http://dx.doi.org/10.14400/JDC.2015.13.12.367

모바일 엔터테인먼트 애플리케이션을 위한 혼합적 시각 효과 생성 방법

김병철
서울대학교 정보문화학전공

A Hybrid Generation Method of Visual Effects for Mobile Entertainment Applications

Byung-Cheol Kim
Information and Culture Technology Studies, Seoul National University

요 약 모바일 장치의 비약적 성능 향상을 바탕으로 스마트폰 게임 등 모바일 엔터테인먼트 애플리케이션들이 3차원 렌더링을 도입하여 사실적인 시각 효과를 제공하는 경우가 증가하고 있다. 그러나 사진 수준의 고품질 시각 효과를 제공하기 위해서는 광선추적법 등의 광학 시뮬레이션이 필수적이지만, 모바일 장치의 애플리케이션 프로세서(AP)와 메모리, 배터리 성능의 한계로 인해 모바일 장치에서 그러한 계산집중적 렌더링 기법을 활용하는 것은 여전히 실용적이지 못하다. 본 논문에서는 사전 계산을 포함한 다양한 렌더링 기법을 혼합적으로 사용하여 모바일 장치에서도 실사 수준의 시각 효과를 생성해 낼 수 있는 혼합 렌더링 방법을 제안한다. 이 방법은 모바일 장치에서도 표준적으로 활용되고 있는 그래픽스 라이브러리인 OpenGL 쉐이더를 바탕으로 여러 기법을 혼합적으로 활용하므로 실제 개발에 바로 사용될 수 있다. 또한 사전 계산의 결과가 렌더링 시에는 단순히 계수의 사칙연산에 사용되므로 응답성을 거의 떨어뜨리지 않아 사용자와의 상호작용이 필수적인 모바일 게임 등에도 실제 활용될 수 있을 것으로 기대된다.

주제어 : 모바일 애플리케이션, 시각 효과, 하이브리드 렌더링, 광학 시뮬레이션, 쉐이더

Abstract This paper proposes a hybrid rendering method which combines pre-computed global illumination results and interactive local illumination techniques and thus could interactively produce photo-realistic visual effects for mobile entertainment applications. The proposed method uses the programmable shading capability of OpenGL, a *de facto* standard for computer graphics library so that it can be deployed in a real-world development environment. Also, it increases the rendering time by a negligible amount compared to normal rendering time since the pre-computed results are used as operands of plain arithmetic operations. Therefore it is expected to be applicable in practice for mobiles games which require real-time responsiveness to users.

Key Words : Mobile Applications, Visual Effects, Hybrid Rendering, Global Illumination, Shader

Received 24 October 2015, Revised 27 November 2015
Accepted 20 December 2015
Corresponding Author: Byung-Cheol Kim
(ITCT Studies, Seoul National University)
Email: clorvie@gmail.com

ISSN: 1738-1916

1. 서론

스마트폰을 필두로 모바일 장치들이 비약적으로 발전하면서 이를 기반으로 하는 모바일 게임 등의 엔터테인먼트 애플리케이션도 크게 늘어나고 있다. 그러나 여전히 PC 등에 비해 상대적으로 부족한 계산 성능 및 메모리 크기, 배터리 용량 제한 등이 있는 모바일 장치에서 PC 수준의 시각 효과를 제공하는 것은 어렵다.

컴퓨터 그래픽스 이미지의 품질을 결정하는 주요인은 물체 표면 색상 결정을 위한 빛 계산(Illumination Method), 즉 광학 현상 시뮬레이션 방식이다. 높은 품질을 위해서는 전역조명계산(Global Illumination) 기법[1,2]이, 빠른 응답 속도를 위해서는 지역조명계산(Local Illumination) 기법[3]이 사용된다.

전자는 빛과 물체 및 물체와 물체 간의 광학 현상을 모두 시뮬레이션하나 후자는 빛과 물체 간의 광학 현상만을 시뮬레이션 하므로 이미지 품질에서 전자는 그림자, 거울 효과 등이 표현 가능하나 후자는 빛 밝기에 따른 음영만 표현 가능한 한계가 있다. 처리 시간은 데이터 수에 따라 전자는 기하급수적으로, 후자는 선형적으로 증가하기 때문에 전자는 실시간성 혹은 상호작용성이 요구되는 애플리케이션에서 사용되기에는 어려운 한계가 있다.

이러한 상황에 대응하기 위해 OpenGL등 최신의 그래픽스 라이브러리에서 사용할 수 있는 쉐이더(shader) 기능은 여러 렌더링 기법을 혼합하여 적용할 수 있는 여지를 제공하므로 이를 바탕으로 본 연구에서는 상기의 두 가지 기법의 장점들을 모두 활용할 수 있는 혼합 렌더링 기법을 제안한다.

2. 알고리즘

본 장에서는 본 연구의 목표를 달성하기 위해 구현해야 할 각 중요 알고리즘에 대해 설명한다.

2.1 구형 하모닉스를 이용한 렌더링

구형 하모닉스(Spherical Harmonics)를 이용한 새로운 실시간 전역조명계산 기법[4-5]은 GDC2003에 정리·소개 되면서[6] 실제적으로 사용가능한 실시간 전역조명계산 기법으로서 대두되었다. 본 연구에서도 이를 바탕으로 전역조명효과를 생성하며 이 절에서는 이에 대해 상세히 설명한다.

우선 2.1.1.에서는 전역조명계산 기법의 기초적 개념인 렌더링 방정식(The Rendering Equation)에 대해서 설명하고, 2.1.2.에서는 컴퓨터로 풀기 난해한 형태의 렌더링 방정식을 근사하는 방법인 몬테 카를로 적분법(Monte Carlo integration)에 대해 설명한다.

몬테 카를로 적분법이 렌더링 방정식의 적분 자체를 샘플링을 통해 근사(approximation)함에도 불구하고 적분식 내에 존재하는 적분대상함수를 매 프레임 이미지를 렌더링 할 때마다 계산(evaluation)한다면 실시간 렌더링을 달성할 수 없다. 대부분의 이전 연구들이 이러한 이유로 실시간에 전역조명계산을 적용해 렌더링을 하기 힘들었고, 점진적 계산법 등[7,8]을 새롭게 고안하려 하였다.

그런데 여기서 우리가 적분할 적분대상함수는 구 좌표계(spherical coordinate system)로 표현되는 광자전달함수(irradiance transfer function)이다. 2.1.4.과 2.1.5.에서 설명할 SH 프로젝션(projection)은 구 좌표계 상의 함수를 주파수 도메인(frequency domain)으로 변환하여 적은 수의 계수로 (근사)분해할 수 있고, 또한 이 계수들끼리의 간단한 내적(inner product)으로 해당 함수를 복원(reconstruction)할 수 있다.

결국 몬테 카를로 적분법과 SH를 이용하면, 전처리를 통해 복잡한 광자전달함수를 적은 수의 SH 계수로 분해·근사·저장한 뒤, 렌더링 시에는 간단한 내적만으로 렌더링 방정식을 계산할 수 있으므로 실시간에 전역조명계산 효과와 거의 유사한 렌더링 결과물을 얻을 수 있다.

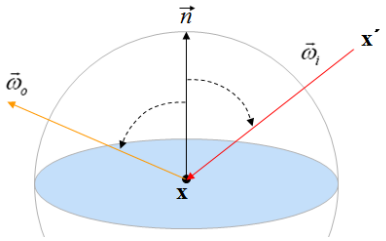
2.1.1 렌더링 방정식 (The Rendering Equation)

렌더링 방정식[9]은 SIGGRAPH86에 제임스 카지야(James Kajiya)가 발표하면서 이후 대부분의 빛 계산 연구의 근간이 되었다. 렌더링 방정식은 [Fig. 1]과 같이 한 물체의 표면이 매우 작은 평평한 조각(microfacet)의 연속된 집합이라고 가정(근사)하고, 이 조각들을 빛 반사의 최소 단위 대상으로 삼는다.

[Fig. 1] A surface as a set of contiguous microfacets

그리고 빛의 단위 대상은 광자(photon or ray)로서 물체 표면의 평평한 조각에 광자가 부딪혀 이동할 때 조각의 특성(반사율)에 따라 각 색상 채널(color channel)별로 잃는 세기의 변화가 색을 결정한다고 가정한다.

렌더링 방정식은 이러한 가정 하에서 물체 표면의 한 조각을 기준으로 [Fig. 2]와 같이 도식화하여 광학 현상을 수학적(혹은 공학적)으로 모델링 하였다.



[Fig. 2] The incidence and reflection of a ray at a microfacet or a vertex)

이는 (수식 1)과 같이 표현할 수 있으며 각 항에 대한 설명은 다음과 같다.

$$L(\mathbf{x},\overrightarrow{\omega_o}) = G(\mathbf{x},\overrightarrow{\omega_o})\left[L_e(\mathbf{x},\overrightarrow{\omega_o}) + \int_S f_r(\mathbf{x},\overrightarrow{\omega_i}\rightarrow\overrightarrow{\omega_o})L(\mathbf{x}',\overrightarrow{\omega_i})\,V(\mathbf{x},\mathbf{x}')d\omega_i\right]$$

(수식 1)

우선 점 x, x′은 언급한 가정 상의 조각 하나씩을 의미한다. 조각은 매우 크기가 작고 평평하므로 빛이 들어올 수 있는 모든 방향의 궤적을 이으면 반구(hemi-sphere) 형태가 된다.

첫 부분의 $G(\mathbf{x}, \mathrm{x}')$ 함수는 빛이 전달되는 매개체의 재질, 즉 매질의 특성까지도 고려할 수 있도록 수식화한 것이나, 일반적으로 매질은 모두 같다고 가정하며 본 연구에서도 매질의 특성은 모두 같다고 가정하였으므로 이를 제거하여도 무방하다.

$L_e(\mathbf{x},\overrightarrow{\omega_o})$ 함수는 형광 혹은 야광 물체처럼 스스로 빛을 내뿜는 물체를 모델링하기 위한 함수이다. $L(\mathbf{x}',\overrightarrow{\omega_i})$ 함수는 다른 점 x′에서 $\overrightarrow{\omega_i}$ 방향으로 점 x에 입사하는 빛의 세기를 뜻한다. 예를 들어 x′이 광원이면 빛의 세기는 최대이고, x′이 물체의 다른 점이면 또 다른 점 x′′에서 들어오는 빛을 자신의 재질에 따라 일정 부분 흡수하고 나머지를 반사하여 x로 전달한다. 즉, 각 점에서 나머지 모든($\int$) 점으로부터 빛을 전달받는 것을 수학적으로 표현한 것이다. 이 함수의 계산 결과는 거울 효과나 투명 효과 등에 영향을 미친다.

$V(\mathbf{x}, \mathbf{x}')$ 함수는 점 x와 x′ 사이에 다른 물체나 자신의 다른 점이 존재해 서로에게 빛을 전달할 수 없는 상태인지 검사한다. 가려지지 않았으면 1, 가려졌으면 0을 반환하는 함수이다. 이 함수의 계산 결과가 셀프 쉐도우의 생성 여부를 결정한다. $f_r(\mathbf{x},\overrightarrow{\omega_i}\rightarrow\overrightarrow{\omega_o})$ 함수는 광자의 입사 방향과 반사 방향에 대한 반사율 함수인 BRDF (Bi-directional Reflectance Distribution Function)를 뜻한다.

(수식 1)의 렌더링 방정식은 광자전달기반 빛 계산 기법을 포괄적 수식으로 잘 정리했다. 그러나 이를 실제 컴퓨터에서 계산하는 데는 크게 네 가지의 난점이 있다.

1. 반구상의 모든 방향에 대한 적분($\int$)을 컴퓨터로 계산하는 것이 힘들다.
2. $L(\mathbf{x}', \overrightarrow{\omega_i})$ 함수를 계산하는 것이 매우 힘들다. 수식적으로 각 점이 다른 모든 점에 대해 회귀적으로 (recursively) 연결되어 있기 때문에 알고리즘화 하였을 때 계산 시작지점과 종료지점을 설정하기 힘들기 때문이다.
3. $V(\mathbf{x}, \mathbf{x}')$ 함수를 계산하는 것이 힘들다. 이 함수는 최악의 경우 한 점에서 다른 모든 점에 대해 검사해야 하므로 $O(n^2)$의 시간 복잡도를 가지기 때문이다.
4. $f_r(\mathbf{x},\overrightarrow{\omega_i}\rightarrow\overrightarrow{\omega_o})$, 즉 어떤 물체의 BRDF를 구하는 것이 매우 힘들다.

렌더링 방정식이 발표된 이후 약 20년 동안 전역 조명 계산 기법 연구자들은 이 네 가지 문제를 해결하는 데 주력했으나 모든 문제를 완전히 해결하는 방법론은 아직까지 제시하지 못했다. 본 연구에서는 1, 2, 3번에 대해서 제한된 해결책을 제안한다. 4번 문제는 본 연구에서 다루지 않으나 다른 연구에서 제시된 대부분의 해결책을 즉시 적용할 수 있다.

1번 문제는 확률 변수와 샘플링을 통한 적분 근사법인 몬테 카를로 적분법을 사용하여 해결하고 2.1.2와 2.1.3에서 자세히 설명한다. 2번 문제는 SH 프로젝션을 이용하여 해결하고 2.1.4~2.1.6에서 구체적으로 설명한다. 3번 문제는 그래픽스 하드웨어를 바탕으로 반구를 반-정육면체(hemi-cube)로 근사하여 해결하고 2.1.7에서 자세히 설명한다.

2.1.2 몬테 카를로 적분법 (Monte Carlo Integration)

몬테 카를로 적분법의 기본 개념은 대상 함수의 계산 영역을 일정한 단위로 분할하고 각 단위 영역에서 함수의 기댓값(expected value)을 구하여 이를 모두 더한 뒤 최종 적분값을 구하는, 다시 말해 영역을 샘플링 하여 근사치를 구하는 방법이다.

만약 분할된 영역 내에서 값의 분산이 크면 이 방법은 정답과 동떨어진 결과를 제시할 것이다. 그러나 가능한 한 영역을 잘게 분할하여 영역 내의 분산을 매우 낮출 수 있다면 이 방법은 정답에 가까운 결과를 제시할 수 있다. 이를 수식적으로 전개하면 다음과 같다.

$\int_{-\infty}^{+\infty} p(x)dx = 1$ … ① 임의의 확률밀도함수 $p(x)$; ($p(x) \geq 0$에 대해서 항상 참)

$E[f(x)] = \int f(x)p(x)dx$ … ② 기댓값 정의

$E[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$ … ③ 대수의 법칙

(Law of Large Numbers; N 이 매우 클 때 성립)

②와 ③의 각 우변으로부터 트릭을 써서 식을 전개하면,

$$\int f(x) = \int \frac{f(x)}{p(x)}p(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x)}{p(x)} \quad \cdots ④$$

④에서 $w(x) = \frac{1}{p(x)}$ 로 치환하면,

$$\int f(x) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)w(x_i). \qquad \text{(수식 2)}$$

결국 임의의 함수의 적분값을 N 개의 샘플에 대해 함수값을 구하고 이에 각 샘플의 가중치($w(x_i)$; 샘플 x_i가 나타날 확률의 역수)를 곱하여 전부 더한 값으로 근사하여 구할 수 있다는 것이 몬테카를로 적분법이다.

2.1.3 구 샘플링 (Spherical Sampling)

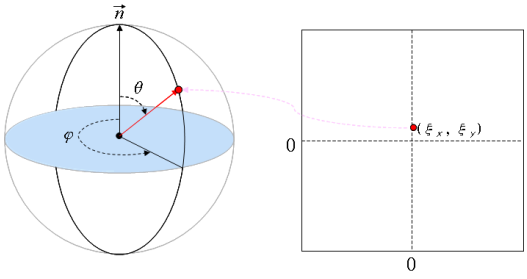
본 연구에서 몬테 카를로 적분법을 적용할 대상은 렌더링 방정식의 적분 계산이다. 그런데 이 적분의 정의역(domain)은 반구이다. 따라서 몬테카를로 적분법을 반구상에서 적용하기 위해서는 구를 단위 영역으로 나누는 것, 즉 구 샘플링(sphere sampling)이 필수적이다.

몬테 카를로 적분법을 적용하기 위해 정의역을 분할(구 좌표계 상에서 방향 샘플링)할 때 주의할 점은 가중치이다. 각 샘플에 대한 계산값은 해당 샘플의 가중치와 곱해지기 때문에 각 샘플의 가중치를 구할 수 있도록 샘플링 해야 한다.

특히 각 샘플 당 가중치가 모두 같다면 (수식 2)의 $w(x_i)$가 상수가 되어 $\sum$ 밖으로 빠져나올 수 있고, 이것은 N번의 곱셈이 1번으로 줄어드는 것을 의미한다. 그러므로 각 샘플의 가중치가 같도록 구 샘플링을 하면, 하나의 샘플이 나올 확률은 $p(x_i) = 1/(\text{구의 표면적}) = 1/4\pi$ 이므로 이의 가중치는 $w(x_i) = 1/p(x_i) = 4\pi$ 가 된다.

구 좌표계 상에서 샘플이란 두 개의 각 변수(an angle-variable pair) 로 정의되고, 이 변수들은 두 개의 실수 변수, (ξ_x, ξ_y)에 의해 다음과 같이 매핑 될 수 있다.

$$(\theta, \phi) = (2\arccos(\sqrt{1-\xi_x}), 2\pi\xi_y) \qquad \text{(수식 3)}$$



[Fig. 3] A planar sample(right) and the corresponding spherical sample(left)

이 때 두 변수 (ξ_x, ξ_y)를 비편향 확률 변수(unbiased random variable)로서 $[0\cdots1, 0\cdots1]$ 범위 내에서 샘플링하면, 각 샘플은 구 표면 상에서 균등하게 분포하게 되어 가중치가 위에서 기술했듯 상수가 된다.

2.1.4 구형 하모닉스 (Spherical Harmonics)

SH(Spherical Harmonics)는 구(sphere)를 구성하는 단위직교좌표계(orthonormal basis)를 형성한다. 이를 직관적으로 표현하면, 마치 푸리에 변환(Fourier transform)이 2차원 이미지를 주파수 영역(frequency domain)으로 분해하는 것처럼, SH는 구 표면의 값(ex. 색상)을 주파수 영역으로 분해하는 역할을 한다. 따라서 SH를 이용하면 두 각으로 표현되는 구 좌표계 상의 임의의 함수를 원하

는 개수의 주파수 영역들로 분해하고, 분해된 주파수 값들을 이용하여 구 좌표계 상의 원래 함수를 복원해 낼 수 있다.

우선 SH 함수의 정의를 기술한다. 전통적으로 SH 함수는 y로 표기하며 주파수의 범위를 결정하는 변수를 l로 표기한다.

$$y_l^m = \begin{cases} \sqrt{2}\,K_l^m \cos(m\phi) P_l^m(\cos\theta), & m>0 \\ \sqrt{2}\,K_l^m \sin(-m\phi) P_l^{-m}(\cos\theta), & m<0 \\ K_l^0 P_l^0(\cos\theta), & m=0 \end{cases} \quad \text{(수식 4)}$$

이 때,

$$l \in \mathbf{R}^+,\ -l \le m \le l$$
$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}} \quad \text{(수식 5)}$$

(수식 4)에서 P_l^m은 Associated Legendre Polynomial 이며 다음과 같이 순환적으로 정의된다.

$$(l-m)P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m \quad \text{(수식 6-1)}$$
$$P_m^m = (-1)^m (2m-1)!!(1-x^2)^{m/2} \quad \text{(수식 6-2)}$$
$$P_{m+1}^m = x(2m+1)P_m^m \quad \text{(수식 6-3)}$$

$$x!! = \begin{cases} n\cdot(n-2)\cdots 5\cdot 3\cdot 1 & x>0 \text{ odd} \\ n\cdot(n-2)\cdots 6\cdot 4\cdot 2 & x>0 \text{ even} \\ 1 & x=-1,0 \end{cases}$$

(수식 5)는 다소 복잡해 보이나 단순히 (수식 4)를 0~1로 정규화(normalization)하기 위한 부분이다.

2.1.5 SH 프로젝션 (SH Projection)

2.1.4에서 기술한 SH 함수의 정의는 보기에 매우 복잡하지만 이를 활용하여 SH 프로젝션 하는 방법은 간단하다. 어떤 구 좌표계 상의 함수 $f(s)$는 다음과 같이 주어진 주파수 l, m에 대해 상수 c_l^m로 분해(SH 프로젝션)되고, 이 상수들을 이용하여 원 함수의 근사값 $\tilde{f}(s)$를 구할 수 있다.

$$c_l^m = \int_S f(s) y_l^m(s) ds \quad \text{(수식 7; SH 프로젝션)}$$

$$\tilde{f}(s) = \sum_{l=0}^{n-1}\sum_{m=-l}^{l} c_l^m y_l^m(s) \quad \text{(수식 8; SH 복원)}$$

이 때 함수 $f(s)$의 s는 2.1.3에서 기술한 구 샘플 (θ, ϕ)이다. 즉, 각 샘플에 대해 y_l^m을 한 번만 구하고 여기에 대상 함수값을 곱하여 SH 계수값을 구할 수 있다. 아래는 (수식 2, 3, 7)을 같이 적용하여 실제 SH 계수를 구하는 식이다.

$$c_l^m = \frac{4\pi}{N}\sum_{i=1}^{N} f(\theta,\phi) y_l^m(\theta,\phi)$$
(수식 9; 구 샘플에 대한 SH 프로젝션)

이제 $f(\theta, \phi)$는 광원 함수(Light Function)가 될 수도 있고, $L(\mathbf{x}', \overrightarrow{\omega_i})V(\mathbf{x}, \mathbf{x}')$가 될 수도 있다.

2.1.6 SH를 이용한 난반사 재질 물체의 렌더링

2.1.2~2.1.5에서는 2.1.1에서 기술한 렌더링 방정식의 수식 자체를 근사하는 방법에 대해 설명하였다. 본 절에서는 렌더링 방정식에서 계산(evaluation)해야 하는 함수 자체를 근사하여 실제 렌더링에 적용하는 방법에 대해 설명한다.

(수식 1)의 원래 렌더링 방정식은 한 점(x)에서 시점 방향(ω_o)으로 나가는 빛의 세기를 구하는 식이다. 그러나 본 연구에서는 대상 물체의 재질이 난반사(diffuse reflection)만 하는 것으로 제한한다. 즉 시점 방향에 상관없이 렌더링 방정식을 재정의 한다. 그리고 물체 스스로 빛을 방출하지 않는 것으로 제한한다. 그러면 (수식 1)은 다음과 같이 재정의 된다.

$$L(\mathbf{x}) = \int_S L(\mathbf{x}', \overrightarrow{\omega_i}) f_r(\mathbf{x}, \overrightarrow{\omega_i})\, V(\mathbf{x}, \mathbf{x}') d\omega_i \quad \text{(수식 10)}$$

이를 2.1.3에서 기술한 구 샘플 $s=(\theta, \phi)$에 대해 다시 쓰면 다음과 같다.

$$L(\mathbf{x}) = \int_S L(\mathbf{x}, s) f_r(\mathbf{x}, s)\, V(\mathbf{x}, s) ds \quad \text{(수식 11)}$$

이제 우리가 계산해야 하는 것은 함수 세 개 $L(\mathbf{x}, s)$, $f_r(\mathbf{x}, s)$, $V(\mathbf{x}, s)$의 값과 이의 곱셈 및 적분이다.

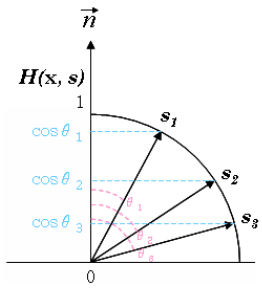
$V(\mathbf{x}, s)$를 구하는 구체적인 방법은 2.1.7에서 설명한다. $L(\mathbf{x}, s)$는 단순히 샘플 s에 대한 빛의 세기를 광원 함수를 통해 구하면 된다. $f_r(\mathbf{x}, s)$에 대해서는 램버트 법칙(Lambert's Law)을 적용한다. 본 연구에서 SHL의 대

상을 난반사 재질의 물체로 한정하였기 때문에 빛의 난반사 법칙(=램버트 법칙)을 수식화한 램버시안 함수(Lambertian Diffuse Function)[10]를 정점의 빛 반사 함수 $f_r(\mathbf{x}, s)$로 사용한다.

$$f_r(\mathbf{x}, s) = \rho_{\mathbf{x}}(\overrightarrow{n_{\mathbf{x}}} \cdot \vec{s}) \qquad \text{(수식 12)}$$

- $\rho_{\mathbf{x}}$: 점 $\mathbf{x}$의 빛 반사율(albedo)
- $\overrightarrow{n_{\mathbf{x}}}$: 점 $\mathbf{x}$의 법선 벡터
- $\vec{s}$: 구 샘플 $s=(\theta, \phi)$ 에 해당하는 방향 벡터

이 때 $\overrightarrow{n_{\mathbf{x}}}$와 $\vec{s}$는 단위 벡터이기 때문에 $\overrightarrow{n_{\mathbf{x}}} \cdot \vec{s} = \cos\theta$ 이다. 즉, 빛의 입사 각도(θ)가 커지면 커질수록 반사하는 빛의 양은 적어진다. 편의상 $(\overrightarrow{n_{\mathbf{x}}} \cdot \vec{s})$를 $H(\mathbf{x}, s)$로 표기하고, 이를 도식화 하면 [Fig. 4]와 같다.



[Fig. 4] A depiction of the Lambertian function

이렇게 구한 $L(\mathbf{x}, s)$, $f_r(\mathbf{x}, s)$, $V(\mathbf{x}, s)$ 를 곱하고 더하면 최종적으로 렌더링에 필요한 한 정점의 빛의 세기 $L(\mathbf{x})$를 구할 수 있다.

여기에서 $V(\mathbf{x}, s)$ 함수는 계산하는 데 매우 오랜 시간이 걸리는 함수이다. 따라서 렌더링 시에 매번 계산하는 것이 아니라, 사전에 한 번만 계산하고 (수식 9)를 이용하여 SH 계수로 분해하여 저장한 뒤 렌더링 시에는 (수식 8)을 이용하여 복원하여 사용한다. $f_r(\mathbf{x}, s)$는 정점 당 계산해야 하므로 $V(\mathbf{x}, s)$의 계산 시에 같이 계산하여 SH 계수화 한다. 마찬가지로 $L(\mathbf{x}, s)$도 SH 계수화 한다. 이 때 $L(\mathbf{x}, s)$도 같이 SH 계수화 하는 까닭은 다음과 같다.

편의상 $L(\mathbf{x}, s) \Rightarrow L(s)$, $f_r(\mathbf{x}, s)\, V(\mathbf{x}, s) \Rightarrow T(s)$로 표기하고, 이 두 함수에 (수식 7, 8)을 이용하여 SH 분해 및 복원하는 식을 다음과 같이 표기한다.

$$p_l^m = \int_S L(s) y_l^m(s) ds \qquad \text{(수식 13-1)}$$

$$q_l^m = \int_S T(s) y_l^m(s) ds \qquad \text{(수식 13-2)}$$

$$\tilde{L}(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} p_l^m y_l^m(s) \qquad \text{(수식 13-3)}$$

$$\tilde{T}(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} q_l^m y_l^m(s) \qquad \text{(수식 13-4)}$$

그리고 2.1.4에서 언급했듯 SH는 단위직교좌표계이다. 즉,

$$y_l^m \cdot y_k^n = \begin{cases} 1 & l=k,\ m=n \\ 0 & \text{otherwise} \end{cases} \qquad \text{(수식 14)}$$

이다. 이제 (수식 11)에 (수식 13)의 네 등식과 (수식 14)를 적용하여 풀면 다음과 같다.

$$\begin{aligned} &\int L(s)\, T(s) ds \\ &\approx \int \tilde{L}(s)\, \tilde{T}(s) ds \\ &= \int \left\{ \sum_{l=0}^{n-1} \sum_{m=-l}^{l} p_l^m y_l^m(s) \right\} \left\{ \sum_{l=0}^{n-1} \sum_{m=-l}^{l} q_l^m y_l^m(s) \right\} ds \\ &= \sum_{l=0}^{n-1} \sum_{m=-l}^{l} p_l^m q_l^m \end{aligned}$$

즉, SH가 단위직교좌표계이기 때문에 SH 복원된 함수들의 곱은 이들의 SH 계수들의 내적(inner product or dot product)만으로 구할 수 있다. 따라서 전처리 시에 대상 함수들을 SH 계수로 분해하여 저장해 두면, 렌더링 시에는 단순히 이 계수들의 내적만으로 각 정점의 색상을 결정할 수 있다.

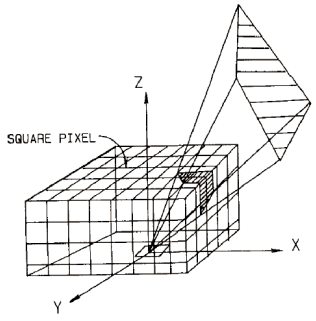
2.1.7 정점 단위의 가시성 처리

(수식 1)의 렌더링 방정식에서 $V(\mathbf{x}, \mathbf{x}')$는 각 점에서 다른 점에 대한 가시성 결정(visibility determination) 함수이다. 단순히 생각하면 이 함수는 한 점에서 나머지 모든 점에 대해 검사해야 하므로 이를 처리하기 위한 알고리즘은 $O(n^2)$, 즉 기하급수적 시간 복잡도를 가진다. 그러나 조금 더 생각하면 다양한, 선형적 시간 복잡도를 가진 방법들을 이에 적용할 수 있다.

예를 들어, BSP trees (Binary Space Partitioning trees)로 모든 정점을 관리하면, 시간복잡도는 크게 줄어든다. 그러나 이 방법은 BSP를 다각형 단위로 만들기 위해서 상당히 복잡한 작업을 수행하여야 하고, 이를 위해 별도

의 메모리를 요구하는 단점이 있다.

본 연구에서는 그래픽스 하드웨어를 이용하여 이를 처리하는 방법을 채용한다. 기본 아이디어는 1985년 Cohen 등이 제안한 반정육면체(The Hemi-Cube)를 이용한 광선 추적 기법[11]이다.



[Fig. 5] Ray tracing using a hemi-cube[11]

[Fig. 5]는 이의 핵심을 보여주는 그림으로서 [Fig. 1]에서의 연속 공간의 반구(continuous space hemi-sphere)를 이산 공간의 반정육면체(discrete space hemi-cube)로 근사했을 때, 각 점의 나머지 물체들과의 기하학적 관계를 보여준다.

이 아이디어를 최신의 그래픽스 하드웨어를 활용하여 다음과 같이 구현하면 매우 높은 성능의 광선 추적 시뮬레이션이 가능하다. 각 점을 가상 시점(viewpoint)으로 삼고 반정육면체의 각 면을 이미지 평면(image plane)으로 삼아 전체 씬을 렌더링 한 뒤 렌더링 결과 이미지 픽셀에 값이 있으면 이 픽셀에 해당하는 방향에는 어떤 물체가 존재하는 것($V(\mathbf{x}, \mathbf{x}') = 0$)이다. 그 외에는 $V(\mathbf{x}, \mathbf{x}') = 1$ 이다. 다시 말해, 각 정점 당 5개의 방향(5면)에 대해 렌더링 하고 결과 이미지의 픽셀을 검사하면 각 정점에서 빛이 바로(directly) 들어올 수 있는 방향과 들어올 수 없는(혹은 간접적으로만, indirectly) 방향을 알 수 있다.

최신의 그래픽스 하드웨어는 현재 수백만 개의 다각형을 실시간(30fps 이상)에 렌더링 할 수 있기 때문에 이를 활용하여 상기와 같이 광선 추적하면 알고리즘의 성능을 크게 높일 수 있다.

2.2 쉐도우 맵(Shadow Map)

쉐도우 맵(Shadow Map)은 1978년 랜스 윌리엄스(Lance Williams)가 처음으로 그 개념을 제시[12]하고 그 이후 1992년 마크 시걸(Mark Segal)이 이를 현실적으로 구체화[13]시키고 쉐도우 맵이라 명명한 뒤 1997년에는 OpenGL의 확장으로서 지원되고[14], 2000년에는 그래픽스 하드웨어에서 직접 지원되면서[15] 현재는 실시간 3차원 컴퓨터 그래픽스 애플리케이션에서 가장 대중적으로 활용되는 그림자 생성 기법이 되었다.

쉐도우 맵 알고리즘은 텍스쳐를 활용하여 구현되었기 때문에 근본적으로 픽셀 단위로 작동하며, 따라서 그림자의 윤곽이 뚜렷한 하드 쉐도우(hard shadow)를 생성하는 것이 장점이자 단점이다. 그러나 본 연구에서는 소프트 쉐도우 생성만 가능한 SHL의 보완책의 하나로서 쉐도우 맵 알고리즘을 채택하였기 때문에 이의 장점만을 취할 수 있다.

2.3 픽셀 단위 전반사 빛 계산

본 연구의 목표인 실시간 전역 조명 계산 방식은 SHL을 기본으로 한다. 그러나 2.1에서 SHL이 렌더링 방정식을 근사하여 적용할 때 대상을 정점 단위의 난반사(diffuse reflection) 빛 계산으로 한정했기 때문에 SHL로는 고주파수 영역(high frequency bandwidth)에 해당하는 빛 반사는 계산하지 못한다. 즉, SHL만으로는 픽셀 단위의 전반사(specular reflection) 빛 계산은 할 수 없다.

렌더링 대상 물체의 재질에 따라 전반사 빛 계산은 물체의 특성을 표현하는 데 있어 매우 중요하다. 본 연구는 SHL로 해결할 수 없는 전반사 빛 계산을 퐁 빛계산 방정식(Phong Lighting Equation)과 퐁 쉐이딩(Phong Shading)을 통해 처리하여 SHL의 단점을 보완하려 한다.

2.3.1 퐁 빛계산 식 (Phong Lighting Equation)

퐁 빛계산 방정식은 빛의 전반사 현상을 시뮬레이션한다. 가정은 한 점의 법선 벡터 $\vec{n}$을 기준으로 빛이 들어오는 각과 반사되는 각이 같다는 것이다. 이러한 가정 하에서 빛의 입사 벡터 $\vec{l}$에 대응하는 반사 벡터 $\vec{r}$은 다음과 같이 간단히 구할 수 있다.

$$\vec{r} = 2(\vec{n} \cdot \vec{l}) - \vec{l} \qquad \text{(수식 15)}$$

이렇게 구한 반사 벡터 $\vec{r}$을 해당 점에서 시점을 향하는 벡터 $\vec{v}$와 내적하고 표면의 재질에 따라 이 값을 증폭하여 전반사 되는 빛의 세기를 결정한다. 즉, 반사되는 방향과 시점의 방향이 가까우면 가까울수록, 재질의 반짝

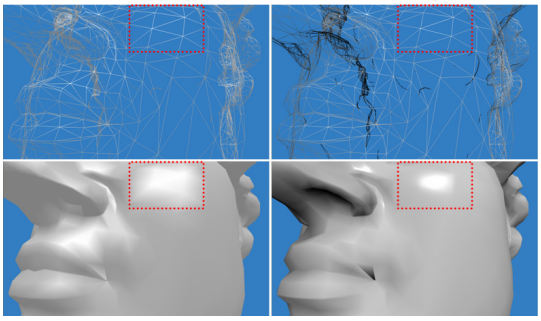
임 정도가 클수록 물체의 해당 점은 밝게 렌더링 된다.

$$L_{\mathrm{specular}} = (\vec{r} \cdot \vec{v})^{m_{\mathrm{shi}}} \quad \text{(수식 16)}$$

- m_{shi} : 물체의 반짝임 정도(shininess)

2.3.2 퐁 쉐이딩 (Phong Shading)

대부분의 그래픽스 하드웨어에서 기본으로 지원되는 구로 쉐이딩(Gouraud Shading)은 이러한 퐁 빛계산 방정식을 삼각형의 각 정점에 적용하고 이의 결과값인 색상을 보간하여 삼각형 내부를 픽셀화(rasterization)한다. 그러나 이런 방식을 이용하면 세밀하게 나타나야할 전반사 효과가 삼각형 단위로 뭉개지는 현상이 발생하여 질감 표현력이 떨어진다([Fig. 6]의 좌측).



[Fig. 6] A comparison of Gouraud shading(left) and Phong shading(right)

퐁 쉐이딩은 이를 극복하기 위해 퐁 빛계산 방정식을 픽셀 수준에서 적용한다. 즉, 정점의 색상을 보간하여 픽셀의 색상을 결정하는 것이 아니라 정점의 법선 벡터를 보간하여 픽셀에서 (수식 15, 16)을 직접 계산하여 색상을 결정하는 것이다.

이러한 퐁 쉐이딩 결과의 우수함([Fig. 6]의 우측)에도 불구하고 기존의 OpenGL v1.5까지는 이를 그래픽스 하드웨어 상에서 직접 구현할 수 없었다. 픽셀 수준의 빛계산은 렌더링 성능을 크게 떨어뜨리므로 OpenGL v1.5에서는 픽셀 수준 빛계산 방법을 제공하지 않았기 때문이다. 그러나 2.4에서 설명할 고수준 쉐이더와 이를 지원하는 최신 그래픽스 하드웨어는 이러한 제약을 크게 완화하였다.

2.4 고수준 쉐이더를 이용한 복합 렌더링

2.1~2.3에서 기술한 기법들은 결국 한 픽셀의 색상값을 결정하는 쉐이딩 기법들이다. 그러나 3차원 그래픽스 프로그래밍 표준인 OpenGL v1.5까지는 쉐이딩 기법이 몇 가지로 고정되어 있기 때문에 본 연구에서 개발하는 다양한 쉐이딩 기법들을 이를 이용해서는 구현하기 힘들 뿐만 아니라 이들을 전부 통합하여 적용하는 것은 더욱 힘들다. OpenGL ES 1.x 버전은 고정 (렌더링) 기능 파이프라인(fixed function pipeline)만을 지원하기 때문에 마찬가지이다.

OpenGL v2.0 및 OpenGL ES 2.x 부터는 쉐이딩 알고리즘을 프로그래머가 직접 작성할 수 있도록 개선됨으로써 본 연구의 목표인 복합 렌더링을 그래픽스 하드웨어 상에서 한번(one-pass)에 처리할 수 있는 근간을 제공한다. 2.4.1에서는 OpenGL에서 채택한 고수준 쉐이딩 언어를 활용하여 복합 렌더링을 처리하는 방법에 대해 설명한다.

2.4.1 GLSL을 이용한 복합 렌더링

본 연구에서는 총 다음의 세 가지의 쉐이딩을 개발하고 이들을 통합하여 씬을 렌더링 하고자 한다.

1. SHL을 이용한 정점 단위 쉐이딩
2. 쉐도우 맵을 이용한 픽셀 단위 쉐이딩
3. 퐁 빛계산 모델과 퐁 쉐이딩을 이용한 픽셀 단위 쉐이딩

GLSL을 이용하면 정점의 색상을 결정하는 단계와 픽셀의 색상을 결정하는 단계를 각각 버텍스 쉐이더와 픽셀 쉐이더를 이용하여 대체(override)할 수 있다. 우선 SHL은 순전히 정점 단위의 작업이기 때문에 버텍스 쉐이더에서 처리한다. 쉐도우 맵은 정점 단위의 작업과 픽셀 단위 작업이 병행되어야 하기 때문에 버텍스 및 픽셀 쉐이더 양쪽에서 처리한다. 퐁 쉐이딩은 픽셀 단위 작업이므로 픽셀 쉐이더에서 처리한다. 다음의 각 절에서는 이들의 알고리즘을 기술한다.

2.4.1.1 SHL을 이용한 정점 단위 쉐이딩

<버텍스 쉐이더>

- SH 계수 개수 : N
- 광원 함수의 SH 계수 : c_l[N]
- 정점 $v0$의 SH 계수 : c_{v_0}[N]

- 정점의 난반사 밝기 : $I_{v_0} = c_l \cdot c_{v_0} = \sum_{i=1}^{N} c_l[i] c_{v_0}[i]$
- 정점의 난반사 재질 : $\{d_r, d_g, d_b\}$
- 정점의 난반사 색상 : $Color_{v_0}\{r,g,b\} = \{d_r, d_g, d_b\} \times I_{v_0}$
- **보간 변수(varying variable)** : $Color_{v_0}$

2.4.1.2 쉐도우 맵을 이용한 픽셀 단위 쉐이딩

<버텍스 쉐이더>

- 정점 v_0의 물체 좌표계 상의 좌표(x, y, z, w) : v_0
- 물체->월드 좌표계 변환 행렬 : M_{world}
- 시점 좌표계 행렬(4x4) : M_{eye}
- 시점 좌표계 행렬의 역행렬 : M_{eye}^{-1}
- OpenGL MODELVIEW 행렬 : $M_{modelview} = M_{eye} M_{world}$
- 시점 좌표계 상의 좌표 : $v_{0_{eye}} = M_{modelview} v_0$
- 광원 좌표계 행렬 : M_{light}
- 광원 좌표계 상의 정점 좌표 : $v_{0_{light}} = M_{light} M_{eye}^{-1} v_{0_{eye}}$
- **보간 변수** : $v_{0_{light}}$

<픽셀 쉐이더>

- 삼각형 세 정점의 보간 변수 $v_{0_{light}}$, $v_{1_{light}}$, $v_{2_{light}}$로부터 보간된 변수: $v_{012_{light}} = \beta(\alpha v_{0_{light}} + (1-\alpha) v_{1_{light}}) + (1-\beta) v_{2_{light}}$
- $v_{012_{light}}$의 x, y, z, w : $x_{light}, y_{light}, z_{light}, w_{light}$
- 현재 픽셀의 쉐도우 맵 상의 좌표 및 깊이값 : $\{x', y', z'\} = \{x_{light}/w_{light}, y_{light}/w_{light}, z_{light}/w_{light}\}$
- 쉐도우 맵의 깊이값 : $z_{shadow-map}$ = texture_access(shadow_map_id, x′, y′)
- **그림자 여부 판별 변수** :

$$p_{shadow} = \begin{cases} 0 & z_{shadow-map} < z' \\ 1 & otherwise \end{cases}$$

2.4.1.3 퐁 쉐이딩을 이용한 픽셀 단위 쉐이딩

<버텍스 쉐이더>

- 현재 정점에서 시점을 향하는 벡터 : $e_{v_0} = -v_{0_{eye}}$
- 법선 벡터 : n_{v0}
- **보간 변수** : e_{v_0}, n_{v0}

<픽셀 쉐이더>

- 현재 픽셀에서 광원으로 향하는 방향 : l
- 빛 반사벡터 : $r = 2(n_{v_{012}} \cdot l) n_{v_{012}} - l$
- 픽셀의 반짝임 정도 : $m_{shininess}$
- 전반사 빛 세기 : $I_{specular} = \max(r \cdot e_{v_{012}}, 0)^{m_{shininess}}$
- 픽셀의 전반사 재질 : $\{s_r, s_g, s_b\}$
- 픽셀의 전반사 색상 : $Color_{specular}\{r,g,b\} = \{s_r, s_g, s_b\} \times I_{specular}$

2.4.1.1~2.4.1.3까지의 각 픽셀 쉐이더의 계산값을 조합하여 최종적으로 해당 픽셀의 색상을 다음과 같이 결정할 수 있다.

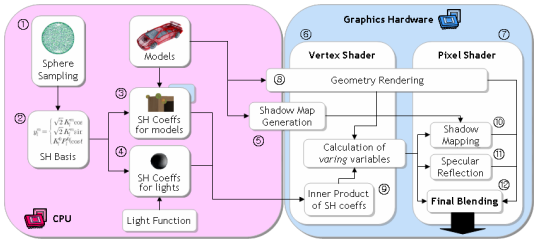
<픽셀 쉐이더 - 최종 블렌딩(final blending)>

- 최종 픽셀 색상 : $Color = p_{shadow} \times (Color_{v_{012}} + Color_{specular})$

3. 구현

본 장에서는 2장에서 설명한 알고리즘을 실제로 구현한 내용에 대해 기술한다. 3.1.에서는 구현한 기법의 전반적 구성 및 작동 방식을 설명하고, 3.2.에서는 각 쉐이딩의 구현 사항에 대해 설명하며, 3.3.에서는 고수준 쉐이더를 이용하여 렌더링을 수행하는 실제 GLSL코드를 설명한다. 3.4.에서는 구현한 기법을 다양한 모델에 적용해 결과를 보이고, 그 성능을 평가한다.

3.1 프레임워크



[Fig. 7] An overview of the hybrid rendering framework

[Fig. 7]은 구현한 소프트웨어의 전체 작업 흐름을 도식화한 것이며 각 단계에 대한 설명은 다음과 같다.

- ① SH 프로젝션을 수행하기에 앞서 3차원 공간을 구 좌표계에서 샘플링 한 뒤 ② 각 샘플링 된 방향에 대해 SH 계수값을 계산하고 이를 별도의 파일로 저장.
- ③ 그래픽스 하드웨어를 활용한 광선 추적 기법을 통해 대상 물체의 광자전달경로를 찾고 이를 샘플링된 방향에 대해 SH 계수로 분해하여 물체의 기하학적 정보(geometry)와 함께 파일로 저장.
- ⑥⑦ 렌더링에 들어가기 전에 GLSL 프로그램을 OpenGL 드라이버를 통해 컴파일 하여 그래픽스 하드웨어의 기존 쉐이딩 프로그램을 대체.
- SH 계수로 광자전달경로가 저장된 3차원 모델을 CPU에 읽어 들임.
- ④ 광원을 구 좌표계에서 샘플링 및 SH 프로젝션 하여 광원의 SH 계수를 구한다. 광원이 움직이는 환경에서는 매 프레임 렌더링시 이를 재계산.
- ⑤ 광원의 시점에서 물체를 렌더링해 쉐도우맵 생성.
- 메인 메모리로 읽어 들인 모델의 SH 계수들과 광원의 SH 계수들을 그래픽스 하드웨어로 전송.
- ⑧ 물체의 기하 정보를 일반적인 OpenGL 명령어를 통해 렌더링 하면서 그래픽스 하드웨어의 버텍스 쉐이더와 픽셀 쉐이더는 새롭게 설정된 쉐이딩 알고리즘에 따라 ⑨ SHL를 수행하고 이의 결과에 ⑩ 쉐도우 맵 및 ⑪ 퐁 쉐이딩의 결과를 ⑫ 섞어(blending) 최종적으로 프레임버퍼 각 픽셀의 색상값을 결정.

3.2 쉐이딩 구현

[Fig. 7]의 ①② 과정에서 2.1의 알고리즘에 따라 구 샘플링과 이의 SH 베이시스를 계산할 때 결정해야 하는 인자는 샘플링 개수와 SH 밴드 개수이다. 본 연구에서는 구를 10,000개의 방향으로 샘플링 하고, SH 밴드(l) 개수를 4개로 설정하고 계산하였다. 따라서 SH 베이시스를 위한 계수들은 $10,000 \times 4^2 = 160,000$ 개가 생성된다.

[Fig. 7]의 ③ 과정에서 각 모델의 SH 계수를 계산할 때 구해야 하는 각 정점에서의 구 샘플 당 가시성 검사는, 각 정점을 시점 위치로, 정점을 중심으로 하는 반정육면체의 각 면과 이에 해당하는 피라미드를 뷰프러스텀(view frustum)으로 삼아 모델을 렌더링 하여 달성한다.

이 때 렌더링 결과 이미지를 프레임버퍼의 색상 부분(rgb)를 쓰지 않는다. 색상은 초기화 시에 배경 색상을 지정하고 렌더링 하여도 렌더링 결과 픽셀 중 배경색과 같은 색이 나타날 수 있기 때문이다. 간단하게는 배경은 검은 색으로, 모든 물체는 흰색으로 그려도 되지만, 일반적으로 구현되는 렌더링 함수들이 매 폴리곤을 렌더링 할 때마다 그에 맞는 색상을 지정하기 때문에 이를 모두 고쳐서 구현하는 것은 비효율적이다.

본 연구에서는 스텐실 버퍼를 사용한다. 스텐실 버퍼를 이용하면 한 프레임의 렌더링 처음과 끝에서만 명령어를 넣어주면 원하는 효과를 얻을 수 있기 때문이다. 즉, 스텐실 버퍼를 렌더링 시작 시 0으로 초기화 하고 픽셀에 실제 렌더링이 되면 해당 픽셀의 스텐실 값을 1로 갱신한 뒤, 이의 결과를 OpenGL의 glReadPixels() 함수를 이용하여 얻어서 픽셀의 스텐실 값이 1이면 가려지는 것으로, 0이면 가려지지 않는 것으로 판단한다.

[Fig. 7]의 ⑤ 과정의 쉐도우 맵 생성은 매 프레임 렌더링 시 이루어진다. 실제 물체 렌더링 직전에 광원의 위치와 방향을 시점으로 삼아 물체를 P-버퍼에 렌더링 하고 이를 OpenGL의 확장 명령어인 wglBindTexImage(), wglReleaseTexImage() 를 이용하여 텍스쳐로 변환한다. 그리고 광원 좌표계 상에서의 각 정점의 위치를 구하기 위해 쉐도우 맵 텍스쳐에 해당하는 텍스쳐 행렬에 쉐도우 맵 생성시 사용했던 프로젝션(GL_PROJECTION) 및 모델뷰(GL_MODELVIEW) 행렬을 넣어주고, 텍스쳐 좌표 자동 생성 기능을 활성화 시킨다.

3.3 고수준 쉐이더를 이용한 혼합 렌더링

3.3.1 버텍스 쉐이더

[Fig. 8]은 GLSL을 이용한 혼한 렌더링용 버텍스 쉐이더 코드를 나타낸다. 주요 계산 내용은 다음과 같다.

2,3번째 줄의 사용자 보간 변수(user-specified varying variables)는 정점 당 법선 벡터와 시점 벡터를 저장한다. 9번째 줄은 현재 정점의 좌표값을 변환(transformation; homogeneous model-view transformation ⇒ projection)하여 정점이 어떤 위치의 픽셀에 해당하는지를 계산한다. 반드시 처리해야 한다. gl_Position에 값을 설정하지 않으면 정상적으로 버텍스 쉐이더 프로그램이 컴파일 되지 않는다.

본 버텍스 쉐이더에서는 기본적으로 멀티텍스처링(multi-texturing)을 사용한다. 0번 텍스쳐 구조는 일반적

인 모델 텍스쳐를 위해 사용하고, 3번 텍스쳐는 쉐도우 맵을 위해 사용한다. 따라서 12번째 줄에서 0번 텍스쳐의 텍스쳐 좌표는 그대로 기본 보간 변수를 통해 픽셀 쉐이더로 넘겨주고, 15번째 줄에서 21번째 줄에서는 정점을 시점 좌표계에서 월드 좌표계, 월드 좌표계에서 광원 좌표계로 변환하여 텍스쳐 좌표 값으로 픽셀 쉐이더에 넘긴다.

```
1 // to fragment shader
2 varying vec3 vNormal;
3 varying vec3 vEye;
4
5 void main()
6 {
7     // calculate a vertex position in the conventional way;
8     // from object-space into clipping-space
9     gl_Position = ftransform();
10
11    // bypass texture coordinates
12    gl_TexCoord[0]  = gl_MultiTexCoord0;
13
14    // generate eye-linear texture coordinates for shadowing
15    vec4 ecPos = gl_ModelViewMatrix * gl_Vertex;
16    gl_TexCoord[3].s = dot(ecPos, gl_EyePlaneS[3]);
17    gl_TexCoord[3].t = dot(ecPos, gl_EyePlaneT[3]);
18    gl_TexCoord[3].p = dot(ecPos, gl_EyePlaneR[3]);
19    gl_TexCoord[3].q = dot(ecPos, gl_EyePlaneQ[3]);
20    // transform them to light frustum
21    gl_TexCoord[3] = gl_TextureMatrix[3] * gl_TexCoord[3];
22
23    // for calculating specular components in fragment shader
24    vEye = -normalize(ecPos.xyz);
25    vNormal = normalize(gl_NormalMatrix * gl_Normal);
26
27    // sh ambient & diffuse
28    vec4 temp = vec4(0.0);
29    temp = temp + gl_BackMaterial.ambient * gl_MultiTexCoord1;
30    temp = temp + gl_BackMaterial.diffuse * gl_MultiTexCoord2;
31    temp = temp + gl_BackMaterial.specular * gl_MultiTexCoord4;
32    temp = temp + gl_BackMaterial.emission * gl_MultiTexCoord5;
33
34    gl_FrontColor = vec4(vec3(dot(temp, vec4(1.0,1.0,1.0,1.0)))
35                         * gl_FrontMaterial.diffuse.rgb, 1.0);
36 }
```

[Fig. 8] Vertex shader code

24, 25번째 줄은 퐁 빛계산 방정식에 필요한 두 인자인 법선 벡터와 시점 벡터를 계산해서 사용자 보간 변수에 저장한다. gl_NormalMatrix(gl_ModelViewMatrix 왼쪽 상위 3x3행렬의 역행렬의 전치행렬-transpose-행렬[16])는 버텍스 쉐이더가 작동되기 전 GLSL에서 자동으로 계산해 주는 행렬로, 여기에 물체 좌표계 상의 법선 벡터를 곱하면 시점 좌표계 상으로 법선 벡터가 변환된다.

28~32번째 줄은 뒷면 재질 변수를 통해 버텍스 쉐이더로 넘어온 광원의 SH 계수 16개와 텍스쳐 좌표 변수(1,2,4,5; 0-실제 텍스쳐, 3-쉐도우 맵)를 통해 넘어온 정점의 SH 계수 16를 내적하는 부분이다.

최종적으로 34,35번째 줄에서는 내적을 통해 구한 정점의 난반사 빛 세기에 실제 색상값을 곱하여 정점의 색상을 결정한다.

3.3.2 픽셀 쉐이더

```
1 // textures
2 uniform sampler2D decalMap;          // 0
3 uniform sampler2DShadow shadowMap;  // 3
4
5 // flags
6 uniform int iTexEnabled;
7 uniform int iShadowMapped;
8 uniform int iCalcSpecular;
9
10 // in the eye coordinate system if not bump mapped.
11 // in the surface local coordinate system, otherwise.
12 varying vec3 vNormal;
13 varying vec3 vEye;
14
15 void main()
16 {
17     // texture
18     vec4 surfaceColor;
19     if (iTexEnabled == 1)
20         surfaceColor = texture2D(decalMap, gl_TexCoord[0].st);
21     else surfaceColor = vec4(1.0);
22
23     // specular
24     vec3 reflectVec0 = normalize(reflect(
25                         -gl_LightSource[0].position.xyz, vNormal));
26     vec3 specular0 = vec3(0);
27
28     // check if SHADOWED
29     vec3 lightdimming = vec3(0.6);
30     vec3 shadow = vec3(0.4);
31     if (iShadowMapped == 1) {
32         shadow = shadow2DProj(shadowMap, gl_TexCoord[3]).rgb * vec3(0.4);
33     }
34
35     // Phong lighting equation
36     if (iCalcSpecular == 1 && shadow.r == 0.4   // only when not shadowed
37         && gl_FrontMaterial.specular.r > 0.1) {
38         specular0 = vec3(pow(max(dot(vEye, reflectVec0), 0.0), 64))
39                     * gl_FrontMaterial.specular.rgb;
40     }
41
42     // final blending
43     gl_FragColor.rgb = (lightdimming + shadow) *
44                         (specular0 * gl_LightSource[0].specular.rgb
45                         + gl_Color * surfaceColor.rgb);
46     gl_FragColor.a = 1.0;
47 }
```

[Fig. 9] Pixel(fragment) shader code

[Fig. 9]는 GLSL을 이용한 혼합 렌더링용 픽셀 쉐이더 코드를 나타내며 주요 계산 내용은 다음과 같다.

2,3번째 줄은 텍스쳐 ID이다. 쉐이더에서는 텍스쳐를 액세스 하기 위해 반드시 sampler 형식으로 텍스쳐 ID를 정의해 두어야 한다.

6~8번째 줄에 정의된 uniform 변수들은 플래그용 변수들이다. 쉐이더에서는 현재 픽셀이 텍스쳐 활성화(texture-enabled)인지 아닌지조차 알 수 없기 때문에 상황에 맞는 렌더링을 위해 uniform 변수를 통해 애플리케이션 수준에서 직접 쉐이더로 이를 알려줘야 한다.

12,13번째 줄의 사용자 보간 변수는 버텍스 쉐이더에서 넘어오는 보간된 입력값들이다. 18~21번째 줄은 우선 현재 픽셀에 텍스쳐가 매핑되어야 하는지 체크하고 그렇다면 텍스쳐 액세스를 통해 현재 픽셀에 맞는 텍셀(texel; texture element)값을 가져온다. 24번째 줄에서는 (수식 15)를 실제로 계산한다. GLSL에서는 이 기능을 하드웨어에 직접 구현할 수 있도록 reflect()라는 명령어를 기본으로 제공한다. 여기에서도 이를 이용한다. 광원의 방향 벡터는 OpenGL 기본 변수에서 바로 얻을 수 있으며, 이때 이 방향 벡터는 이미 시점 좌표계로 기술되어 있다.

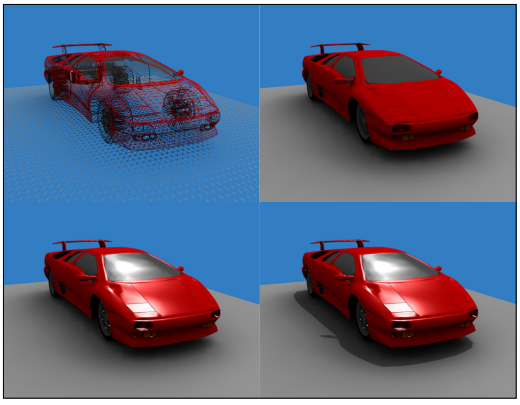
32번째 줄에서는 쉐도우 맵 액세스를 통해 하드 쉐도우를 입힐지 말지를 결정한다. 셀프 쉐도우는 SHL을 통해 결정하기 때문에 하드 쉐도우는 완전히 픽셀을 검게 만들지 않고 앰비언트(ambient)와 같이 어느 정도의 기본 밝기를 남겨 둔다. 36~40번째 줄에서는 (수식 16)을 계산한다. 현재는 반짝임 정도값(shininess)를 64로 고정해 두었으나, OpenGL 기본 변수를 통해 재질에 정의된 값을 쓸 수도 있다. 46번째 줄은 현재 기법은 투명 물체는 처리하지 않는 것을 뜻한다. 즉 언제나 알파(alpha; inverse transparency)값은 1이다. 43~45번째 줄은 최종적으로 SHL과 쉐도우 맵, 퐁 쉐이딩을 하나의 색상으로 조합(blending)한다.

이제 이 색상은 프레임버퍼로 가기 직전 프레임버퍼와의 픽셀 테스트(e.g., 스텐실 테스트, 깊이 테스트, 등등)를 거친 뒤 최종적으로 스크린에 맺히게 된다.

3.4 실험 및 성능분석

렌더링은 △스포츠카, △두상, △책장 모델들에 대해 수행하였다. 각 실험 결과 이미지를 다음 각 절에서 보이고, 이들 모델의 특성과 이에 따른 각 세부 기법의 적용 장단점에 대해 기술한다.

3.4.1 자동차 모델의 렌더링



[Fig. 10] A rendering of a car model (53,126 tris.)
Wireframe(top-left) ⇒ SHL only(top-right)
⇒ +Specular reflection(bottom-left)
⇒ +Shadow map(bottom-right)

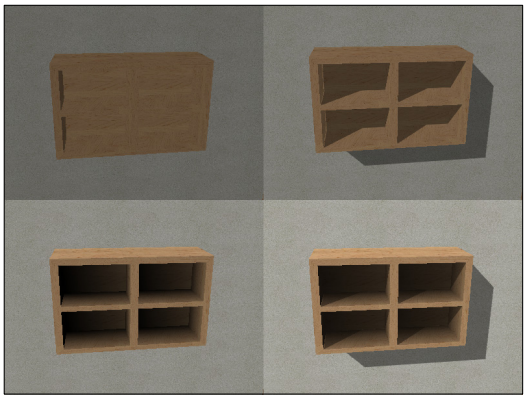
[Fig. 10]은 폴리곤 44,934개의 스포츠카 모델을 폴리곤 8192개의 바닥에 올려놓고 렌더링 한 결과이다. 바닥에 하드 쉐도우가 맺히고 스포츠카의 하부에는 모델 자체에 의한 소프트 쉐도우가 효과적으로 표현되며, 유리창이나 보닛 부분 등에는 전반사 효과가 잘 표현된다.

3.4.2 두상 모델의 렌더링

[Fig. 11]은 다비드 상(David statue)의 모델 중에서 머리 부분만 렌더링한 결과이다. 두상은 부드러운 굴곡이 오밀조밀하게 많은 형태이기 때문에 SHL의 셀프 쉐도우 효과를 가장 잘 보여준다. 그러나 이러한 굴곡의 형태는 쉐도우 맵을 이용할 경우의 단점인 노이즈 발생을 늘려서 쉐도우 맵을 적용하기 힘들게 한다.

[Fig. 11] A rendering of David head (16,111 tris.)
(Top: diffuse only, Bottom: with specular reflection)
(Left: without SHL, Right: with SHL)

3.4.3 책장 모델의 렌더링



[Fig. 12] A rendering of book shelf model (76 tris.)
(Top: using OpenGL fixed functions, Bottom: using SHL)
(Left: without a shadow map, Right: with a shadow map)

책장 모델은 76개의 폴리곤에 하나의 텍스쳐가 전체적으로 매핑 되어 있다. 모델의 면들이 평평하거나 서로 수직을 이루고 있고, 주로 안쪽에 공간이 들어가서 형성돼 있다. 재질은 전반사가 없이 난반사만 한다. 이러한 경우 굉장히 적은 수의 폴리곤 모델임에도 불구하고, SHL과 쉐도우 맵을 이용하면 훌륭한 렌더링 결과물을 볼 수 있다.

3.4.4 성능 분석

우선 표 3에서 실험한 모델들의 폴리곤 개수와 처리 시간을 정리하였다. 이 중 렌더링 속도는 광원이 매 프레임 변화하고 사용자의 인터랙션이 지속적으로 가해지는 상황, 즉 제약이 매우 적은 일반적 환경에서 측정하였다.

〈Table 1〉 Statistics of models

Model	No. of Tris.	File Size(MB)			Pre-computing Time (sec.)	Note
		Model	SH Coeff.	Total		
Car	53K	2.630	58.37	61.00	804	SHL+SM+PS
David	16K	0.980	17.62	18.60	207	SHL[1)]+PS
David (High res.)	300K	18.300	330.70	349.00	33,739	SHL+PS[2)]
Book shelf	76	0.008	0.001	0.009	1	SHL+SM[3)]

[1)]SHL : SH Lighting, [2)]PS : Phong Shading, [3)]SM : Shadow Map

[Fig. 10~12]는 본 연구 결과의 정성적 실험 결과를, <Table 1>은 정량적 실험 결과를 보였다. 아래의 <Table 2>는 이를 정리하여 하나의 틀에서 본 연구 결과를 평가한다.

〈Table 2〉 Quantitative/Qualitative evaluations

Categories				Level
Quantitative	Pre-computing	Time	*About a small scene of 50K polygons	800 sec.
		Memory		60 MB
	Rendering	Time		30 fps
		Memory		65 MB
Qualitative	Optic-phenomenally	Ambient	Inter-reflection	Yes
			caustics	No
		Diffuse		Yes
		Shadow	Self	Yes
			Soft	Yes
			Hard	Yes
		Specular		Yes
	Methodologically	Processing Unit	Vertex	Yes
			Pixel	Partly Yes
		Light Sources	Area	Yes
			Directional	Yes
			Point (local)	No
		Flexibility	Movable Light Sources	Yes
			Movable Objects	Yes

4. 결론

4.1 성과

본 연구의 목표는 상호작용성을 보장하는 고수준 렌더링 기법의 개발이다. 이의 구체적 방법으로서 본 연구에서는 SHL과 쉐도우 맵 및 퐁 쉐이딩을 고수준 쉐이더를 통해 혼합 렌더링 하는 방법을 개발하였다. 정량적으로는 약 15만개의 텍스쳐 있는 배경 모델 위에 약 5만개 폴리곤 개수의 SH 계수가 계산된 모델을 30 fps 이상의 속도로 렌더링이 가능했다. 정성적으로는 배경 모델은 하드 쉐도우, 대상 모델은 셀프 쉐도우, 하드 쉐도우 및 전반사 효과까지 포함한 렌더링이 가능했다.

그러나 현재는 전처리 시에 정점 당 가시성 검사를 CPU와 GPU를 모두 사용해서 하고 있다. 그래픽스 하드웨어 구조 특성상 CPU ⇒ GPU보다 GPU ⇒ CPU가 훨씬 느리다. 그러나 현재 고수준 쉐이더와 그래픽스 하드웨어의 성능으로 보아 GPU ⇒ CPU로 데이터를 옮기지 않고도, 즉 GPU만 사용하여 본 연구 결과와 동일한 결과를 얻을 수 있을 것으로 생각한다. 따라서 CPU없이 그래픽스 하드웨어 상에서 SH 계수 계산이 가능한 기법의 개발이 향후 필요하고, 이것이 달성되면 상대적으로 큰 전처리 시간을 획기적으로 줄일 수 있을 것으로 기대한다.

4.2 활용방안

본 연구에서 개발한 렌더링 기법를 사용하면, 실제 사진과 유사한 이미지 품질로 가상 씬을 렌더링 할 수 있으므로 게임에 본 기술을 사용하면 훨씬 강력한 시각적 몰입감(immersion)을 유저에게 제공할 수 있다. 특히 이 기술은 고정된 시점이나 빛 방향뿐만 아니라 원하는 방향의 광원으로 원하는 시점에서 자유로이 씬을 렌더링 할 수 있기 때문에 게임의 필수 요구 조건인 상호작용성을 지원할 수 있다.

또한 군사 작전이나 작업 자동화 시에, 실제 세상(real worlds)에 가상 물체(virtual objects)를 정합하여 작업 적용 범위(coverage)를 증가시키고 효율화하는 증강현실(Augmented Reality)의 응용에서는 가상 물체가 실제 세상과 차이 없게 느껴지게 하는 것이 핵심적인 기술 중의 하나이다. 그러므로 본 기술을 이에 적용하면 정합된 가상 물체를 마치 실제로 존재하는 듯한 몰입감을 사용자에게 제공할 수 있을 것으로 기대된다.

REFERENCES

[1] T. Whitted. An Improved Illumination Model for Shaded Display. Communications of the ACM, 23(6), June 1980, 343-349.

[2] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. ACM SIGGRAPH '88, 75-84.

[3] B. T. Phong. Illumination for computer generated pictures, Communications of ACM, Vol. 18 (1975), no. 6, 311-317.

[4] R. Ramamoorthi and P. Hanrahan. An Efficient Representation for Irradiance Environment Maps. ACM SIGGRAPH '01, 497-500.

[5] P. Sloan, J. Kautz, and J. Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low Frequency Lighting Environments, SIGGRAPH '02, 527-536.

[6] R. Green. Spherical Harmonic Lighting: The Gritty Details. Proceedings of the Game Developers Conference, 16 Jan, 2003, 1-47.

[7] A. Keller. Instant Radiosity. ACM SIGGRAPH '97, 49-56.

[8] K. Dmitriev, S. Brabec, K. Myszkowski and H.-P. Seidel. Interactive Global Illumination Using Selective Photon Tracing. Proceedings of 13th Eurographics Workshop on Rendering, 2002, 25-36.

[9] J. T. Kajiya. The Rendering Equation, SIGGRAPH '86, 143-150

[10] T. Akenine-Möller and E. Haines. Real-Time Rendering, 2nd Ed., A K Peters: Natick, 2002. Cited on p. 73.

[11] M. F. Cohen and D. P. Greenberg. The Hemi Cube: A Radiosity Solution For Complex Environments, SIGGRAPH'85, 31-40.

[12] L. Williams. Casting Curved Shadows on Curved Surfaces, SIGGRAPH'78, 270-274.

[13] M. Segal, C. Korobkin, R. v. Widenfelt, J. Foran and P. Haeberli. Fast Shadows and Lighting Effects Using Texture Mapping, SIGGRAPH'92, 249-252.

[14]http://oss.sgi.com/projects/ogl-sample/registry/SGIX/shadow.txt

[15] ATI Technologies Inc. Radeon charisma engine and pixel tapestry architecture. White Paper, 2000.

[16] K. Turkowski. Transformations of Surface Normal Vectors with applications to three dimensional computer graphcis, Apple Technical Report No. 22, 6 July 1990.

김 병 철(Kim, Byung-Cheol)

- 2002년 2월 : 아주대학교 정보 및 컴퓨터공학부(공학사)
- 2004년 2월 : 한국과학기술원 전자전산학과 전산학 전공(공학석사)
- 2011년 8월 : 한국과학기술원 전산학과(공학박사)
- 2011년 9월 ~ 현재 : 서울대학교 정보문화학 전공 강사
- 관심분야 : 가상현실, 컴퓨터그래픽스, 물리기반 시뮬레이션
- E-Mail : clorvie@gmail.com