Gama Network Presents:

Gamasutra.com

Hardware Accelerating Art Production

(하드웨어 가속기를 이용한 아트 작업)

By Steve Hill

Gamasutra

March 19, 2004

URL: http://www.gamasutra.com/features/20040103/fristrom_01.shtml

쉐이더의 이용은 실시간 렌더링에 심대한 변화를 가져왔을뿐 아니라, 아트 툴이나 작업과정에서 시각 효과를 극대화하는 데에도 큰 영향을 주었다. 최종적으로 게임 캐릭터와 배경이 화면에 어떻게 나타날 지에 영향을 주는 변수들과 표면부 맵이 갈수록 많아짐에 따라, 아티스트들이 그들의 작업을 정교하게 튜닝할 수 있는 툴의 중요성이 더욱 커졌다.

다행스럽게도 최근의 주요 3D 그래픽 패키지들은 쉐이더를 기본으로 지원한다. 그러나 개발자들이 자체 제작한 툴(커스텀 플러그인, 에디터, 마무리 프로세스 등) 역시 이와 같은 수준으로 사용이 용이해야 할 것이다.

이 글은 프로그램 가능 그래픽 하드웨어들을 실시간 렌더링 뿐만 아니라 게임 개발과정에서 아트 효과의 처리 등에도 사용하도록

제안하기 위한 것이다. 이 글에서는 메쉬에 대한 고강도 사전계산 그림자 처리기법(Ambient Occlusion)을 GPU 에 매핑하는 것을 중심으로 이 기법의 강력한 능력을 보여주고 있다. 이 방법은 컴퓨터의 CPU 만을 이용하는 소프트웨어 방식보다 최대 15 배가 빠르다.

또한 이 기법은 아트 처리에도 강력하지만, 동시에 아티스트들이 3D 아트 패키지를 이용해서 손쉽게 버텍스나 텍스처 데이터를 만들어내고, 조정하고, 표현하는 데 쓸 수 있는 플러그인을 위해서도 여러 가능성을 제시한다. 여기서 설명하는 기법은 Direct3D 의 1.1 버전의 쉐이더를 사용하므로, 요즘 보편적으로 사용되는 그래픽 카드들에 적용될 수 있을 것이다.

개요

첫번째 섹션에서는 기술적 변화가 어떻게 아트 제작에 영향을 미치는지, 그리고 툴이 어떤 기능들을 제공하여야 하는지에 대해 살펴본다. 그리고 AO(Ambient Occlusion)에 대한 개요를 설명한다. 이어서 처음에는 그다지 효율적이지 못한 하드웨어 기반 처리방법을 3 단계에 걸쳐 개선하여 깔끔하고 매끄러운 것으로 만드는 과정을 보여줄 것이다.

이 기법을 변형한 몇 가지의 다른 기법들도 선보이며, 실제 작업과정에서의 장단점을 소개할 것이다. 또한 이 기법을 확장하면 어떻게 고차원 어클루젼(Higher-order occlusion, 즉 percomputed radiance transfer)을 처리할 수 있는지를 보여줄 것이다. 끝으로는 한번 고려해볼 만한 다른 하드웨어 가속 기법들에 대하여 검토한다.

아트 제작과정

먼저 전반적인 아트 제작과정의 개요를 살펴보고, AO(Ambient Occlusion)가 프리프로세스로써 어떻게 쓰이는지를 알아보자. AO

쉐도우를 만들어내는 기법은 가장 기본적인 아트 제작과정에 쉽게 통합될 수 있다. 즉 3DS MAX 와 같이 이미 실시간 쉐이더를 제공하는 3D 모델링 툴을 쓰면 3D 아티스트가 즉각적으로 결과물이 어떻게 보일 것인지 확인할 수 있다. ATI 의 렌더몽키(Rendermonkey)와 같이 3D 메쉬 데이터의 처리가 가능한 상용 툴이나 자체 제작 툴을 쓸 수도 있고, 자체 제작한 모델링 툴이나 텍스처 툴을 쓸 수도 있다.

사례 연구

AO 프로세스(적어도 게임 용으로)는 하나 이상의 메쉬를 입력하면 버텍스나 텍스처 데이터를 산출해 낸다. 뒤에 여러 가지의 변형을 설명하겠지만 기본적으로 이 데이터는 각 표면의 모든 점에 대해 각각의 그림자 값을 제공한다. 이들 값은 추가적인 버텍스 속성이나 어클루젼 맵에 저장되는데, 이는 주변 조명(또는 산란된 환경 조명)에서는 물론 게임에서도 사용된다.

게임 실행 시에 소요되는 처리는 간단하지만 게임 제작 시에는 각 지점을 나타내는 데 필요한 가시도(Visibility) 정보의 샘플이 매우 많이 필요하기 때문에 계산을 아주 많이 해야 한다. CPU 를 이용하는 통상적인 방법으로는 AO 구성에 소요되는 계산이 너무 오래 걸려, 편집이나 프리뷰를 하는 데 아주 힘들다. 이 계산을 빨리 처리할 수 있다면 제작 시간을 단축하는 데 좋을 뿐 아니라, AO 를 모델링 패키지의 플러그인으로 만들어 아티스트들이 콘텐트 반복을 더욱 쉽게 할 수 있도록 하는 가능성을 열어 놓을 수 있다.

여기서 AO 에 대한 개요를 설명하겠지만 이 주제에 대한 훨씬 더 상세한 정보를 향후 출간될 GPU Gems 시리즈에서 찾아볼 수 있을 것이다(추가 도서 목록 참조). 여기서의 개요 설명은 이 글에서, 이 기법에 익숙하거나 그렇지 않거나, 앞으로 AO 를 그래픽 하드웨어에 매핑하는 것을 논의하는 데 출발점으로 세우기 위한 것이다.

Ambient Occlusion: 기초

AO 란 한 지점에 대해 외부로부터의 빛이 장애물에 가로막혀 도달하지 못하는 정도를 나타내는 것이다. 각 지점 표면의 구성요소 즉 버텍스와 텍셀마다 평균 폐쇄 값을 구하여 자체 그림자를 시뮬레이션하는 데 사용된다(그림 1 참조).

그림 1: 주변 조명과 함께 나타낸 AO의 분해도: 메쉬가 어클루젼 맵에 의해 싸여져 있음(왼쪽), 주변 조명 산란에 의한 조명(중간), 주변 조명 산란과 어클루젼 맵을 결합한 조명(오른쪽)

아주 부드러운 자체 그림자는 조명의 사실감을 크게 높여준다. 그 예는 렌더몽키 쉐이더 개발 스위트의 효과 툴에서 찾아볼 수 있다.

이 기법은 영화업계에서 사용되면서 널리 알려졌는데, ILM 이 *Dinosaur* 에서 처음으로 이를 사용하였다[Landis02]. 이것은 주변의 빛을 특히 벽, 하늘, 바닥으로부터 반사되어 나오는 간접 조명을 흐릿하게 만들어 정교한 화면을 만들어내는 것이다. 반사 조명을 인위적으로 만들거나 대규모의 광역 조명을 이용하던 것을, 이 기법을 통해 저렴한 비용으로 처리할 수 있었다.

AO 가 이런 예전의 방법들보다 훨씬 낫다고 하는 이유는 바로 조명을 움직이거나 정교하게 조정하는 것을 훨씬 쉽게 해주기 때문이다. 이것은 영화에서 후반작업에 자주 필요한 것인데 이제 게임에서도 필요한 부분이다.

전제

가시도(Visibility) 정보를 재사용하는 것이 가능 하려면 다음 몇 가지 전제가 성립되어야 한다.

- 확산된 표면
- 기하학적 견고성
- 지속적 조명

위들 전제 중 처음 두 가지는 전통적 방식의 사전계산 라디오시티(Radiosity)에서도 필요한 사항이다. '확산된 표면'이란 전제조건은 저장된 가시도 정보는 어떤 각도에서 보느냐에 관계없이 일정해야 한다는 것이고, 시간이 경과해도 변형되지 않아야 한다는 것이다. 이 조건들이 만족될 때 가시도 정보들의 재사용이 가능하다. 조명과 가시도 정보의 분리로 인해 AO 는 정적인 조명보다 더욱 자유롭다. 이것은 독립적으로 폐쇄된 단일 메쉬를 쉽게 변형하고, 회전하고, 균일하게 확장 시킬 수 있기 때문이다. 그러나 물론 오브젝트들의 그룹은 서로들 사이의 그림자 값을 유지하기 위해서 반드시 한꺼번에 변환되어야 한다.

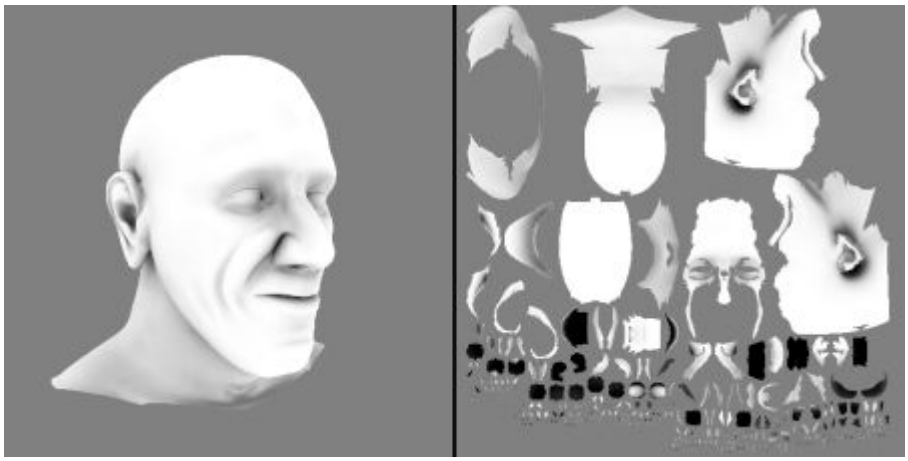

그림 2: 기본적 방식의 AO :버텍스 데이타(왼쪽) - 귀 뒤에 있는 단선적으로 인터폴레이트(Interpolate, 보간)한 방식의 물체들은 낮은 테셀레이션(Tessellation, 다각형화)때문에 그림자의 변화들을 정확하게 잡아내지 못함.
텍스처 데이터(오른쪽)- 더 부드러운 그림자 형성.

프로세스

등식 1 은 어클루젼 O_p 를 적분 공식으로 표현한 것이다. 이 적분 공식의 해는 통상 몬테 카를로 적분법으로 구할 수 있다. 이에 대한 자세한 논의는 “추가 도서 목록”을 참조하기 바란다.

$$O_p = \frac{1}{\pi} \int_{\Omega} V_p(\vec{\omega})(N \cdot \vec{\omega})\, d\omega$$

등식 1: 가시도 정보의 반구형 적분값

먼저 어떤 주어진 점 *p* 로부터 광선을 *N* 이라는 노멀(Normal) 반구에 투사한다. 가시도 정보를 나타내는 함수 *V* 를 이 각각의 광선에 대하여 계산한다. 광선이 공간의 경계에 도달하기 전에 어떤 물체에 부딪히면 이 값은 0 이 되고 그렇지 않으면 1 이다. 그림 3 은 2 차원에서 반면이 폐쇄된 점으로부터 이를 계산하는 것을 보여준다. 실제로는 그림자를 부드럽게 표현하기 위해서는 수백 수천의 광선에 대하여 이를 계산해야 한다.

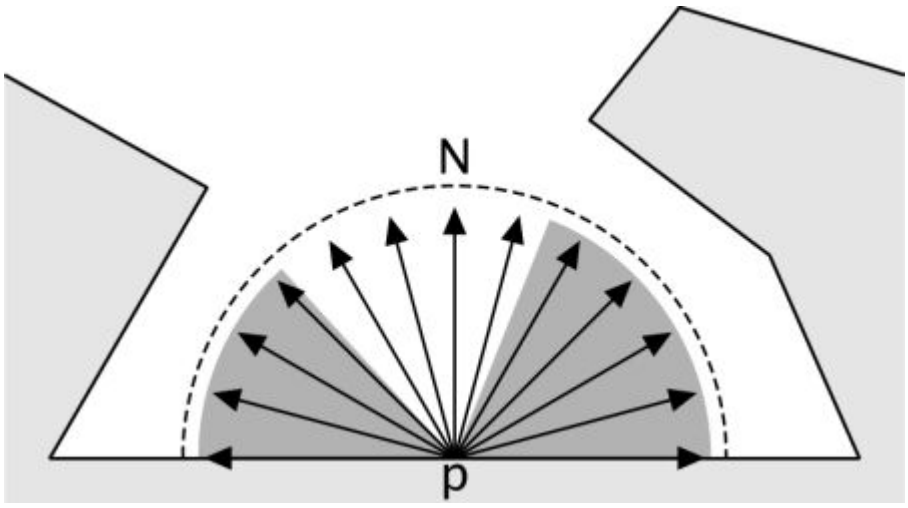

그림 3: 점 *p*에 대해 반구로부터 가시도를 샘플링하는 것을 보여주는 단면도. 다수의 광선을 투사하여 장애물체와 만나는지를 검사하고 이들의 평균을 구한다.

광선의 방향이 균일하게 분포되어 있는 경우 각 가시도 값은 노멀 방향과 샘플 방향간의 코사인 값에 의해 가중치가 부여된다(자세한

설명은 추가 도서 목록을 참조). 이들의 평균을 구하면 폐쇄정도를 나타내는 스칼라 값이 나온다. 효율적 계산을 위해 ILM 은 광선의 코사인 분포를 이용하였다. 이를 통하면 등식 2 에서 볼 수 있는 바와 같이 코사인 값을 나타낼 필요가 없고, 통계적으로 중요한 방향에 대한 것으로 집중할 수 있다.

$$O_p = \frac{1}{n} \sum_{i=1}^{n} V_p(\vec{s_i})$$

등식 2: 광선의 코사인 분포를 이용한 몬테 카를로 적분

게임에서의 사용

게임에서 AO 를 이용하면 그림자 효과를 입체화하여 불변적 주변 조명을 즉각 개선할 수 있다. NVIDIA 의 오가 데모(그림 4)는 바로 이와 같은 방법으로 주요 방향성 조명 및 그림자 맵과 함께 AO 를 사용하고 있다.

그림 4: NVIDIA 오가 데모의 분석: AO를 이용한 위 그림과, 주조명(방향성 조명 및 그림자 맵)에 덧붙여 불변적 주변 조명과 AO를 결합한 아래 그림. 그림 **Spellcraft Studio GmbH 와 NVIDIA 제공.**

그러나 조명이 불변한다는 가정 하에서만 이러한 방법이 가능하여 그것을 계산에서 제외할 수 있다. 그러나 현실에서는 개요에서 언급하였던 것처럼 이것을 약간 더 확장하여 적용할 수 있다. 예를 들어 ILM 은 확산된 표면의 2 차 조명들에 대해 균일 주변 조건 대신에 사전 여과(Pre-filtered)된 환경 맵을 사용한다. 여기서는 조명이 천천히 변하기 때문에 AO 를 이용해 사실감 있는 결과를 낼 수 있다. 또한 폐쇄되지 않은 방향들의 평균값을 저장하기 때문에 근접성도 개선된다. 이것은 이른바 맵을 인덱스할 때 표면부의 노멀을 “구부려 펴낸(bent) 노멀”로 대체하는 것을 말한다.

그러나 사전 여과(Pre-filtered)된 맵보다 더 좋은 방법은 구형 조율(Spherical harmonic) 조명을 쓰는 것이다[Forsyth03]. 다시 말하지만 AO 는 빛의 희석을 위해서 사용될 수 있고, Precomputed Radiance Transfer(SH 를 이용)에도 쓸 수 있다. 이에 대해서는 뒤에서 설명한다.

요약하면 AO 는 산란 된 주변 조명에 대해 쓸 수 있는 저렴하고 효율적인 그림자 처리 기법이다. 이것은 전통적 방식의 게임 조명에 광범위한 사실적 효과를 가져다 준다. 또 다음에 설명하는 바와 같이, 최근의 프로그램 가능 그래픽 카드들을 이용하여 3D 모델의 제작이나 아트 작업에서 쓸 수도 있다.

AO 의 GPU 로의 매핑

재구성 (Reformulation)

AO 의 반구형 샘플들을 하드웨어에 직접적으로 매핑하는 방법의 하나는 헤미큐브이다[Purvis03]. 그러나 이것은 너무 많은 계산을 요구한다. 각각의 표면부 구성요소에 대해 장면 전체를 변환하고 여러 차례 렌더링하여야 한다. 이보다 더 효율적인 방법은 서로 다른 방향의 일정하게 동질한 광선들 여러 개의 궤적을 함께 구하는 것이다.

[Whitehurst03]에서 설명된 바와 같이 Weta Digital 이 개발한 이 방식의 직관적인 구현 방법은 구형으로 장면을 둘러싸는 것이다(그림 5 참조). 샘플의 가중치를 설정하기 위해 빛의 방향들을 이용하였고, 가시도를 결정하기 위해 이에 해당하는 심도 맵(일정하게 동질한 광선들)을 이용하였다.

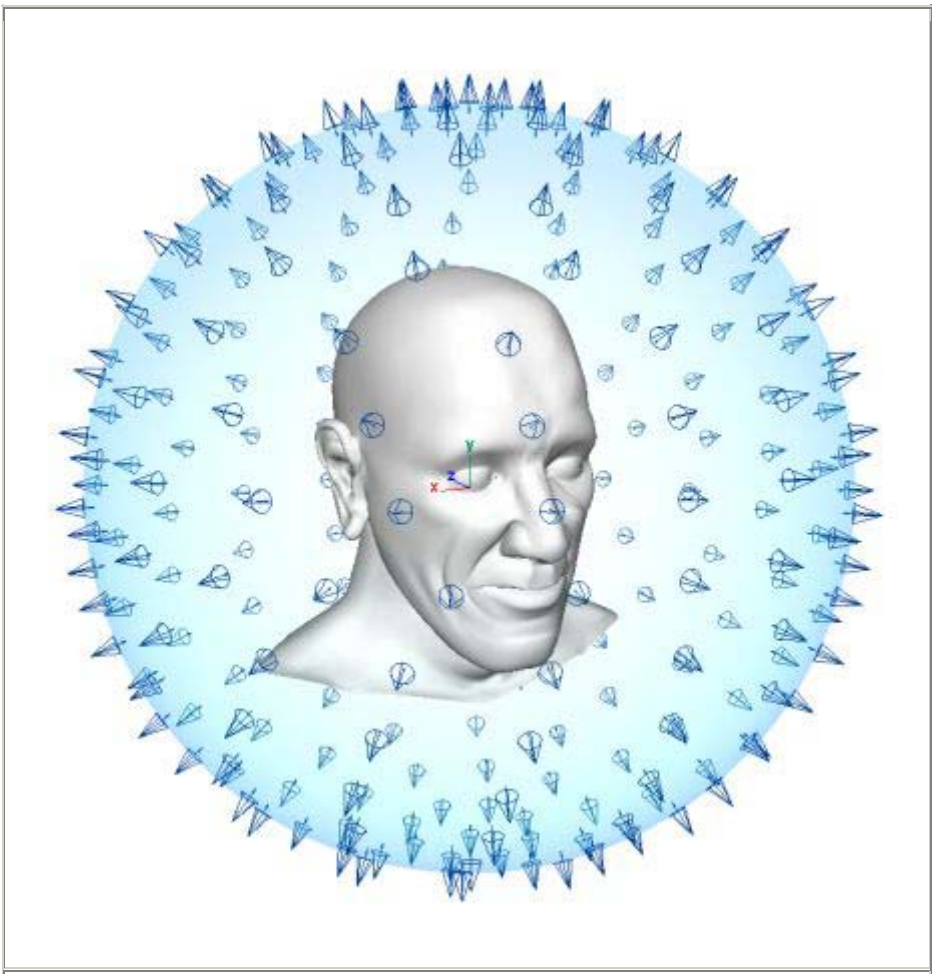

그림 5: 구형상의 샘플링. 원뿔들로 표시된 각 시점(Viewpoint)들을 하나씩 차례로 계산한다. 이는 각 방향에 대해 그림자 심도 맵에 의한 가시도 광선을 종합해낸다.

등식 3 은 이 예를 몬테 카를로 적분 공식으로 나타낸 것이다. 각 구성요소 p 에 대해 가중치가 적용된 가시도 샘플들(이하 샘플)의 합을 구하고, 가중치의 합계인 w 로 평균을 낸다. 샘플의 방향 s_i 가 구의 한 단위에 해당하기 때문에 각 지점에서 시야 밖의 것들에 대해서는 반구함수 H 에 의해 가중치 0 이 부여된다. 가시도 함수 V 는 심도 맵 공간에서의 표면부 구성요소와 이에 해당하는 맵 값과의 심도 차이 비교인 것이다.

$$O_p = \frac{1}{w} \sum_{i=1}^{n} V_p(\vec{s_i}) H_N(\vec{s_i})$$

등식 3a: 원구상의 몬테 카를로 적분값

$$\begin{aligned} w &= \sum_{i=1}^{n} H_N(\vec{s_i}) \\ H_N(\vec{s}) &= \max(N \cdot \vec{s}, 0) \end{aligned}$$

등식 3b: 정의

그래픽 하드웨어를 이용하면 심도 맵을 빠르게 렌더링 할 수 있다. 그러나 몇 개의 문제를 해결하면 GPU 의 능력을 최대한 이용하고 이보다 더 나아갈 수 있다. 다음 설명은 어떻게 그것을 가능하게 하는지를 보여준다.

1 차 시도

더 깊은 논의를 하기 전에 기본적인 라스터화(Rasterization) 하드웨어와 소프트웨어 프로세싱을 이용한 처리 방법을 살펴보자. 그 방법은 다음과 같다:

1. 장면의 중심으로부터 방향 하나를 선택
2. 그 시점(Viewpoint)으로부터 보이는 물체들의 렌더링
3. 심도 값을 되 읽어 들임(Read-back)
4. 각 표면부 요소들을 심도 버퍼로 변환
5. 심도를 비교하여 가시도를 측정
6. 위의 과정을 반복하여 각 구성요소의 샘플들을 모으고 가중치 부여
7. 샘플과 가중치의 합을 이용하여 각 구성요소의 AO 를 계산

프로그래밍 기능이 없는 그래픽 칩을 이용할 때는 위의 과정이 필요하다. 그러나 프로그래밍 기능이 있는 것이라면 이것을 더 효율적으로 만들 수 있다; 구성요소별 변환과 삼각 래스터화(텍스처 구성)는 전용 하드웨어를 이용하면 훨씬 더 쉽게 해결될 수 있다. 또한 위의 방법은 심도 값을 읽어 들이는 데 많은 시간이 든다. 이것은 노이즈(즉 시점을 공유하는 데서 생기는 엮임 현상)를 없애기 위해 아주 많은 반복(iteration)이 필요하다는 것을 생각하면 큰 문제이다.

프로세싱 속도를 현저히 빠르게 만들기 위해서는 GPU 를 더 효율적으로 사용하여야만 한다. 아래에서 설명하는 바와 같이 이는 되 읽기(Read-back)도 줄일 수 있다.

2 차 시도

시간이 많이 소요되는 막대 심도 변환(Bar-depth transfer)은 스트림 프로세싱으로 잘 처리할 수 있으므로, 먼저 이것을 해결해 보자. 심도를 렌더링하는 방법은 전과 크게 다를 바 없다. 그러나 소프트웨어를 이용해서 처리하던 표면 구성요소의 변환과 비교를 쉐이더 하드웨어를 이용하여 처리할 수 있다. 버텍스 쉐이더는 각 점의 구성요소와 방향에 대해 가중치를 계산한다. 이에 해당하는 픽셀 쉐이더는 가중치를 이용하여 필요한 심도 측정을 처리하고 결과를 기록한다. 그림 6 에서 보는 바와 같이, 이제 가속기를 이용하여 두 개의 단계, 심도 처리와 샘플링을 처리하는 것이다.

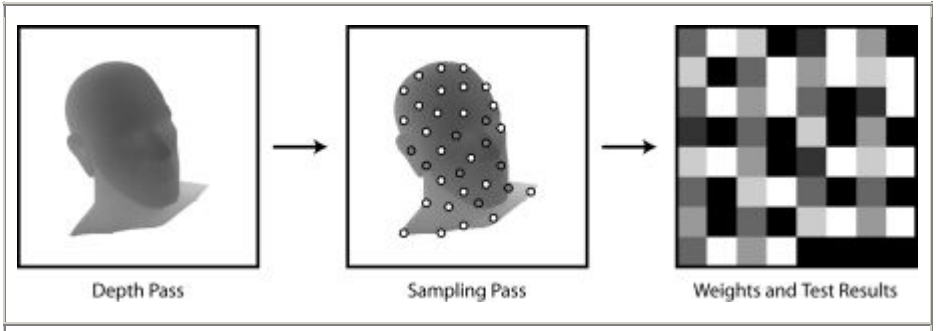

그림 6: AO를 GPU에 매핑할 때 쉐이더의 이용

버텍스 AO 에서는 각 지점들(예를 들어 D3DPT_POINTLIST)이 두 개의 쉐이더에 전달되는데, 쉐이더는 가중치와 테스트 결과를 각각 기록하며 렌더링 목표물에서 하나의 픽셀을 차지한다. 이들이 버텍스 순서대로 연속적으로 놓여지면 렌더링 목표물의 내용은 소프트웨어로 간단하게 이터레이션 될 수 있다.

텍스처 구성에서는 아티스트가 지정하였거나 자동으로 생성된 비교차(non-overlapping) UV 를 이용하는 대신에 삼각형들을 래스터화하기 때문에, 약간의 프로세스가 더 필요하다. 또 버텍스 쉐이더가 설정을 실행하여 픽셀마다 가중치가 계산된다.

Read-back 또한 1 차 시도에 비해 어느 정도 감소되었다. 단 이는 표면 해상도(표면 구성요소의 갯수)가 심도 버퍼 해상도보다 작다는 것을 가정한다. 그렇지 않으면 과소샘플링(under-sampling) 현상이 일어날 것이다. GPU 나 CPU 의 속도에 따라 다르겠지만 1 차 시도 방식을 최적화함으로써 단축된 시간을 포함하여 상당한 속도 개선이 이루어질 것이다. 또한 이 방법에서는 전용 하드웨어가 하위레벨 계산을 처리하므로 더 단순하다.

이러한 개선에도 불구하고 read-back 때문에 진행이 지연되는 문제에 대해서는 더 많은 개선의 여지가 있다. 통합 처리 단계를 그래픽 카드에서 할 수 있다면 GPU 를 더 많이 이용할 수 있고, 우리는 계산이 다 끝난 후에 마지막 결과만을 읽어 들이면 될 것이다. 이것은 ps2.0 쉐이더를 이용한 고정밀도 목표물에서 가능하다. 그러나 이 기능을 지원하는 카드들이 흔하지 않고, 추가의 정밀도를 위해서는 속도저하와 저장 공간 소요를 감수하여야 한다. 그러나 다음에 설명할 마지막 방법에서 알 수 있듯이, 방금 언급한 기능들 없이도 GPU 를 이용해 부분적으로 통합 처리하는 것이 가능하다. 이를 통해 되읽어들이는 시간을 엄청나게 절약할 수 있다.

3 차 시도

스플릿 램프(Split ramp)는 [James03]에서 소개된 참신한 아이디어로, 요소 당 8 비트 버퍼에 값들을 축적하여 기록하는 방법이다. 앞의 이차 시도에서 설명한 버텍스 AO 의 예를 생각해 보자. 가중치를 픽셀 쉐이더에 그냥 전달하는 데 그치는 것이 아니라, 그 값들을 이용하여 색상 요소별로 분할되는 특수 텍스처들을 인덱스할 수 있다. 한 채널에서 고위치 비트 값을 나타내고 다른 채널에서 저위치 비트 값을 나타내는 것이다.

정밀도와 관계된 문제에 대하여 지금까지 다루지 않았지만 , 아래의 결과에서 볼 수 있듯이, 8 비트는 하나의 가중치나 샘플(프리뷰를 위한

것들)을 저장하고도 남는다. 스플릿 램프 기법에서는 가중치를 R 과 G 에 대한 4 비트 값으로 분할한다. 샘플을 처리하는 경우에도 마찬가지로 B 와 A 의 가중치를 가시도 결과로 마스크하여 처리할 수 있다. 모든 구성 요소들에서 고위치 비트 값이 없음으로 인해 알파 블렌딩을 통해 값들을 축적하는 것이 가능하다. 16 차례 정도의 계산을 반복하면 고위치 비트에 값이 부여될 것이다. 그림 7 은 단일 구성요소를 이 방법으로 처리하는 것을 보여주고 있다.

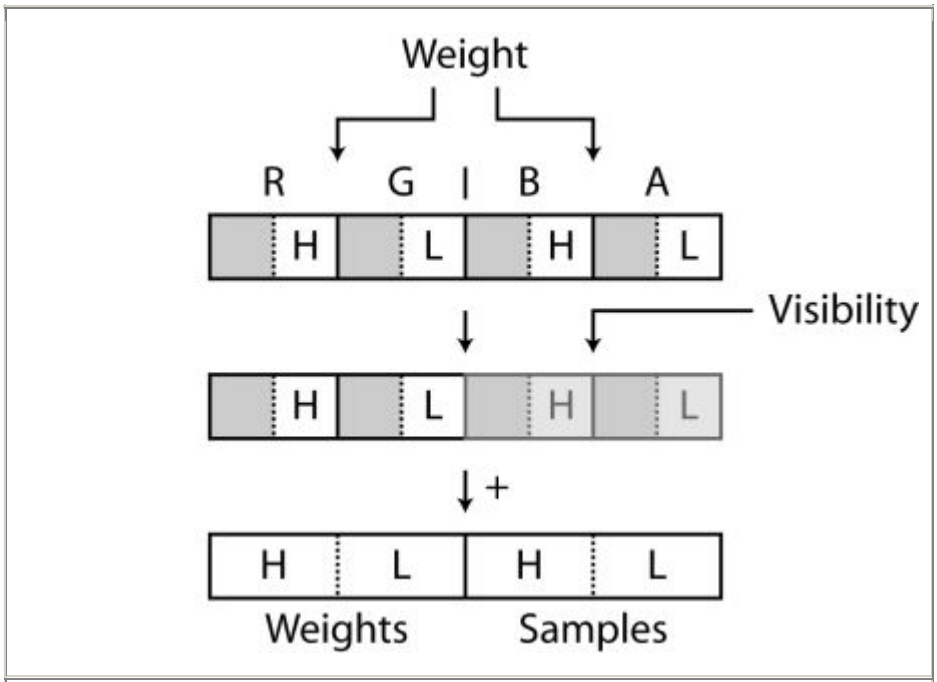

그림 7: 복수의 색상 구성 요소별로 가중치와 샘플들을 분할하는 과정

한 세트의 이터레이션이 끝나면 타겟에 대한 합을 재구성하고, 이를 각 구성 요소별로 주 메모리에 저장된 본 합계 값에 더한다. 그리고 다음 세트의 이터레이션을 위해 타겟은 모두 0 으로 재설정된다.

이로써 전송 횟수를 1/16 으로 줄일 수 있는데, 앞의 단계들에서 단축된 시간들과 함께 전형적인 게임 메쉬들과 텍스처들을 처리할 때 속도가 엄청나게 빨라졌다. 타이밍과 분석에 대해서는 이 논문의 뒷부분에서 다루고 있다.

현실성

지금까지의 논의에서는 그림자 처리의 품질과 정확성에 영향을 주는 몇 가지 사항들에 대해서 그냥 지나쳤다. 이들 중 일부는 여기의 사례 연구에만 해당하는 것이 아니다. 심도 맵 렌더링을 위해 원근법 프로젝션을 사용할 수도 있지만, 앞서 “광선 원구”에 관하여 언급할 때 제시한 바와 같이 정사도법 (orthographic projection) 우리의 작업에 아주 적합하다. 전자의 방법으로는 시각(또는 반대로 말하면 샘플의 방향)의 변화 때문에 추가의 쉐이더 계산이 필요하고, 가까운 물체의 해상도가 먼 곳의 물체의 해상도보다 높게 나타나는 문제가 있다. 반면에 정사도법을 쓰면 균일성을 얻을 수 있고 촘촘하게 프러스텀(frustum)을 구성하는 것이 아주 쉬워진다. 왜냐하면 한 장면의 경계 원구에 기반한 방향 축들을 사용하는 것이 모든 시점(Viewpoint)에서의 일관성을 형성하는 데 최상이기 때문이다.

여기서의 예와 같이 Direct3D 를 이용하면, 래스터화(Rasterization) 규칙들과 관련한 문제들이 생긴다. 기하학적 특성들은 각 픽셀의 중심에서 샘플이 구해진다. 이것은 텍셀의 모서리에서 값을 읽어들이는 텍스처 검색과 일치하지 않는 방식이다. 또 한 차례의 계산 반복에서 후속으로 읽어들이는 텍스처를 렌더링할 때, 래스터 좌표를 조정하는 것이 최상의 방법이다[Brown03]. 이러한 속성들을 설정하고 전반적인 상황에서 예측 가능한 데이터들로 테스트하면, 나중에 발생할 수 있는 미묘한 문제들을 피해갈 수 있다.

또 필터링은 특정 검색 테이블을 사용하는 데서 간과되기 쉬운 중요한 문제들 중의 하나이다. 스플릿 램프에서는 저위치 비트의 값이 교차되는 지점들에서 정확한 값을 구해내기 위해 포인트 샘플링이 필요하다.

다음 섹션에서는 8 비트의 가상 심도 버퍼를 사용하는 다양한 변형 기법들을 설명한다. 대부분의 경우에 스플릿 기법을 이용하여 심도

비교를 위한 별도의 재계산 없이 더 정확한 16 비트 방식으로 바꾸는 것이 가능하다. 경우에 따라 하드웨어 쉐도우 매핑도 사용할 수 있다.

구현

각각의 메쉬 버텍스에 대해 AO 를 계산하고 저장하거나 별도의 어클루젼 맵을 계산하는 것이 가능하다는 점을 앞서 지적하였다. 전자는 표면부가 충분히 테셀화되어(Tessellated, 모자이크식) 그림자의 변화를 잘 나타내 줄 수 있다는 전제 하에 불변의 주변 조명의 경우에 적합하다. 후자에서는 이런 조건이 필요치 않다. 또 텍스처 단계를 건너뛰려면 AO 저장에 공간이 소요되지만 또 다른 맵을 채널에 같이 포함할 수 있을 것이다.

노멀 맵을 생성하기 위해 고극점 모델(High-poly model)을 사용하는 경우에는 다른 대안이 있다. 더욱 자세한 AO 텍스처를 계산하기 위해 추가 정보가 사용될 수 있다. 이를 위해서는 프로세싱에 더 많은 시간이 소요될 것이다.

또한 Precompouted Radiance Transfer 는 추가적인 변수들을 이용하면 AO(또는 구부려 펴진 노멀) 이외에 저주파 조명을 더욱 정확히 표현하는 데도 사용할 수 있다. 데이터의 압축이 가능하다 하더라도 이 방법에도 추가 저장 공간이 소요된다. SH 조명이 추가적인 지원을 요구하므로 렌더링도 영향을 받게 된다.

이하 이 섹션에서는 이러한 문제들 특히 가속기를 사용한 프리프로세싱에 대해 다룰 것이다.

버텍스 구성(Vertex Baking)

버텍스 AO 와 세이더 코드에 대해서는 리스트 1 의 코드들이 자명하며 달리 더 설명이 필요 없을 것이다.

```
//////////////////////////////////////////////
// depth.vsh
//////////////////////////////////////////////

vs.1.1

// c0 : Rasterization offset
// c1-4 : World*View*Proj. matrix

dcl_position v0

// Output projected coordinates
m4x4 r0, v0, c1
mad oPos, r0.w, c0, r0

// Output depth via diffuse color register
mov oD0, r0.z


//////////////////////////////////////////////
// sampling_v.vsh
//////////////////////////////////////////////

vs.1.1

// c0 : Rasterization offset
// c1-4 : World*View*Proj. matrix
// c5-7 : World matrix

def c8, 2.0, -2.0, -1.0, 1.0
def c9, 0.5, -0.5, 0.0, 1.0

dcl_position v0
```

```
dcl_normal v1
dcl_texcoord v2

// Scale and offset texture coordinates
// to [-1, 1] range for render target
mad r0.xy, v2.xy, c8.xy, c8.zw
mov r0.zw, c9.zw

// Output coordinates for rasterizing
mad oPos, r0.w, c0, r0

// Project vertex coordinates
m4x4 r0, v0, c1

// Output depth via diffuse colour register
// (for consistency with depth pass)
mov oD0, r0.z

// Scale and offset projected coordinates
// for depth map lookup:
// x' = x*0.5 + 0.5*w
// y' = -y*0.5 + 0.5*w
// z' = 0
// w' = w
mul r0, r0, c9
mad oT0, r0.w, c9.xxzz, r0

// Cosine weighting: max(N.s_i, 0)
// Note: sample direction s_i is world z axis
dp3 r0.z, v1, c7
max r0.z, r0.z, c9.z
```

```
// Output weight, to be split via ramp look-up
mov oT1.x, r0.z
mov oT1.yzw, c9.zzw


////////////////////////////////////////////////
// sampling_v.psh
////////////////////////////////////////////////

ps.1.1

def c0, 0.0, 0.0, 0.0, 0.508 // Depth bias
def c1, 1.0, 1.0, 0.0, 0.0 // Sample mask

tex t0 // Depth
tex t1 // Weight (R & G), sample (B & A)

// Compute depth difference, with
// a bias added for cnd (0.5 + epsilon),
// plus a bit extra to avoid acne/aliasing
sub r0.a, t0.a, v0.a
add r0.a, r0.a, c0.a

// Output weight and sample (zero if occluded)
mul r1, t1, c1
cnd r0, r0.a, t1, r1
```

리스트 1

텍스처 구성(Texture Baking)

텍스처 AO 에서는 픽셀마다 가중치가 계산되어야 한다. 이를 위해 앞서 버텍스 쉐이더에서 사용되었던 가중치 계산 대신에 실제공간에 대한 노멀의 변환이 필요하다. 인터폴레이트(interpolate)된 노멀은 3 차의 스플릿 램프를 인덱스하는 데 사용된다. 스플릿 램프는 픽셀별로 필요한 노멀화를 실행한다. 나머지의 수학 계산은 앞서 본 바와 같이 분할 가중치를 계산하는 것이다. Z 축이 샘플의 방향이고 시각은 이 주축의 반대방향으로 고정되어 있기 때문에 이것이 가능하다.

이 두 개의 버전들은 약간의 노력으로 쉽게 통합될 수 있다. 큐브 맵 검색은 버텍스 AO 에 부담을 줄 것이지만, 약간 조정하면 경우에 따라 스플릿 텍스처만 변경하면 되도록 만들 수 있다.

텍스처 구성(Texture Baking) II

버텍스 어클루젼에 비해 텍스처를 통해 얻을 수 있는 화면의 충실도는 더 높지만, 노멀 매핑이 이루어진다는 것을 생각할 때 해상도를 더 높이는 것이 가능해야 한다. HP(high-poly) 레퍼런스 지오메트리가 가능하다면 텍스처의 세밀도를 높일 수 있는 두 가지 방법이 있을 수 있다.

첫번째 방법은 도출된 노멀 맵을 이용하고 픽셀 쉐이더에 있는 가중치를 계산한 뒤, 종속적인 텍스처 검색을 통해 분할하는 것이다. ps1.1 에서도 몇 차례에 걸쳐 샘플링을 반복하지 않아도 가능한 일이다. 이를 이용하면 LP(low-poly) 메쉬의 거거친 기하학적 어클루젼과 함께 HP 모델에서 추가적인 세밀도를 얻을 수 있다.

세밀도를 더욱 충실하게 재생산하기 위해서, ATI 의 노멀 매퍼 툴에서는 노멀 맵을 생성하는 동안에 텍셀당 AO 샘플을 처리하는 선택기능을 제공하고 있다. 그러나 이것은 다소 비효율적이다; 낭비를

줄이는 더 좋은 방법은 하드웨어를 이용하여 HP 메쉬의 버텍스들에서 AO 를 사전계산하는 것이다. 그렇게 하면 어클루젼은 레이 캐스터(Ray caster)에 의해서 추출된 속성의 하나로 처리될 수 있는 것이다([Cignoni99]에서 기술된 것과 같이).

Precompouted Radiance Transfer (사전계산 빛 전송)

단일 어클루젼 값은 대단히 압축적이고 현재의 게임에서 유용한 조명 방식이다. 또 그 제한 조건들 이외에도 가능하도록 조정할 수 있지만 한계를 갖고 있다. Precomputed radiance transfer(이하 PRT)은 SH 기반함수를 써서([Sloan02]) AO(0 차 값)를 일반화하고, 추가적인 방향성을 구한다. 이것은 다른 부대효과를 가져온다. 9 개 이상의 트랜스퍼 구성요소가 있으면 소프트 쉐도우가 빛의 움직임을 따라서 눈에 띄게 움직이는 것을 볼 수 있다. 비록 이를 위해서는 추가의 저장공간이 필요하지만 Clustered Principle Component Analysis (CPCA) [Sloan03]를 이용하여 벡터들을 효율적으로 압축할 수 있을 것이다.

PRT 가 가능하려면 최소한 여기에 제시된 방식에서는 AO 에 필요한 조건들과 같은 조건들이 필요하다. 단 입사되는 빛 불변이 아니라 거리에 따라 가변적이어야 한다는 조건이 성립되어야 한다. 베이시스 계수를 이용해 근사값을 구하는 조명은 이제 모든 표면 지점에 대해 남아있는 적분값들과 함께 계산에서 제외될 수 있다. 그렇게 하여 트랜스퍼 벡터만이 남게 된다. 압축이 없다면 방출되는 빛 R_p 는 벡터 L(주변 조명을 나타냄)과 확산된 표면 반응에 의해 측정된 트랜스퍼 벡터 T_p의 내적(inner product)이다. 등식 4 는 이것을 나타낸다.

$$
\begin{aligned}
R_p &\approx \rho_d \sum_i l_i \int_S \frac{1}{\pi} B_i(\vec{s}) V_p(\vec{s}) H_N(\vec{s}) ds \\
&\approx \rho_d (L \cdot T_p)
\end{aligned}
$$

등식 4: 출사(확산된 표면과 원근감 있는 조명으로부터) 래디언스의 근사값을 원구 조율을 통해 구하는 등식

이제 AO 를 구할 때 사용했던 방법과 거의 같은 방법으로, 어클루젼 값을 벡터로 표현하고 있다. 다만 기반함수 B_i 가 여기에 포함되고 있다는 것이 다르다. 등식 5 는 등식 3 과 매우 비슷하지만 각 변수들을 통합하여 나타내고 있다.

$$
T_{p,i} = \frac{1}{w} \sum_{j=1}^{n} B_i(\vec{s_j}) V_p(\vec{s_j}) H_N(\vec{s_j})
$$

등식 5: 트랜스퍼 벡터 (몬테 카를로 적분)

이를 하드웨어를 이용하여 구현하기 위해서는 각 기반 함수 B_i 를 소프트웨어로 계산하여 상수로 하드웨어에 전달하여야 한다. 왜냐하면 샘플링의 방향이 각 시점에 대해 고정되어 있기 때문이다. 각 변수들이 [0,1] 사이의 값을 갖도록 스케일링과 바이아싱(biasing)을 하면, 스플릿 램프 검색 시 클리핑을 방지할 수 있다.

상당히 많은 수의 PRT 구성요소가 있다고 가정하면, 모든 구성요소에 대해 불변적인 샘플링 프로세스의 일부를 계산에서 제외하는 것이 가능할 것이다. 말하자면 이것이 가중치이고 현재는 샘플 및 가시도 측정값과 함께 저장되어 있는 것들이다. 이제 추가 텍스처에 결과를

기록하도록 작동을 별도의 계산으로 처리하므로, 쉐이더를 다른 일에 쓸 수 있다. 따라서 샘플링 계산마다 두 개의 PRT 구성요소를 병렬로 처리할 수 있다. 적어도 버텍스 구성의 경우에는 그렇다.

이러한 추가적인 분할의 부대효과는 더 정교한 심도 렌더링과 심도 측정 계산에서 스왑이 더 쉬워진다는 것이다. 이것은 품질 설정에 따라 게임 실행 시 가능할 수 있다. 리스트 2 는 PRT 샘플링을 위한 쉐이더의 코드이다. 나머지(심도, 심도 비교, 가중치 계산)는 리스트 1 로부터 도출할 수 있을 것이다.

```
////////////////////////////////////////////////
// sampling_prt_v.vsh
////////////////////////////////////////////////

vs.1.1

// c0 : Rasterization offset
// c1-4 : World*View*Proj. matrix
// c5 : Sample direction
// c6 : Packed SH basis terms:
// : t0*scale0, t1*scale1, bias0, bias1

def c8, 2.0, -2.0, -1.0, 1.0
def c9, 0.5, -0.5, 0.0, 1.0

dcl_position v0
dcl_normal v1
dcl_texcoord v2

// Scale and offset texture coordinates
// to [-1, 1] range for render target
mad r0.xy, v2.xy, c8.xy, c8.zw
```

```
mov r0.zw, c9.zw

// Output coordinates for rasterising
mad oPos, r0.w, c0, r0

// Output coordinates for depth result lookup
mov oT0, v2

// Cosine weighting: max(N.s_i, 0)
dp3 r0.z, v1, c5
max r0.z, r0.z, c9.z

// Sample = B(s)*[V(s)]*Hn(s), scaled and
// biased to [0, 1] range
// V(s) is evaluated in the pixel shader
//
// The following values are passed through
// a pair of 2D split ramps:

// Scaled and biased samples in x and y
mad oT1.xy, r0.zz, c6.xy, c6.zw
mov oT1.zw, c9.zw

// Biases in x and y (for occluded case)
// Note: these are fixed, so with care a pixel
// shader constant could be used instead
mov oT2.xy, c6.zw
mov oT2.zw, c9.zw


////////////////////////////////////////////
// sampling_prt_v.psh
```

```
////////////////////////////////////////////////

ps.1.1

tex t0 // Depth test result
tex t1 // sample0, sample1
tex t2 // bias0, bias1

// Use depth test result to mask packed samples
cnd r0, t0.a, t1, t2
```

리스트 2

결과

Direct3D Extensions Library (D3DX)의 최근 출시판(DirectX SDK Update Summer 2003)에는 SH PRT 의 소프트웨어 처리를 위한 함수들을 많이 포함하고 있다. 이 방법은 강력하면서도 쉽게 구현할 수 있기 때문에 앞서 이 글에서 제시한 하드웨어 이용 방법과 비교하기에 아주 좋을 것이다.

표 1 은 완전하지는 않지만 D3DX 와 9 개 구성요소 버텍스 PRT 방식의 하드웨어를 이용한 것과의 성능차이를 보여준다. 표면 구성요소의 수가 증가함에 따라 둘 사이에 분명한 차이가 발생함을 알 수 있다. 많은 표면 구성요소는 텍스처 프로세싱에 도움이 되는 것이다.

버텍스당 추출된 샘플의 수에 따라 HW1 과 HW2, 두 개의 하드웨어별로 측정된 시간을 표시하였다. 전자는 D3DX 와 직접적인 비교가 가능하다; 하드웨어 방식은 원구에 대해 샘플을 추출하므로 두

배 이상의 샘플들이 사용되었다. 따라서 평균적으로는 샘플의 절반만이 효과가 있다(소프트웨어에서 반구에 대해서만 샘플을 추출하는 것과 확실한 차이가 있다). 그러나 여전히 공정한 비교는 아니다. 왜냐하면 D3DX 는 코사인 분포를 통해 샘플의 가중치를 주어서 수속에 걸리는 시간을 단축시키기 때문이다.

HW2 는 많지 않은 수의 샘플을 이용하여 프리뷰를 할 때 어느 정도의 성능을 낼 것인지를 대략 보여줄 수 있다. 가시도의 분산 정도에 따라 장면마다 최적치는 다를 것이다.

Model	Vertices	D3DX (2048 samples)	HW1 (4096 samples)	HW2 (1024 samples)	D3DX/HW1
shapes1	2814	14.27s	4.94s	1.36s	2.9x
head	10596	151.66s	13.73s	3.52s	11.0x
skullocc	31076	581.95s	37.77s	9.49s	15.4x

표 1: 결과

위의 결과들은 Athlon XP 2400+ PC 에서 GeforceFX 5800 로 공개된 소프트웨어 빌드와 라이브러리를 써서 얻어낸 것이다. 하드웨어 이용방법에 대해 최상의 결과를 제공하기 위해 메쉬들을 사전에 “버텍스 캐쉬 오더”로 최적화하였다. 그러나 이것은 소프트웨어에서도 레퍼런스의 근접성을 향상시킨다. 하드웨어 방법에서 사용한 심도 해상도는 두 가지 경우 모두 512x512x8 비트이다.

보기에는 두 방법으로 만든 조명에 경미한 차이가 있을 뿐이다(그림 8). 이것은 샘플의 저장 정밀도가 한정되었기 때문인데, 프리뷰를 하는 동안에는 그 정도는 문제가 되지 않을 것으로 생각한다. 정밀도와 read-back 는 스플릿 램프시 서로 상극관계에 있다. 그러나

반복 횟수와 심도 맵(해상도와 정밀도)은 그림의 품질에 지대한 영향을 준다.

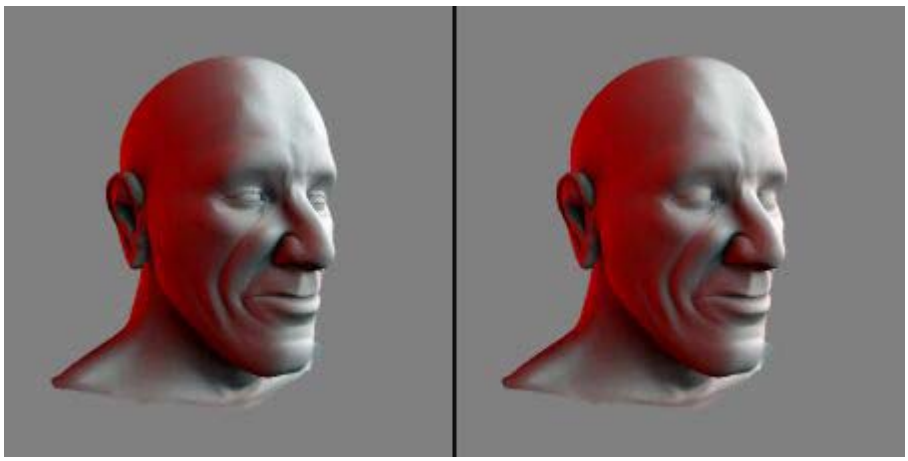

그림 8: D3DX 소프트웨어 방식(왼쪽)과 HW1 하드웨어 방식(오른쪽) 결과

유일한 주된 차이는 눈꺼풀 아래의 그림자들이다(그림 9 참조). 하드웨어 방식에서는 이것이 표현되지 않았다. 이것은 심도 정밀도가 부족하여 발생하는 문제인데 앞서 설명한 것처럼 이것을 증가시킬 수 있다. 버텍스가 (노멀을 따라) 바깥으로 살짝 경사된 조건에서, 감소된 심도 바이아스는 정확성을 증대시킬 수 있다. 또 붉은 조명과 흰 조명 사이의 시안 색상은 저준위 SH 조명의 근접성 때문이다.

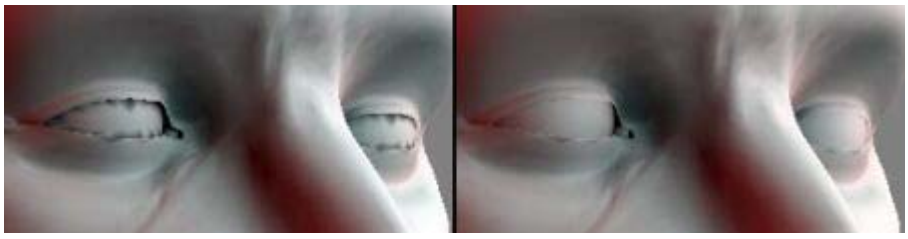

그림 9: 하드웨어로 구현된 그림에서 D3DX(왼쪽) 방식에 비해 심도 해상도가 떨어짐을 볼 수 있다. HW1(오른쪽)에서 8비트 심도 맵을 사용함에 따라 눈꺼풀 아래에 그림자가 나타나지 않고 있다.

[Sloan04]에서는 플로팅 포인트 하드웨어를 이용하여 비슷한 결과를 보여주고 있다. 그가 사용한 PRT 가속기 방식-본질적으로 이 논문에서 소개한 것과 같은 방식이나, 2.0 버젼의 쉐이더와 고정밀도 버퍼를 사용-은 향후 출시될 DirectX SDK Update 에 포함될 것이다. 이와 함께 소프트웨어 시뮬레이터에 대해서도 상당한 최적화가 이루어졌다. 개선된 API 는 더 많이 모듈러 방식으로 되어 있고, 일반화된 CPCA 로 사용자에 의해 생성된(이 논문에서 만들어진 것과 같은) 데이터를 압축하는 것을 가능케 하고 있다.

확장

앞서 설명한 방법들을 전용 하드웨어로 매핑하는 것도 가능하다. 그러나 그런 하드웨어가 잘 구현된 소프트웨어보다 더 강력할 것인지는 확실하지 않다. 또 전체 작동을 몇 차례의 계산으로 나누어 처리할 필요가 있을 수도 있어 성능은 더 저하될 수도 있다.

버텍스 AO 의 성능을 더 향상시키는 한 방법으로 유연한 광선 샘플링을 생각해 볼 수 있다. 예를 들어 몇 차례의 지정된 횟수만큼 반복 한 후 버텍스를 검사하는 것이다. 이 검사를 통해 완전히 또는 약간의 오차 범위 내에서 가시도가 열려있는 버텍스들을 찾아내고, 다이나믹 인덱스 버퍼(또는 다른 방법으로)와 추가적인 기록을 이용하여 이들을 다음 단계 처리를 위해 남겨두는 것이다. 더 일반적인 방법은 몇 차례의 반복 사이에 어떤 차이가 있는지를 검사하고, 지정된 오류 범위를 넘어서면 중단하는 것이다. 이것은 고려해 볼만한 방법이지만, 호스트 컴퓨터와 GPU 사이에 더 많은 통신이 필요하여 속도 향상을 저해할 수도 있다.

다른 적용 방법

다음 사항들은 하드웨어 가속기를 이용한 사전처리의 다른 예들이다. 이들 중 일부는 게임 개발에 이미 사용되고 있는 것들이고, 다른

것들도 앞으로 이용될 수 있는 것들이다.

라디오시티(**Radiosity)**

Coombe 등은 [Coombe03]에서 프로그레시브 라디오시티 방식을 거의 완벽히 GPU 에 매핑하는 것을 보여준다. 그들의 방법은 헤미-큐브(hemi-cube)를 이용할 때 생기는 몇 가지 전형적인 문제들을 해결하고 있다. 첫째 델타 폼 팩터를 소프트웨어로 처리하기 위해 헤미-큐브 표면들을 주 메모리로 다시 읽어 들여야 하는 문제를 해결하고 있다. 각 면들에 걸쳐 이터레이션을 하고 임의로 구성 요소들을 갱신하는 것이 아니라-현재의 하드웨어로는 거의 불가능한 일- 각 텍스처에 직접 저장되는 표면 구성 요소들을 ID 헤미-큐브 표면으로 역변환하는 방식을 이용한다. 둘째로 ID 를 각 구성요소가 아니라 패치에 부여하기만 하면 되므로 앨리어싱(Aliasing)도 감소된다(슈팅을 할 때 헤미-큐브에 있는 문제의 하나).

이 논문에는 표면 슈팅 선택과 유연한 하위분할(subdivision)을 위해 고안된 기발한 방법들이 소개되어 배울점이 많다. 가속 처리를 한다는 관점에서 보았을 때, 그들의 시스템은 간단한 장면에 대해서는 결과를 빨리 얻을 수 있고, 다른 경우들에서는 텍스처 저장과 프로그레시브 방식의 속성상의 제한 때문에 보통의 결과를 얻을 수 있다.

정적 SH 볼륨(Static SH Volume)

Max Payne 2 는 게임 실행 시 모델의 주변 조명을 위해 게임의 한 레벨 전체에 걸친 SH 예상치의 정적 볼륨을 이용한다. Remedy 는 오프라인(offline) 계산을 가속하는 데[Lehtinen04], ps2.0 쉐이더를 통한 주변 맵 SH 프로젝션과 부동 소수점 저장을 이용한다.

주어진 좌표 점에서 먼저 사전에 조명 된 장면이 HDR 큐브-맵에 렌더링된다. 그리고 표면 텍셀을 해당하는 방향에서 측정한 현재의

기반함수로 곱한다. 이 방향값들은 모든 큐브에서 고정되어 있으므로 당연히 이것들은 사전에 계산될 수 있고 텍스처로부터 읽어 들일 수 있다. 이 결과들은 4 개의 근접 텍셀의 그룹들을 감축 통합(reductive summing)하는 것을 통해 축적된다. 이는 여러 차례의 계산을 통해 이루어지는데 반복 박스 필터링과 비슷한 과정으로, 단지 평균을 구하지 않고 처리한다는 차이가 있다. 그런 뒤 마지막 레벨을 다시 읽어 들이고 적당한 크기로 확장하여 SH 계수를 구하는 것이다. 모든 계수에 대해 프로젝션을 반복하고 모든 좌표점에 대해 렌더링을 실행한다.

노멀 매핑(Normal Mapping)

Wang 등은 [Wang03]에서 그래픽 하드웨어를 이용하여 노멀 맵 프로세싱을 가속하는 이미지-스페이스 방법을 소개하고 있다. 이것은 이 논문에서 소개한 첫번째 AO 방법과 유사하다. 다수의 시점에서 레퍼런스 메쉬를 렌더링하고 소프트웨어로 심도를 비교하여 가장 가까운 표면 지점과 노멀을 계산한다.

이들은 타겟 삼각 노멀을 삽입함으로써 복잡한 메쉬에서 드러나지 않은 표면점으로 인해 생기는 문제를 피해가고 있다. 이들의 논문에서 제시한 방법은 레퍼런스 노멀을 포함하고 있는 프레임 버퍼와 심도 버퍼를 다시 읽어 들이는 것에 의존하는 것이다. 버텍스와 프래그먼트 프로그램을 이용하는 것은 상당히 가능성이 있는 것으로, 이 논문의 사례 연구에서 보이는 바와 같이, 더 많은 작업을 GPU 로 이전시켜 프로세스를 더욱 가속시키는 것이다.

ORB 의 크리스쳔 세거(Christian Seger)는 노멀 맵 가속에서 드러나지 않은 표면점의 문제를 피하기 위해 다른 방법을 시도한다[Seger03]. LP 모델에서 주어진 하나의 삼각형에 대해 검색 반경 내의 HP 레퍼런스에 의한 삼각형들을 먼저 모두 취합한다. 이들은 타겟 삼각형의 인터폴레이트된 노멀에 기반하여 압축된다.

광선추적 방식의 입장에서 보면 이것은 최근접점이 아니라 노멀 샘플링을 에뮬레이션하는 것이다. 이것은 물체들을 축소시킨다([Seger03] 참조). 이 압축 과정을 통해 계산된 좌표들을 이용하여 하드웨어들이 이 삼각형들을 노멀 맵에 렌더링한다. 여기서 텍스처 공간에서 타겟 삼각형의 외부에 있는 텍셀들을 집어내기 위해 스텐실링(stenciling)을 이용한다.

결론

이 논문의 사례 연구와 다른 예들을 통해서, 더 반응성이 좋은 아트 툴을 개발하는데 프로그램 가능 그래픽 카드를 이용할 수 있다는 것이 입증 되었기를 바란다. 나아가, 가능한 가속의 정도가 반드시 그래픽 프로세서의 부동 소수점에 제한될 필요도 없다는 것 또한 보여주었기를 바란다.

궁극적으로 소프트웨어가 더 일반적인 기능을 갖고 있으며 디버그하기 편리하고 확장성이 있으며 더 높은 정밀도를 제공할 수 있다는 것은 사실이다. 그러나 GPU 가 갖고 있는 한계들이 사라지고 있으며, 쉐이더의 디버깅도 더 쉬워지고 있다. 그리고 결과물에서 볼 수 있듯이 높은 정확도가 프리뷰에 반드시 필요한 것은 아니다.

프로세스를 효율적으로 그래픽 하드웨어에 매핑하면 상당한 속도 향상을 얻을 수 있고, GPU 는 CPU 보다 기본 성능에서 더 많은 확장성을 갖추고 있다. 구현 방법 또한 소프트웨어의 경우보다 더 단순하다. 소프트웨어를 이용할 경우에는 처리 속도를 줄이기 위해 추가의 데이터 구조, 알고리즘, 저단계 최적화가 필요하다.

앞으로 나올 쉐이더들은 더 넓은 범위의 알고리즘들을 그래픽 프로세서에 매핑하도록 하여줄 것이다. 또 BrookGPU 와 같은 고단계 추출도 희망적인 진전이다. 또한 PCI-Express 를 이용한 더 빠른

통신은 (하드웨어와 소프트웨어의) 복합 방식을 더욱 현실적으로 만들 것이다.

감사의 말씀

다음 분들께 감사의 말씀을 드린다; 논평과 지원을 보내준 Simon Brown, Willem de Boer, Heine Gundersen, Peter McNeill, David Pollak, Richard Sim 과 Neil Wakefield; 세밀한 검토의견과 정보 및 아이디어를 제공한 Peter-Pike Sloan 과 Rune Vendler; 자신들의 하드웨어 처리기법을 설명해준 Jaakko Lehtinen 과 Christian Seger; 불안정한 코드를 가지고 다양한 GPU 에 걸쳐 테스팅을 해 준 Media Mobsters 에 있는 친구; 오거 사진 및 스티브 버크가 모델링한 헤드 메쉬의 사용을 허락해 준 Simon Green 과 NVIDIA; 출시 예정인 D3DX 의 상세한 기능들을 기술할 수 있도록 허락해 준 마이크로소프트.