Gamasutra.com

빛이 있으라! 새로운 게임 세대를 위한 통합 조명 기술

Let There Be Light!: A Unified Lighting Technique for a New Generation of Games

Jason Lacroix

2005 년 7 월 29 일

http://www.gamasutra.com/features/20050729/lacroix_01.shtml

게임 개발자들은 플레이어들이 개발자들이 만든 우주로 한층 더 몰입할 수 있도록 하기 위해 보다 현실성 있는 게임을 제공하고자 항상 노력하고 있다. 버텍스(Vertex) 및 픽셀 쉐이더(Pixel shader)의 출현으로 개발자들이 제작한 장면에 다양한 조명과 고급 비주얼 이펙트를 적용할 수 있게 됨으로써 위의 목표를 향해 진일보 할 수 있었다. 지난 세대의 게임(즉, Xbox/PS2/GC era)에서 프로그래머블 버텍스(programmable vertex)와 픽셀 파이프라인(pixel pipeline) 덕분에 역동적인 조명이 크게 발전할 수 있었다. 이제 대부분의 게임은, 보다 후기에 시장 진입한 좀 더 심미적인 픽셀당 조명(per-pixel lighting) 솔루션과 함께 버텍스(Vertex)를 기반으로 하여 산출된 조명을 지원한다. 이제 버텍스(Vertex)에서 픽셀 쉐이더(Pixel shader)로 또 다른 도약을 이루어 낼 시기이다.

차세대 게임 콘솔을 추진하는 요소 중 가장 두드러지는 현대의 GPU(Graphics Processing Units)는 믿을 수 없을 만큼의 프로세싱 파워를 제공한다. 이제 다양한 기술을 사용하여 픽셀 레벨당 연산된 장면에 모든 조명을 적용할 수 있다. 픽셀당 조명(per-pixel lighting)이 지난 쉐이더 모델(SM)을 통해 이용할 수 있는 것은 사실이나, SM3.0 의 도입으로 개발자들은 전통적으로 패스(pass) [1] 에서 산출할 수 있었던 조명 수의 제한을 없앨 수 있게 되었다.

이 글의 목적은 독자들에게 역동적 조명의 임의의 수를 다룰 수 있는 통합된 픽셀당 조명(per-pixel lighting) 솔루션과 그러한 솔루션이 채택해야 하는 최적화 기술에 대한 설명을 제시하는 것이다. 조명 기술의 현재와 그 한계에 대해 살펴봄으로써 이야기를 시작할 것이다. 그 다음, 어플리케이션(application)이 솔리드 프레임 레이트(solid frame rate)에서 적용됨을 보장할 수 있는 최적화와 함께 통합 모델에 대해 자세히 설명하고 그 한계에 대해 논의해보고자 한다.

이 글 곳곳에는 실행에 관한 세부 설명을 돕기 위해 여러 가지 쉐이더 코드 비트(bits of shader code)가 제시될 것이다. 이 때문에 기존의 조명 기술의 여러 한계와, 통합 조명(unified lighting) 솔루션이 동일한 시나리오로 어떻게 더 우수한 결과를 창출하는지에 대해 설명하기 위한 테스트 시나리오가 작성되었다. 테스트 시나리오는 조명 수(즉 환경)에 영향을 받는 정적이고 스킨이 없는 물체로 구성될 것이다. 모든 조명 소스는 포인트 조명(point light)으로 가정한다. 다른 유형의 조명뿐 아니라 정반사 조명 모델(specular lighting model)을 지원하는 코드의 추가는 독자의 몫으로 남겨두겠다. 조명 방정식(lighting equation)에 관심 있는 사람은 인터넷에서 쉽게 이용할 수 있다.

조명 방정식(lighting equation)에 익숙하지 않은 독자들을 위해 Point light's contribution 은 다음과 같이 연산된다.

전제 조건: N, 조명이 비춰지는 표면 위에 포인트의 평균
$P_{point,}$ 조명이 비춰지는 포인트의 위치
P_{light}, 조명의 위치
A0, A1, A2, 조명에 대한 감쇠 계수 [2]

(1) $Dist = |P_{point} - P_{light}|$, 조명과의 거리

(2) $L_{point} = P_{point} - P_{light} / |P_{point} - P_{light}|$, 조명에서 표면 포인트까지의 평균 방위

(3) $att = 1.0/(A0 + (A1*dist) + (A2*dist^2))$, 조명의 감쇠도

(4) $C_{point} = att * (N \cdot -L_{point})$, 전체 조명에 대한 색 분포

이 아티클에서 우리는 감쇠 계수로 인버스 선형 드롭 오프(inverse linear drop off)만을 사용할 것이다. 인버스 선형 감쇠도(Inverse linear attenuation)는 1/dist 로, A0 = 0.0, A1 = 1.0, A2 = 0.0 로 설정하는 것과 동일하다.

현재의 조명 기술에 대한 개요

버텍스 조명 (Vertex Lighting)

오늘날 게임에서 조명의 가장 일반적인 형태는 버텍스 조명(Vertex Lighting)이다. 이것은 본문에서 소개된 역동적인 조명 중에서 가장 빠른 형태이다. 버텍스 조명(Vertex Lighting)은 그 단어가 내포하고 있는 대로 작동한다. 각 조명의 색상 분포는 표면에 각 버텍스에서 연산되고 래스터화(rasterization) 과정에서 표면 전체에 보정된다(interpolated).

빠른 조명 수정이 필요한 경우, 이 솔루션은 결함이 적고 비용이 적게 든다. 그러나 이 방법에는 나중에 다루게 될 단점이 있다. 아울러, 표 1.1 에서 HLSL 에서 버텍스 조명(Vertex Lighting)에 대한 짧은 예를 보여주고 있다.

```
float4x4 matW;      // World matrix
float4x4 matVP;     // View-Projection matrix
// Let's assume there are 5 lights
float3 fLightPos[5];
float3 fLightColor[5];

struct VSInput
{
 float3 pos          : POSITION0;      // Position
 float3 normal       : NORMAL0;        // Normal
 float4 color        : COLOR0;         // Color
};

struct VSOutput
{
 float4 pos          : POSITION;
 float4 color        : COLOR0;
};

VSOutput VertexShader( in VSInput input )
{
 VSOutput output;

 // Transform the vertex into world space
 float3 pos = mul( input.pos, (float3x3)matW );

  // Complete vertex transformation
  output.pos = mul( float4(input.pos, 1.0), matVP );

 // ... and the normal
 float3 normal = mul( input.normal, (float3x3)matW );

 // Setup a color register and copy the emmissive color (vert color) and alpha of the vertex
 float4 tmpColor = input.color;

 // Do the lighting
 for( int i = 0; i < 5; i++ )
 {
  // Compute distance to light
  float dist = length(pos – fLightPos[i]);
  // … and direction …
  float3 direction = normalize(pos – fLightPos[i]);
  // … and attenuation
  float attenuation = 1/dist;
  // Add light contribution
  tmpColor.xyz += fLightColor[i] * attenuation * dot( normal, -direction );
 }

 // Write the color out
 output.color = tmpColor;

 return output;
}
```

표 1.1: HLSL 을 이용한 간단한 포인트 조명(Point lighting)

픽셀당 노말 맵(Per-pixel normal map) 기반의 조명

노말 맵 조명(Normal map lighting)은 각 버텍스가 아닌 각 픽셀에서 조명 계산식을 산출하기 위해, 텍스처에 공간에 대한 탄젠트 공간 법선(tangent-space normal)[3]을 인코딩 함으로써 다른 방법으로 조명에 접근하고 있다. 옵젝트 공간 노말 맵(Object space normal maps) 또한 가능하며 일반적으로 역동적인 물체에 사용된다. 이러한 범프 매핑(bump mapping)의 형태는 그래픽의 경계를 허물고자 하는 게임에 대한 기준으로 빠르게 자리잡았다. 대부분의 새로운 게임은 주요 조명 기술로서 이것에 의존하고 있는데, 그 이유는 이 조명 기술로써 저량의 폴리곤(polygon)을 유지하면서 디테일의 레벨을 엄청난 수준으로 끌어올려 주기 때문이다. 그리고 패럴랙스 맵핑(parallax mapping)이라고 불리는 노말 맵 조명(Normal map lighting)의 변형물이 있는데, 이것은 패럴랙스 효과를 시뮬레이트 하기 위해 일반 텍스처에 추가적인 하이트 맵(height map)을 인코드하는 것이다. ATI 나 Nvidia 의 개발자 사이트에서 이것에 대한 자세한 정보 및 실행 디테일을 이용할 수 있다.

노말 맵 조명(Normal map lighting) 실행을 위 해서는 세 단계의 접근 방법이 필요하다. 노말 맵이 우선 생성되어 모델에 적용되어야 하며 탄젠트 공간(tangent-space) 정보와 함께 보낸다 [4]. 다음으로, 버텍스 쉐이더(vertex shader)에 노말 매핑(normal mapping)할 표면에 버텍스(vertex)들을 처리할 때, 모든 조명의 위치 정보를 탄젠트 공간(tangent-space)으로 변형시키기 위해 탄젠트 매트릭스(tangent matrix)를 만들어야 한다(모든 조명 수식은 동일한 좌표 공간에서 수행되어야 한다). 탄젠트 공간 매트릭스(Tangent space matrix)는 버텍스의 접선(vertex's tangent), 종법선(binormal), 법선 벡터(normal vector)로 구성된 3 x 3 매트릭스이다. 종법선 벡터는 접선과 법선의 벡터곱을 계산하여 얻을 수 있다. 마지막으로, 각 조명의 색상 분포는 버텍스 쉐이더(vertex shader)에서 변형된 데이터로부터 산출한 탄젠트 공간 조명 벡터(tangent space lighting vector)와 노말 맵(normal map)에서 가져 온 정보를 이용하여 픽셀 쉐이더(pixel shader)에서 계산한다. 노말 맵 조명(Normal map lighting)의 HLSL 실행을 나타낸 표 1.2 를 참조한다.

VertexShader.vsh :

```
float4x4 matW;
float4x4 matVP;
// Let's assume there are no more than 5 lights
float3 fLightPos[5];

struct VSInput
{
  float3 pos            : POSITION0;      // Position
  float3 normal         : NORMAL0;        // Normal
  float3 tangent        : TANGENT0;       // Tangent
  float4 color          : COLOR0;         // Color
  float2 uv             : TEXCOORD0;      // UVs for normal map
};

struct VSOutput
{
  float4 pos            : POSITION;
  float4 color          : COLOR0;
  float2 uv             : TEXCOORD0;
  // Need to export a world position and all of our light positions
  float3 worldPos       : COLOR1;
  float3 lightPos[5]    : TEXCOORD1;
};

VSOutput VertexShader( in VSInput input )
{
  VSOutput output;

  // Transform the vertex
  float3 worldPos = mul( input.pos, (float3x3)matW );
  // Complete vertex transformation
  output.pos = mul( float4(worldPos, 1.0), matVP );
  output.worldPos = worldPos;

  // ... and the normal
  float3 normal = mul( input.normal, (float3x3)matW );
  // ... and the tangent
  float3 tangent = mul( input.tangent, (float3x3)matW );

  // Compute binormal vector
  float3 binormal = cross( tangent, normal );

  // Copy the emmissive color (vert color) and alpha of the vertex
  output.color = input.color;
  // Copy uv coordinates for normal map out
  output.uv = input.uv;

  // Do the lighting information conversion
  for( int i = 0; i < 5; i++ )
  {
    // Compute tangent space light position
    output.lightPos[i] = mul( fLightPos[i], float3x3(tangent, binormal, normal) );
  }

  return output;
}
```

PixelShader.psh :

```
// Again .. assuming 5 lights ...
float3      fLightColor[5];

// Need a sampler to fetch from the normal map
sampler sNormalMap;

struct PSInput
{
  float4 color          : COLOR0;
  float2 uv             : TEXCOORD0;
  // Import all lighting positions and world position
  float3 worldPos       : COLOR1;
  float3 lightPos[5]    : TEXCOORD1;
};

float4 PixelShader( inPSInput input ) : COLOR0
{
  // Setup the temperary color register
  float4 tmpColor = input.color;

  // Fetch the normal from the texture
  // Need to bias the result to get it
  // in the range of -1 to 1
  float3 normal = 2.0 * (tex2D( sNormalMap, input.uv ) - 0.5);

  float3 lightDir;

  // Do the lighting
  for( int i = 0; i < 5; i++ )
  {
    // Compute distance to light
    float dist = length(worldPos – input.lightPos[i]);
    // ... and direction ...
    float3 direction = normalize(worldPos – input.lightPos[i]);
    // ... and attenuation
    float att = 1/dist;
    // Add light contribution
    tmpColor.xyz += fLightColor[i] * att * dot( normal, -direction );
  }

  return tmpColor;
}
```

표 1.2 : 방향성 조명(directional light)과 함께 간단한 노말 맵 조명(normal map lighting)에 대한 버텍스/픽셀 쉐이더(Vertex/Pixel shader)

기존 조명 모델의 한계

버텍스(Vertex)와 노말 맵 조명(Normal map lighting)이 지난 5-6 년간 널리 이용되었지만, 이들에게 단점이 없는 것은 아니다. 통합된 픽셀당 솔루션(unified per-pixel solution)에 대해 언급하기 전에, 왜 새로운 조명 모델이 필요한지, 그 이유를 이해하는 것이 도움이 된다.

보정된 조명(버텍스 조명)

대부분의 경우 버텍스 조명(Vertex Lighting)이 빠르고 효과적이지만, 얻어진 결과가 기대에 못 미치는 시나리오가 일부 존재한다. 두 개의 삼각형으로 이루어진 대형 폴리곤에 영향을 미치는 포인트 조명(point lighting)을 예로 들어보자. 버텍스 조명(Vertex Lighting)은 각 버텍스(vertex)에 도달한 색을 보정하여 만들어지기 때문에, 이 시나리오는 쿼드(quad)가 전체 표면에 걸쳐 동일하게 조명을 받는 결과를 빚어낸다(포인트 조명(point lighting)이 가장자리에 도달하지 않으면 빛이 생기지 않는다 – 그림 1). 이 문제를 해결하기 위해 쿼드(quad)는 중앙에서 가장자리로 폴오프(falloff)를 만들어 내기 위해 모자이크 처리를 해야 한다(그림 2). 이것은 예술팀에게 역효과를 불러일으키며, 픽셀당 조명(per-pixel lighting) 방법을 사용하여 쉽게 수정될 수 있는 문제점을 갖고 있다.

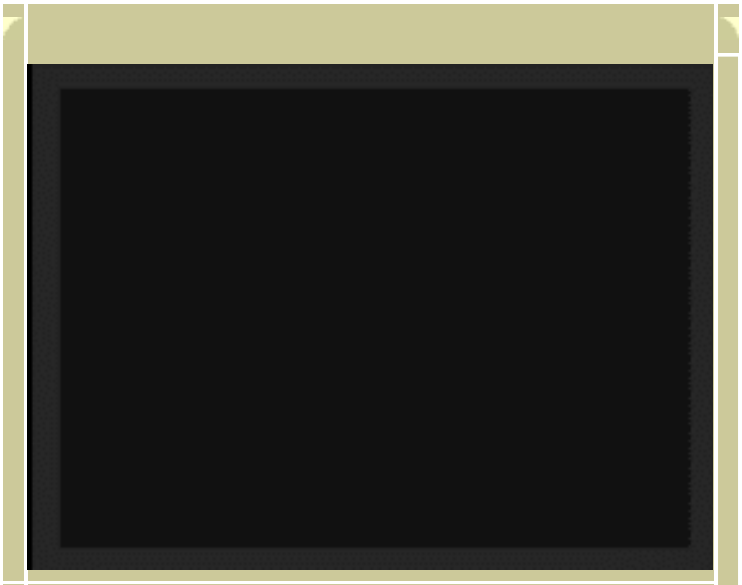

그림 1 – 로우 폴리(low poly) 표면에서의
포인트 조명(point lighting) (버텍스 4)

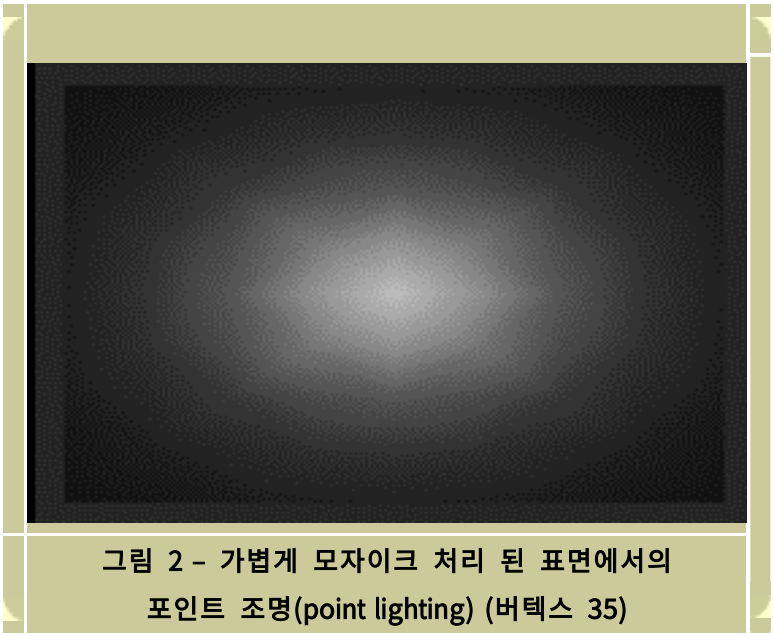

그림 2 – 가볍게 모자이크 처리 된 표면에서의 포인트 조명(point lighting) (버텍스 35)

조명 수의 제한(노말 맵 조명)

노말 맵 조명(Normal map lighting)은 버텍스 조명(Vertex Lighting)의 고유한 문제점을 비켜갈 수는 있지만 자체의 한계를 갖고 있다. 문제는 모든 조명 연산이 동일한 좌표 시스템에서 수행되어야 하는 것에서 기인한다. 노말 맵 조명(Normal map lighting)의 현재 실행은 버텍스 쉐이더(vertex shader)가 탄젠트 공간(tangent space)에 필요한 조명 정보와 픽셀 쉐이더(pixel shader)에서 산출된 색 분포를 변형시킨다. 문제는 명확하다. 표면이 빛을 받을 수 있는 조명 공급원의 개수가 레지스터의 수로 제한되면 버텍스 쉐이더(vertex shader)가 픽셀 쉐이더(pixel shader)로 통과할 수 있다.

텍스처 레지스터(texture register)를 통해 조명 정보를 통과시키고자 하는 경우를 생각해보라. GeForce 6800 에서 4 자리의 부동 소수점 엔트리(floating point entries)의 벡터 10 개를 통과시킬 수 있다. 노말 맵(Normal map)에서 레귤러 텍스처(regular texture), 방향성 조명(directional lighting)뿐이라고 가정할 때 언제든 최대 8 개의 조명이 표면에 적용된다. 많은 사람들은 여기에서 “충분하지 않나” 라고 생각한다. 그렇다. 조명 8 개는 적절한 조명과 물체가 실제처럼 보이도록 하는 데는 충분한 개수로 생각될 수 있다. 하지만 다음을 고려해 보자. 포인트 조명(Point lighting)은 감쇠와 방향을 연산하기 위한 위치 저장을 위해 1 개의 레지스터를 여분으로 필요로 하고 (즉, 7 개의 조명), 스포트 라이트(spot light)는 조명당 두 개의 레지스터를 필요로 한다 (4 개까지로 제한을 둔다). 여러 가지 형태의 조명을 섞는 것은 일을 더욱 망칠 뿐이고, 차세대 게임에 출시될 다른

복잡한 효과나 환경 맵(environment map)의 주체, 2 차 텍스처(secondary texture)는 아직 도입되지 않았다. 향후 필요하게 될 그러한 복잡한 효과로, 하나 또는 두 개의 레지스터 만으로도 일을 마칠 수 있게 된다.

통합 픽셀당 조명 솔루션(Unified per-pixel lighting solution)

버텍스(Vertex)와 노말 맵 조명(Normal map lighting)에 대해 제안된 솔루션은 실제로 간단한 것이다. 모든 공간의 조명 정보를 픽셀 쉐이더(pixel shader)에 직접 업로드하고 픽셀당 조명(per-pixel lighting) 연산을 수행하는 것이다.

이 솔루션의 실행은 노말 맵(normal map)이 존재 하느냐의 여부에 따라 약간 다른 방법이 필요하다. 다음으로는, 노말 맵(normal map)이 존재하지 않을 때 어떻게 픽셀 쉐이더(pixel shader)에서 우주 공간 법선(world space normal)을 구하고, 존재할 경우에는 어떻게 탄젠트 공간 법선(tangent space normal)을 구하는지 검토해 볼 것이다. 두 개의 솔루션 모두 이전(버텍스 조명(Vertex Lighting) 및 노말 맵 조명(normal map lighting))과 마찬가지로 동일한 데이터 집합을 필요로 한다. 이렇게 통합 조명 솔루션(unified lighting solution)과 함께 실제 조명 코드는 항상 노말 맵(normal map)의 존재 여부와 상관없이 동일하게 남아있을 것이다. 이는 시나리오를 바탕으로 하여 법선을 산출하는 프래그먼트 쉐이더(fragment shader)를 설치하는데 도움을 주고, 조명 프래그먼트(lighting fragment)에 그것을 통과시킨다. 두 가지 시행에 모두 필요한 조명 쉐이더 코드(lighting shader code)는 표 2.1 에서 볼 수 있다.

```
float3      fLightColor[X];
float3      fLightPos[X];

struct PSInput
{
  float4 color          : COLOR0;
  float3 worldPos    : COLOR1;
  // Other inputs depending on technique used
};

float4 PixelShader( in PSInput input ): COLOR0
{
  // Setup the temperary color register
  float4 tmpColor = input.color;

// ....
// Compute normal from one of 2 presented techniques
// ....

  // Do the lighting
  for( int i = 0; i < X; i++ )
  {
    // Compute distance to light
    float dist = length(input.worldPos – flightPos[i]);
    // ... and direction ...
    float3 direction = normalize(input.worldPos – flightPos[i]);
    // ... and attenuation
    float att = 1/dist;
    // Add light contribution
    tmpColor.xyz += fLightColor[i] * att * dot( normal, -direction );
  }

  return tmpColor;
}
```

표 2.1: 조명으로부터의 색 분산(color contribution)을 구하기 위한 픽셀당 조명(per-pixel lighting) 연산

색이 아닌 법선 보정

검토해 볼 첫 번째 솔루션은 어떤 것이 버텍스 조명(Vertex Lighting) 문제를 해결하는 것인지 이다. 앞서 주어진 대형 표면 예제로 돌아가보자. 각 버텍스(vertex)의 색은 폴리곤의 표면에 걸쳐 보정되어 버텍스(vertex)가 거의 사용되지 않기 때문에 결국, 결과는 어느 정도 고른 조명 분포를 지닌 대형 쿼드(quad)로 나타나거나 이 경우에서는 아무런 결과도 나타나지 않았다. 이 문제를 해결하기 위해(동일한 버텍스 수를 사용하여) 우리는 각 픽셀을 개별적으로 비추어 표면의 조명 분산이 우리가 지정한 (이 경우에서는 인버스 선형(inverse linear)) 특정 감쇠에 따라 적절히 폴오프(falloff) 될 것이다.

이렇게 하기 위해, 우리는 픽셀 쉐이더(pixel shader) 위에 버텍스 쉐이더(vertex shader)에서 국소 사용되고 변형되는 표면 법선을 구해야 한다. 이는 픽셀 쉐이더(pixel shader)를 공급할 때 하드웨어가 표면에 데이터를 보정한다는 사실을 이용하여 쉽게 수행할 수 있는 것으로 드러났다. 버텍스(vertex)에 대한 색을 산출하는 것 대신, 우주 공간에 법선을 단순 변형하고 픽셀 쉐이더(pixel shader)에서의 처리를 위한 방사 색과 나란히 레지스터에 위치시킬 것이다. 또한 우리는 포인트 조명(point lighting)을 실행하고 있으므로, 쉐이더(shader) 전체에 버텍스(vertex)의 공간 위치를 발송하여 보정된 우주 공간 픽셀 값을 얻을 수 있다. 그 후에, 버텍스 쉐이더(vertex shader)에서와 마찬가지 방법으로 픽셀 쉐이더(pixel shader)에서 조명 연산을 수행하기만 하면 된다. HLSL 시행에 대한 표 2.2 와 비주얼 결과에 대한 그림 3 을 참조한다.

VertexShader.vsh:

```
float4x4 matW;
float4x4 matVP;

struct VSInput
{
  float3 pos          : POSITION0;      // Position
  float3 normal       : NORMAL0;        // Normal
  float4 color        : COLOR0;         // Color
};

struct VSOutput
{
  float4 pos          : POSITION;
  float4 color        : COLOR0;
  // Export normal and worldspace position
  float3 normal       : TEXCOORD0;
  float3 worldPos     : TEXCOORD1;
};

VSOutput VertexShader( in VSInput input )
{
  VSOutput output;

  // Transform the vertex
  float3 worldPos = mul( input.pos, (float3x3)matW );
  output.pos = mul( float4(worldPos, 1.0), matVP );

  // ... and the normal
  output.normal = mul( input.normal, (float3x3)matW );

  // Write out world position
  output.worldPos = worldPos;

  // Copy the emmissive color (vert color) and alpha of the vertex
  output.color = input.color;

  // copy
  return output;
}
```

PixelShader.psh:

```
float3    fLightColor[X];
float3    fLightPos[X];

struct PSInput
{
  float4 color        : COLOR0;
  // Import normal and worldspace position
  float3 normal       : TEXCOORD0;
  float3 worldPos     : TEXCOORD1;
};

float4 PixelShader( in PSInput input ) : COLOR0
{
  // Setup the temperary color register
  float4 tmpColor = input.color;

  // Get the normal and renormalize it
  // (as interpolation messes with unit vectors)
  float3 normal = normalize(input.normal);

  // ...
  // Perform lighting according to listing 2.1
  // ...

  return tmpColor;
}
```

표 2.2: 보정된 노말 쉐이더(Interpolated normal shader)

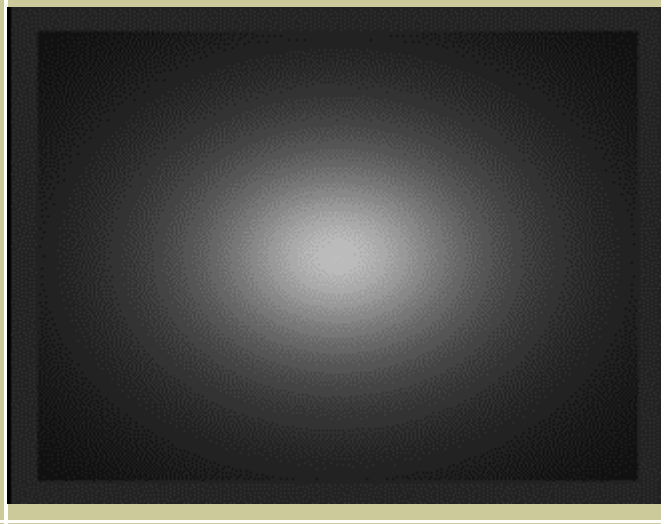

그림 3 – 픽셀당 조명(per-pixel lighting)와 로우 폴리(low poly) 표면에 포인트 조명(Point light)(버텍스 4)

우주 공간에서 노말 맵 조명(normal map lighting) 실행

두 번째 솔루션은 현재의 노말 맵 조명(normal map lighting) 기술에 내재하는 조명 제한 문제를 해결한다. 다시 말해 이 솔루션은 우주 공간을 비추는 것이다. 그러므로 우리는 우리가 관리할 수 있는 한도 내에서 가능한 한 많은 조명을 업로드 할 수 있다. 여전히 픽셀 쉐이더(pixel shader)에 업로드 할 수 있는 지속적인 레지스터의 수에도 2005 년 4 월을 기준으로 DirectX 9c 버전에서 224 개라는 한계를 가지고 있지만, 아티스트가 꿈꿀 수 있는 어떤 복잡한 효과라도 표현하기에 충분하다.

우주 공간에 탄젠트 공간 법선(tangent space normal)을 구하기 위해서는 버텍스의 법선 벡터(normal vectors of the vertex), 접선(tangent), , 종법선(binormal)을 이용하여 탄젠트 공간 매트릭스(tangent space matrix)를 제작할 필요가 있다. 그러나 탄젠트 공간(tangent space)으로의 데이터 변환을 위해 이러한 매트릭스를 사용하는 대신, 우리는 인버스 탄젠트 공간 매트릭스(inverse tangent space matrix)(매트릭스가 직교인 경우 이 작업을 위해 우리는 전환 고유 기능(transpose instrinsic function)을 사용할 수 있다 – 예술 팀에 몇 가지 제한을 둠으로써 쉽게 진행할 수 있음)를 연산하고, 이를 우주 공간 매트릭스(world space matrix)와 곱한다. 이것은 탄젠트 공간 법선(tangent space normal)을 우주 공간으로 변형하고 모든 조명 연산을 수행하는데 이용할 수 있도록 해줄 것이다.

노말 맵(normal map)에서 탄젠트 공간 법선(NT)가 주어진다면 다음과 같이 우주 공간 법선(NW)을 산출할 수 있다.

$(NT)(T^{-1}W) = N(TT^{-1})W = N(I)W = NW$ (우주 공간에서의 법선)

쉐이더(shader) 시행은 표 2.3 을 참조하기 바란다.

이 솔루션을 사용하여 얻은 우주 공간 법선(world space normal)에는 일부 정확성 오류가 발생할 수도 있음에 주의한다. 이는 노말 맵(normal map) 생성시 연산되고 법선을 변형하는데 사용된 실제의 텍셀(texel) 기반의 탄젠트 공간(tangent space)과는 대조적으로, 보정된 버텍스 데이터(interpolated vertex data)로부터 탄젠트 공간 매트릭스(tangent space matrix)를 재 생성 한다는 사실에 기인한다. 대부분의 경우 이 솔루션은 완벽하게 적용된다. 그러나 여러 버텍스(vertex)가 하나의 법선을 공유하지만, 접선 정보는 공유하지 않을 때에는 shading seams 이 기하학적 배열에서 나타날 것이다(Morten Mikkelsen 에게 이 점을 지적해 준 것에 대해 감사의 말을 전한다).

VertexShader.vsh :

```
float4x4 matW;
float4x4 matVP;

struct VSInput
{
  float3 pos          : POSITION0;      // Position
  float3 normal       : NORMAL0;        // Normal
  float3 tangent      : TANGENT0;       // Tangent
  float4 color        : COLOR0;         // Color
  float2 uv           : TEXCOORD0;      // UVs for normal map
};

struct VSOutput
{
  float4 pos          : POSITION;
  float4 color        : COLOR0;
  float2 uv           : TEXCOORD0;
  // Need to export tangent, normal and world pos
  float3  worldPos    : COLOR1;
  float3  tangent     : TEXCOORD1;
  float3  normal      : TEXCOORD2;
};

VSOutput VertexShader( in VSInput input )
{
  VSOutput output;

  // Transform the vertex
  float3 worldPos = mul( input.pos, (float3x3)matW );
  // Complete vertex transformation
  output.pos = mul( float4(worldPos, 1.0), matVP );
  output.worldPos = worldPos;

  // Copy normal and tangent vectors directly into output
  output.tangent = input.tangent;
  output.normal = input.normal;

  // Copy the emmissive color (vert color) and alpha of the vertex
  output.color = input.color;
  // Copy uv coordinates for normal map out
  output.uv = input.uv;

  return output;
}
```

PixelShader.psh :

```
float3      fLightColor[X];
float3      fLightPos[X];
float4x4    matW;                   // The World matrix

// Need a sampler to fetch from the normal map
sampler sNormalMap;

struct PSInput
{
  float4 color          : COLOR0;
  float2 uv             : TEXCOORD0;
  // Import world position, tangent and normal
  float3 worldPos       : COLOR1;
  float3 tangent        : TEXCOORD1;
  float3 normal         : TEXCOORD2;
};

float4 PixelShader( inPSInput input ) : COLOR0
{
  // Setup the temperary color register
  float4 tmpColor = input.color;

  // Fetch the normal from the texture
  // Need to bias the result to get it
  // in the range of -1 to 1
  float3 tmpNormal = 2.0 * (tex2D( sNormalMap, input.uv ) - 0.5);

  // Renormalize our resulting inputs and get binormal
  float3 tangent = normalize( input.tangent );
  float3 normal = normalize( input.normal );
  float3 binormal = normalize( cross( tangent, normal ) );

  // Compute matrix inverse
  Float3x3 matInverse = transpose( float3x3(tangent, binormal, normal) );

  // Bring normal into world space
  normal = mul( mul( tmpNormal, matInverse ), matW );

  // ...
  // Perform lighting according to listing 2.1
  // ...

  return tmpColor;
}
```

표 2.3: 우주 공간 조명에서(world space light) 탄젠트 공간 (Tangent space)으로의 전환 쉐이더(conversion shader)