아이폰에서 스텐실 그림자 만들기
Creating Stencil Shadows on iPhone

브라이언 할(Brian Hal)

가마수트라 등록일(2010. 6.16)

http://www.gamasutra.com/view/feature/5867/creating_stencil_shadows_on_iphone.php

캐릭터가 장면 위에 덧그려진 것처럼 보이지 않고 장면 속에 녹아 들어 보이게 만드는 가장 좋은 방법은 그림자를 그리는 것이다. 그림자는 장면에 깊이를 부여하고 캐릭터에게 현실감을 준다. 캐릭터가 뛰게 되면 캐릭터의 그림자와 캐릭터가 분리되면서 높이를 더욱 실감할 수 있게 된다. 그림자는 대부분의 3D 엔진에서 표준 사양으로 자리잡아가고 있다.

그렇지만 아이폰에서는 아이폰이 지닌 하드웨어적인 제한으로 인해 전형적인 3D 엔진을 사용하기는 문제가 있다. 간단히 말해서 그림자를 만들기 위해 사용되는 공통적인 기술을 아이폰에서 사용할 수 없다는 것이다. 처리 능력이 제한되기 때문에 섀이더(shader) 기반의 기술이 적합하지 않으며 비싼 메모리에 대한 투자가 필요한 경우가 많다. 동적인 텍스처가 필요한 그림자 솔루션에서도 하드웨어 비용보다 더 비싼 메모리 투자가 필요하므로 이 역시 적절하지 않다.

아이폰에는 스텐실 버퍼가 없으니까 스텐실 버퍼 기반의 그림자 역시 불가능할 것이다. 정말 그럴까?

스텐실 그림자 기법을 아이폰에서 사용할 수 있는 방법에 대해 자세히 알아보기 전에 먼저 스텐실 그림자가 어떻게 만들어지는 것인지 알아보는 것이 중요할 것이다. 최상위 수준에서는 이 기법은 직선적이다. 먼저 장면 내 오브젝트가 렌더링된다.

두 번째로 그림자를 그려줄 오브젝트의 모서리가 처리되고 볼륨으로 연장되어 볼륨이 스텐실 버퍼로 렌더링된다. 마지막으로 스텐실 버퍼를 마스크로 사용하여 전체 화면의 그림자가 그려진 쿼드(quad)가 장면 전체에 걸쳐 렌더링된다.

오브젝트의 모서리를 볼륨으로 연장하는 방법에는 여러 가지 방법이 있으며 이와 같은 일을 할 수 있는 다른 소스도 많이 있다. 이 기사에서는 그림자 볼륨을 생성하는 방법을 알고 있으며 기하학적 구조의 볼륨을 연장하는 방법을 사용해 보았다고 가정하고 이 방법에서 보다 중요한 부분인 볼륨의 렌더링에 대해 집중적으로 다루기로 한다.

카메라가 그림자 볼륨 내부에 있을 경우와 그림자 볼륨 외부에 있을 경우 어떻게 할지에 대한 논란이 있지만 이는 지금 다룰 주제를 넘어선 것이다. 이 기법의 아이디어는 볼륨을 두 번 렌더링하는 것이다. 처음에는 컬링되고 z-테스팅이 이루어진 뒷면으로 볼륨을 렌더링하며 스텐실에 1 을 더한다. 두 번째로 앞면을 컬링하고 z-테스팅하여 볼륨을 렌더링한 후 스텐실에서 1 을 뺀다.

두 번의 처리가 끝나면 스텐실 버퍼의 값은 그림자가 없는 곳은 0 이고 기타 부위는 0 보다 커진다. 기본적으로 뒷면과 앞면이 있고 다른 기하학적 구조가 없는 곳은 그림자가 없다. 뒷면과 앞면 사이에 어떤 기하학적 구조가 있다면 앞면 렌더링되지만 뒷면은 렌더링되지 않기 때문에 스텐실 버퍼의 값은 양수가 된다.

스텐실 버퍼가 없는 아이폰에서는 이런 처리과정을 어떻게 할 수 있을까? 3D 기능이 강화된 하드웨어가 장착된 데스크톱이 흔하지 않던 수년 전에는 제한된 기능만을 제공하는 하드웨어에서 일정한 함수 파이프라인만을 사용할 수 있었다. 스텐실 버퍼는 첨단적인 하드웨어에서만 필요한 것으로 여겨졌지만 거의 모든 하드웨어에 알파 채널을 가진 버퍼는 있었다. 이 알파 채널이 많은 기능을 수행했으며 일부 경우에서는 스텐실의 역할로 사용되기도 했다.

언뜻 보기엔 스텐실 버퍼에서 알파 버퍼로 바꾸기만 하는 간단한 작업처럼 보인다. 스텐실과 알파 버퍼 모두 더하기 연산을 할 수 있고 스텐실과 알파 버퍼 모두 빼기 연산을 할 수 있다. 그러므로 첫 번째 시도로서 모든 스텐실 연산을 이와 비슷한 알파 연산으로 바꾼다.

스텐실 구축 과정의 첫 번째 단계. 알파 버퍼의 값을 초기화한다.

그림자 볼륨을 렌더링하는 첫 번째 단계에서는 뒷면의 컬링과 z-테스팅이 이루어진 후 Add 연산을 하는데 소스 블렌딩과 목적 블렌딩을 위해 모두 1 값을 사용한다.

스텐실 구축 과정의 두 번째 단계. 그림자 영역을 어둡게 처리했으며 그림자가 없는 영역을 강제로 1 로 설정하였다.

두 번째 단계에서는 앞면의 컬링과 z-테스팅을 수행한 후 Reverse Subtract 연산을 한다. 이 때 소스 블렌딩과 목적 블렌딩을 위해 모두 1 값을 사용한다. 아쉽게도 결과물이 그리 훌륭하진 않다.

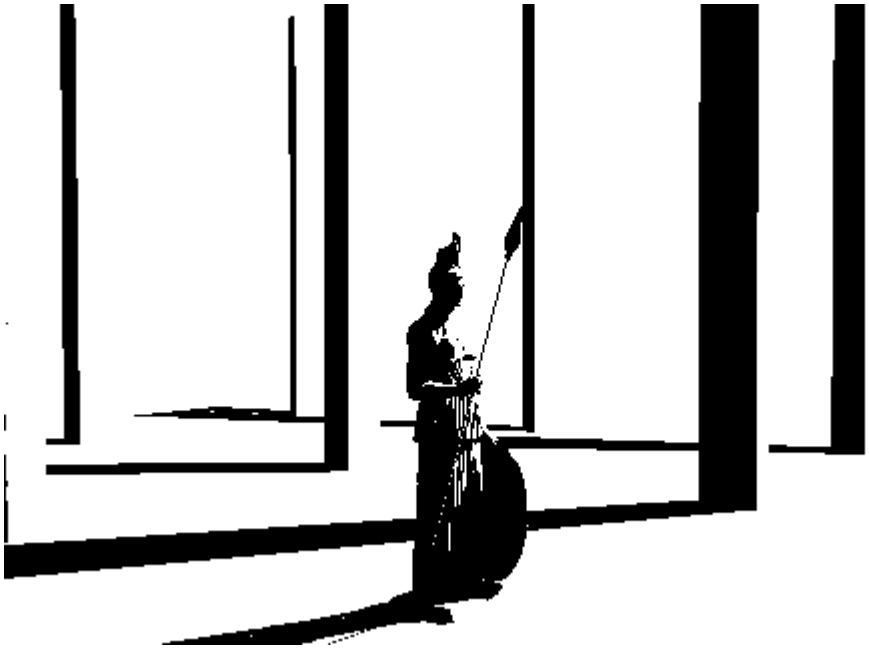
스텐실 구축 과정의 세 번째 단계. 이제 복잡한 그림자가 보다 분명해졌다. 조각상의 그림자 영역은 다른 값을 갖는다.

문제점 중 하나는 목적 버퍼의 알파 값이 1 이고 렌더링된 오브젝트의 알파 값이 1 이라면 목적 버퍼의 값은 1 로 포화상태가 된다는 것이다. 이는 그림자 볼륨이 복잡하고 동일한 픽셀에 여러 개의 앞면이 대응되는 경우에 일어날 수 있다. 예를 들어 도넛모양(Torus)은 두 개의 마주보는 앞 벽과 두 개의 마주보는 뒤 벽이 있다. 이것을 버퍼로 적절하게 렌더링하려면 빼기 연산이 이루어지기 전에 합계 2 로 만드는 알고리즘이 필요하다.

스텐실 버퍼는 스텐실 버퍼의 비트 수에 따라 256 개 값에서 수백만 개에 이르는 값을 가질 수 있다. 그러므로 스텐실이 알고리즘의 빼기 연산이 이루어지기 이전에 반복적으로 더해질 수 있다. 알파 버퍼로 이 문제를 해결하기 위해서는 렌더링될 때 그림자 볼륨의 알파 값으로 1/256 을 사용할 수 있다. 이렇게 하면 보다 복잡한 기하학적 구조의 버퍼 값이 축적될 수 있고 포화상태가 되지 않는다.

스텐실 구축 과정의 4 번째 단계. 그림자 스텐실이 보다 진해졌으며 복잡한 그림자가 두드러져 보인다.

그렇지만 이렇게 하게 되면 그림자를 적용할 경우에는 더 많은 작업이 필요해지기 때문에 장면에서의 오브젝트의 복잡도에 따라 4/256 이나 8/256 의 배수를 사용하는 것이 더 효율적이다. 장면의 복잡함을 표현하기 위한 값이 너무 크면 버퍼가 포화상태가 되어 그림자가 드리워진 것처럼 표현된다.



스텐실 구축 과정의 5 번째 단계. 그림자가 있는 모든 곳의 그림자 스텐실 값이 거의 0 이 된다.

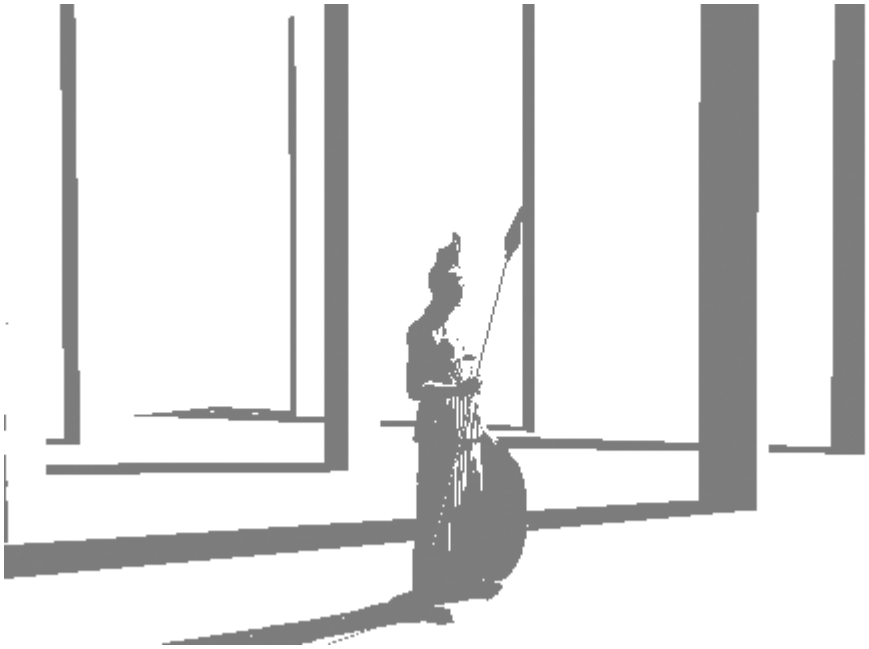
포화 문제가 해결되었지만 또 다른 문제가 있다. 처리 결과 계산된 알파 마스크가 너무 밝다는 것이다. 그 결과 전체 화면이 렌더링되면 그림자가 보이지 않을 것이다. 마스크도 하나의 값으로 일정하지 않기 때문에 복잡한 오브젝트에 의해 그림자가 생길 때 어두운 정도가 다양하게 나타날 수 있다. 이에 대한 해결법은 전체 화면의 쿼드를 이용하여 버퍼에 정규화 처리를 여러 번 하는 것이다.

스텐실 정규화 과정의 첫 번째 단계. 여기서 그림자들은 0 이 되며 그림자가 없는 영역은 (1 - 원하는 알파 값)이 된다.

초기화 단계에서는 Subtract 블렌딩 연산이 이루어지는데 소스 블렌딩에는 1-목적 알파 값이, 목적 블렌딩에서는 1 값이 사용되어 1/256 의 알파 값으로 쿼드가 렌더링된다. (쿼드의 알파 값이 이전에 설명한 것처럼 큰 값인 경우 처리 과정이 이보다 적어질 수 있다).
계속되는 처리 단계에서는 Reverse Subtract 블렌딩 연산이 이루어지는데 이 때 소스 블렌딩에는 1- 목적 알파 값이, 목적 블렌딩에는 목적 알파 값이 쓰인다. 이런 과정을 계속하게 되면 스텐실에서와 같이 그림자가 없는 곳은 1 값이 되고 그림자가 있어야 할 곳은 작은 값이 된다.

단계	알파 값 (그림자가 없는 곳)	알파 값 (그림자가 있는 곳)
초기	0.0000	0.0040
단계 1	1.0000	0.9922
단계 2	1.0000	0.9767
단계 3	1.0000	0.9306
단계 4	1.0000	0.7966
단계 5	1.0000	0.4312

복잡한 기하학 구조와 일치하지 않는 알파 값에 대한 문제점을 해결하기 위해서는 두 가지 방법을 사용할 수 있다. 첫 번째 방법은 그림자 스텐실과 같은 알파 값을 이용하여 전체 화면의 쿼드에 두 번의 처리 과정을 더 거치는 것이다. 첫 단계는 Reverse Subtract 연산을 하는데 소스 블렌딩과 목적 블렌딩을 위해 모두 1 값을 사용한다. 그 결과 모든 알파 값은 원하는 알파 값보다 작아지면서 0 값에 가까워진다. 모든 알파 값이 원하는 알파 값보다 작다고 가정한다면 그림자 영역은 순수한 0 이 되며 그림자가 없는 영역은 (1-원하는 알파 값)이 된다.



스텐실 정규화 과정의 두 번째 단계. 그림자들은 원하는 알파 값으로 설정되며 그림자가 없는 영역은 1 으로 돌아간다.

두 번째 단계에서는 소스 블렌딩과 목적 블렌딩에 모두 1 값을 사용하여 Add 블렌딩 연산을 함으로써 원하는 알파 값을 다시 알파 값에 더한다. 이로써 그림자 영역은 원하는 알파 값을, 그림자 없는 영역은 0 값이 된다. 이는 Max 처리와 같은 효과를 준다.

아이폰이 Max 처리를 지원하긴 하지만 개인적으로 이것이 제대로 동작하는 것을 보지는 못했다. 그렇지만 이론적으로는 위의 두 단계는 한번의 Max 처리로 대체될 수 있다. Max 처리는 위의 두 단계의 처리를 더 적은 필레이트(fill rate)로 한번에 효율적으로 할 수 있다.

이제 알파 스텐실이 구축되었으므로 그림자를 적용할 수 있다. 그러나 마지막 처리 과정이 하나 더 필요한데, 이는 1 인 알파 값으로 원하는 그림자 색으로 칠해진 전체 화면의 쿼드를 처리하는 것이다. Add 블렌딩 연산이 이루어지며 소스 블렌딩은 0 이, 목적 블렌딩은 목적 알파 값이 이용된다. 색 처리는 해당 장면에서 그림자의 영향을 받는 영역을 어둡게 하기 위해 Modulate 연산이 이루어져야 한다.

그림자가 적용된 장면.

이런 종류의 기교를 위해 알파 버퍼를 사용하는 것은 제한된 기능 파이프라인과 애증관계를 형성하면서 커온 사람들이 오래 전부터 사용해 왔던 전통이다. 우리 중 일부에게 모바일 플랫폼은 과거와 같이 또 다시 제한된 하드웨어로 원하는 일을 하고 멋진 비주얼을 얻기 위해 속임수를 써야 하는 환경을 의미한다. 그 결과 아이폰이 가진 한계에도 불구하고 아이폰에서 동적인 그림자를 구현할 수 있었다.

결론적으로 보면 6 번의 스텐실 구축 단계, 두 번의 정규화 단계, 한 번의 적용 단계를 거쳐 스텐실에 버금가는 품질의 그림자를 이루어낼 수 있었다. 채우기 비율(Fill rate)이 문제가 되지 않는다면 아이폰에서 실시간 그림자도 표현할 수 있을 것이다. 여기서는 Ogre 3D 엔진 버전 1.7 로 이 방법을 구현했으며 아이폰 내의 그림자 샘플에서 이를 테스트해 보았다. 앞으로 출시될 Ogre 3D 에서는 이 기능을 사용할 수 있을 것이다.