

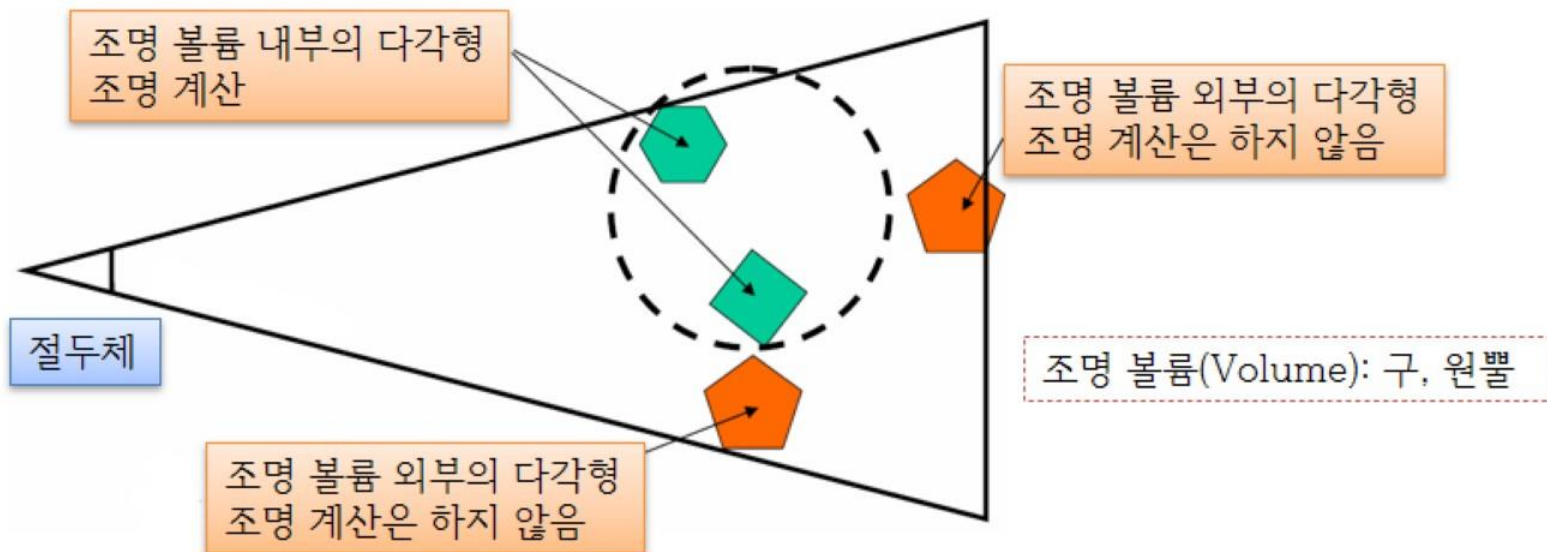
Game Programming with DirectX

지연 조명 (Deferred Shading)

조명 처리(Lighting)

• 조명 처리의 최적화

- 조명 볼륨 내부의 표면을 셰이딩(최적화)



- 스텐실 컬링(Stencil Culling)

첫번째 단계는 계산량이 많지 않으므로 색상을 출력하지 않으면서 렌더링

1. 색상 출력(Color Write)을 비활성화하여 조명 볼륨을 렌더링

`DepthFunc = D3D12_COMPARISON_FUNC_LESS;`

`FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_ALWAYS;`

`FrontFace.StencilDepthFailOp = D3D12_STENCIL_OP_REPLACE; //참조값(a)`

나머지 스텐실 연산: `D3D12_STENCIL_OP_KEEP;`

2. 조명 계산 셰이더를 사용하여 출력

`DepthFunc = D3D12_COMPARISON_FUNC_ALWAYS;`

`FrontFace.StencilFunc = D3D12_COMPARISON_FUNC_EQUAL; //참조값(a)`

조명이 영향을 주지 않는 픽셀은 컬링됨(스텐실 값이 참조값과 일치하지 않으므로)

D3D12_DEPTH_STENCIL_DESC

지연 셰이딩(Deferred Shading)

- 조명(Lighting)

- 많은 조명과 많은 객체들을 렌더링하는 경우 계산량이 많음
- 단일-패스, 다중-패스, 지연 셰이딩

- 단일-패스 조명(Single-Pass Lighting)

For each object:

Render object(Apply all lighting in one shader)

$O(\text{Objects} * \text{Lights} * \text{ScreenPixels})$

- 가려진 표면들에 대한 조명 효과는 화면에 나타나지 않음(낭비: Overdraw)
- 여러 조명을 관리하기 어렵고 그림자(Shadow)와 통합하기 어려움

- 다중-패스 조명(Multi-Pass Lighting)

Bidirectional Reflectance Distribution Function

For each light:

For each object affected by light

Framebuffer += BRDF(object, light)

- 가려진 표면들에 대한 조명 효과는 화면에 나타나지 않음(낭비)

깊이 검사
(Depth Test)

- 지연 조명(Deferred Shading)

For each object:

Render lighting(material) properties into "G-buffer"

For each light:

Framebuffer += BRDF(**G-buffer**, light)

$O((\text{Objects} + \text{Lights}) * \text{ScreenPixels})$

- 픽셀에 대한 조명 계산을 하지 않고 정보를 모아 나중에 조명 효과를 계산

지연 셰이딩(Deferred Shading)

• 지연 셰이딩(Deferred Shading)

- 화면-좌표계의 셰이딩(조명 처리) 기법
- 정점 셰이더 또는 픽셀 셰이더에서 셰이딩(조명 처리)을 두 단계로 수행함
 - 첫 번째 단계
셰이딩을 먼저 수행하지 않고 보이는 픽셀에 대한 조명 처리 데이터를 수집
각 표면(픽셀)을 위한 위치 벡터, 법선 벡터, 재질 등을 일련의 텍스처로 렌더링
이러한 일련의 텍스처를 기하 버퍼(**G-Buffer**: Geometry Buffer)라고 함
 - 두 번째 단계
각 픽셀에 대하여 **화면 좌표계(Screen Space)**에서 조명 효과를 계산함
- 씬과 조명을 분리
각 조명의 효과는 화면의 각 픽셀에 대하여 한번만 계산함
조명이 많은 경우 성능 저하의 문제없이 렌더링할 수 있음
조명 관리를 쉽게 할 수 있음
투명 효과를 처리할 수 없음(투명한 객체들은 따로 렌더링)
여러 재질을 사용하는 경우 어려움이 있음
- G-버퍼
실수 텍스처(위치 벡터 등을 저장할 수 있어야 함)
- 다중 렌더 타겟(MRT: Multiple Render Targets)
단일 패스로 모든 G-버퍼 속성을 출력

$O(\text{Lights} * \text{Triangles})$



$O(\text{Lights}) + O(\text{Triangles})$

씬 복잡도
(Scene Complexity)

조명 복잡도
(Light Complexity)

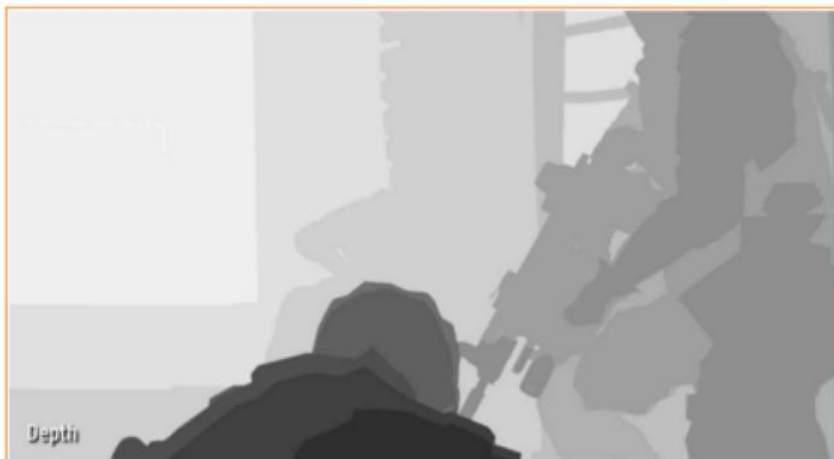
지연 셰이딩(Deferred Shading)

- 지연 셰이딩(Deferred Shading): Killzone 2
 - <http://www.guerrilla-games.com/>

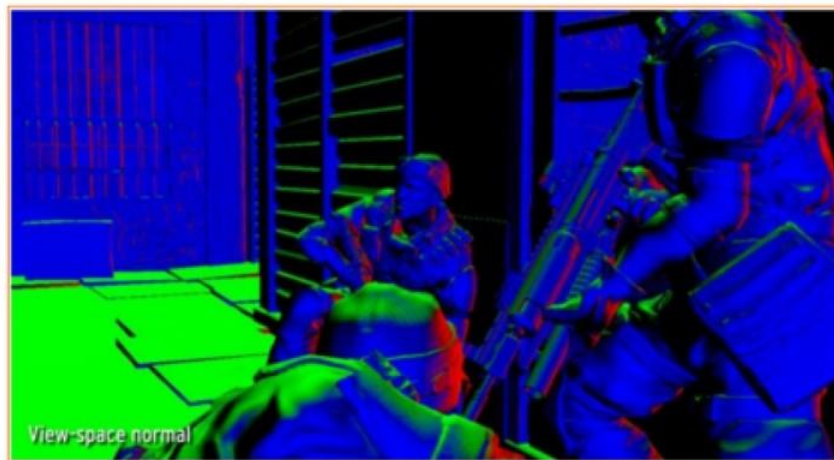


지연 셰이딩(Deferred Shading)

- 지연 셰이딩(Deferred Shading): Killzone 2



깊이값(Depth)



법선 벡터(Normal)



스펙쿨러(Specular Intensity)



스펙쿨러(Specular Power)

지연 셰이딩(Deferred Shading)

- 지연 셰이딩(Deferred Shading): Killzone 2



화면좌표계 2D 모션 벡터(Motion Vector)



색상(Albedo: Texture Color)



지연 결합(Deferred Composition)



후처리(Post Processing)

지연 셰이딩(Deferred Shading)

- 지연 셰이딩(Deferred Shading): Killzone 2

- G-Buffer(Killzone 2)

$4 * R8G8B8A8 + (24\text{-Bit Depth} + 8\text{-Bit Stencil}) \approx 36\text{MB}$

R8	G8	B8	A8	Buffer
Depth 24-Bit			Stencil 8-Bit	Depth Stencil
Lighting Accumulation RGB 24-Bit			Intensity 8-Bit	Render Target 0
Normal X (Floating Point) 16-Bit		Normal Y (Floating Point) 16-Bit		Render Target 1
Motion Vector XY 16-Bit		Specular Power	Specular Intensity	Render Target 2
Diffuse Albedo RGB 24-Bit			Occlusion	Render Target 3

- G-Buffer(Battlefield 3)

R8	G8	B8	A8	Buffer
Depth 24-Bit			Stencil 8-Bit	Depth Stencil
Lighting Accumulation RGB 24-Bit			Intensity 8-Bit	Render Target 0
Normal X (Floating Point) 16-Bit		Normal Y (Floating Point) 16-Bit		Render Target 1
Motion Vector XY 16-Bit		Specular Power	Specular Intensity	Render Target 2
Diffuse Albedo RGB 24-Bit			Occlusion	Render Target 3

지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

① 기하 버퍼(GBuffer) 생성 단계

썬을 텍스처(GBuffer) 로 렌더링(조명 계산을 하지 않음)

- 법선 벡터
- 디퓨즈 반사
- 스펙큘러 반사
- 위치 벡터
- ...

② 조명 계산 단계

화면 전체 크기의 사각형 렌더링(조명 계산)
조명 정보는 텍스처(GBuffer)에서 로드(샘플링)

```
struct VS_GB_INPUT
{
    float4 position : POSITION;
    float2 uv : TEXCOORD;
    float3 normal : NORMAL;
    float3 tangent : TANGENT;
};
```

```
struct VS_GB_OUTPUT
{
    float4 position : SV_Position;
    float2 uv : TEXCOORD;
    float3 normalW : NORMALW;
    float3 position : POSITIONW;
    float3 tangent : TANGENTW;
    float3 bitangentW : BITANGENTW;
};
```

```
VS_GB_OUTPUT VSDeferredLighting(VS_GB_INPUT input)
```

```
{
    VS_GB_OUTPUT output;

    output.position = mul(input.position, gmtxWorld);
    output.positionW = output.position.xyz;
    output.position = mul(output.position, gmtxView);
    output.position = mul(output.position, gmtxProjection);
    output.normalW = normalize(mul(input.normal, (float3x3)gmtxWorld));
    output.tangentW = normalize(mul(input.tangent.xyz, (float3x3)gmtxWorld));
    output.bitangentW = normalize(cross(output.normalW, output.tangentW));
    output.uv = input.uv;
    return(output);
}
```

지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

① GBuffer 생성 단계

[earlydepthstencil]

PS_GB_OUTPUT **PSDeferredLighting**(VS_GB_OUTPUT input)

```
{
    PS_GB_OUTPUT output;
    float3 T = normalize(input.tangentW);
    float3 B = normalize(input.bitangentW);
    float3 N = normalize(input.normalW);
    float3x3 TBN = float3x3(T, B, N);
    float3 normal = normalize(gtxTextureMap.Sample(gssNormalMap, input.uv).rgb * 2.0f - 1.0f);
    output.normal = float4(mul(normal, TBN), 1.0f);
    float3 diffuseAlbedo = gtxTextureMap.Sample(gssDiffuseMap, input.uv).rgb;
    //매핑할 텍스처의 색상을 디퓨즈 반사 색상으로 사용하기도 함
    output.diffuseAlbedo = float4(diffuseAlbedo, 1.0f);
    output.specularAlbedo = float4(0.7f, 0.7f, 0.7f, 64.0f / 255.0f);
    //output.specularAlbedo = gcMaterialSpecular; //power : (0~255) → (0 ~ 1.0)
    output.position = float4(input.positionW, 1.0f);
    return(output);
}
```

```
cbuffer cbMaterial : register(b1)
{
    float4 gcMaterialDiffuse;
    float4 gcMaterialAmbient;
    float4 gcMaterialSpecular; //(r, g, b, power)
    float4 gcMaterialEmissive;
};
```

```
Texture2D gtxtDiffuseMap : register(t0);
SamplerState gssDiffuseMap : register(s0);

Texture2D gtxtNormalMap : register(t1);
SamplerState gssNormalMap : register(s1);
```

```
struct PS_GB_OUTPUT
{
    float4 normal : SV_Target0;
    float4 diffuseAlbedo : SV_Target1;
    float4 specularAlbedo : SV_Target2;
    float4 position : SV_Target3;
};
```

```
float4 Lighting(float3 vPosition, float3 vNormal);
```


지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

② 조명 계산 단계(화면 전체 크기의 사각형 렌더링)

```
VS_OUTPUT VS(in VS_INPUT input)
{
    VS_OUTPUT output;
    output.position = input.position;
    return(output);
}
```

```
Texture2D gtxtNormal : register(t0);
Texture2D gtxtDiffuseAlbedo : register(t1);
Texture2D gtxtSpecularAlbedo : register(t2);
Texture2D gtxtPosition : register(t3);

Texture2D gtxtDepth : register(t4);
```

```
struct VS_INPUT
{
    float4 position : POSITION;
};
```

```
struct VS_OUTPUT
{
    float4 position : SV_Position;
};
```

```
ret Object.Load(
    int# Location, //배열 인덱스(u, v, mip)
    [int SampleIndex, ][int Offset]
);
```

```
void Object.GetDimensions(
    [in] UINT MipLevel,
    [out] UINT Width, UINT Height,
    [out] UINT NumberOfLevels
);
```

```
float4 PS(VS_OUTPUT input) : SV_Target
{
    int3 uvm = int3(input.position.xy, 0); //(u, v, mipmap level)
    float3 normal = gtxtNormal.Load(uvm).xyz; //gtxtNormal[uvm]
    float3 position = gtxtPosition.Load(uvm).xyz;
    float3 diffuseAlbedo = gtxtDiffuseAlbedo.Load(uvm).xyz;
    float4 specular = gtxtSpecularAlbedo.Load(uvm);
    float4 clllumination = Lighting(position, normal, diffuseAlbedo, specular.xyz, specular.w*255.0f);
    return(clllumination);
}
```

지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

– 법선 벡터의 인코딩(Encoding Normal Vector)

▪ 법선 벡터 $\mathbf{n} = (x, y, z)$ 는 구 표면의 점으로 취급할 수 있음

▪ XYZ

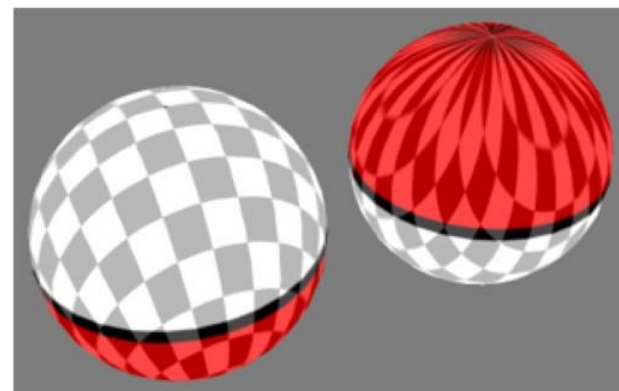
$$\mathbf{n} = (x, y, z) \rightarrow \mathbf{c} = 0.5 * \mathbf{n} + 0.5 \quad \mathbf{c} = (r, g, b) \rightarrow \mathbf{n} = 2.0 * \mathbf{c} - 1.0$$

▪ LAEAP(Lambert Azimuthal Equal-Area Projection)

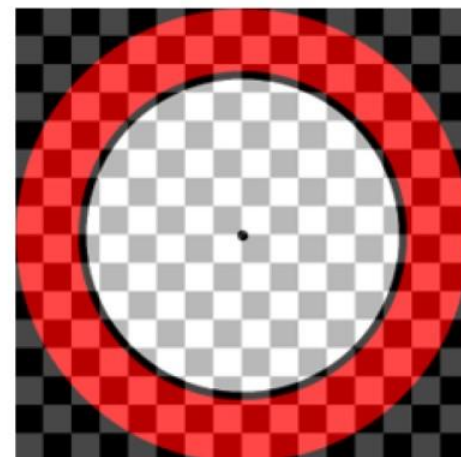
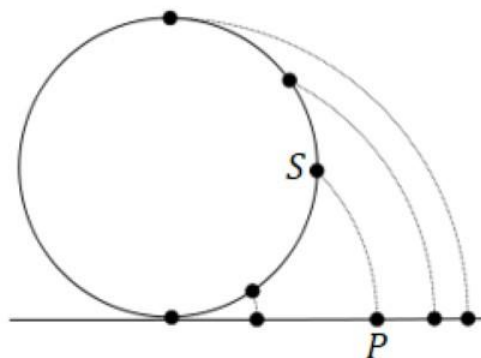
구의 방정식: $x^2 + y^2 + z^2 = 1$

$$(X, Y) = \left(\sqrt{\frac{2}{1-z}} x, \sqrt{\frac{2}{1-z}} y \right)$$

$$(x, y, z) = \left(\sqrt{1 - \frac{X^2+Y^2}{4}} X, \sqrt{1 - \frac{X^2+Y^2}{4}} Y, -1 + \frac{X^2+Y^2}{2} \right)$$



$$\begin{aligned} x^2 + y^2 + z^2 &= 1 \\ z^2 &= 1 - (x^2 + y^2) \\ z &= \pm \sqrt{1 - (x^2 + y^2)} \end{aligned}$$



Compact Normal Storage for small G-Buffers

Survey of Efficient Representations for Independent Unit Vectors

Octahedron normal vector encoding

지연 셰이딩(Deferred Shading)

- 지연 렌더링(Deferred Rendering)

- 법선 벡터의 인코딩(Encoding Normal Vector)

- 구면 좌표계(Spherical Coordinates)

- 단위벡터의 길이를 알고 있으므로 2개의 각도만 저장
 - 일반적인 법선벡터들의 인코딩에 적합

$$\mathbf{f} = \text{float2}(\text{atan2}(y, x) * \frac{1}{\pi}, z)$$

$$\text{angle} = 2.0 * \mathbf{f}.x - 1.0 \quad s\theta = \sin(\text{angle}.x * \pi) \quad c\theta = \cos(\text{angle}.x * \pi)$$

$$s\varphi = \text{sqrt}(1.0 - \text{angle}.y * \text{angle}.y) \quad c\varphi = \text{angle}.y$$

$$\mathbf{n} = (c\theta * s\varphi, s\theta * s\varphi, c\varphi)$$

```
#define PI 3.1415926536f
```

```
float4 Encode(float3 n)
```

```
{
```

```
    return(float4((float2(atan2(n.y, n.x) / PI, n.z) + 1.0) * 0.5, 0, 0));
```

```
}
```

```
float3 Decode(float2 enc)
```

```
{
```

```
    float2 ang = enc * 2.0 - 1.0;
```

```
    float2 scth;
```

```
    sincos(ang.x * PI, scth.x, scth.y);
```

```
    float2 scphi = float2(sqrt(1.0 - ang.y * ang.y), ang.y);
```

```
    return(float3(scth.y * scphi.x, scth.x * scphi.x, scphi.y));
```

```
}
```

지연 셰이딩(Deferred Shading)

- 지연 렌더링(Deferred Rendering)

- 법선 벡터의 인코딩(Encoding Normal Vector)

- Store X&Y, Reconstruct Z (Killzone 2)

간단하게 인코딩/디코딩

```
float2 Encode(float3 n)
{
    return(float2(n.xy * 0.5 + 0.5));
}
```

```
float3 Decode(float2 enc)
{
    float2 dec = enc * 2 - 1;
    return(float3(dec, sqrt(1 - dot(dec, dec))));
}
```

- Spheremap Transform (Cry Engine 3)

구를 원으로 매핑

```
float2 Encode(float3 n)
{
    float2 enc = normalize(n.xy) * (sqrt(-n.z * 0.5 + 0.5));
    return(enc * 0.5 + 0.5);
}
```

```
float3 Decode(float2 enc)
{
    float4 nn = float4(enc, 0, 0) * float4(2.0, 2.0, 0, 0) + float4(-1.0, -1.0, -1.0, -1.0);
    float f = dot(nn.xyz, -nn.xyw);
    nn.xy *= sqrt(f);
    return(float3(nn.xy, f) * 2.0 + float3(0, 0, -1.0));
}
```


지연 셰이딩(Deferred Shading)

- 지연 렌더링(Deferred Rendering)

- 법선 벡터의 인코딩(Encoding Normal Vector)

- Lambert Azimuthal Equal-Area Projection

```
float2 Encode(float3 n)
{
    return(n.xy / sqrt(8.0 * n.z + 8.0) + 0.5);
}
```

```
float3 Decode(float2 enc)
{
    float2 fenc = enc * 4.0 - 2.0;
    float f = dot(fenc, fenc);
    return(float3(fenc * sqrt(1 - (f / 4.0)), 1.0 - (f / 2.0)));
}
```

- Octahedron-Normal Vectors

```
float2 Encode(float3 n)
{
    n /= (abs(n.x) + abs(n.y) + abs(n.z));
    n.xy = (n.z >= 0.0) ? n.xy : OctWrap(n.xy);
    n.xy = n.xy * 0.5 + 0.5;
    return(n.xy);
}
```

```
float2 OctWrap(float2 v)
{
    return((1.0 - abs(v.yx)) * (v.xy >= 0.0) ? 1.0 : -1.0);
}
```

```
float3 Decode(float2 f)
{
    f = f * 2.0 - 1.0;
    float3 n = float3(f.x, f.y, 1.0 - abs(f.x) - abs(f.y));
    float t = saturate(-n.z);
    n.xy += (n.xy >= 0.0) ? -t : t;
    return(normalize(n));
}
```

지연 셰이딩(Deferred Shading)

- 지연 렌더링(Deferred Rendering)

- 법선 벡터의 인코딩(Encoding Normal Vector)

- Crytek Spheremap Transform

```
G.xy = normalize(normal.xy) * (sqrt(normal.z * 0.5f + 0.5f));
```

```
G.xy = normalize(normal.xy) * (sqrt(-normal.z * 0.5f + 0.5f));
```

```
G.xy = normalize(normal.xy) / sqrt(8.0f * normal.z + 8.0f) + 0.5f;
```

```
uint nPackedG = (f32tof16(G.y) << 16) | f32tof16(G.x);
```

```
float2 G = float2(f16tof32(nPackedG), f16tof32(nPackedG >> 16));
```

```
normal.z = length(G.xy) * 2.0f - 1.0f;
```

```
normal.xy = normalize(G.xy) * sqrt(1.0f - normal.z * normal.z);
```

$$(X, Y) = \left(\frac{x}{\sqrt{x^2+y^2}}, \frac{y}{\sqrt{x^2+y^2}} \right) \sqrt{z * 0.5 + 0.5} = \left(\frac{x\sqrt{z*0.5+0.5}}{\sqrt{x^2+y^2}}, \frac{y\sqrt{z*0.5+0.5}}{\sqrt{x^2+y^2}} \right)$$

$$X^2 + Y^2 = \left(\frac{x\sqrt{z*0.5+0.5}}{\sqrt{x^2+y^2}} \right)^2 + \left(\frac{y\sqrt{z*0.5+0.5}}{\sqrt{x^2+y^2}} \right)^2 = \frac{(x^2+y^2)}{(x^2+y^2)} (z * 0.5 + 0.5) = z * 0.5 + 0.5 = \frac{(z+1)}{2}$$

$$z = 2(X^2 + Y^2) - 1 = 2 \frac{(z+1)}{2} - 1 = z$$

$$(x, y) = \left(\frac{X}{\sqrt{X^2+Y^2}}, \frac{Y}{\sqrt{X^2+Y^2}} \right) \sqrt{1.0 - z * z} = \frac{1}{\sqrt{X^2+Y^2}} \left(\frac{x\sqrt{z*0.5+0.5}}{\sqrt{x^2+y^2}}, \frac{y\sqrt{z*0.5+0.5}}{\sqrt{x^2+y^2}} \right) \sqrt{1.0 - z * z}$$

$$= \frac{1}{\sqrt{\frac{(z+1)}{2}}} \sqrt{\frac{(z+1)}{2}} \left(\frac{x}{\sqrt{x^2+y^2}}, \frac{y}{\sqrt{x^2+y^2}} \right) \sqrt{1.0 - z^2} = \frac{1}{\sqrt{\frac{(z+1)}{2}}} \sqrt{\frac{(z+1)}{2}} \left(\frac{x}{\sqrt{x^2+y^2}}, \frac{y}{\sqrt{x^2+y^2}} \right) \sqrt{x^2 + y^2} = (x, y)$$

$$\text{length}(G.xy) = \sqrt{X^2 + Y^2}$$

$$\begin{aligned} x^2 + y^2 + z^2 &= 1 \\ z^2 &= 1 - (x^2 + y^2) \\ z &= \pm \sqrt{1 - (x^2 + y^2)} \\ 1 - z^2 &= (x^2 + y^2) \end{aligned}$$

지연 셰이딩(Deferred Shading)

- 지연 렌더링(Deferred Rendering)

- 최적화된 GBuffer 생성 단계

```
VS_GB_OUTPUT VSOptimizedDeferredLighting(in VS_GB_INPUT input)
{
    VS_GB_OUTPUT output;

    matrix mtxWorldView = mul(gmtxWorld, gmtxView);
    output.position = mul(input.position, mtxWorldView);
    output.position = mul(output.position, gmtxProjection);

    output.normal = normalize(mul(input.normal, (float3x3)mtxWorldView));
    output.tangent = normalize(mul(input.tangent.xyz, (float3x3)mtxWorldView));
    output.bitangent = normalize(cross(output.normal, output.tangent));

    output.uv = input.uv;

    return(output);
}
```

```
struct VS_GB_INPUT
{
    float4 position : POSITION;
    float2 uv : TEXCOORD;
    float3 normal : NORMAL;
    float3 tangent : TANGENT;
};
```

```
struct VS_GB_OUTPUT
{
    float4 position : SV_Position;
    float2 uv : TEXCOORD;
    float3 normal : NORMAL;
    float3 tangent : TANGENT;
    float3 bitangent : BITANGENT;
};
```


지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

- 최적화된 GBuffer 생성 단계

```
PS_GB_OUTPUT PSOptimizedDeferredLighting(in VS_GB_OUTPUT input)
```

```
{  
    PS_GB_OUTPUT output;  
    float3 T = normalize(input.tangent);  
    float3 B = normalize(input.bitangent);  
    float3 N = normalize(input.normal);  
    float3x3 TBN = float3x3(T, B, N);  
    float3 normal = normalize(gtxtNormalMap.Sample(gssNormalMap, input.uv).rgb * 2.0f - 1.0f);  
    normal = normalize(mul(normal, TBN));  
    output.normal = normal.xy * (sqrt(-normal.z * 0.5f + 0.5f)); //Encode Azimuthal  
    float3 diffuseAlbedo = gtxtDiffuseMap.Sample(gssDiffuseMap, input.uv).rgb;  
    output.diffuseAlbedo = float4(diffuseAlbedo, 1.0f);  
    output.specularAlbedo = gcMaterialSpecular; //power : (0~255) → (0 ~ 1.0)  
  
    return(output);  
}
```

```
Texture2D gtxtDiffuseMap : register(t0);  
SamplerState gssDiffuseMap : register(s0);  
  
Texture2D gtxtNormalMap : register(t1);  
SamplerState gssNormalMap : register(s1);
```

```
normal.xy / sqrt(8.0f * normal.z + 8.0f) + 0.5f;
```

```
cbuffer cbMaterial : register(b1)  
{  
    float4 gcMaterialDiffuse;  
    float4 gcMaterialAmbient;  
    float4 gcMaterialSpecular;  
    float4 gcMaterialEmissive;  
};
```

```
struct PS_GB_OUTPUT  
{  
    float2 normal : SV_Target0;  
    float4 diffuseAlbedo : SV_Target1;  
    float4 specularAlbedo : SV_Target2;  
};
```

```
float4 Lighting(float3 vPosition, float3 vNormal);
```

지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

- 최적화된 GBuffer를 사용한 조명 계산 단계

```
VS_OUTPUT VS(in VS_INPUT input)
```

```
{  
    VS_OUTPUT output;  
    output.position = input.position; //투영 좌표계  
    float3 positionCS = mul(input.position, gmtxInvProjection).xyz;  
    output.positionCS = float3(positionCS.xy / positionCS.z, 1.0f);  
    return(output);  
}
```

```
Texture2D gtxtNormal : register(t0);  
Texture2D gtxtDiffuseAlbedo : register(t1);  
Texture2D gtxtSpecularAlbedo : register(t2);  
Texture2D gtxtDepth : register(t3);
```

```
struct VS_INPUT  
{  
    float4 position : POSITION;  
};
```

```
struct VS_OUTPUT  
{  
    float4 position : SV_Position;  
    float3 positionCS : POSITION;  
};
```

```
cbuffer cbCamera : register(b1)  
{  
    ...  
    matrix gmtxInvProjection;  
};
```

```
float4 PS(VS_OUTPUT input, uint coverage : SV_Coverage) : SV_Target
```

```
{  
    int3 uvm = int3(input.position.xy, 0); //(u, v, mipmap level)  
    float3 position = PositionFromDepth(gtxtDepth.Load(uvm).x, input.positionCS);  
    float3 normal = SphereMapDecode(gtxtNormal.Load(uvm).xy);  
    float3 diffuseAlbedo = gtxtDiffuseAlbedo.Load(uvm).xyz;  
    float4 specular = gtxtSpecularAlbedo.Load(uvm);  
    float4 cillumination = Lighting(position, normal, diffuseAlbedo, specular.xyz, specular.w*255.0f);  
    return(cillumination);  
}
```


지연 셰이딩(Deferred Shading)

• 지연 렌더링(Deferred Rendering)

- 최적화된 GBuffer를 사용한 조명 계산 단계

```
float3 PositionFromDepth(float depth, float3 positionCS)
{
    float fz = gmtxProjection[3][2] / (depth - gmtxProjection[2][2]);
    return(positionCS * fz);
}
```

$$(x, y, z, 1) * PM = (x * xScale, y * yScale, \frac{(z - z_n) * z_f}{(z_f - z_n)}, z)$$

$$d = \frac{(z - z_n) * z_f}{(z_f - z_n)} * \frac{1}{z}$$

$$d * z = \frac{(z - z_n) * z_f}{(z_f - z_n)}$$

$$d * z * (z_f - z_n) = (z - z_n) * z_f$$

$$d * z * (z_f - z_n) - z * z_f = -z_n z_f$$

$$z * \{d * (z_f - z_n) - z_f\} = -z_n z_f$$

$$z = \frac{-z_n z_f}{d * (z_f - z_n) - z_f} = \frac{\frac{-z_n * z_f}{(z_f - z_n)}}{d - \frac{z_f}{z_f - z_n}}$$

$$\begin{pmatrix} xScale & 0 & 0 & 0 \\ 0 & yScale & 0 & 0 \\ 0 & 0 & \frac{z_f}{(z_f - z_n)} & 1 \\ 0 & 0 & \frac{-z_n * z_f}{(z_f - z_n)} & 0 \end{pmatrix}$$

```
float3 PositionFromDepth(float2 uv)
{
    float z = gtxDepth.Sample(gssDefault, uv);
    float x = uv.x * 2.0f - 1.0f;
    float y = uv.y * 2.0f - 1.0f;
    float4 posPS = float4(x, y, z, 1.0f);
    float4 posCS = mul(posPS, gmtxInvProjection);
    return(posCS.xyz / posCS.w);
}
```

지연 셰이딩(Deferred Shading)

- 지연 렌더링(Deferred Rendering)

- 최적화된 GBuffer를 사용한 조명 계산 단계

```
float3 SphereMapDecode(float2 enc)
{
    float4 nn = float4(enc, 1.0f, -1.0f);
    float l = dot(nn.xyz, -nn.xyw);
    nn.z = l;
    nn.xy *= sqrt(l);
    return(nn.xyz * 2.0f + float3(0.0f, 0.0f, -1.0f));
}
```

```
output.normal = normal.xy * (sqrt(-normal.z * 0.5f + 0.5f));
```

```
float2 Encode(float3 n)
{
    float2 e = normalize(n.xy) * sqrt(-n.z * 0.5f + 0.5f);
    return(e * 0.5f + 0.5f);
}
```

```
float2 Encode(float3 n)
{
    return(n.xy / sqrt(8.0f * n.z + 8.0f) + 0.5f);
}
```

```
float3 Decode(float4 e)
{
    float4 nn = e * float4(2, 2, 0, 0) + float4(-1, -1, 1, -1);
    nn.z = dot(nn.xyz, -nn.xyw);
    nn.xy *= sqrt(nn.z);
    return(nn.xyz * 2 + float3(0, 0, -1));
}
```

```
float3 Decode(float2 e)
{
    float2 fe = (e * 4.0f) - 2.0f;
    float f = dot(fe, fe);
    float3 n;
    n.xy = fe * sqrt(1.0f - (f / 4.0f));
    n.z = 1.0f - (f / 2.0f);
    return(n);
}
```