Lookup Table을 활용한 Mobile Platform에서의 Ambient Occlusion 렌더링 기법

박주상
고려대학교 컴퓨터정보통신대학원 디지털 미디어공학과
e-mail : jusang.park@gmail.com

Ambient Occlusion Rendering Technique in the Mobile Platform utilizing Lookup Table

Ju-Sang Park
*Dept. of Degital Media, Computer Information Communication , Korea University

요 약

게임, 영화, 애니메이션 분야에 이르기까지 3D 렌더링 기술은 많은 분야에 걸쳐 활용되고 있으며, 이러한 3D 렌더링 기술의 발전으로 현실감있는 표현이 점차 가능해지고 있다. 영화나 애니메이션이 많은 시간과 비용을 들여 고품질의 영상을 만들어 내는 반면에, 게임은 실시간으로 고품질의 영상을 만들어 내며, 이를 위해서는 많은 연산을 필요로 한다. 그래서 게임에 고품질의 렌더링 기술을 적용하기 위해서는 상당히 높은 성능의 하드웨어를 필요로 하며, 현재 점차 높은 성능의 하드웨어가 개발되고 보급되기 시작하면서, 게임에 실시간으로 적용 가능한 다양한 렌더링 기술이 개발되고 있는 상황이다. 하지만 이것은 PC 플랫폼만 국한된 상황이며, 모바일 기기가 가지는 성능상의 제약으로 인해서 모바일 기기에 이러한 PC 플랫폼 기반에서 적용되는 3D 렌더링 기술을 적용하기란 여간 힘든 일이 아닐 수 없다. 본 논문에서는 3D 렌더링 기술 중에 하나인 Ambient Occlusion 기법을, 모바일 디바이스가 제공하는 하드웨어적인 한계를 극복하고 보다 향상된 렌더링 속도로 기존 PC 환경과 유사한 효과를 표현하기 위한 렌더링 기법을 제안하고자 한다.

1. 서론

본 논문에서 제안하는 Ambient Occlusion 렌더링 기술은 이미 잘 알려져 있는 SSAO(Screen Space Ambient Occlusion) 렌더링 기술을 기반으로 하며, 이 기법은 크게 두 프로세스로 처리가 되는데, 샘플링된 커널 벡터와 랜덤한 노말맵의 랜덤 벡터를 이용한 Reflect 계산 과정의 전 처리를 통한 Lookup Table 생성을 위한 과정과, 전 처리된 Lookup Table 을 이용한 모바일 디바이스에서의 Ambient Occlusion 렌더링 과정으로 처리가 된다.

본 논문에서 제안하는 Ambient Occlusion 렌더링 기법을 통해서 기존 Shader 를 통한 연산과정에서 성능 저하에 많은 영향을 끼치는 부분을 전 처리된 Lookup Table 로 대체하여 향상된 렌더링 성능을 얻게 된다. 샘플링된 커널 벡터와 랜덤 벡터를 이용한 Relfect 연산 과정은 성능 저하에 많은 영향을 끼치는 부분 이기도 하고, Lookup Table 을 이용한 전 처리가 가능한 부분이기도 하여, 이를 전 처리하는 과정은 모바일 환경에서 좀 더 향상된 성능을 내기 위한 중요한 준비 과정이다. 마지막으로 전 처리되어 생성된 Looup Table 을 활용하여 Ambient Occlusion 렌더링 처리를 하게 되는데, 성능 저하에 많은 영향을 끼치는 부분을 제거하고 이를 전 처리된 Looup Table 로 대체함으로써 향상된 성능의 출력 결과를 모바일 기기에서 보여줄 수 있게 된다.

2. 관련 연구

2.1 Ambient Occlusion 개념

컴퓨터에서 생성되는 이미지의 현실성을 높이기 위한 다양한 방법들이 존재하는데, 이러한 방법들 중 하나는 오브젝트에 그림자 효과를 부여하는 것이다. 실시간 컴퓨터 그래픽스에는 다양한 그림자 기법이 존재하는데, 이러한 기법들 중 하나가 Ambient Occlusion 기법이다.

먼저 Occlusion 이란 용어를 우리말로 풀이하면 "겹침"이란 뜻인데, 현실에서도 표면 위에 어떤 물체가 올라가서 겹쳐진 경우 겹치게 되는 영역은 일반적으로 어두워지게 되는데, Ambient Occlusion 은 계산하는 점이 주변에 다른 면들에 의해 얼마나 가리워지는지를 0 ~ 1 의 범위의 수치로 계산을 하게 된다. 그리고 이 계산된 수치를 이용하여 이미지에 밝고 어두운 정도를 최종적으로 표현하게 된다.

2.2 Screen Space Ambient Occlusion 기법

Ambient Occlusion 렌더링 기술 중 하나로 이미 널리 알려져 있는 Screen Space Ambient Occlusion 은 2007 년 SIGGRAPH 컨퍼런스에서 “Finding Next Gen” 발표 때 Martin Mittring 에 의해 처음으로 논의가 되었으며, 이것은 2007 년 Crysis 라는 PC Game 에서 처음으로 사용이 되었다.

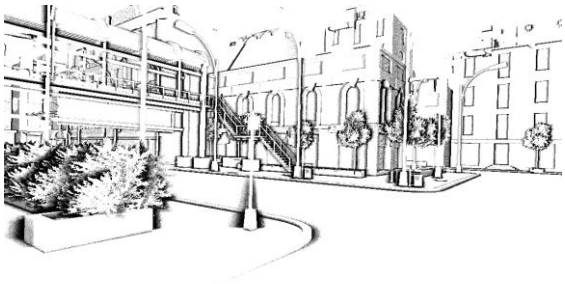
이 기법은 주어진 점의 Occlusion 정도를 계산하기 위해서 오직 깊이 버퍼만을 사용하며, 총 2 단계를 거쳐서 최종 이미지가 만들어 진다. 먼저 화면의 깊이 값을 렌더링 하기 위한 깊이 텍스처를 생성하고 생성

된 텍스처에 장면의 깊이 값을 렌더링 한다. 그리고 이 깊이 텍스처에 저장된 깊이 값을 이용하여 현재 장면의 모든 픽셀에 대한 Occlusion 정도를 결정한다. 각 픽셀의 Occlusion 정도를 계산하기 위해서 쉐이더 리소스로 깊이 버퍼를 연결하고 화면 전체를 덮는 사각형의 버퍼를 생성하는데, 여기에는 화면의 각각의 픽셀에 대응되는 2 차원 UV 정보가 담기게 되며 이 정보는 Vertex Shader 를 통해서 Pixel Shader 로 전달된다. 픽셀 쉐이더는 버텍스 쉐이더로부터 전달받은 값들을 이용해서 Occlusion 정도를 계산하게 되는데, 먼저 현재 위치 주변의 깊이 값을 얻어 오기 위한 샘플링 커널 벡터를 먼저 정의하고, 준비한 Noise Texture 로부터 랜덤 백터를 얻어와 샘플링 커널 벡터를 반사시키는데 사용하며, 이 때 샘플링 커널 벡터를 사용해서 다양한 변화를 줄 수 있으며, Occlusion 테스트에 있어서 수준 높은 결과를 얻을 수 있다.

이렇게 모든 샘플링 커널 벡터로부터 얻은 Occlusion 값을 모두 더하고 평균 값을 계산해서 최종 색상을 결정하게 된다. 본 논문에서는 테스트를 위한 샘플링 커널을 16 Sample 만을 사용하였으며, 더 많은 샘플을 사용할 경우 더욱 부드러운 결과를 보여줄 수 있게 된다.

2.3 Screen Space Ambient Occlusion 효과

SSAO 픽셀 쉐이더 연산을 마치게 되면 아래와 같은 결과물을 얻게 된다. 그리고 SSAO Buffer(그림 1)와 이미지를 합성하여 최종 결과물을 만들어낸다.



(그림 1) SSAO Buffer

2.4 성능 비교 및 분석

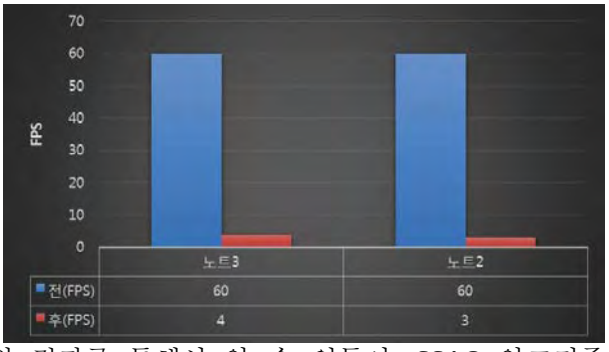
모바일 기기에서 Screen Space Ambient Occlusion 알고리즘이 성능에 끼치는 정도를 실험을 통해서 알아보자. 실험을 위해서 준비한 모바일 기기의 하드웨어 사양은 <표 1>과 같으며, 준비한 각각의 디바이스에서 SSAO 알고리즘을 적용하기 전 후의 성능 차이를 비교해 보았다. <표 2>

<표 1> 실험용 모바일 기기 사양

갤럭시 노트2
- 삼성 엑시노트 4412 1.6GHz 쿼드코어
- Mali-400
- 3GB DDR2

갤럭시 노트3
- 퀄컴 스냅드래곤 800 2.3 GHz 쿼드코어
- 퀄컴 Adreno 330 550 MHz
- 3 GB DDR3

<표 2> SSAO 적용 전 후 프레임 비교



위 결과를 통해서 알 수 있듯이, SSAO 알고리즘을 적용했을 때 평균 5 프레임(FPS)도 나오지 않음을 알 수 있다.

다음으로 Ambient Occlusion 계산의 대부분을 차지하는 샘플링 과정(그림 2) 중에서 성능 저하에 크게 영향을 끼치는 부분을 찾기 위해서, 샘플링 처리를 위한 쉐이더 코드들 중에서 전 처리된 Lookup Table 로 대체가 가능한 부분을 먼저 파악하고, 이를 임시로 준비한 Lookup Table 로 하나하나 대체해가며 렌더링 속도를 비교해 보았다.

먼저 (그림 2) 샘플링 처리 코드 중에서, 샘플링 벡터와 Noise Texture 의 랜덤 벡터를 이용한 Relfect 연산과정[1]과, 픽셀의 현재 위치와 인접한 위치의 Depth 차이를 이용한 Occlusion 수치 계산과정[2]을 전 처리된 Lookup Table 로 대체 가능함을 파악하였다. 그리고 Reflect 연산[1] 및 Occlusion 수치 계산 과정[2]을 제거 하고 임시로 준비한 Lookup Table 로부터 저장된 결과 값을 가져오도록 코드를 수정하고, Lookup Table 에 전 처리된 값을 가져오도록 수정하였을 때 어느 정도의 속도향상을 가져 올 수 있을지 실험을 통해서 알아보았다.

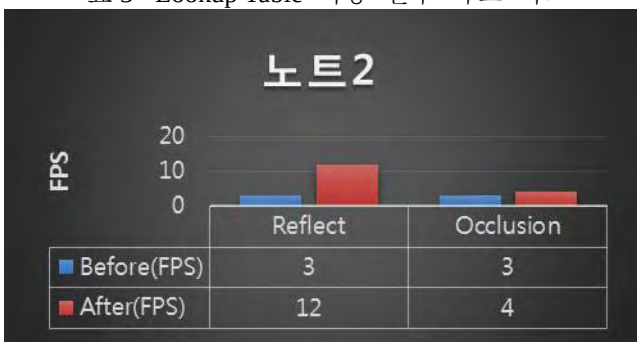
```
for(int i=0; i < 16; i++)
{
[1]  float3 randomvec = normalize(tex2D(_NoiseTex,
                pixel.scrPos.xy * _NoiseFactor).rgb);
     float3 ray = reflect(RAND_SAMPLES[i], randomvec);
     float fSignValue = sign(dot(ray,normal));
     float3 position = float3(pixel.scrPos.xy,
                      depth * _ProjectionParams.z);

     float3 hemi_ray = position +
            (fSignValue * ray * _SampleRange / 2048.0f / depth);

     float near_depth;
     float3 near_normal;
     DecodeDepthNormal(
               tex2D(_CameraDepthNormalsTexture, hemi_ray.xy),
                   near_depth, near_normal
               );

[2]  float falloff = _Falloff / 2048.0f;
     float Area = _Area / 2048.0f;
     float difference = (depth - near_depth);
     float fCalcStep = step(falloff, difference);
     float fSmoothStep = smoothstep(falloff, Area, difference);
     occlusion += fCalcStep * (1.0f - fSmoothStep);

}
```

(그림 2) Ambient Occlusion Pixel Shader 코드

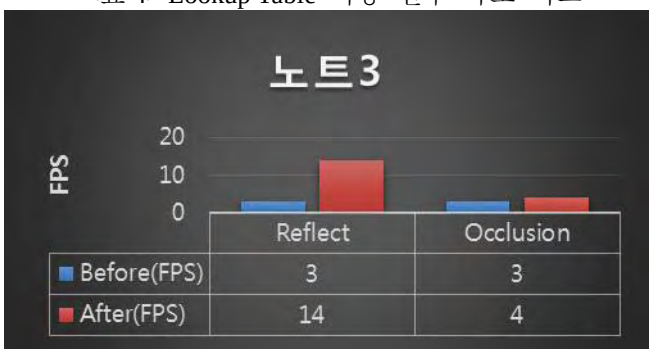
노트 2 와 노트 3 를 대상으로 한 실험 결과<표 3,4>를 통해서 볼 수 있듯이, 기존의 Relect 연산[1]을 Lookup Table 을 활용한 방식으로 대체하였을 때 노트 2 에선 9 프레임, 노트 3 에서는 11 프레임의 렌더링 성

능을 향상 시킬 수 있음이 증명되었다.

<표 3> Lookup Table 적용 전후 속도 비교

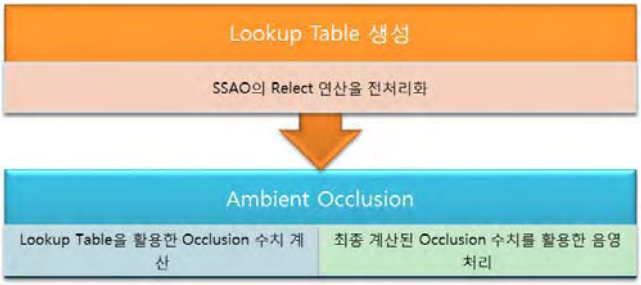


<표 4> Lookup Table 적용 전후 속도 비교



3. 설계 및 구현

3.1 설계



(그림 3) 시스템 프로세스

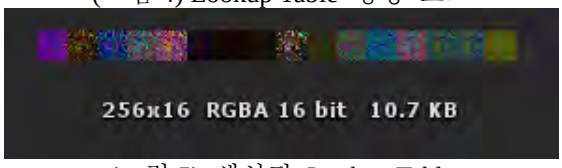
(그림 3)과 같이 크게 두 개의 단계로 구성이 되며, 하나는 기존 SSAO 의 Reflect 연산을 Lookup Table 에 전처리 하기 위한 단계이고, 다른 하나는 전 처리된 Lookup Table 을 활용하여 Occlusion 계산을 수행하고 최종적인 음영처리를 하기 위한 단계이다.

3.2 Lookup Table 구현

16 개의 샘플 벡터와 준비된 한 개의 Noise Texture 를 이용하여, 0 번에서 15 번 샘플벡터 각각에 해당하는 Lookup Table 을 16 x 16 크기로 가로로 16 개 생성한다. (그림 4) Lookup Table 의 RGB 값에는 샘플벡터와 Noise 벡터를 이용한 연산결과의 절대값이 저장되며, Alpha 값에는 연산결과 x y z 각각의 값의 부호가 +인지 –인지 여부를 16 비트 연산을 이용하여 저장하는데, x 가 음수인 경우 2^2 은 1, y 가 음수인 경우 2^1 을 1, z 가 음수인 경우 2^0 을 1 로 처리하고 3 개의 총합을 0 ~ 1 의 범위로 변환하여 저장한다. 생성된 최종 Lookup Table 은 (그림 5)과 같다.

```
for (int iSam = 0; iSam < 16; iSam++)
{
    for (int x = 0; x < 16; x++)
    {
        for (int y = 0; y < 16; y++)
        {
            Color _noise = NoiseTex.GetPixel(x, y);

            Vector3 random;
            random.x = _noise.r;
            random.y = _noise.g;
            random.z = _noise.b;

            Vector3 ray = sample_sphere16 [iSam] -
                (2.0f * random * Vector3.Dot (sample_sphere16 [iSam], random));

            float fSignR = ray.x < 0 ? 1 : 0;
            float fSignG = ray.y < 0 ? 1 : 0;
            float fSignB = ray.z < 0 ? 1 : 0;
            float fSign = (4*fSignR) + (2*fSignG) + (1*fSignB);

            Color _col;
            _col.r = Mathf.Abs(ray.x);
            _col.g = Mathf.Abs(ray.y);
            _col.b = Mathf.Abs(ray.z);
            _col.a = fSign / 256.0f;

            int iAddX = (iSam % 16) * 16;
            int iAddY = (iSam / 16) * 16;
            ray_texture.SetPixel (iAddX+x, iAddY+y, _col);
        }
    }
}
```

(그림 4) Lookup Table 생성 코드



(그림 5) 생성된 Lookup Table

3.3 Ambient Occlusion 구현

앞서 생성된 Lookup Table 을 활용한 샘플링 처리는 (그림 6)과 같은 방식으로 처리가 된다.

```
for(int i=0; i < SAMPLES; i++)
{
[1] float4 texray = tex2D ( _SampleTex,
                            float2( (i*16.0f/256.0f)+(pixel.scrPos.x*16.0f/256.0f),
                                    pixel.scrPos.y ) );

[2] int iRayAlpha = texray.a * 256.0f;
    texray.x *= iRayAlpha / 2 == 2 || iRayAlpha / 2 == 3 ? -1 : 1;
    texray.y *= iRayAlpha / 2 == 1 || iRayAlpha / 2 == 3 ? -1 : 1;
    texray.z *= iRayAlpha % 2 == 1 ? -1 : 1;

    float fSign = sign(dot(texray.xyz,normal));
    float3 hemi_ray = position + fSign * texray.xyz * radius_depth;

    float occ_depth;
    float3 occ_normal;
    DecodeDepthNormal(tex2D(_CameraDepthNormalsTexture, hemi_ray.xy), occ_depth, occ_normal);

    float difference = (depth - occ_depth);
    float fCalcStep = step(falloff, difference);
    float fSmoothStep = smoothstep(falloff, Area, difference);
    occlusion += fCalcStep * (1.0f - fSmoothStep);
}
```

(그림 6) Lookup Table 을 활용한 샘플링 처리

먼저 생성된 Lookup Table 에서 각각의 Sample 인덱스에 해당하며 Scene 의 UV 좌표에 매칭되는 값을 읽어오고[1], 다음으로 Lookup Table 을 생성하는 과정에서 Alpha 채널에 저장한 Reflect 연산 결과의 부호값을 읽어와서 역으로 변환하고, 최종적으로 RGB 채널로부터 읽어온 값에 곱해 준다[2].

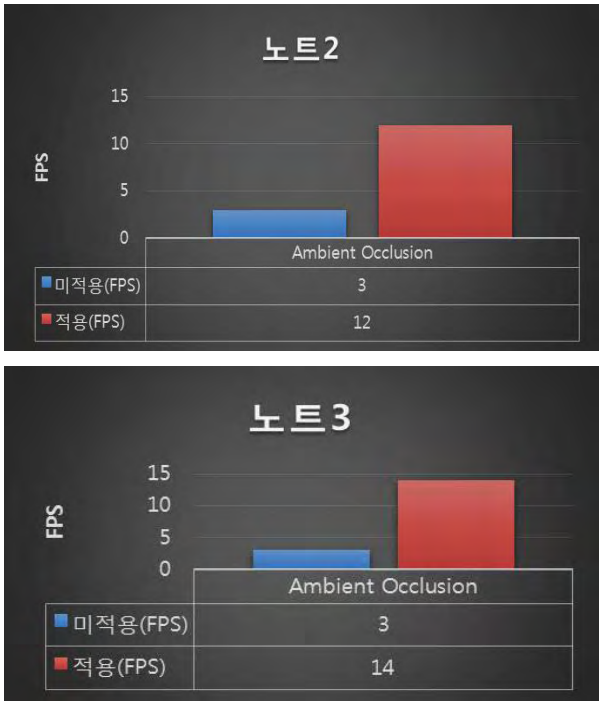
4. 결과 및 평가

새로 제안된 알고리즘의 성능을 확인하기 위해서 다양한 모델을 사용하여 기존 알고리즘과의 렌더링 결과 및 속도를 비교해 보았다.

(그림 7) Lookup Table 적용 전

(그림 8) Lookup Table 적용 후

뒤 배경과 오브젝트의 경계 부분을 유심히 살펴 보면 기존의 SSAO 알고리즘을 적용한 (그림 7)과 Lookup Table 을 활용한 Ambient Occlusion 알고리즘을 적용한 (그림 8) 두 개의 결과물에 있어서 큰 차이가 없음을 확인 할 수 있다. 다시 말해서 기존의 알고리즘과 새로운 알고리즘에 있어서 큰 차이가 없음이 확인이 되었다.



<표 5> Lookup Table 적용 전후 속도 비교

또한 <표 5>를 통해서 두 알고리즘간의 성능을 비교해 보면, 기존의 Ambient Occlusion 알고리즘을 이용한 렌더링 속도와 비교해서 Lookup Table 을 이용한 Ambient Occlusion 알고리즘을 적용 했을 때의 속도가 평균적으로 3~4 배 이상 빠른 것을 확인 할 수 있다.

5. 결론

본 논문에서 제안하는 Lookup Table 을 활용한 새로운 Ambient Occlusion 렌더링 기법에서는, 기존의 Ambient Occlusion 처리 과정에서 성능 저하에 많은 영향을 끼치는 부분을 제거하고 이를 전 처리된 Lookup Table 을 사용하도록 함으로써 성능 향상을 가져 올 수 있게 된다.

또한 본 논문에서는 Screen Space Ambient Occlusion 계산 과정 중에서 많은 연산량을 보이는 Relfect 연산에 한정해서 Lookup Table 을 적용해 보았지만, 이 연산뿐만 아니라 성능상의 저하를 가져오는 다양한 부분에 있어서 Lookup Table 을 활용할 수 있는 방법이 연구될 수 있을 것으로 기대한다. 그리고 본 논문에서 소개한 Screen Space Ambient Occlusion 뿐만 아니라 본 논문에서 소개하지는 않았지만 AMD 에서 발표한 High Definition Ambient Occlusion 과 NVIDIA 에서 발표한 Horizon Based Ambient Occlusion 등 다양한 Ambient Occlusion 렌더링 기법으로의 확장이 가능할 것으로 보이며, 적용 대상을 모바일 기기 뿐만 아니라 PC 콘솔 등 다양한 플랫폼으로 확장하여 성능에 크게 영향을 주는 부분을 Lookup Table 로 대체함으로써 연산량을 줄여 성능을 향상 시키는데 많은 도움이 될 것으로 기대한다.

참고문헌

[1]Matt Pharr and Simon Green: Ambient Occlusion. GPU Gems (2004), pp. 279-292.

[2]Michael Bunnell: Dynamic Ambient Occlusion and Indirect Lighting. GPU Gems 2(2005), pp. 223-233.

[3]Mittring, M.: Finding next gen: Cryengine 2. In: ACM SIGGRAPH 2007 Courses (2007), pp. 97–121.

[4]Engel W, Hoxley J, Kornmann R, Suni N, Zink J : Screen Space Ambient Occlusion. Programming vertex, geometry, and pixel shaders(2008), pp. 79-91.

[5]Hoberock, J., Jia, Y.: High-quality ambient occlusion. GPU Gems 3(2007), pp. 257–274.

[6]Pharr, Matt: Programming Techniques for High-Performance Graphics and General-Purpose Computation. GPU Gems 2 (2005).

[7]“Ambient Occlusion” http://en.wikipedia.org/wiki/Ambient_occlusion