 해상도에서 그림자의 구체적인 경계를 살펴보면 어색한 경계선이 나타남을 확인할 수 있다. 이러한 문제는 투영 텍스쳐 자연스러운 해칭 그림자를 표현하기 위해서는 문제가 될 수 있으며 해결해야할 요소로 존재한다.

▌참 고 문 헌▐

[1] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein, "Real-Time Hatching", 2001.

[2] 타카시이마기레, 김용준 역, "DirectX 9 셰이더 프로그래밍"

[3] Tomas Akenine-Moller, Eric Haines, "Real-Time Rendering Second Edition"

[4] Wolfgang F. Engel, "Direct3D ShaderX : Vertex and Pixel Shader Tips and Tricks"

[5] Jennifer Fung, Oleg Veryovka, "Pen-and-ink textures for real-time rendering",

[6] Randima Fernando, Mark J. Kilgard, 김규열, "Cg로 배우는 셰이더 프로그래밍"

[7] Wolfgang Engel, "ShaderX2 : DirectX 9 셰이더 프로그래밍"

[8] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein, "Fine Tone Control in Hardware Hatching", 2002

[9] Bert Freudenberg, Maic Masuch, and Thomas Strothotte "Real-Time Halftoning : A Primitive For Non-Photorealistic Shading", 2002.

[10] ETRI, "Recent Trends in Non-Photorealistic Rendering"

[11] Rami Annala, "Shadow Maps", 2002.

[12] Eric Haines, Tomas Moller, "Real-Time Shadows"

[13] Randima Fernando, "GPU Gems"

[14] Emil Praun, Adam Finkelstein, Hugues Hoppe, "Lapped Textures", 2000.

[15] 김용준, "IT EXPERT 3D 게임 프로그래밍"

[16] 김용준, 셰이더의 종류

17] 김용준, HLSL문법(www.3dstudy.net 자료실)

[18] http://developer.nvidia.com

[19] http://research.microsoft.com/%7Ehoppe/