II. Direct3D Foundations
8. Lighting

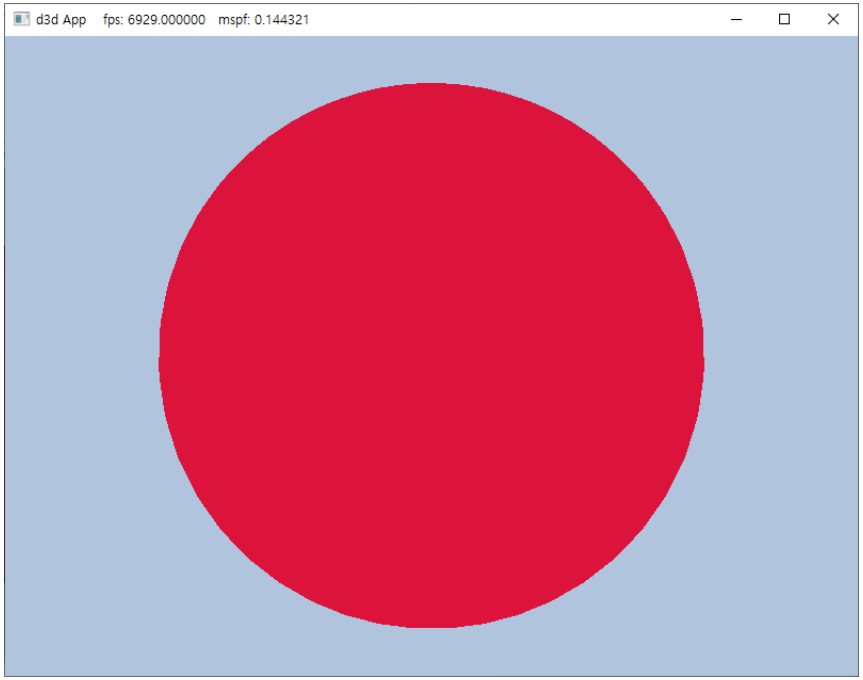
Game Graphic Programming

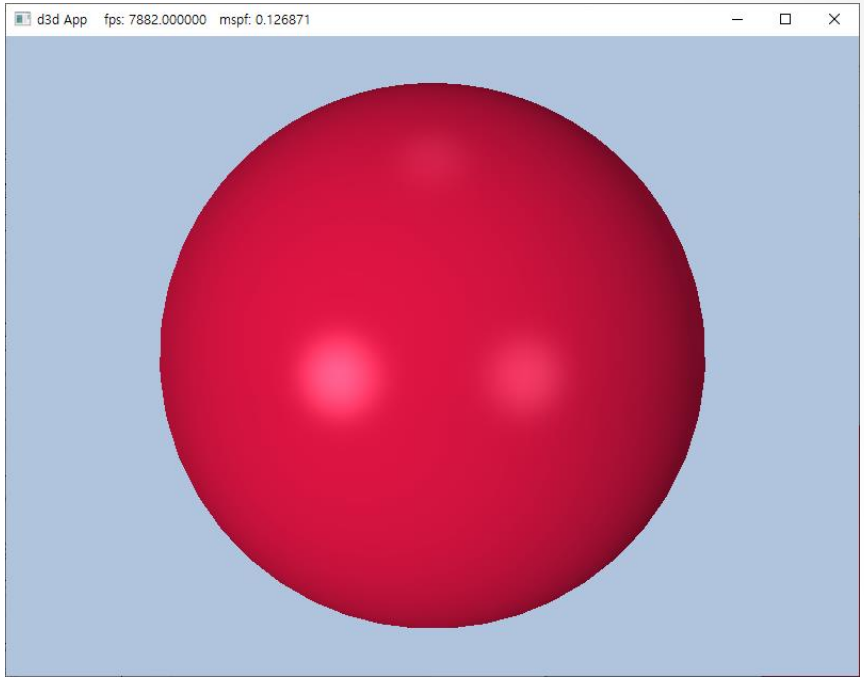
Kyung Hee University

Software Convergence

Prof. Daeho Lee

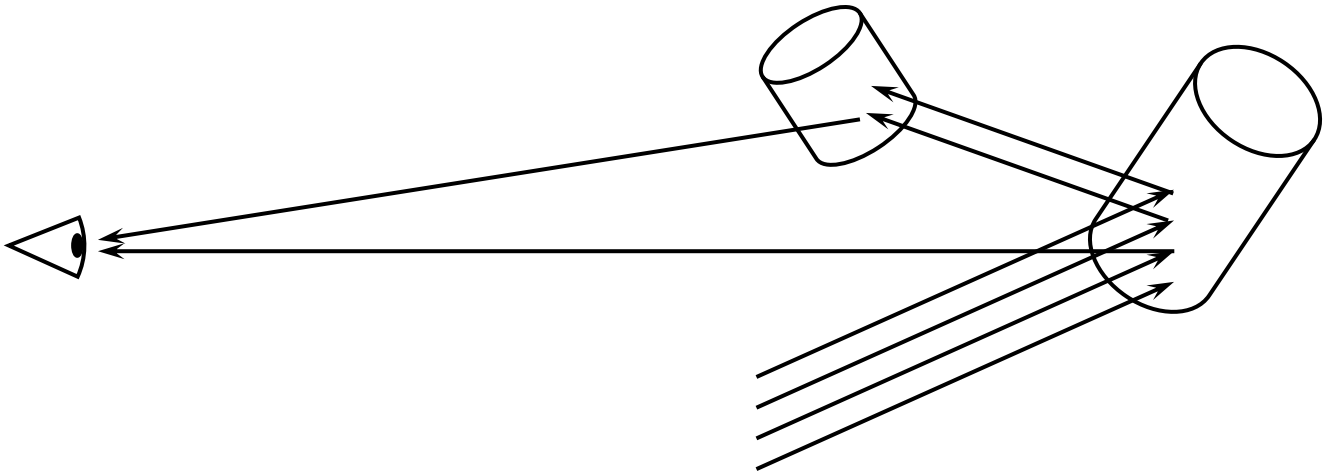
Lighting


d3d App fps: 6929.000000 mspf: 0.144321


d3d App fps: 7882.000000 mspf: 0.126871

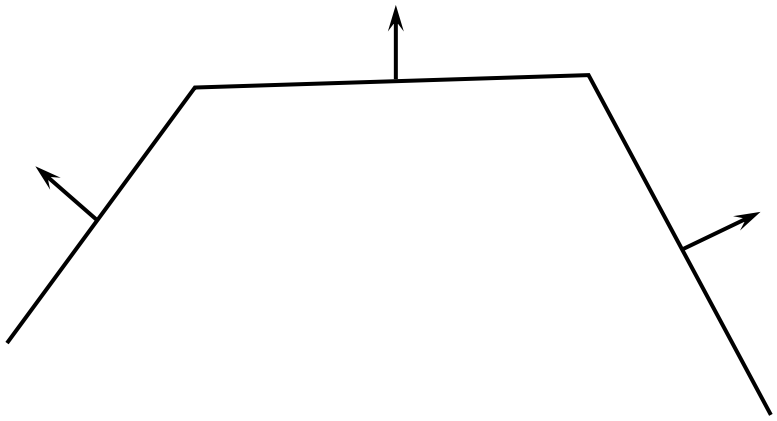
Lighting & Material Interaction (1)

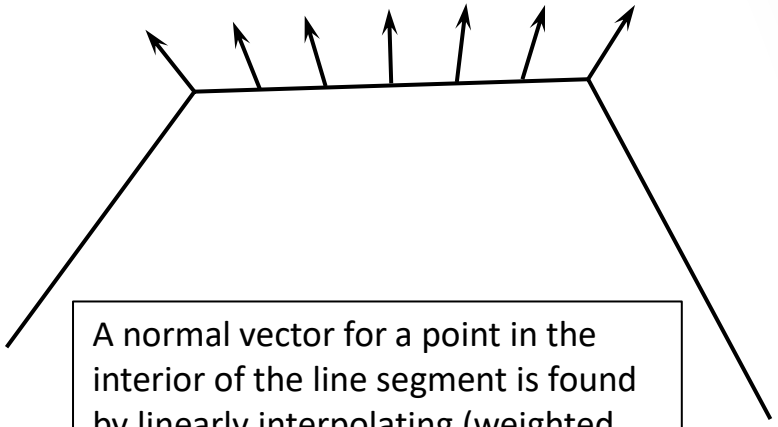
- When using lighting, we no longer specify vertex colors directly; rather, we specify materials and lights, and then apply a lighting equation, which computes the vertex colors for us based on light/material interaction.
  - Materials can be thought of as the properties that determine how light interacts with the surface of an object.
    - Examples of such properties are the color of light the surface reflects and absorbs, the index of refraction of the material under the surface, how smooth the surface is, and how transparent the surface is. By specifying material properties we can model different kinds of real-world surfaces like wood, stone, glass, metals, and water.



Normal Vectors

- A **face normal** is a unit vector that describes the direction a polygon is facing.
- A **vertex normal** at a vertex of a polyhedron is a directional vector associated with a vertex, intended as a replacement to the true geometric normal of the surface.





A normal vector for a point in the interior of the line segment is found by linearly interpolating (weighted average) between the vertex normal.

Computing Normal Vectors (1)

- To find the face normal of a triangle $\Delta\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ we first compute two vectors that lie on the triangle's edges:
  - $\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$
  - $\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$
  - The face normal

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|}$$

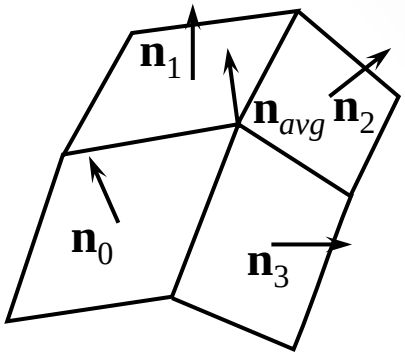
```
XMVECTOR ComputeNormal(FXMVECTOR p0, FXMVECTOR p1, FXMVECTOR p2) {
  XMVECTOR u = p1 - p0;
  XMVECTOR v = p2 - p0;

  return XMVector3Normalize(XMVector3Cross(u,v));
}
```

Computing Normal Vectors (2)

- Vertex normal averaging

$$\mathbf{n}_{avg} = \frac{\mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3}{\left\| \mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 \right\|}$$




```
for(UINT i = 0; i < mNumTriangles; ++i) {
  UINT i0 = mIndices[i*3+0];  UINT i1 = mIndices[i*3+1];
  UINT i2 = mIndices[i*3+2];

  Vertex v0 = mVertices[i0];  Vertex v1 = mVertices[i1];
  Vertex v2 = mVertices[i2];

  Vector3 e0 = v1.pos - v0.pos;  Vector3 e1 = v2.pos - v0.pos;
  Vector3 faceNormal = Cross(e0, e1);

  mVertices[i0].normal += faceNormal;
  mVertices[i1].normal += faceNormal;
  mVertices[i2].normal += faceNormal;
}
for(UINT i = 0; i < mNumVertices; ++i)
  mVertices[i].normal = Normalize(&mVertices[i].normal))
```

Transforming Normal Vectors

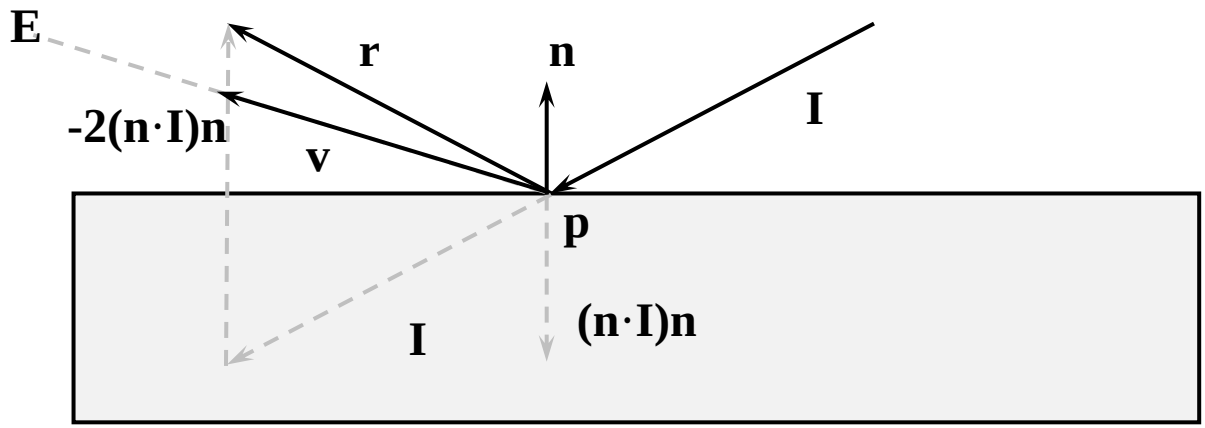
- A tangent vector: $\mathbf{u} = \mathbf{v}_1 - \mathbf{v}_0$
- A normal vector: $\mathbf{n}$
- A non-uniform scaling transformation: $\mathbf{A}$
  - Transformed tangent vector: $\mathbf{uA} = \mathbf{v}_1\mathbf{A} - \mathbf{v}_0\mathbf{A}$
  - $\mathbf{uA}$ does not remain orthogonal to $\mathbf{nA}$.
  - Find $\mathbf{B}$, where $\mathbf{nB}$ is orthogonal to $\mathbf{uA}$.
    - $\mathbf{uA} \cdot \mathbf{nB} = 0$
    - $\mathbf{B}$ is the inverse transpose of $\mathbf{A}$.

$$\mathbf{u} \cdot \mathbf{n} = 0$$
$$\mathbf{u}\mathbf{n}^T = 0$$
$$\mathbf{u}(\mathbf{A}\mathbf{A}^{-1})\mathbf{n}^T = 0$$
$$(\mathbf{u}\mathbf{A})(\mathbf{A}^{-1}\mathbf{n}^T) = 0$$
$$(\mathbf{u}\mathbf{A})(((\mathbf{A}^{-1}\mathbf{n}^T)^T)^T = 0$$
$$(\mathbf{u}\mathbf{A})(\mathbf{n}(\mathbf{A}^{-1})^T)^T = 0$$
$$(\mathbf{u}\mathbf{A}) \cdot \mathbf{n}(\mathbf{A}^{-1})^T = 0$$
$$\mathbf{B} = (\mathbf{A}^{-1})^T$$

Reflection & View Vectors

- Reflection vector

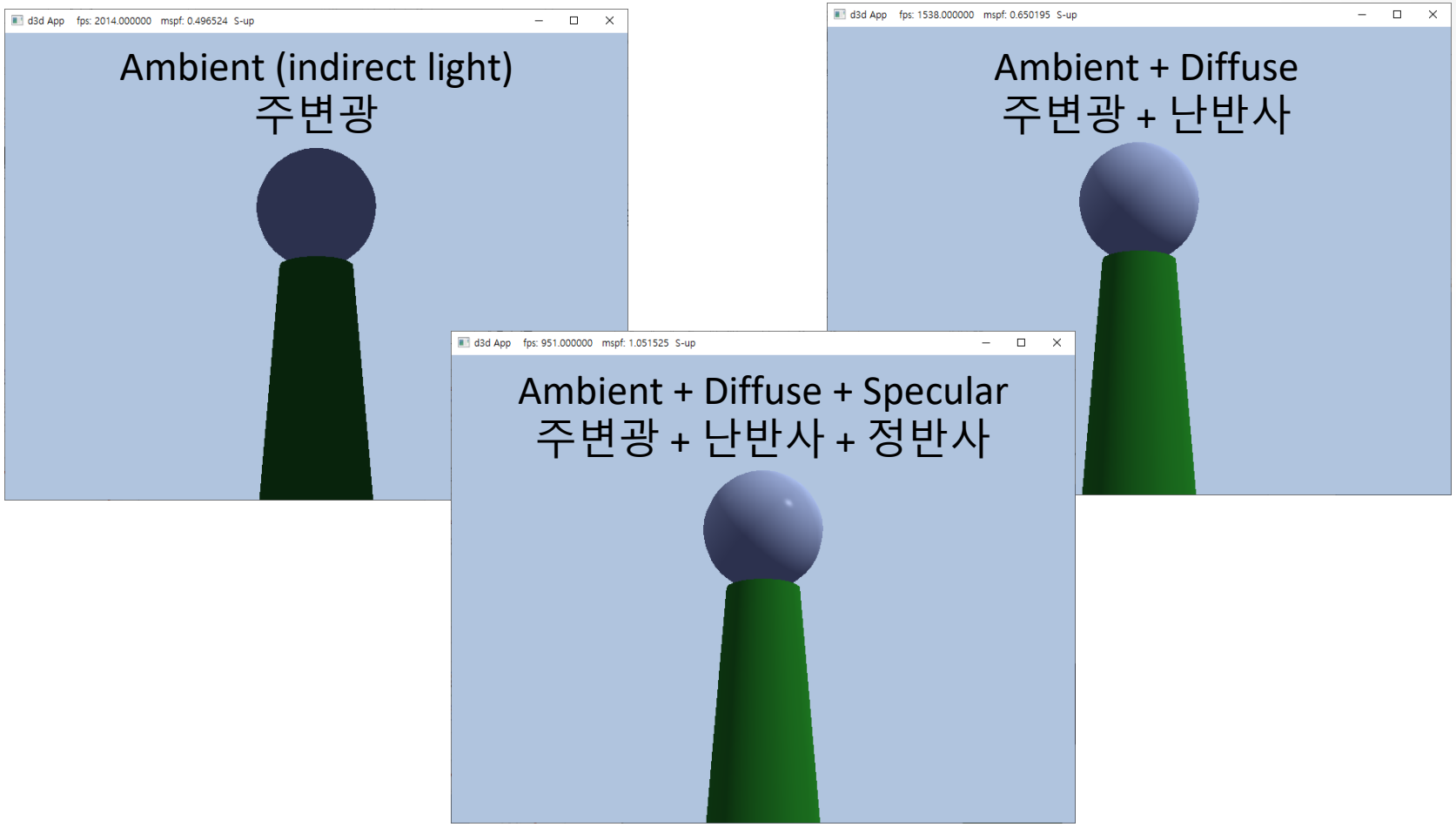
$$\mathbf{r} = \mathbf{I} - 2(\mathbf{n} \cdot \mathbf{I})\mathbf{n}$$



- View vector (to-eye vector)

$$\mathbf{v} = \text{normalize}(\mathbf{E} - \mathbf{p})$$

Ambient, Diffuse, Specular

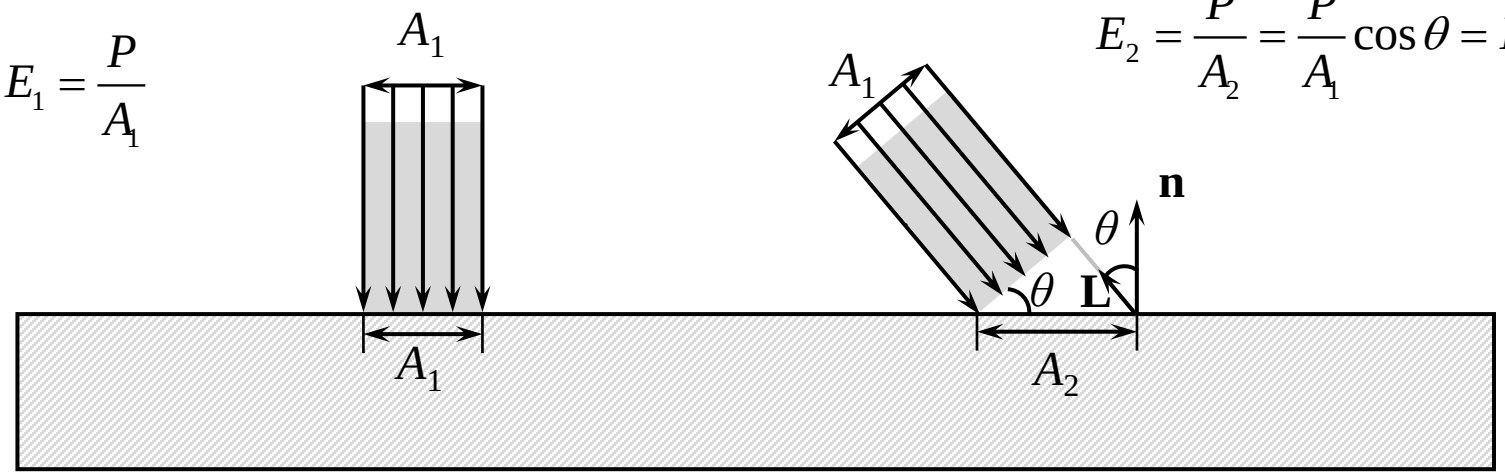


Lambert's Cosine Law (1)

- We can think of light as a collection of photons traveling through space in a certain direction.
  - Each photon carries some (light) energy.
  - The amount of (light) energy emitted per second is called radiant flux (복사속, 복사 흐름).
  - The density of radiant flux per area (called irradiance, 복사 조도) is important because that will determine how much light an area on a surface receives (and thus how bright it will appear to the eye).
- Irradiance: E

$$E_1 = \frac{P}{A_1}$$

$$E_2 = \frac{P}{A_2} = \frac{P}{A_1}\cos\theta = E_1(\mathbf{n}\cdot\mathbf{L})$$



Lambert's Cosine Law (2)

- Lambert's cosine law
  - The irradiance striking area (A_2) is equal to the irradiance at the area (A_1) perpendicular to the light direction scaled by $\mathbf{n}\cdot\mathbf{L} = \cos\theta$.

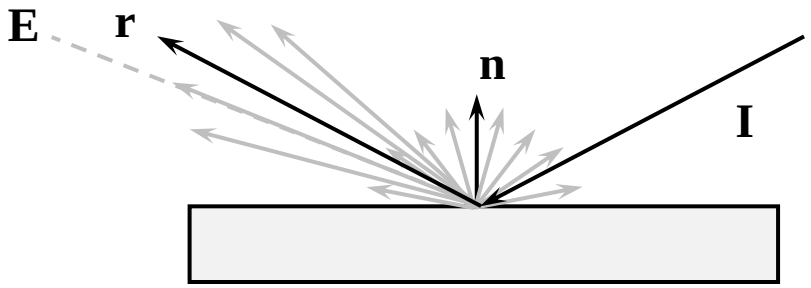
$$f(\theta) = \max(\cos\theta, 0) = \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)$$

Diffuse Lighting

- Diffuse reflection
  - When light strikes a point on the surface, some of the light enters the interior of the object and interacts with the matter near the surface.
  - The light will bounce around in the interior, where some of it will be absorbed, and **the remaining part will be scattered out of the surface in every direction**.
  - Light color + diffuse albedo color
  - Diffuse albedo
    - The amount of incoming light that the surface reflects due to diffuse reflectance (By energy conservation, the amount not reflected is absorbed by the material).
  - The amount of light reflected off the points ($\mathbf{c}_d$)
    - Componentwise multiplication of incoming light ($\mathbf{B}_L$) and diffuse albedo ($\mathbf{m}_d$)

$$\mathbf{c}_d = \mathbf{B}_L \otimes \mathbf{m}_d$$

$$\mathbf{c}_d = \max(\mathbf{n} \cdot \mathbf{L}, 0) \cdot \mathbf{B}_L \otimes \mathbf{m}_d$$



Ambient Lighting

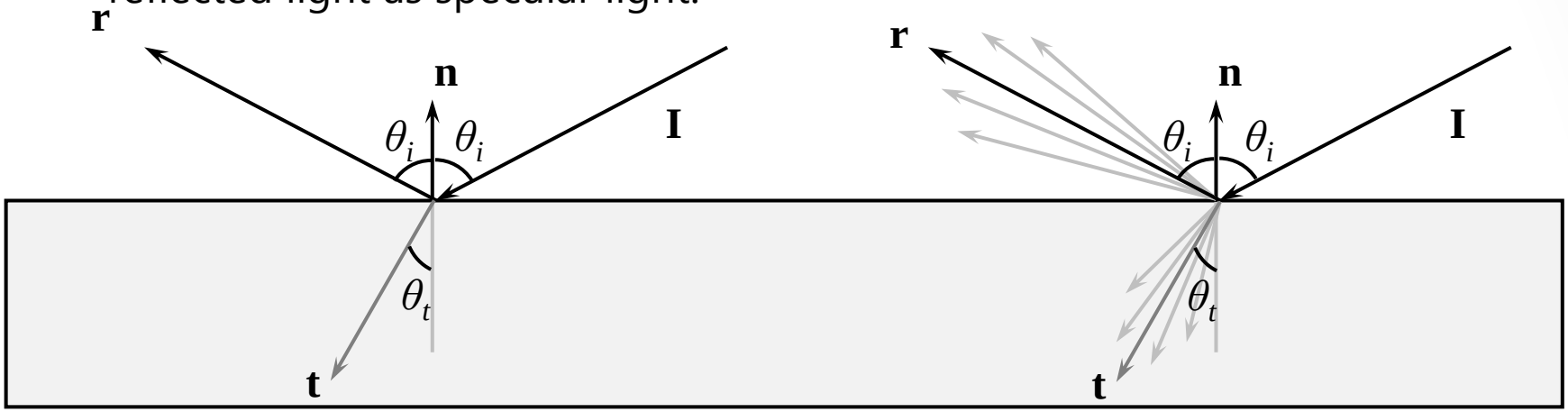
- Only one side of the teapot is in the direct line of site of light source; nevertheless, the backside of the teapot would not be completely black. This is because some light scatters off the walls or other objects in the room and eventually strikes the backside of the teapot.
- To sort of hack this indirect light, we introduce an ambient term to the lighting equation:

$$\mathbf{c}_a = \mathbf{A}_L \otimes \mathbf{m}_d$$

  - $\mathbf{A}_L$ specifies the total amount of indirect (ambient) light a surface receives, which may be different than the light emitted from the source due to the absorption that occurred when the light bounced off other surfaces.

Specular Lighting (1)

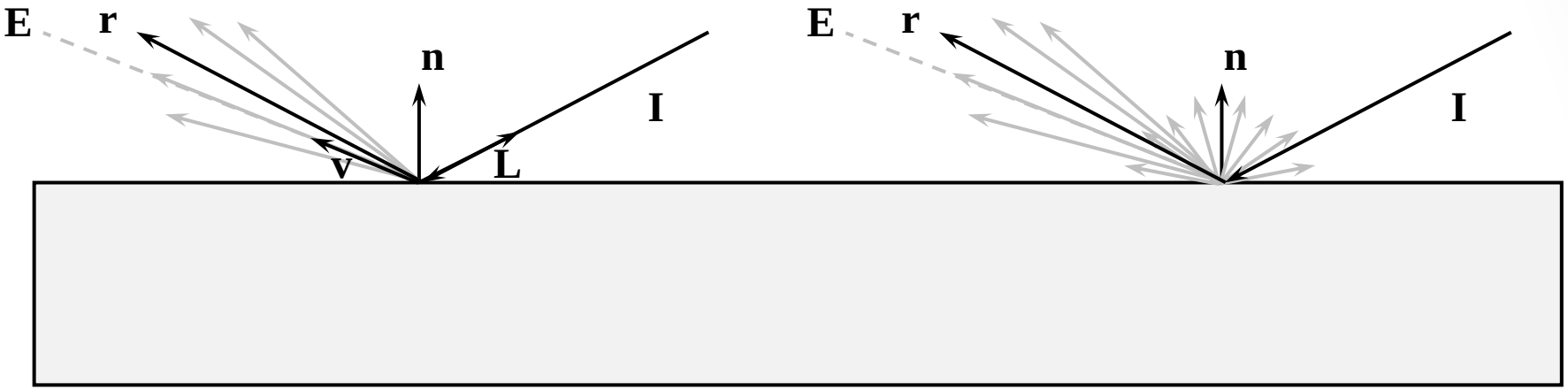
- A kind of reflection happens due to the Fresnel effect, which is a physical phenomenon.
  - When light reaches the interface between two media with different indices of refraction, some of the light is reflected, and the remaining light is refracted.
  - The index of refraction is a physical property of a medium that is the ratio of the speed of light in a vacuum to the speed of light in the given medium.
  - We refer to this light reflection process as specular reflection and the reflected light as specular light.



Most objects are not perfectly flat mirrors but have microscopic roughness. This causes the reflected and refracted light to spread about the reflection and refraction vectors.

Specular Lighting (2)

- For opaque objects, the refracted light enters the medium and undergoes diffuse reflectance.



The reflected light that makes it into the eye is a combination of specular reflection and diffuse reflection.

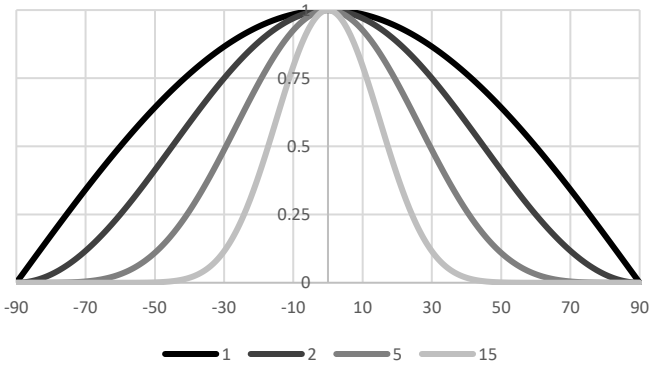
Phong Shader (1)

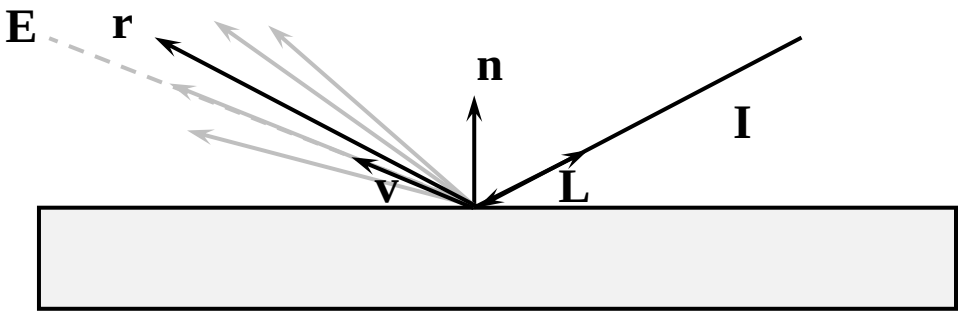
- Phong shader
  - For a perfect mirror, incident light is reflected only in the specular reflection direction.
  - A Shiny surface has a narrow specular reflection range. (shininess: m)
  - The angle between the view direction and the reflection direction: θ
  - Specular reflection: $\cos^m \theta$

$$\cos^m \theta = (\mathbf{v} \cdot \mathbf{r})^m$$

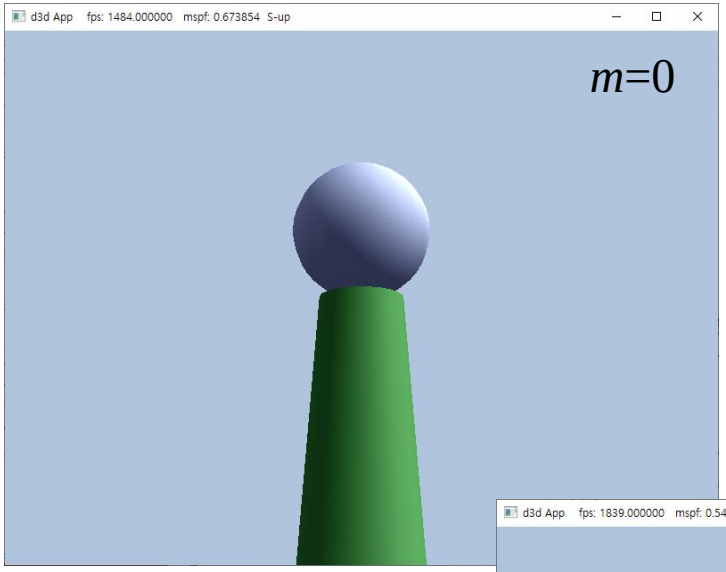
$$\mathbf{r} = \text{reflect}(\mathbf{I}, \mathbf{n})$$

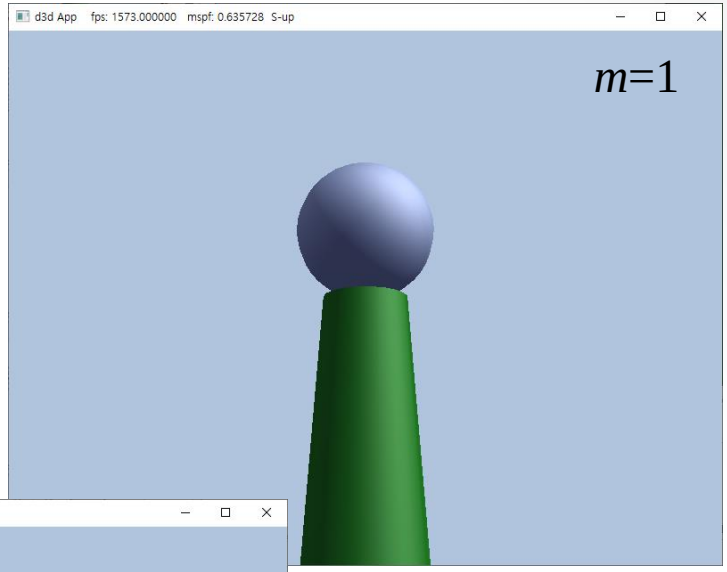
$$\mathbf{c}_s = \max(\mathbf{n} \cdot \mathbf{L}, 0) \cdot \mathbf{B}_L \otimes (\mathbf{v} \cdot \mathbf{r})^m$$

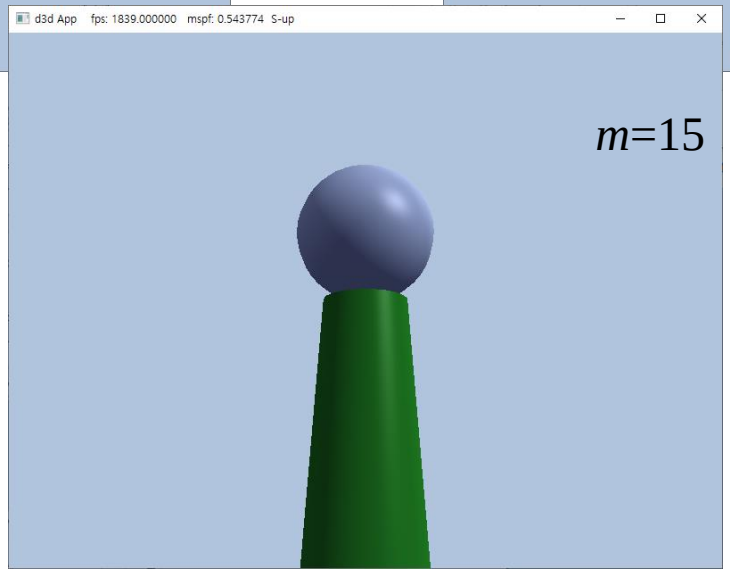




Phong Shader (2)







Fresnel Effect

- The Fresnel effect is the phenomenon where a surface's reflectivity changes based on the viewing angle.
- The Fresnel equations describe how much energy is reflected at a surface boundary.
  - These describe the percentage of incoming light that is reflected, $0 \leq \mathbf{R}_F \leq 1$. By conservation of energy, if $\mathbf{R}_F$ is the amount of reflected light, then $(1 - \mathbf{R}_F)$ is the amount of refracted light.
  - How much light is reflected depends on the medium (some materials will be more reflective than others) and also on the angle θ_i between the normal vector $\mathbf{n}$ and light vector $\mathbf{L}$.
  - Due to their complexity, the full Fresnel equations are not typically used in real-time rendering; instead, the Schlick approximation is used:

$$\mathbf{R}_F(\theta_i) = \mathbf{R}_F(0°) + \left(1 - \mathbf{R}_F(0°)\right)\left(1 - \cos\theta_i\right)^5$$

  - $\mathbf{R}_F(0°)$ is a property of the medium; below are some values for common materials:
    - Water (0.02, 0.02, 0.02), glass (0.08, 0.08, 0.08)
    - Plastic (0.05, 0.05, 0.05)
    - Gold (1.0, 0.71, 0.29), silver (0.95, 0.93, 0.88)
    - Copper (0.95, 0.64, 0.54)

$$\mathbf{R}_F(0°) = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

Two media of refractive indices n_1 and n_2

Fresnel /freɪˈnɛl/

Roughness

- Reflective objects in the real world tend not to be perfect mirrors. Even if an object's surface appears flat, at the microscopic level, we can think of it as having roughness.
  - The micro normal is an additional normal map on top of the regular normal map (macro normal map).
  - As the roughness increases, the directions of the micro normal diverge from the macro-normal, causing the reflected light to spread out into a specular lobe.
  - The shininess of a surface is just the inverse of the roughness.

Blinn-Phong Shader (1)

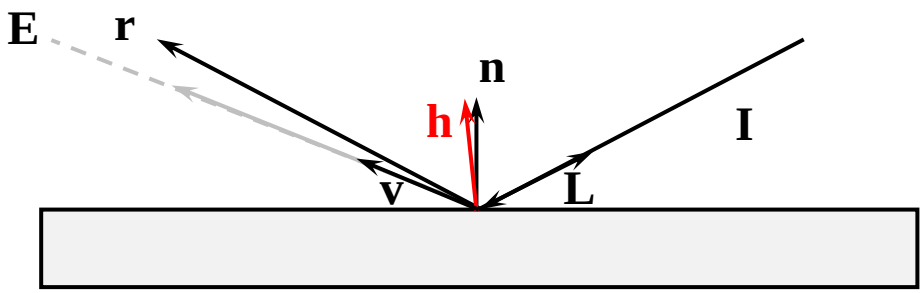
- Blinn-Phong reflection
  - Phong

$$\cos^m \theta = (\mathbf{v} \cdot \mathbf{r})^m$$

  - Blinn-Phong
    - Halfway vector: between $\mathbf{v}$ and $\mathbf{L}$
    - The angle between the normal direction and the halfway direction: θ_h
      - θ_h is likely to be smaller than θ for Phong reflection.

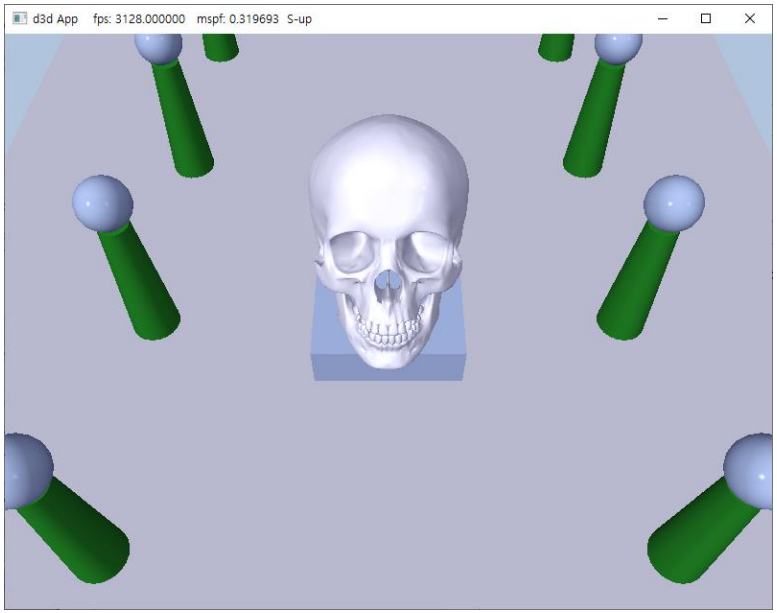
$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{L}}{\|\mathbf{v} + \mathbf{L}\|} \qquad \cos^m \theta_h = (\mathbf{n} \cdot \mathbf{h})^m$$

    - Blinn-Phong is faster than Phong shader.

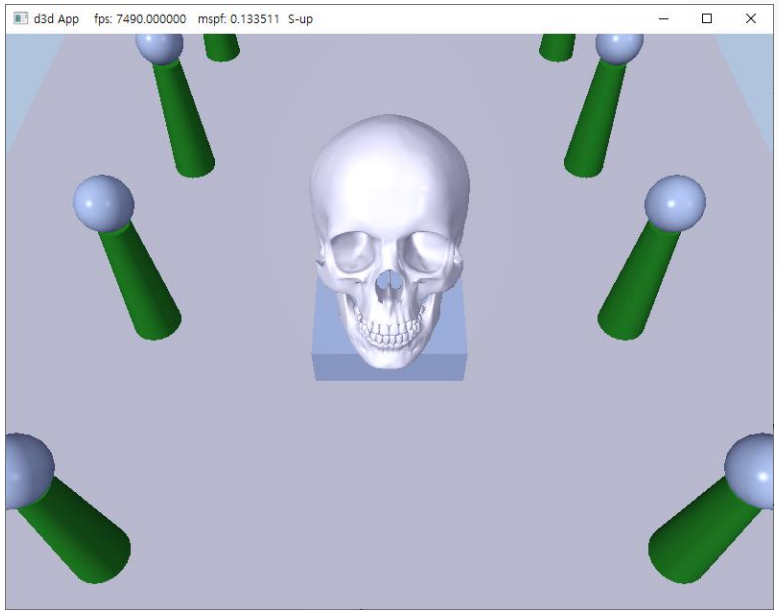


Blinn-Phong Shader (2)

Phong



Blinn-Phong

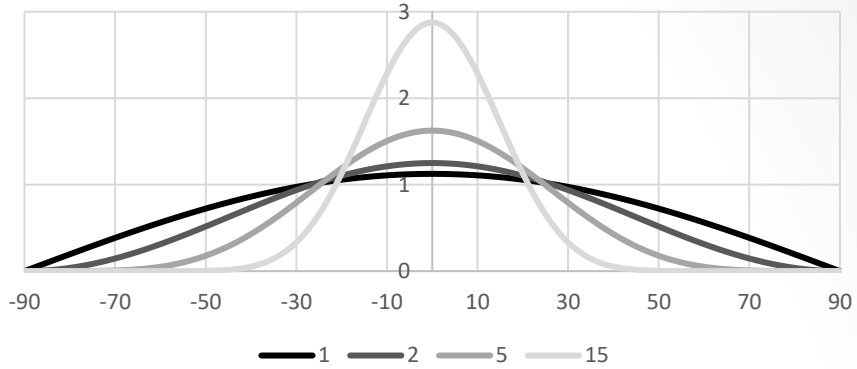


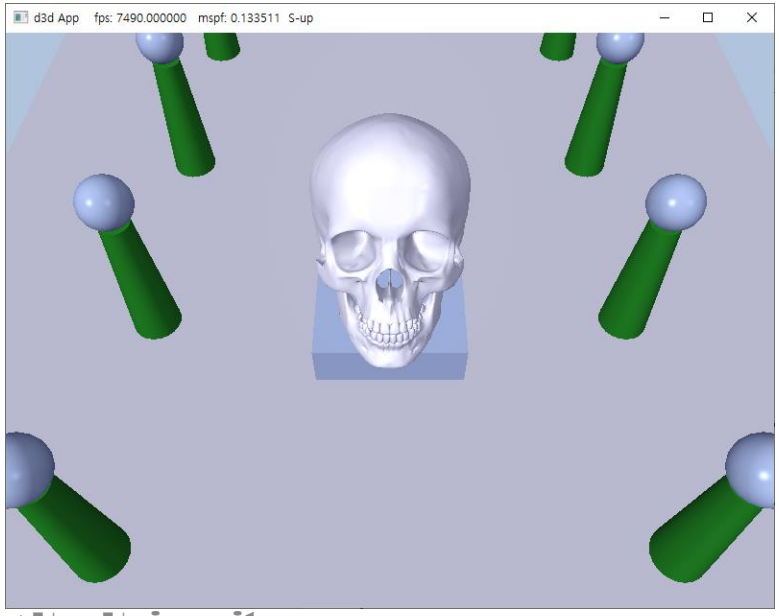
Blinn-Phong Shader (3)

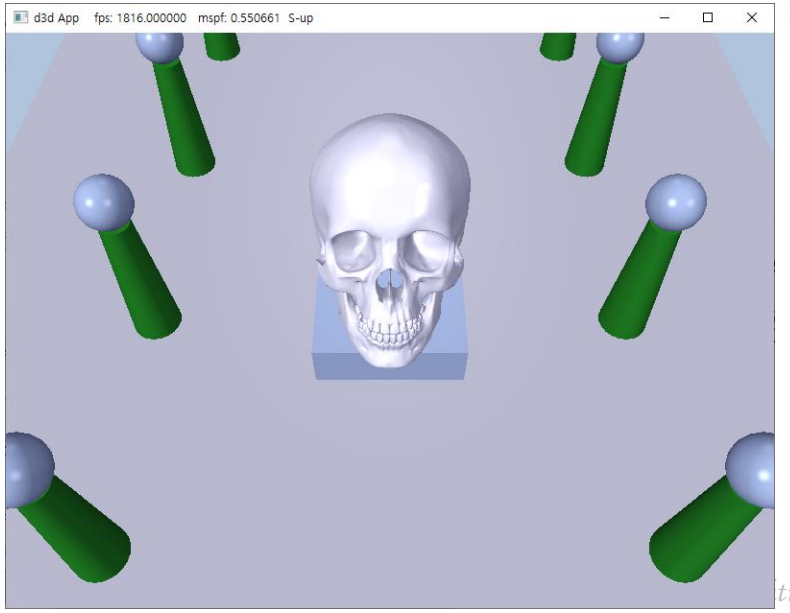
- We can add the normalization factor so that light energy is conserved.

$$\frac{m+8}{8}\cos^m \theta_h$$

$$\mathbf{c}_s = \max(\mathbf{n}\cdot\mathbf{L}, 0)\cdot\mathbf{B}_L \otimes R_F(\alpha_h)\frac{m+8}{8}(\mathbf{n}\cdot\mathbf{h})^m$$







Lighting Model Recap

$$\mathbf{c}_a = \mathbf{A}_L \otimes \mathbf{m}_d$$

$$\mathbf{c}_d = \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)\cdot\mathbf{B}_L \otimes \mathbf{m}_d$$

$$\mathbf{c}_s = \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)\cdot\mathbf{B}_L \otimes R_F(\alpha_h)\frac{m+8}{8}(\mathbf{n}\cdot\mathbf{h})^m$$

$$\begin{aligned}\mathbf{c}_a + \mathbf{c}_d + \mathbf{c}_s &= \mathbf{A}_L \otimes \mathbf{m}_d + \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)\cdot\mathbf{B}_L \otimes \mathbf{m}_d + \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)\cdot\mathbf{B}_L \otimes R_F(\alpha_h)\frac{m+8}{8}(\mathbf{n}\cdot\mathbf{h})^m \\ &= \mathbf{A}_L \otimes \mathbf{m}_d + \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)\cdot\mathbf{B}_L \otimes \left(\mathbf{m}_d + R_F(\alpha_h)\frac{m+8}{8}(\mathbf{n}\cdot\mathbf{h})^m\right) \\ &= \mathbf{A}_L \otimes \mathbf{m}_d + \max\left(\mathbf{n}\cdot\mathbf{L}, 0\right)\cdot\mathbf{B}_L \otimes \left(\mathbf{m}_d + \mathbf{m}_s\right)\end{aligned}$$

Vertex Types (1)

- Chap. 6 & 7
  - HLSL

```
struct VertexIn{
    float3 PosL: POSITION;
    float4 Color : COLOR;
};
struct VertexOut{
    float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};
```

  - Application.cpp or FrameResource.h

```
struct Vertex{
    DirectX::XMFLOAT3 Pos;
    DirectX::XMFLOAT4 Color;
};
std::vector<Vertex> vertices;
```

Vertex Types (2)

- Chap. 8
  - HLSL

```
cbuffer cbMaterial : register(b1) {
    float4 gDiffuseAlbedo;
    float3 gFresnelR0;
    float  gRoughness;
    float4x4 gMatTransform;
};
struct VertexIn {
    float3 PosL    : POSITION;
    float3 NormalL : NORMAL;
};
struct VertexOut {
    float4 PosH    : SV_POSITION;
    float3 PosW    : POSITION;
    float3 NormalW : NORMAL;
};
```

Vertex Types (3)

- FrameResource.h

```
struct Vertex {
    DirectX::XMFLOAT3 Pos;
    DirectX::XMFLOAT3 Normal;
};
```

- Application.cpp

```
std::unordered_map<std::string, std::unique_ptr<Material>>
mMaterials;
// struct Material is defined in d3dUtil.h.
// Elements of mMaterials are created in BuildMaterials().
struct RenderItem {
// …
    Material* Mat = nullptr;
    MeshGeometry* Geo = nullptr;
// …
};
void Application::UpdateMaterialCBs(const GameTimer& gt);
// Materials are copied to the GPU.
```

Implementing Materials (1)

```
// d3dUtil.h
struct Material {
    std::string Name;

    // Index into constant buffer corresponding to this material.
    int MatCBIndex = -1;
    // Index into SRV heap for diffuse texture.
    int DiffuseSrvHeapIndex = -1;
    // Index into SRV heap for normal texture.
    int NormalSrvHeapIndex = -1;

    int NumFramesDirty = gNumFrameResources;

    // Material constant buffer data used for shading.
    DirectX::XMFLOAT4 DiffuseAlbedo = { 1.0f, 1.0f, 1.0f, 1.0f };
    DirectX::XMFLOAT3 FresnelR0 = { 0.01f, 0.01f, 0.01f };
    float Roughness = .25f;
    DirectX::XMFLOAT4X4 MatTransform = MathHelper::Identity4x4();
};
```

Implementing Materials (2)

```
// d3dUtil.h
struct MaterialConstants {
    DirectX::XMFLOAT4 DiffuseAlbedo = { 1.0f, 1.0f, 1.0f, 1.0f };
    DirectX::XMFLOAT3 FresnelR0 = { 0.01f, 0.01f, 0.01f };
    float Roughness = 0.25f;

    // Used in texture mapping.
    DirectX::XMFLOAT4X4 MatTransform = MathHelper::Identity4x4();
};
```

Implementing Materials (3)

```
// Applicaton.cpp
class Application: public D3DApp {
// …
    void BuildMaterials();
    void UpdateMaterialCBs(const GameTimer& gt);
    void BuildFrameResources();
// …
};
struct RenderItem {
// …
    XMFLOAT4X4 World = MathHelper::Identity4x4();
    XMFLOAT4X4 TexTransform = MathHelper::Identity4x4();
    Material* Mat = nullptr;
    MeshGeometry* Geo = nullptr;
// …
};
struct FrameResource {
// …
     std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
     std::unique_ptr<UploadBuffer<MaterialConstants>> MaterialCB = nullptr;
     std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;

// …
};
```

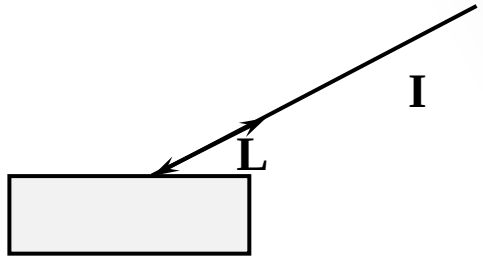
Parallel Lights

- Parallel light (directional light)
  - This light approximates a light source that is very far away.
  - We can ignore the effects of distance and just specify the light intensity where the light strikes the scene.

Point Light

- Point light
  - This is a light that gets emitted from a single point in all directions.
  - Light vector: $\mathbf{L}$
    - Light position: $\mathbf{Q}$, Arbitrary point: $\mathbf{P}$

$$\mathbf{L} = \frac{\mathbf{Q} - \mathbf{P}}{\|\mathbf{Q} - \mathbf{P}\|}$$



  - Attenuation
    - Physically, light intensity weakens as a function of distance based on the inverse square law.

$$I(d) = \frac{I_0}{d^2}$$

    - We use linear falloff.

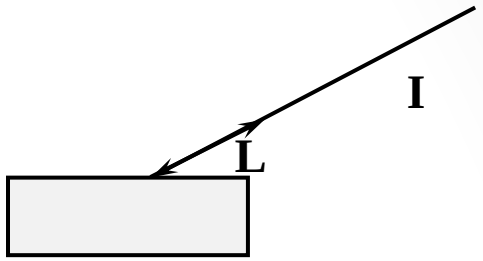
$$\text{att}(d) = \text{saturate}\left(\frac{falloffEnd - d}{falloffEnd - falloffStart}\right)$$

Quadratic attenuation: $\text{att}(d) = \frac{1}{a + bd + cd^2}$

Spotlights

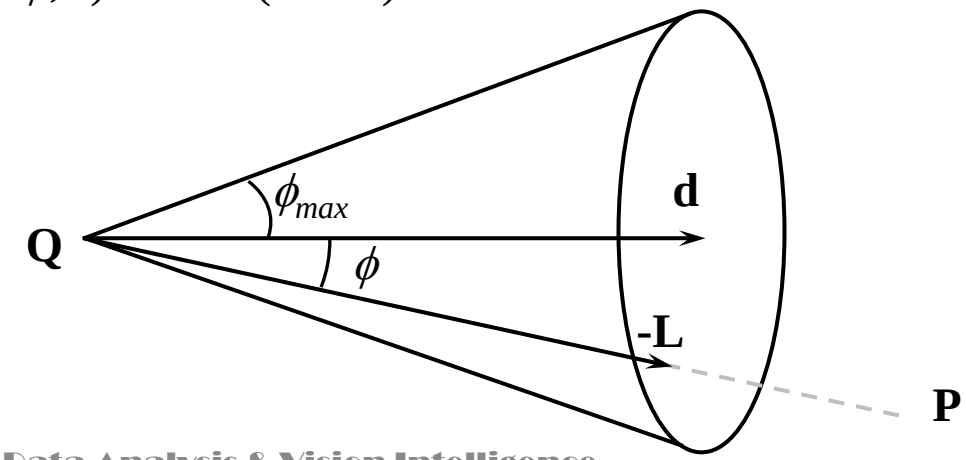
- Spotlight
  - This is a light source that emits light in a cone-shaped beam, creating a focused area of illumination.
  - Light vector: $\mathbf{L}$
    - Light position: $\mathbf{Q}$, Arbitrary point: $\mathbf{P}$

$$\mathbf{L} = \frac{\mathbf{Q} - \mathbf{P}}{\|\mathbf{Q} - \mathbf{P}\|}$$



  - Intensity falloff

$$k_{spot}(\phi) = \max(\cos\phi, 0)^s = \max(-\mathbf{L} \cdot \mathbf{d})^s$$



Kyung Hee University
nize@khu.ac.kr

Lighting Structure (1)

```
// d3dUtil.h
struct Light {
    DirectX::XMFLOAT3 Strength = { 0.5f, 0.5f, 0.5f };
    float FalloffStart = 1.0f;          // point/spot light only
    DirectX::XMFLOAT3 Direction = { 0.0f, -1.0f, 0.0f };
        // directional/spot light only
    float FalloffEnd = 10.0f;           // point/spot light only
    DirectX::XMFLOAT3 Position = { 0.0f, 0.0f, 0.0f };
        // point/spot light only
    float SpotPower = 64.0f;            // spot light only
};
```

Lighting Structure (2)

```
// hlsl
#define MaxLights 16
struct Light {
    float3 Strength;
    float FalloffStart; // point/spot light only
    float3 Direction;   // directional/spot light only
    float FalloffEnd;   // point/spot light only
    float3 Position;    // point light only
    float SpotPower;    // spot light only (s)
};

struct Material {
    float4 DiffuseAlbedo;
    float3 FresnelR0;
    float Shininess;
};
```

$$k_{spot}(\phi) = \max(\cos\phi, 0)^s = \max(-\mathbf{L}\cdot\mathbf{d})^s$$

Constant Buffers (1)

```
// hlsl
cbuffer cbPerObject : register(b0) {
   float4x4 gWorld;
};

cbuffer cbMaterial : register(b1) {
   float4 gDiffuseAlbedo;
   float3 gFresnelR0;
   float  gRoughness;
   float4x4 gMatTransform;
};
cbuffer cbPass : register(b2) {
   // …
   float4 gAmbientLight;
// Indices [0, NUM_DIR_LIGHTS) are directional lights;
// indices [NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS) are point lights;
// indices [NUM_DIR_LIGHTS+NUM_POINT_LIGHTS,
//           NUM_DIR_LIGHTS+NUM_POINT_LIGHT+NUM_SPOT_LIGHTS)
//     are spot lights for a maximum of MaxLights per object.
    Light gLights[MaxLights];
};
```

Constant Buffers (2)

```
// FrameResource.h
struct PassConstants {
 DirectX::XMFLOAT4X4 View = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 InvView = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 Proj = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 InvProj = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 ViewProj = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 InvViewProj = MathHelper::Identity4x4();
    DirectX::XMFLOAT3 EyePosW = { 0.0f, 0.0f, 0.0f };
    float cbPerObjectPad1 = 0.0f;

    // …

    DirectX::XMFLOAT4 AmbientLight = { 0.0f, 0.0f, 0.0f, 1.0f };

// Indices [0, NUM_DIR_LIGHTS) are directional lights;
// indices [NUM_DIR_LIGHTS, NUM_DIR_LIGHTS+NUM_POINT_LIGHTS) are point lights;
// indices [NUM_DIR_LIGHTS+NUM_POINT_LIGHTS,
//           NUM_DIR_LIGHTS+NUM_POINT_LIGHT+NUM_SPOT_LIGHTS)
//     are spot lights for a maximum of MaxLights per object.
    Light Lights[MaxLights];
};
```

Common Helper Functions

```
// hlsl
float CalcAttenuation(float d, float falloffStart, float falloffEnd) {
    return saturate((falloffEnd-d) / (falloffEnd - falloffStart));
}
float3 SchlickFresnel(float3 R0, float3 normal, float3 lightVec) {
    float cosIncidentAngle = saturate(dot(normal, lightVec));
    float f0 = 1.0f - cosIncidentAngle;
    float3 reflectPercent = R0 + (1.0f - R0)*(f0*f0*f0*f0*f0);

    return reflectPercent;
}
float3 BlinnPhong(float3 lightStrength, float3 lightVec,
    float3 normal, float3 toEye, Material mat) {
    const float m = mat.Shininess * 256.0f;
    float3 halfVec = normalize(toEye + lightVec);
    float roughnessFactor
        = (m + 8.0f)*pow(max(dot(halfVec, normal), 0.0f), m) / 8.0f;
    float3 fresnelFactor = SchlickFresnel(mat.FresnelR0, halfVec, lightVec);
    float3 specAlbedo = fresnelFactor*roughnessFactor;
    specAlbedo = specAlbedo / (specAlbedo + 1.0f);

    return (mat.DiffuseAlbedo.rgb + specAlbedo) * lightStrength;
}
```

Implementing Directional Lights

```
// hlsl
float3 ComputeDirectionalLight(Light L, Material mat,
   float3 normal, float3 toEye) {
   float3 lightVec = -L.Direction;

   // Scale light down by Lambert's cosine law.
   float ndotl = max(dot(lightVec, normal), 0.0f);
   float3 lightStrength = L.Strength * ndotl;

   return BlinnPhong(lightStrength, lightVec, normal, toEye, mat);
}
```

Implementing Point Lights

```
// hlsl
float3 ComputePointLight(Light L, Material mat, float3 pos,
    float3 normal, float3 toEye) {
    float3 lightVec = L.Position - pos;
    float d = length(lightVec);

    if(d > L.FalloffEnd) return 0.0f;
    lightVec /= d;

    // Scale light down by Lambert's cosine law.
    float ndotl = max(dot(lightVec, normal), 0.0f);
    float3 lightStrength = L.Strength * ndotl;

    // Attenuate light by distance.
    float att = CalcAttenuation(d, L.FalloffStart, L.FalloffEnd);
    lightStrength *= att;

    return BlinnPhong(lightStrength, lightVec, normal, toEye, mat);
}
```

Implementing Spotlights

```
// hlsl
float3 ComputeSpotLight(Light L, Material mat, float3 pos,
    float3 normal, float3 toEye) {
    float3 lightVec = L.Position - pos;
    float d = length(lightVec);

    if(d > L.FalloffEnd) return 0.0f;
    lightVec /= d;

    float ndotl = max(dot(lightVec, normal), 0.0f);
    float3 lightStrength = L.Strength * ndotl;

    float att = CalcAttenuation(d, L.FalloffStart, L.FalloffEnd);
    lightStrength *= att;

    float spotFactor
        = pow(max(dot(-lightVec, L.Direction), 0.0f), L.SpotPower);
    lightStrength *= spotFactor;

    return BlinnPhong(lightStrength, lightVec, normal, toEye, mat);
}
```

Accumulating Multiple Lights (1)

```
// hlsl
#ifndef NUM_DIR_LIGHTS
    #define NUM_DIR_LIGHTS 3          // # of directional lights
#endif

#ifndef NUM_POINT_LIGHTS
    #define NUM_POINT_LIGHTS 0        // # of point lights
#endif

#ifndef NUM_SPOT_LIGHTS
    #define NUM_SPOT_LIGHTS 0         // # of spot lights
#endif
```

Accumulating Multiple Lights (2)

```
// hlsl
float4 ComputeLighting(Light gLights[MaxLights], Material mat,
                       float3 pos, float3 normal, float3 toEye,
                       float3 shadowFactor) {
    float3 result = 0.0f;
    int i = 0;
#if (NUM_DIR_LIGHTS > 0)
    for(i = 0; i < NUM_DIR_LIGHTS; ++i)
        result += shadowFactor[i]
          * ComputeDirectionalLight(gLights[i], mat, normal, toEye);
#endif
#if (NUM_POINT_LIGHTS > 0)
    for(i = NUM_DIR_LIGHTS; i < NUM_DIR_LIGHTS+NUM_POINT_LIGHTS; ++i)
        result += ComputePointLight(gLights[i], mat, pos, normal, toEye);
#endif
#if (NUM_SPOT_LIGHTS > 0)
    for(i = NUM_DIR_LIGHTS + NUM_POINT_LIGHTS;
      i < NUM_DIR_LIGHTS + NUM_POINT_LIGHTS + NUM_SPOT_LIGHTS; ++i)
        result += ComputeSpotLight(gLights[i], mat, pos, normal, toEye);
#endif
    return float4(result, 0.0f);
}
```

VertexIn & VertexOut

```
// hlsl
struct VertexIn
{
    float3 PosL     : POSITION;
    float3 NormalL  : NORMAL;
};

struct VertexOut
{
    float4 PosH     : SV_POSITION;          // Projected position
    float3 PosW     : POSITION;             // World position vector
    float3 NormalW  : NORMAL;               // World normal vector
};
```

VS

```
// hlsl
VertexOut VS(VertexIn vin) {
   VertexOut vout = (VertexOut)0.0f;

    // Transform to world space.
    float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
    vout.PosW = posW.xyz;

    // Assumes uniform scaling;
    // otherwise, need to use inverse-transpose of world matrix.
    vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

    vout.PosH = mul(posW, gViewProj);

    return vout;
}
```

PS

```
// hlsl
float4 PS(VertexOut pin) : SV_Target {
    // Interpolating normal can unnormalize it, so renormalize it.
    pin.NormalW = normalize(pin.NormalW);

    // Vector from point being lit to eye.
    float3 toEyeW = normalize(gEyePosW - pin.PosW);

    // Indirect lighting.
    float4 ambient = gAmbientLight*gDiffuseAlbedo;

    const float shininess = 1.0f - gRoughness;
    Material mat = { gDiffuseAlbedo, gFresnelR0, shininess };
    float3 shadowFactor = 1.0f;
    float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
        pin.NormalW, toEyeW, shadowFactor);
    float4 litColor = ambient + directLight;

    // Common convention to take alpha from diffuse material.
    litColor.a = gDiffuseAlbedo.a;
    return litColor;
}
```

Skull Example (1)

- ./Models/skull.txt, ./Models/car.txt

```
VertexCount: 31076
TriangleCount: 60339
VertexList (pos, normal)
{
        0.592978 1.92413 -2.62486 0.572276 0.816877 0.0721907
        0.571224 1.94331 -2.66948 0.572276 0.816877 0.0721907
        0.609047 1.90942 -2.58578 0.572276 0.816877 0.0721907
...
}
TriangleList
{
        0 1 2
        3 4 5
        6 7 8
...
}
```

Skull Example (2)

```
void Application::BuildSkullGeometry() {
    std::ifstream fin("Models/skull.txt");
// …
    std::vector<Vertex> vertices(vcount);
// …
    std::vector<std::int32_t> indices(3 * tcount);
// …
    fin.close();

    const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
    const UINT ibByteSize = (UINT)indices.size() * sizeof(std::int32_t);

    auto geo = std::make_unique<MeshGeometry>();
    geo->Name = "skullGeo";

    ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
    CopyMemory(geo->VertexBufferCPU->GetBufferPointer(),
        vertices.data(), vbByteSize);
    ThrowIfFailed(D3DCreateBlob(ibByteSize, &geo->IndexBufferCPU));
    CopyMemory(geo->IndexBufferCPU->GetBufferPointer(),
        indices.data(), ibByteSize);
```

Skull Example (3)

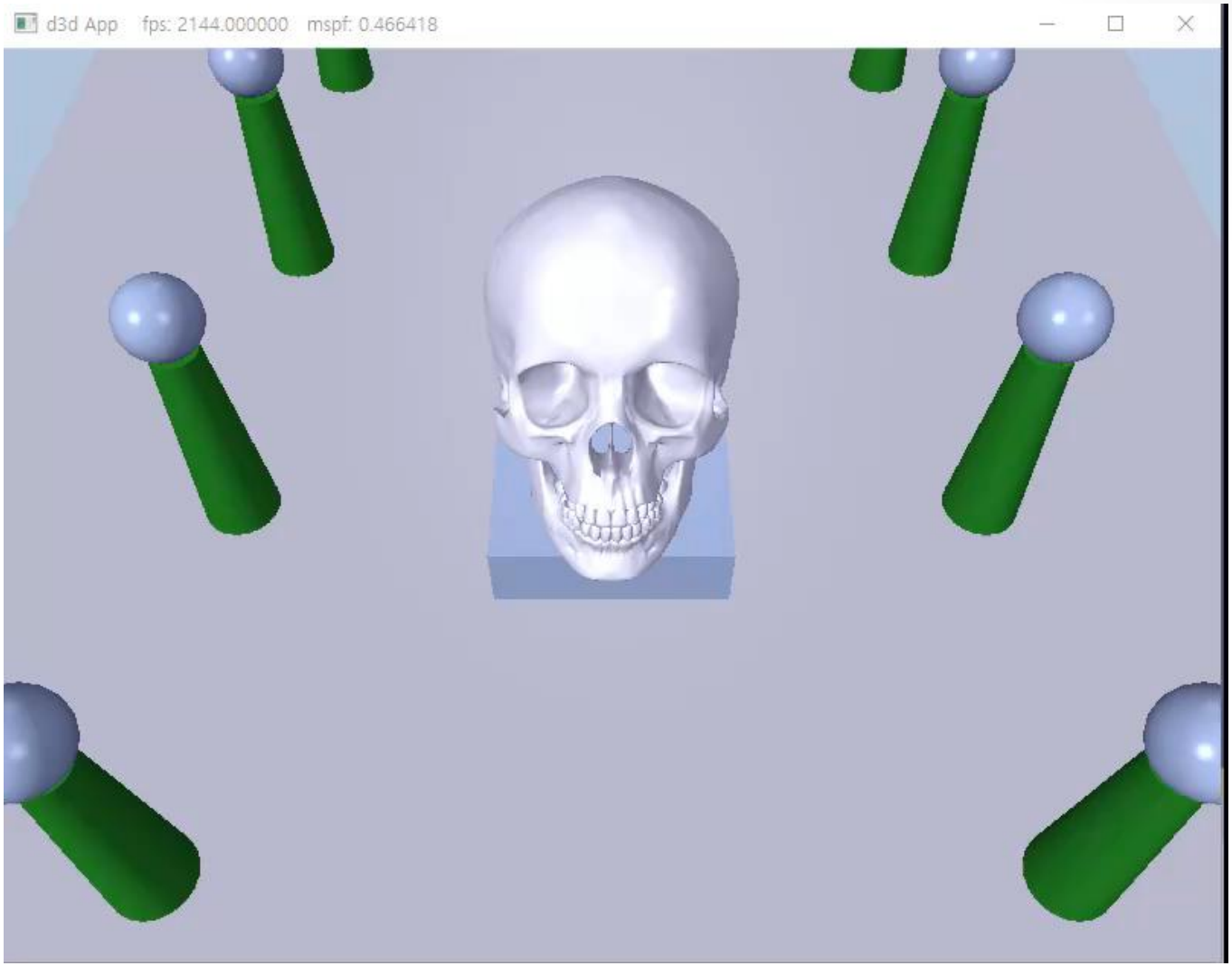
```
    geo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
        mCommandList.Get(), vertices.data(), vbByteSize,
        geo->VertexBufferUploader);
    geo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
        mCommandList.Get(), indices.data(), ibByteSize,
        geo->IndexBufferUploader);

    geo->VertexByteStride = sizeof(Vertex);
    geo->VertexBufferByteSize = vbByteSize;
    geo->IndexFormat = DXGI_FORMAT_R32_UINT;
    geo->IndexBufferByteSize = ibByteSize;

    SubmeshGeometry submesh;
    submesh.IndexCount = (UINT)indices.size();
    submesh.StartIndexLocation = 0;
    submesh.BaseVertexLocation = 0;

    geo->DrawArgs["skull"] = submesh;
    mGeometries[geo->Name] = std::move(geo);
}
```

Shapes

- 

Normal Computation (1)

- Land
  - Function

$$f(x,z) = 0.3z\sin(0.1x) + 0.3x\cos(0.1z)$$

  - Tangent & normal

$$\mathbf{T}_x = \begin{pmatrix} 1 & \frac{\partial f}{\partial x} & 0 \end{pmatrix}$$

$$\mathbf{T}_z = \begin{pmatrix} 0 & \frac{\partial f}{\partial z} & 1 \end{pmatrix}$$

$$\mathbf{n} = \mathbf{T}_z \times \mathbf{T}_x = \begin{pmatrix} -\frac{\partial f}{\partial z} & 1 & -\frac{\partial f}{\partial z} \end{pmatrix}$$

$$= \begin{pmatrix} -0.03z\cos(0.1x) - 0.3x\cos(0.1z) \\ 1 \\ -0.3z\sin(0.1x) + 0.03x\sin(0.1z) \end{pmatrix}^T$$

Normal Computation (2)

```
XMFLOAT3 LitWavesApp::GetHillsNormal(float x, float z) const {
    // n = (-df/dx, 1, -df/dz)
    XMFLOAT3 n(
        -0.03f*z*cosf(0.1f*x) - 0.3f*cosf(0.1f*z),
        1.0f,
        -0.3f*sinf(0.1f*x) + 0.03f*x*sinf(0.1f*z));

    XMVECTOR unitNormal = XMVector3Normalize(XMLoadFloat3(&n));
    XMStoreFloat3(&n, unitNormal);

    return n;
}
```

Normal Computation (3)

- Wave
  - 4 points

$$y(i+1, j, k), y(i-1, j, k), y(i, j+1, k), y(i, j-1, k)$$

$$\mathbf{T}_x = \begin{pmatrix} 1 & \frac{\partial f}{\partial x} & 0 \end{pmatrix}$$

$$\mathbf{T}_z = \begin{pmatrix} 0 & \frac{\partial f}{\partial z} & 1 \end{pmatrix}$$

$$\mathbf{n} = \mathbf{T}_z \times \mathbf{T}_x = \begin{pmatrix} -\frac{\partial f}{\partial z} & 1 & -\frac{\partial f}{\partial z} \end{pmatrix}$$

$$\frac{\partial f}{\partial x} = \frac{y(i+1, j, k) - y(i-1, j, k)}{2}$$

$$\frac{\partial f}{\partial z} = \frac{y(i, j+1, k) - y(i, j-1, k)}{2}$$

Normal Computation (4)

```
void Waves::Update(float dt) {          // Waves.cpp
   // …
   for(int j = 1; j < mNumCols-1; ++j) {
      float l = mCurrSolution[i*mNumCols+j-1].y;
      float r = mCurrSolution[i*mNumCols+j+1].y;
      float t = mCurrSolution[(i-1)*mNumCols+j].y;
      float b = mCurrSolution[(i+1)*mNumCols+j].y;
      mNormals[i*mNumCols+j].x = -r+l;
      mNormals[i*mNumCols+j].y = 2.0f*mSpatialStep;
      mNormals[i*mNumCols+j].z = b-t;

   XMVECTOR n =
      XMVector3Normalize(XMLoadFloat3(&mNormals[i*mNumCols+j]));
   XMStoreFloat3(&mNormals[i*mNumCols+j], n);

   // …
}
```

Updating the Light Direction

```
// float mSunTheta = 1.25f*XM_PI;
// float mSunPhi = XM_PIDIV4;
void LitWavesApp::OnKeyboardInput(const GameTimer& gt) {
   const float dt = gt.DeltaTime();

   if(GetAsyncKeyState(VK_LEFT) & 0x8000)
      mSunTheta -= 1.0f*dt;

   if(GetAsyncKeyState(VK_RIGHT) & 0x8000)
      mSunTheta += 1.0f*dt;

   if(GetAsyncKeyState(VK_UP) & 0x8000)
      mSunPhi -= 1.0f*dt;

   if(GetAsyncKeyState(VK_DOWN) & 0x8000)
      mSunPhi += 1.0f*dt;

   mSunPhi = MathHelper::Clamp(mSunPhi, 0.1f, XM_PIDIV2);
}
```

Demo