Week5

Lighting & Texturing

Hooman Salamat

Objectives

1. To gain a basic understanding of the interaction between lights and materials

2. To understand the differences between local illumination and global illumination

3. To find out how we can mathematically describe the direction a point on a surface is “facing” so that we can determine the angle at which incoming light strikes the surface

4. To learn how to correctly transform normal vectors

5. To be able to distinguish between ambient, diffuse, and specular light

6. To learn how to implement directional lights, point lights, and spotlights

7. To understand how to vary light intensity as a function of depth by controlling attenuation parameters

Lighting

Our visual perception of the world depends on light and its interaction with materials.

On the left we have an unlit sphere, and on the right, we have a lit sphere.

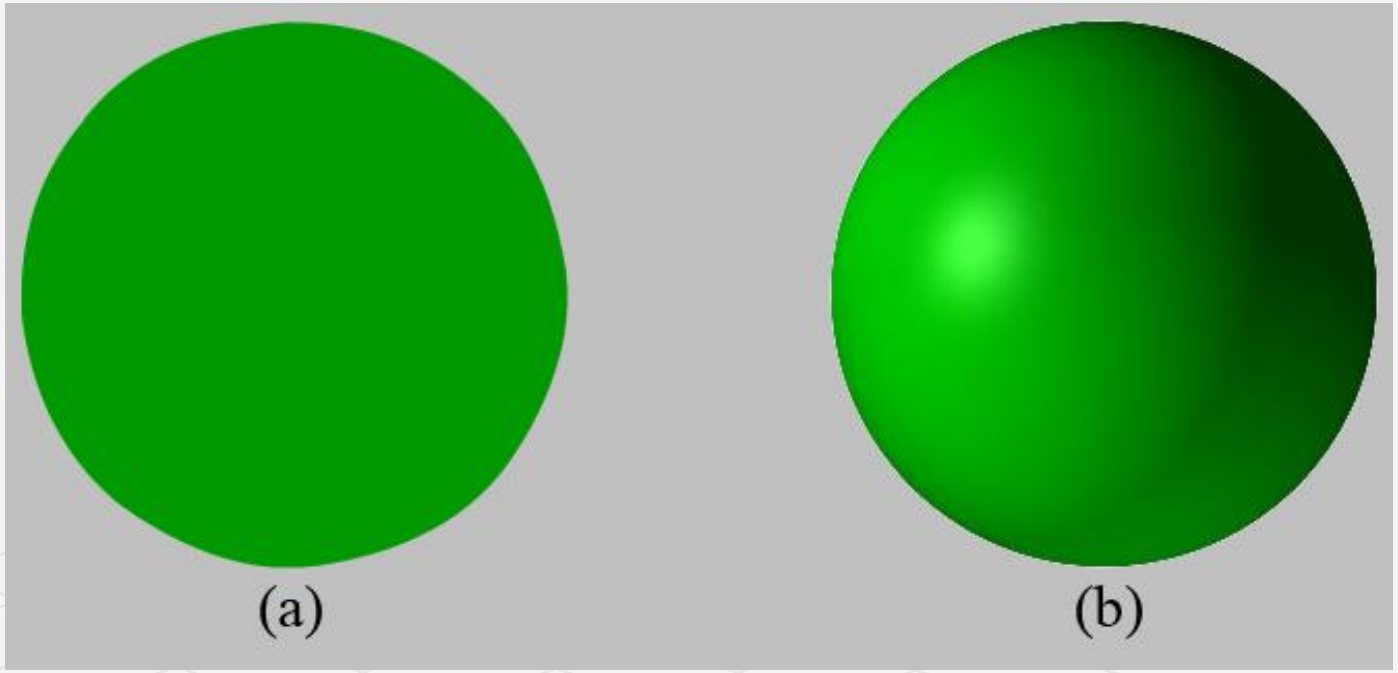
The sphere on the left look rather flat, it is not even a sphere at all, but just a 2D circle!

On the other hand, the sphere on the right does look 3D

The lighting and shading aid in our perception of the solid form and volume of the object.

3D special FX (Special Effects) scenes for films can be much more complex and utilize very realistic lighting models than a game because the frames for a film are pre-rendered.

Games, on the other hand, are real-time applications, and therefore, the frames need to be drawn at a rate of at least 30 frames per second.



Lighting

When using lighting, we no longer specify vertex colors directly

We specify materials and lights then apply a lighting equation, which computes the vertex colors for us based on light/material interaction

This leads to a much more realistic coloring of the object

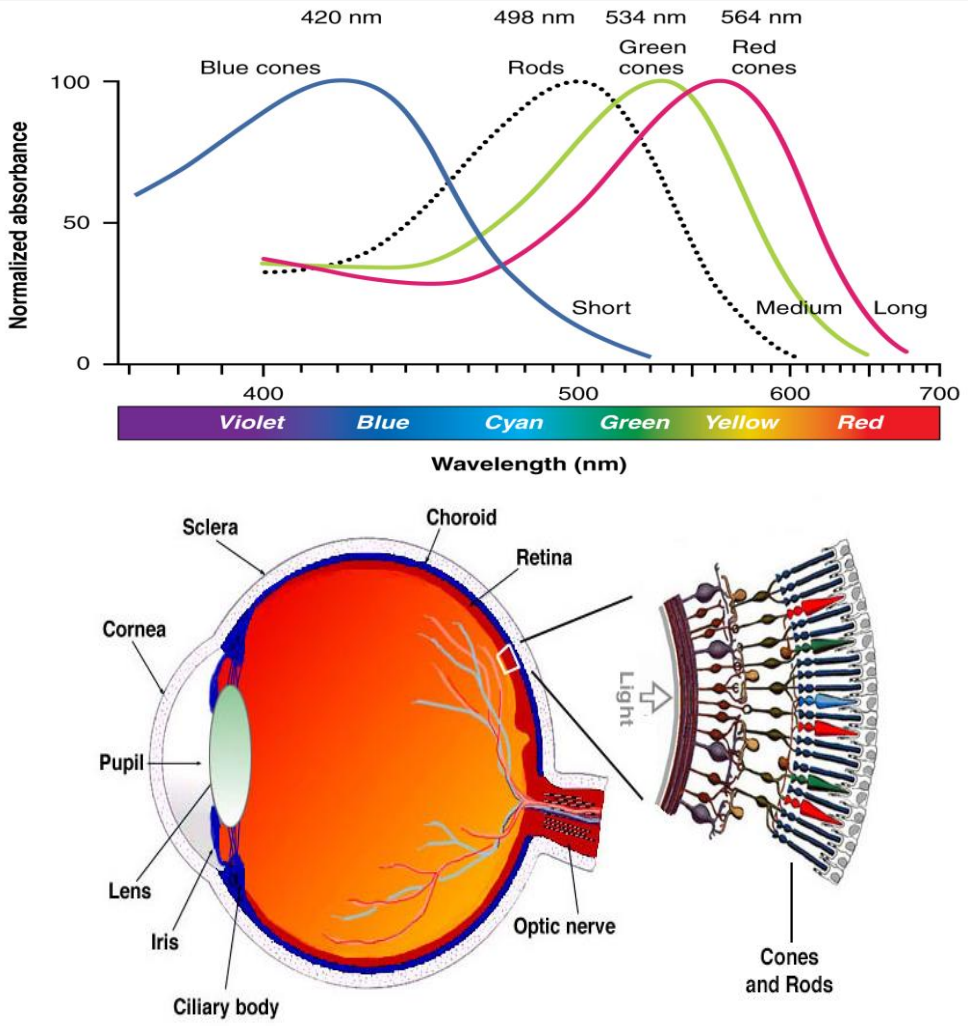
A light source can emit various red, green, and blue light

Some of that light may be absorbed and some may be reflected

When the light mixture enters the eye and strikes an area of the retina, cone receptor cells are stimulated and neural impulses are sent down the optic nerve toward the brain.

The brain interprets the signal and generates a color.

As the light mixture varies, the cells are stimulated differently, which in turn generates a different color in the mind.



Lighting & Materials

(a) Flux of incoming white light.

(b) The light strikes the cylinder and some rays are absorbed and other rays are scatted toward the eye and sphere.

(c)The light reflecting off the cylinder toward the sphere is absorbed or reflected again and travels into the eye.

(d) The eye receives incoming light that determines what the eye sees.

Materials can be thought of as the properties that determine how light interacts with a 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.

Suppose that the material of the cylinder reflects 75% red light, 75% green light, and absorbs the rest, and the sphere reflects 25% red light and absorbs the rest.

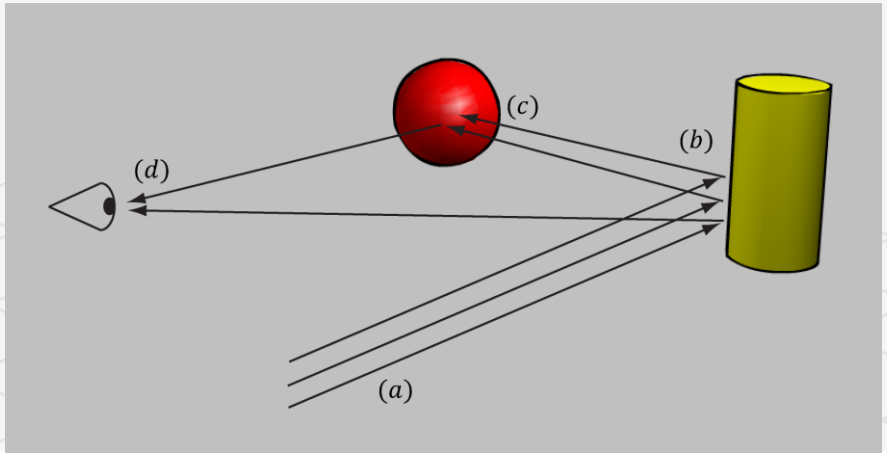
Assume that pure white light is being emitted from the light source.

As the light rays strike the cylinder, all the blue light is absorbed and only 75% red and green light is reflected (i.e., a medium-high intensity yellow).

This light is then scattered—some of it travels into the eye and some of it travels toward the sphere.

The part that travels into the eye primarily stimulates the red and green cone cells to a semi-high degree; hence, the viewer sees the cylinder as a semi-bright shade of yellow.

The other light rays travel toward the sphere and strike it. The sphere reflects 25% red light and absorbs the rest; thus, the diluted incoming red light (medium-high intensity red) is diluted further and reflected, and all of the incoming green light is absorbed.



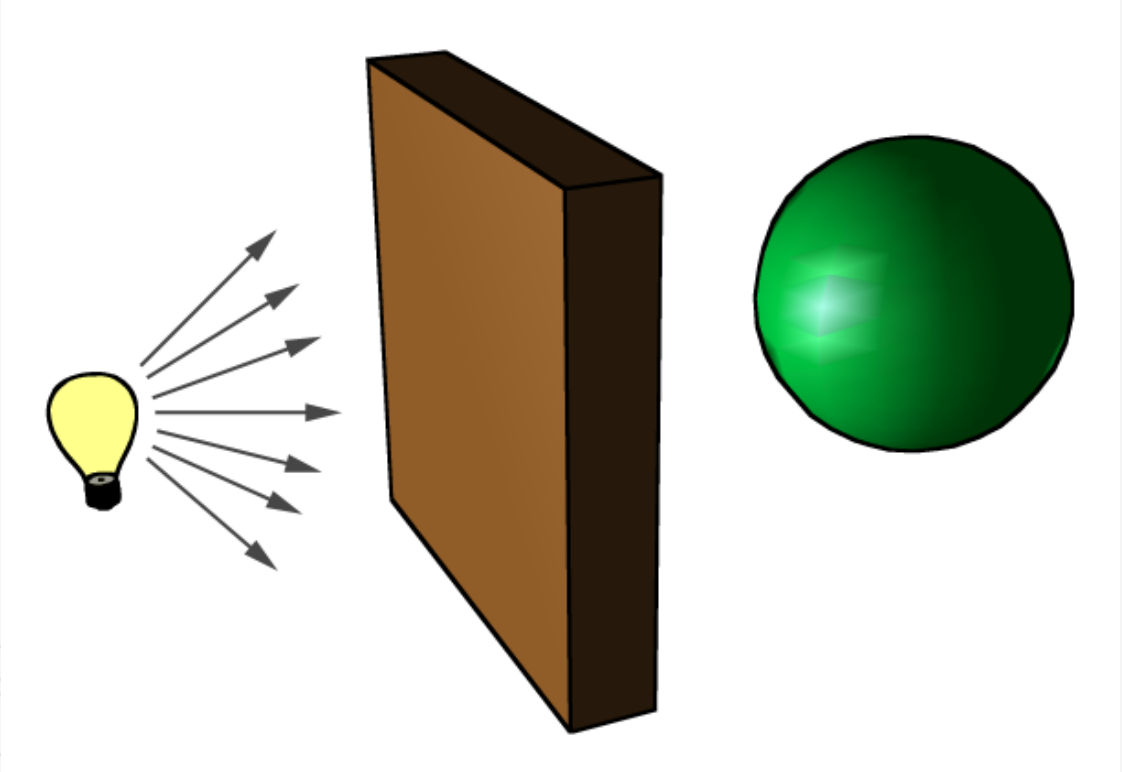
Lighting Model

The lighting models we are going to look at are local illumination models

Each object is lit independently of another object, and only the light directly emitted from light sources is taken into account

It looks like the wall would block the rays emitted by the bulb and the sphere is in the shadow of the wall

But in local illumination, the sphere is lit as if the wall were not there



Global illumination models light

Global illumination models light objects by taking into consideration:

- The light directly emitted from light sources
- The indirect light that has bounced off other objects in the scene

Global illumination models are generally expensive for real-time games

However, they come very close to generating photorealistic scenes.

Finding real-time methods for approximating global illumination is an area of ongoing research; see, for example, voxel global illumination.

http://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf

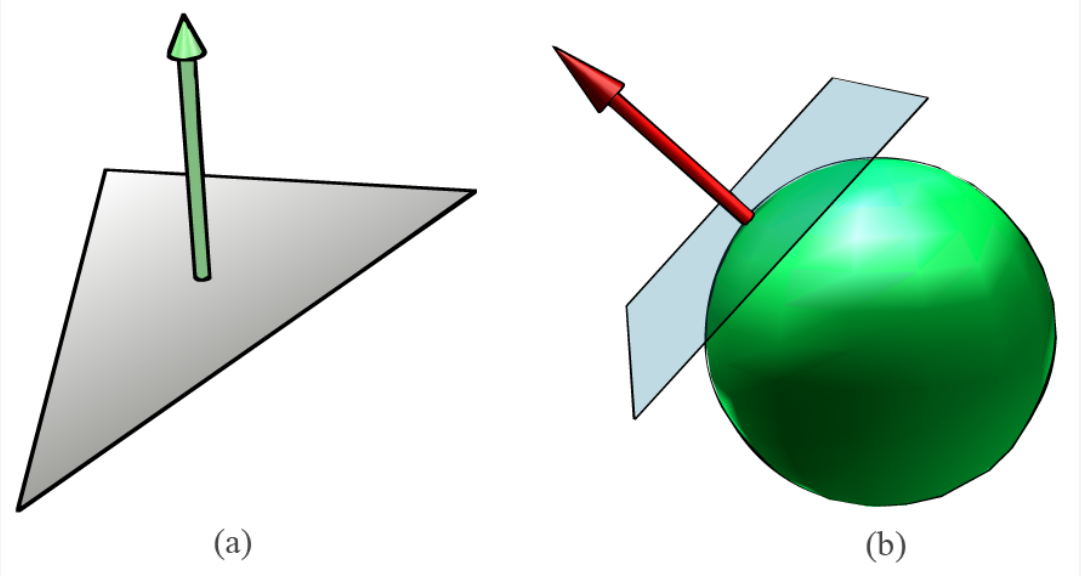
Other popular methods are to precompute indirect lighting for static objects (e.g., walls, statues), and then use that result to approximate indirect lighting for dynamic objects (e.g., moving game characters).

Normal Vectors

A face normal is a unit vector that describes the direction a polygon is facing

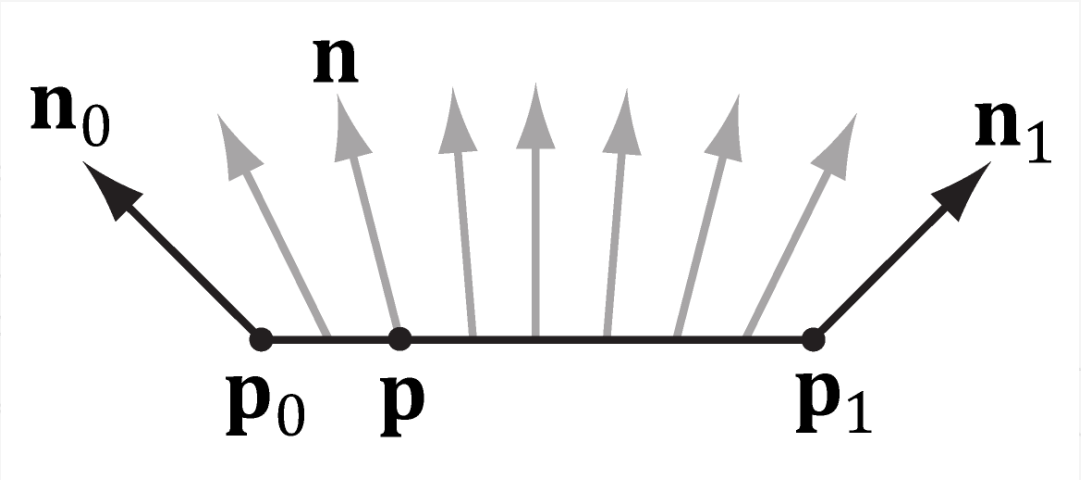
(a) The face normal is orthogonal to all points on the face

(b) The surface normal is the vector that is orthogonal to the tangent plane of a point on a surface



The vertex normals **n**0 and **n**1 are defined at the segment vertex points **p**0 and **p**1.

A normal vector **n** for a point **p** in the interior of the line segment is found by linearly interpolating (weighted average) between the vertex normals; that is, **n** = **n**0 + *t*(**n**1 – **n**0) where *t* is such that **p** = **p**0 + *t*(**p**1 – **p**0)



Although we illustrated normal interpolation over a line segment for simplicity, the idea straightforwardly generalizes to interpolating over a 3D triangle.

Face Normal

To find the face normal of a triangle Δp0, p1, p2 we first compute two vectors that lie on the triangle's edges:

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$$

$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$$

Then the face normal is:

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

XMVector3Normalize

Here is a function that computes the face normal of the front side of a triangle from the three vertex points of the triangle:

```
XMVECTOR ComputeNormal(FXMVECTOR p0, FXMVECTOR p1, FXMVECTOR p2)

{

XMVECTOR u = p1 - p0;

XMVECTOR v = p2 - p0;

return XMVector3Normalize( XMVector3Cross(u, v));

}
```

```
XMVECTOR p0 = XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f);

XMVECTOR p1 = XMVectorSet(1.0f, 2.0f, 3.0f, 0.0f);

XMVECTOR p2 = XMVectorSet(-2.0f, 1.0f, -3.0f, 0.0f);

XMVECTOR facenormal = ComputeNormal(p0, p1, p2);
```

Vertex Normal Averaging

For lighting calculations, we need the surface normal at each point on the surface of a triangle mesh so that we can determine the angle at which light strikes the point on the mesh surface.

To obtain surface normals, we specify the surface normals only at the vertex points (so-called ***vertex normals***).

In order to obtain a surface normal approximation at each point on the surface of a triangle mesh, these vertex normals will be interpolated across the triangle during rasterization

Interpolating the normal and doing lighting calculations per pixel is called pixel lighting or phong lighting.

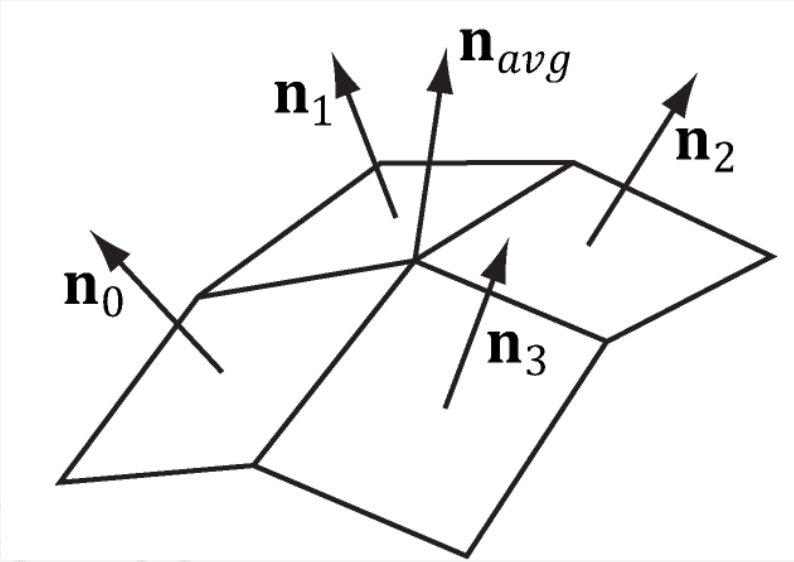
For a differentiable surface, the middle vertex is shared by the neighboring four polygons, so we approximate the middle vertex normal by averaging the four polygon face normals.

The technique that is generally applied to triangle meshes is called ***vertex normal averaging***

The vertex normal n or an arbitrary vertex v in a mesh is found by averaging the face normal of every polygon in the mesh that shares the vertex v

The vertex normal for v is given by:

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



Transforming Normal Vectors

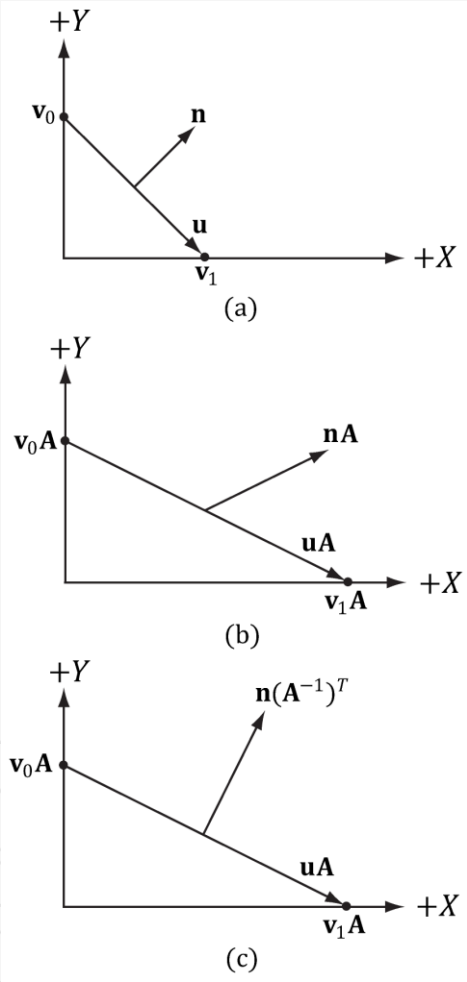
(a) we have a tangent vector $\mathbf{u} = \mathbf{v}_1 - \mathbf{v}_0$ orthogonal to a normal vector **n**.

If we apply a non-uniform scaling transformation **A** we see from Figure *b* that the transformed tangent vector $\mathbf{uA} = \mathbf{v}_1\mathbf{A} - \mathbf{v}_0\mathbf{A}$ does not remain orthogonal to the transformed normal vector **nA**.

(b) After scaling by 2 units on the *x*-axis the normal is no longer orthogonal to the surface.

(c) The surface normal correctly transformed by the inverse-transpose of the scaling transformation.

Given a transformation matrix **A** that transforms points and vectors (non-normal), we want to find a transformation matrix **B** that transforms normal vectors such that the transformed tangent vector is orthogonal to the transformed normal vector (i.e., $\mathbf{uA} \cdot \mathbf{nB} = 0$).



Transforming normal vectors

Thus $\mathbf{B} = (\mathbf{A}^{-1})^T$ (the inverse transpose of **A**) does the job in transforming normal vectors so that they are perpendicular to its associated transformed tangent vector **uA**.

Note that if the matrix is orthogonal ($\mathbf{A}^T = \mathbf{A}^{-1}$), then $\mathbf{B} = (\mathbf{A}^{-1})^T = (\mathbf{A}^T)^T = \mathbf{A}$; that is, we do not need to compute the inverse transpose, since **A** does the job in this case.

In summary, when transforming a normal vector by a nonuniform or shear transformation, use the inverse-transpose.

Even with the inverse-transpose transformation, normal vectors may lose their unit length; therefore, they may need to be renormalized after the transformation.

$\mathbf{u}\cdot\mathbf{n} = 0$	Tangent vector orthogonal to normal vector
$\mathbf{u}\mathbf{n}^T = 0$	Rewriting the dot product as a matrix multiplication
$\mathbf{u}(\mathbf{A}\mathbf{A}^{-1})\mathbf{n}^T = 0$	Inserting the identity matrix $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$
$(\mathbf{u}\mathbf{A})(\mathbf{A}^{-1}\mathbf{n}^T) = 0$	Associative property of matrix multiplication
$(\mathbf{u}\mathbf{A})(\mathbf{A}^{-1}\mathbf{n}^T)^T)^T = 0$	Transpose property $(\mathbf{A}^T)^T = \mathbf{A}$
$(\mathbf{u}\mathbf{A})(\mathbf{n}(\mathbf{A}^{-1})^T)^T = 0$	Transpose property $(\mathbf{A}\mathbf{B}^T)^T = \mathbf{B}^T\mathbf{A}^T$
$\mathbf{u}\mathbf{A}\cdot\mathbf{n}(\mathbf{A}^{-1})^T = 0$	Rewriting the matrix multiplication as a dot product
$\mathbf{u}\mathbf{A}\cdot\mathbf{n}\mathbf{B} = 0$	Transformed tangent vector orthogonal to transformed normal vector

MathHelper.h

Inverse-transpose should be applied to normal to keep normal orthogonal to tangent vectors in case of transformations.

We implement a helper function in *MathHelper.h* for computing the inverse transpose:

We clear out any translation from the matrix because we use the inverse-transpose to transform vectors, and translations only apply to points.

```
static DirectX::XMMATRIX InverseTranspose(DirectX::CXMMATRIX M)

{// B = (A^-1)^T (the inverse transpose of A)

DirectX::XMMATRIX A = M;

A.r[3] = DirectX::XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);

DirectX::XMVECTOR det = DirectX::XMMatrixDeterminant(A);

return DirectX::XMMatrixTranspose(DirectX::XMMatrixInverse(&det, A));

}
```

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\left(\mathbf{A}^{-1}\right)^{T} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & 0 & -2 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Important Vectors in Lighting

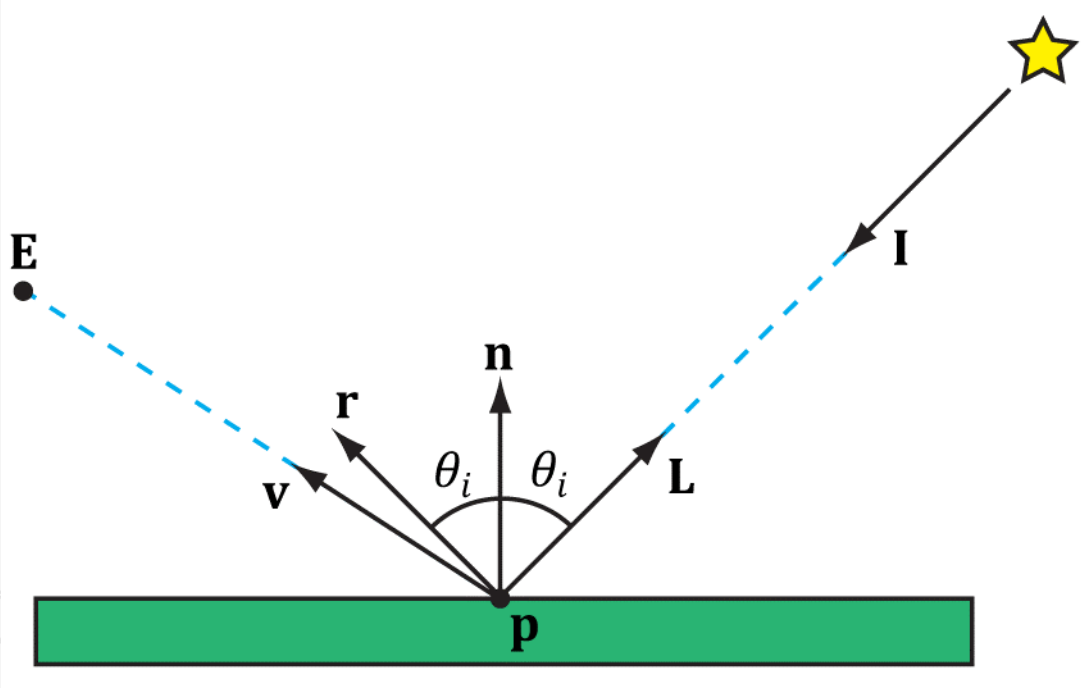
- E is the eye position

- We are considering the point p the eye sees along the line of site defined by the unit vector **v**

- At the point p the surface has normal n, and the point is hit by a ray of light traveling with incident direction **I**

- The light vector L is the unit vector that aims in the opposite direction of the light ray striking the surface point

- The *view vector* (or *to-eye vector*) **v** = normalize(**E** − **p**) is the unit vector from the surface point **p** to the eye point **E** that defines the line of site from the eye to the point on the surface being seen.



The reflection vector

▪ The reflection vector **r** is the reflection of the incident light vector about the surface normal **n**.

r is the reflection vector is given by $r = I - 2(n \cdot I)n$

▪ n is a unit vector

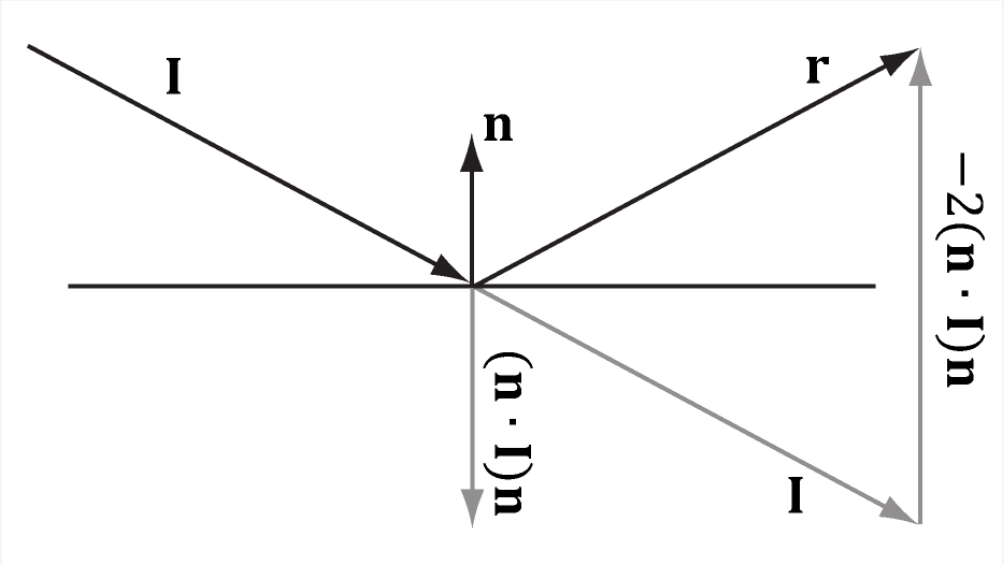
▪ We can actually use the HLSL intrinsic reflect function to compute r for us

Lambert's Cosine Law:, the radiant intensity observed from a surface is directly proportional to the cosine of the angle θ between the direction of the incident light

The vector **L** is used to evaluate $L \cdot n = \cos\theta_i$, where θ_i is the angle between L and n.

▪ The amount of (light) energy emitted per second is called **radiant flux**

▪ The density of radiant flux per area (**irradiance**) is important because that will determine how much light an area on a surface receives

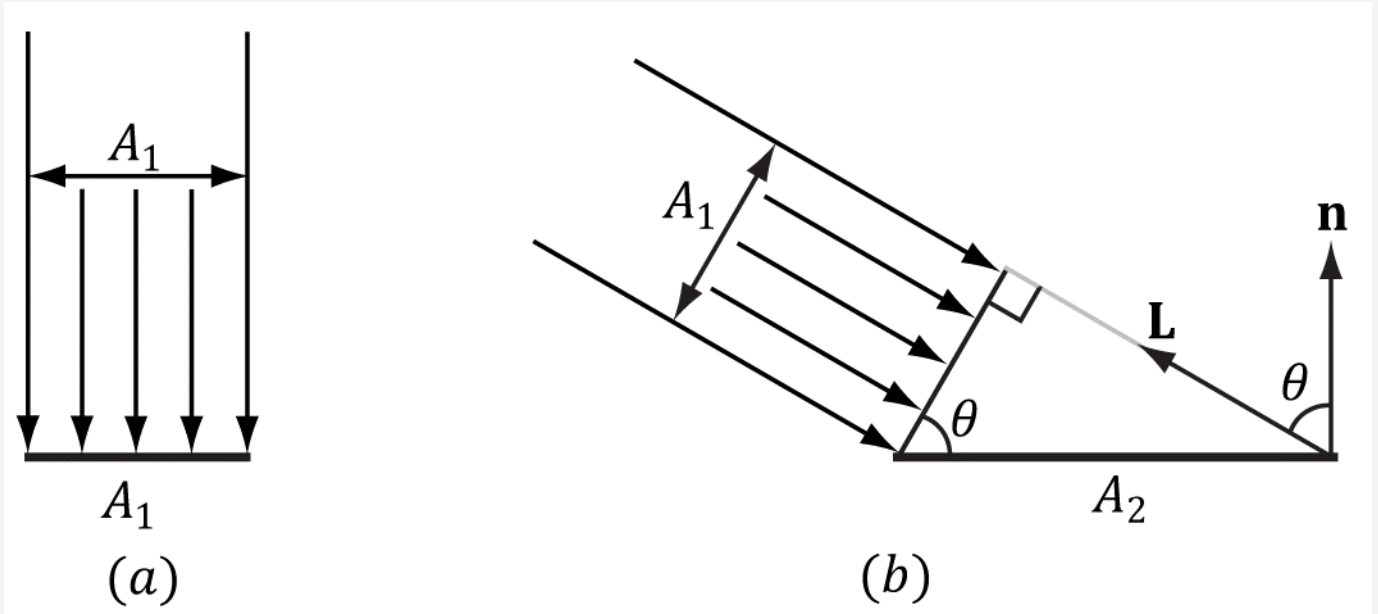


Irradiance

(a) shows a light beam with cross sectional area A1 strikes a surface head-on▪ A light beam with cross sectional area A1 strikes a surface at an angle to cover a larger area. Consider the area A1 with radiant flux P.

Irradiance is $E1 = \frac{P}{A1}$

(b) A2 on the surface, spreading the light energy over a larger area, making the light appear "dimmer". $E2 = \frac{P}{A2}$



A1 and A2 relationship:

$$\cos\theta = \frac{A_1}{A_2} \Rightarrow \frac{1}{A_2} = \frac{\cos\theta}{A_1}$$

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

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

Intensity

▪ The density of radiant flux per area (**irradiance**) striking area *A*2 is equal to the irradiance at the area *A*1 perpendicular to the light direction scaled by **n·L** = cos θ. This is called *Lambert's Cosine Law*.

To handle the case where light strikes the back of the surface (which results in the dot product being negative), we clamp the result with the max function.

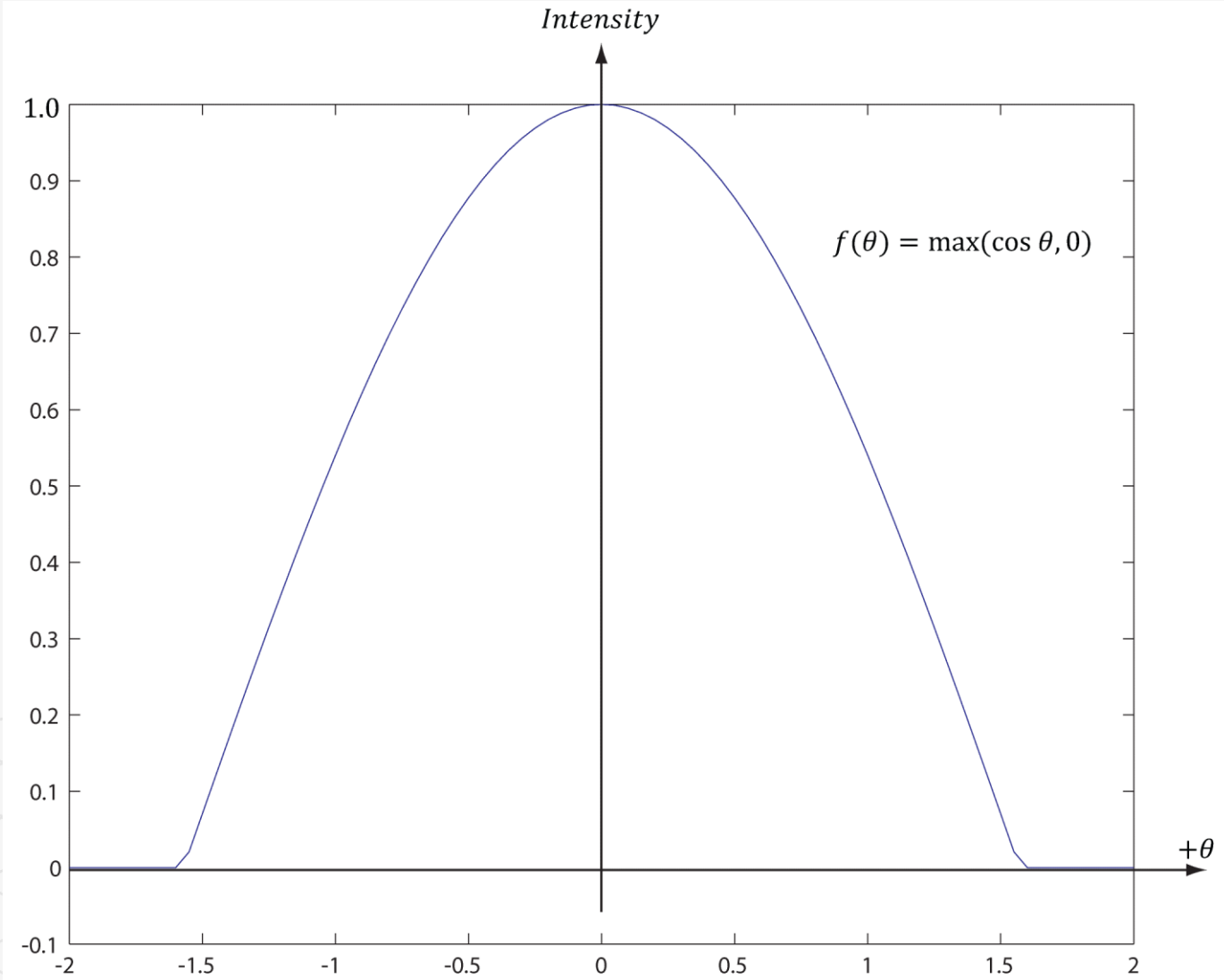
This figure shows a plot of *f*(θ) to see how the intensity, ranging from 0.0 to 1.0 (i.e., 0% to 100%), varies with θ.

Plot of the function

$$f\ (\theta) = \max(\cos\theta, 0) = \max(L \cdot n, 0)$$

$$for\ -2 \leq \theta \leq 2.$$

Note that π/2 ≈ 1.57



DIFFUSE LIGHTING

Consider the surface of an opaque object:

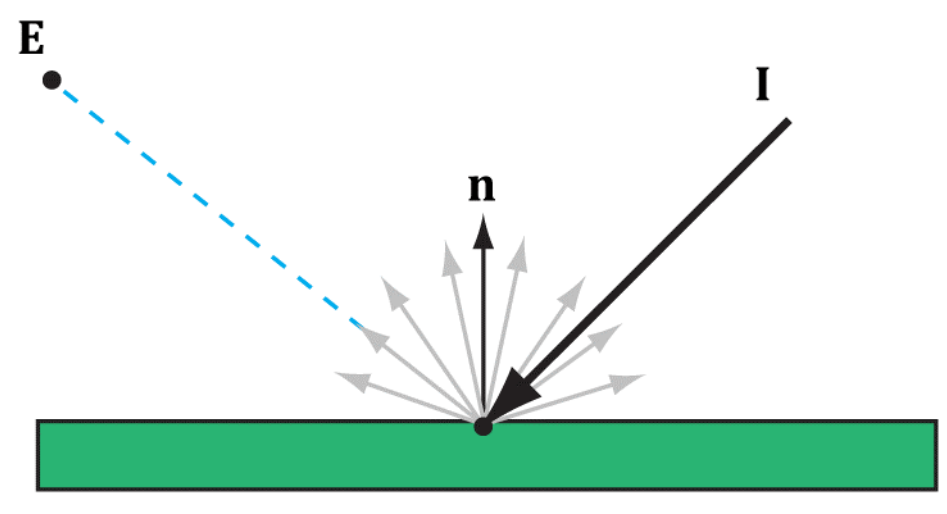
When light strikes a point on the surface, some of the light enters the interior of the object and interacts with the matter near the surface.

The light will bounce around in the interior, where some of it will be absorbed and the remaining part scattered out of the surface in every direction; this is called a *diffuse reflection*.

Because it is difficult to model this subsurface scattering, we assume the re-emitted light scatters out equally in all directions above the surface about the point the light entered.

We stipulate that the reflected light will reach the eye no matter the viewpoint (eye position). Therefore, we do not need to take the viewpoint into consideration (i.e., the diffuse lighting calculation is viewpoint independent).

The amount of absorption and scattering out depends on the material; for example, wood, dirt, brick, tile, and stucco would absorb/scatter light differently



Calculation of diffuse lighting

Two parts: a light color and a *diffuse albedo* color.

The diffuse albedo specifies the amount of incoming light that the surface reflects due to diffuse reflectance.

The diffuse albedo components must be in the range 0.0 to 1.0 so that they describe the fraction of light reflected.

This is handled with a component-wise color multiplication

For example, suppose some point on a surface reflects 50% incoming red light, 100% green light, and 75% blue light, and the incoming light color is 80% intensity white light.

$\boldsymbol{c}_d = \boldsymbol{B}_L \otimes \boldsymbol{m}_d = (0.8, 0.8, 0.8) \otimes (0.5, 1.0, 0.75) = (0.4, 0.8, 0.6)$

We still need to include Lambert's cosine law (which controls how much of the original light the surface receives based on the angle between the surface normal and light vector)

Then the amount of diffuse light reflected off a point is given by:

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

$\mathbf{B}_L$: the quantity of incoming light,

$\boldsymbol{m}_d$: the diffuse albedo color,

L: the light vector,

n: be the surface normal.

AMBIENT and SPECULAR LIGHTING

- Much light we see in the real world is indirect

- Some light scatters off the walls or other objects in the room and eventually strikes an object on the side that is not directly lit.

- To sort of hack this indirect light, we introduce an ambient term to the lighting equation:

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

$\mathbf{A}_L$: the total amount of indirect (ambient) light a surface receives

$\boldsymbol{m}_d$: The diffuse albedo **m***d* specifies the amount of incoming light that the surface reflects due to diffuse reflectance.

- A second kind of reflection happens due to the Fresnel effect, which is a physical phenomenon. (pronounced "fre-nel," the "s" is silent)

. the observation that the amount of reflectance you see on a surface depends on the viewing angle. If you look straight down from above at a pool of water, you will not see very much reflected light on the surface of the pool, and can see down through the surface to the bottom of the pool. At a glancing angle (looking with your eye level with the water, from the edge of the water surface), you will see much more specularity and reflections on the water surface, and might not be able to see what's under the water.

- 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

- We refer to this light reflection process as specular reflection and the reflected light as specular light.

Specular Lighting

The principle of the Fresnel effect is simple:
Steep angle = weak reflection, shallow angle = strong reflection.

(a)The Fresnel effect for a perfectly flat mirror with normal n.

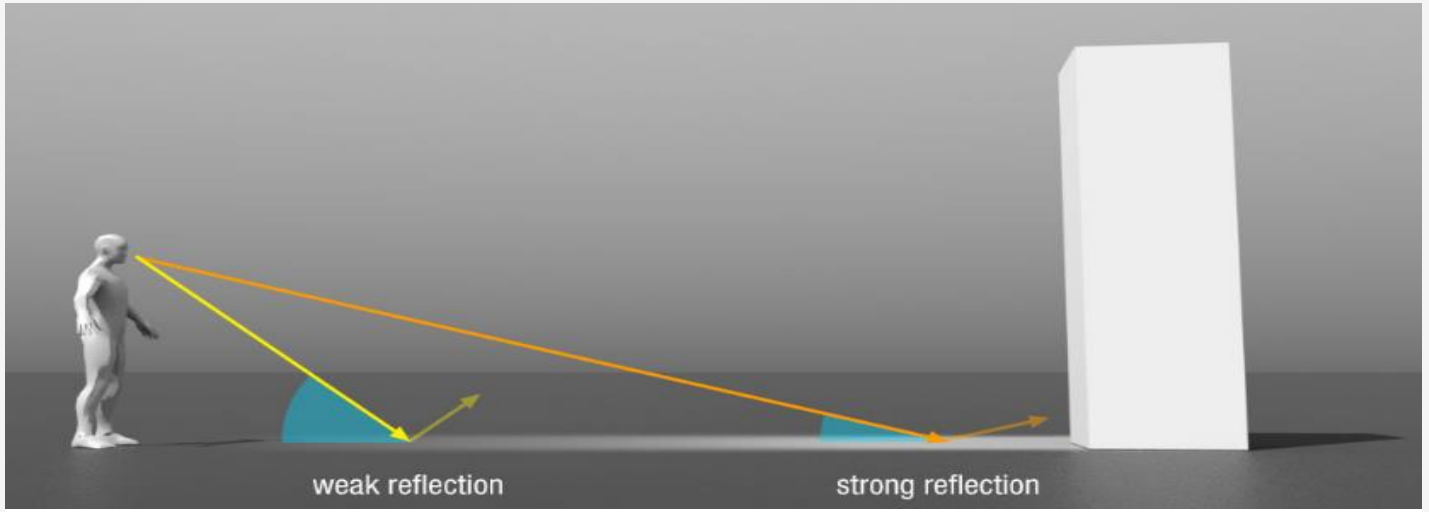
▪ The incident light I is split where some of it reflects in the reflection direction r and the remaining light refracts into the medium in the refraction direction t

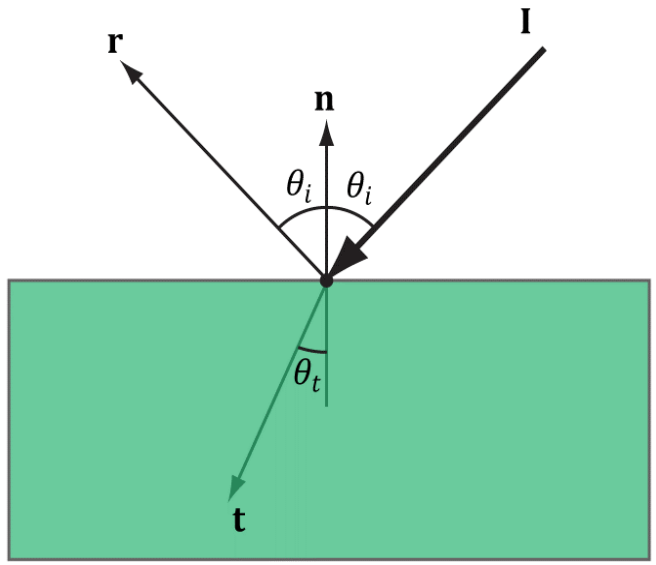
▪ The angle between the reflection vector and normal is always θ_i, which is the same as the angle between the light vector L = −**I** and normal n

▪ The angle θ_t between the refraction vector and −n depends on the indices of refraction between the two mediums and is specified by Snell's Law

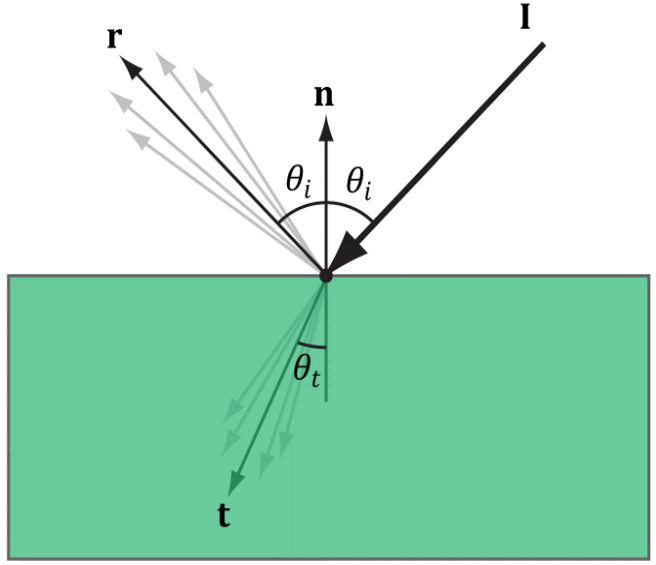
(b) Most objects are not perfectly flat mirrors but have microscopic roughness. This causes the reflected and refracted light to spread about the reflection and refraction vectors.

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*.





(a)



(b)

Specular Lighting

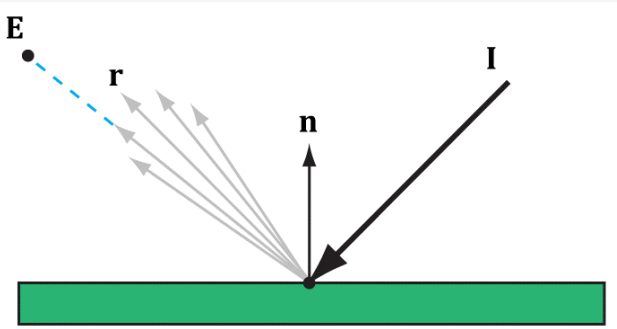
(a) Specular light of a rough surface spreads about the reflection vector r

(b) For opaque objects, the amount of light that reflects off a surface and makes it into the eye is a combination of body reflected (diffuse) light and specular reflection.

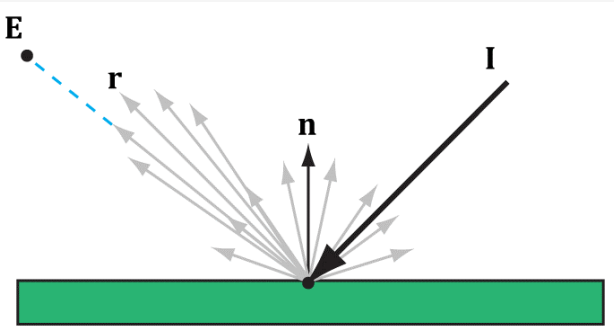
In contrast to diffuse light, specular light might not travel into the eye because it reflects in a specific direction; that is to say, the specular lighting calculation is viewpoint dependent. This means that as the eye moves about the scene, the amount of specular light it receives will change.

(a) Looking down in the pond, reflection is low and refraction high because the angle between L and n is small

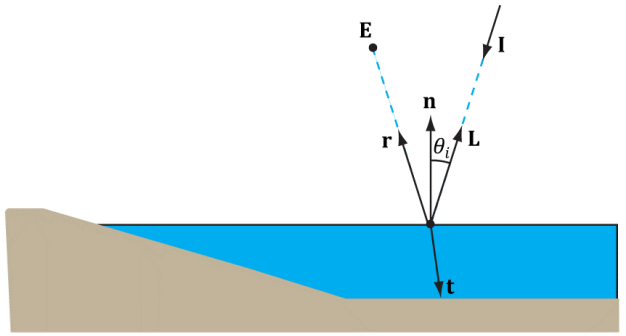
(b) Looking towards the horizon and reflection is high and refraction low because the angle between L and n is closer to 90°



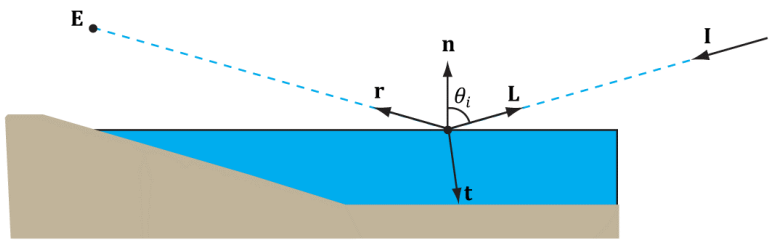
(a)



(b)



(a)



(b)

Fresnel Effect

Consider a flat surface with normal **n** that separates two mediums with different indices of refraction.

Due to the index of refraction discontinuity at the surface, when incoming light strikes the surface some reflects away from the surface and some refracts into the surface.

The *Fresnel equations* mathematically describe the percentage of incoming light that is reflected, $0 \leq \mathbf{R}_F \leq 1$.

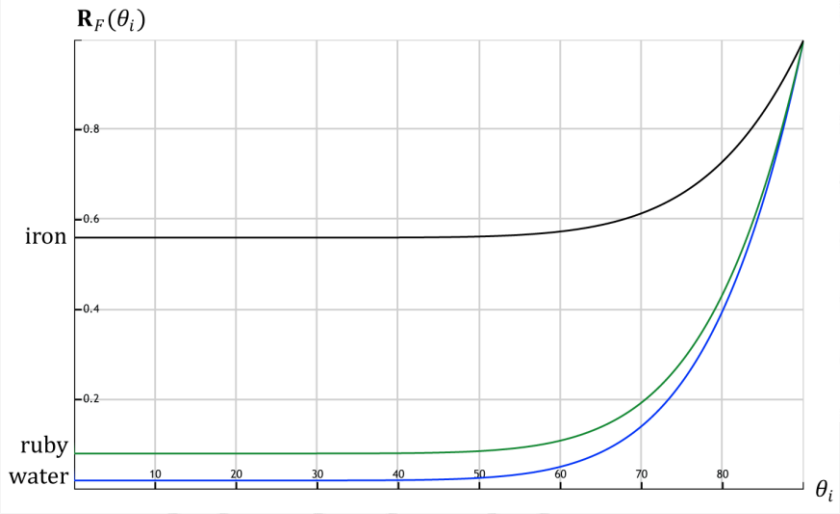
By conservation of energy, if $\mathbf{R}_F$ is the amount of reflected light then $(1 - \mathbf{R}F)$ is the amount of refracted light.

The value $\mathbf{R}_F$ is an RGB vector because the amount of reflection depends on the light color.

Due to their complexity, the full Fresnel equations are not typically used in real time Rendering; instead, the *Schlick approximation* is used.

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

The Schlick approximation plotted for different materials: water, ruby, and iron. The key observation is that the amount of reflection increases as $\theta_i \rightarrow 90°$.



Medium	$\mathbf{R}_F(0°)$
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)

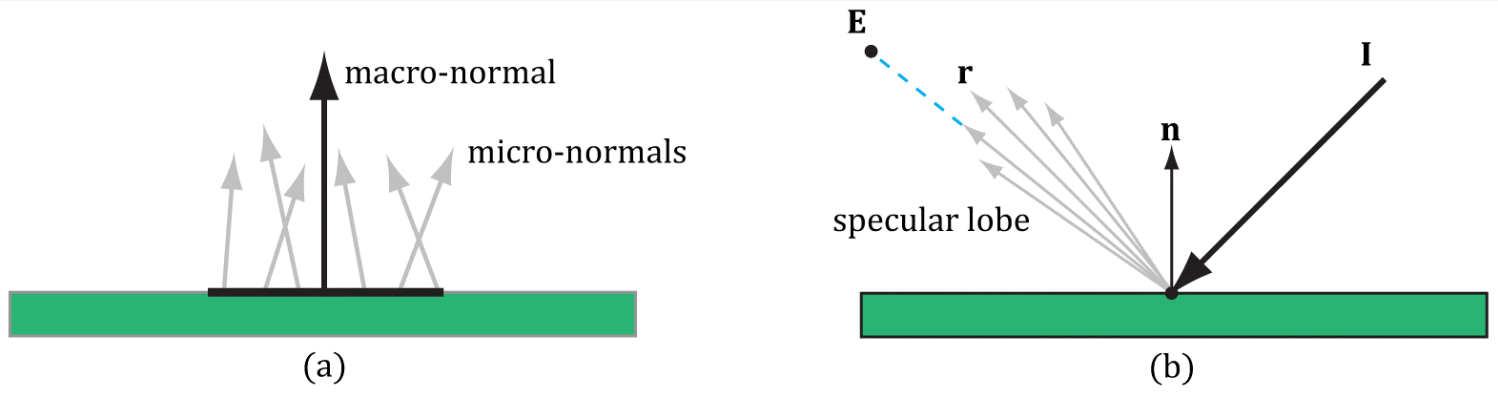
Roughness

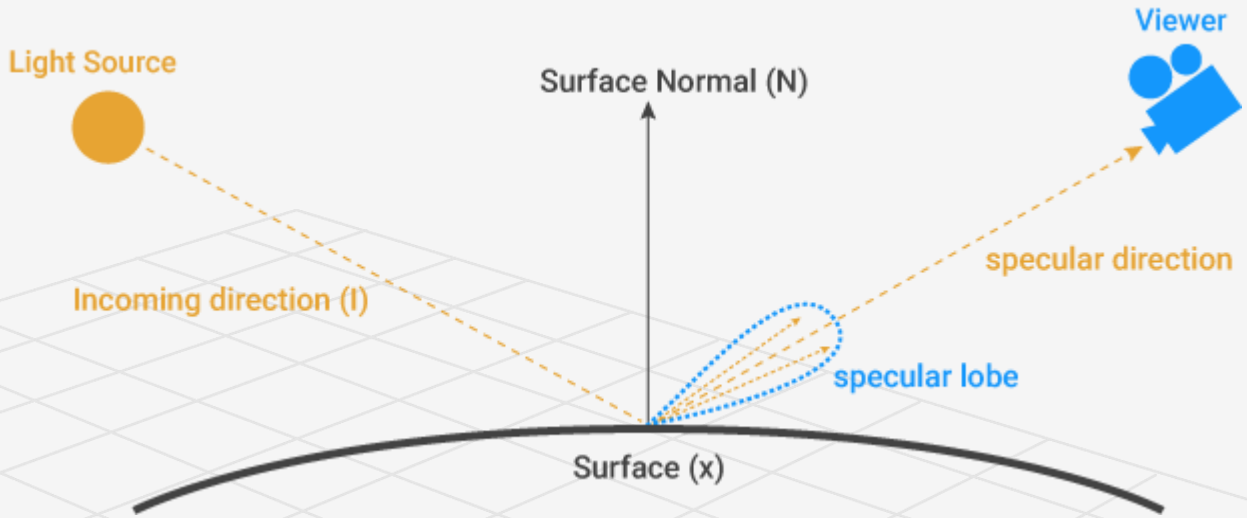
- Reflective objects in the real world tend not to be perfect mirrors

- Even if an object's surface appears flat, at the microscopic level we can think of it as having roughness

- We can think of a perfect mirror as having no roughness and its micro-normals all aim in the same direction as the macro-normal

- As the roughness increases, the direction of the micro-normals diverge from the macro normal, causing the reflected light to spread out into a specular lobe





Modeling the roughness

To model roughness mathematically, we employ the *microfacet* model, where we model the microscopic surface as a collection of tiny flat elements called microfacets;

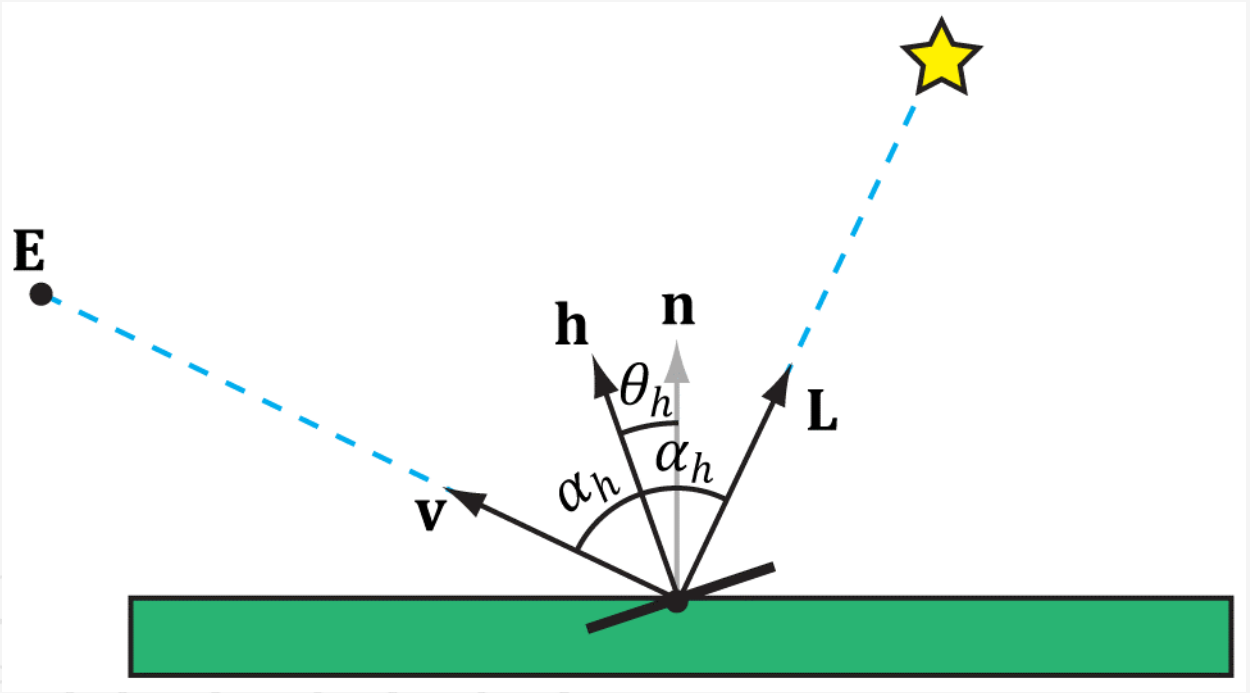
The micro-normals are the normals to the microfacets

For a given view **v** and light vector **L**, we want to know the fraction of microfacets that reflect **L** into **v**; in other words, the fraction of microfacets with normal $\boldsymbol{h} = normalize(\boldsymbol{L} + \boldsymbol{v})$

This will tell us how much light is reflected into the eye from specular reflection—the more microfacets that reflect **L** into **v** the brighter the specular light the eye sees.

The vector **h** is called the ***halfway vector*** as it lies halfway between **L** and **v**.

The angle θ_h between the halfway vector **h** and the macro-normal **n**.



The normalized distribution function

We define the normalized distribution function $\rho(\theta_h) \in [0, 1]$ to denote the fraction of microfacets with normals **h** that make an angle θ_h with the macro-normal **n**.

$\rho(\theta_h)$ achieves its maximum when $\theta_h = 0°$.

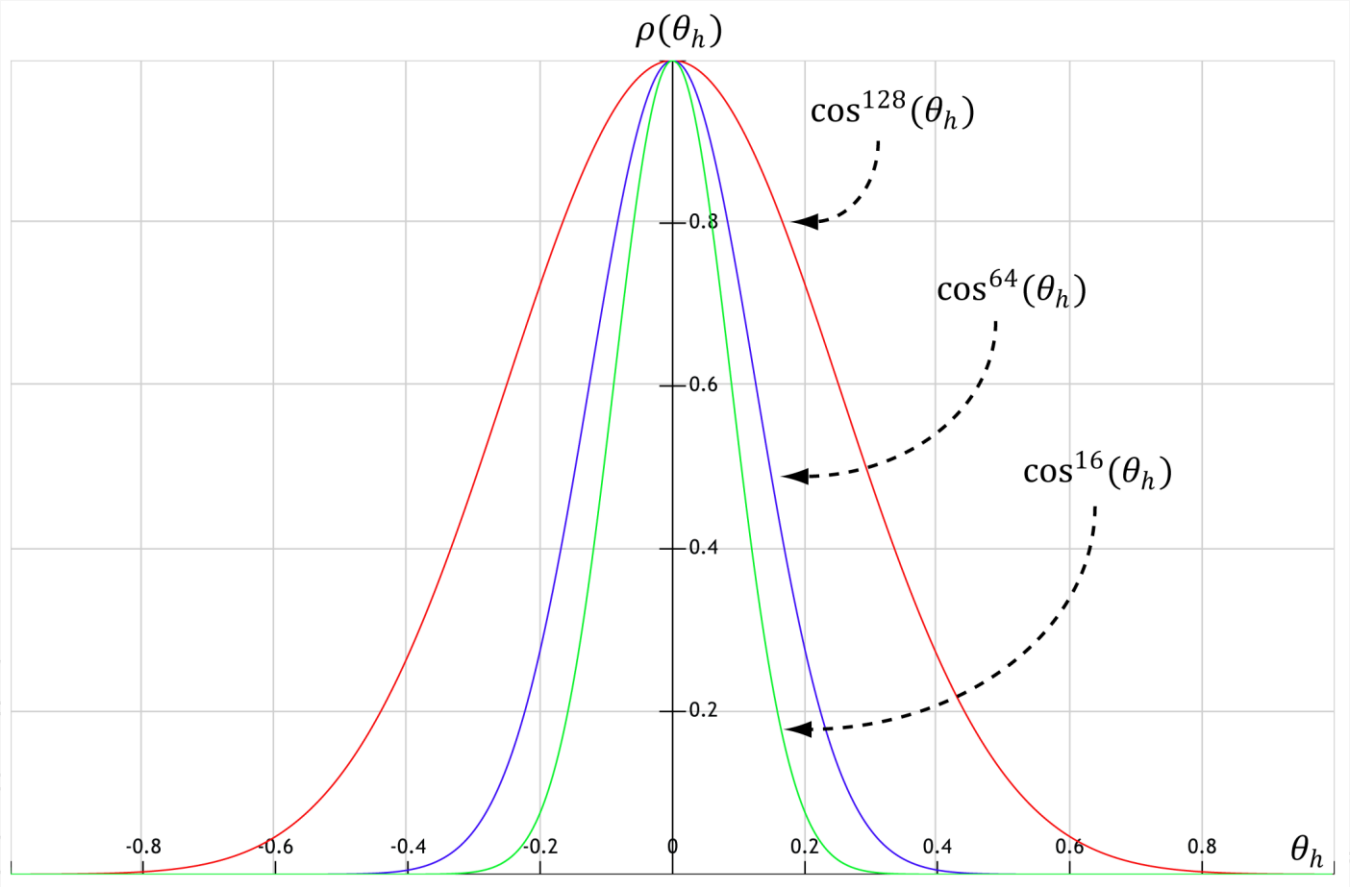
As θ_h increases (as **h** diverges from the micro-normal **n**) we expect the fraction of microfacets with normal **h** to decrease.

A popular controllable function to model $\rho(\theta_h)$ that has the expectations just discussed is:

$$\boldsymbol{\rho(\theta_h) = cos^m(\theta_h) = cos^m(n \cdot h)}$$

Here *m* controls the roughness, which specifies the fraction of microfacets with normals **h** that make an angle θ_h with the macro-normal **n**.

As *m* decreases, the surface becomes rougher, and the microfacet normals increasingly diverge from the macro-normal. As *m* increases, the surface becomes smoother, and the microfacet normals increasingly converge to the macro-normal.



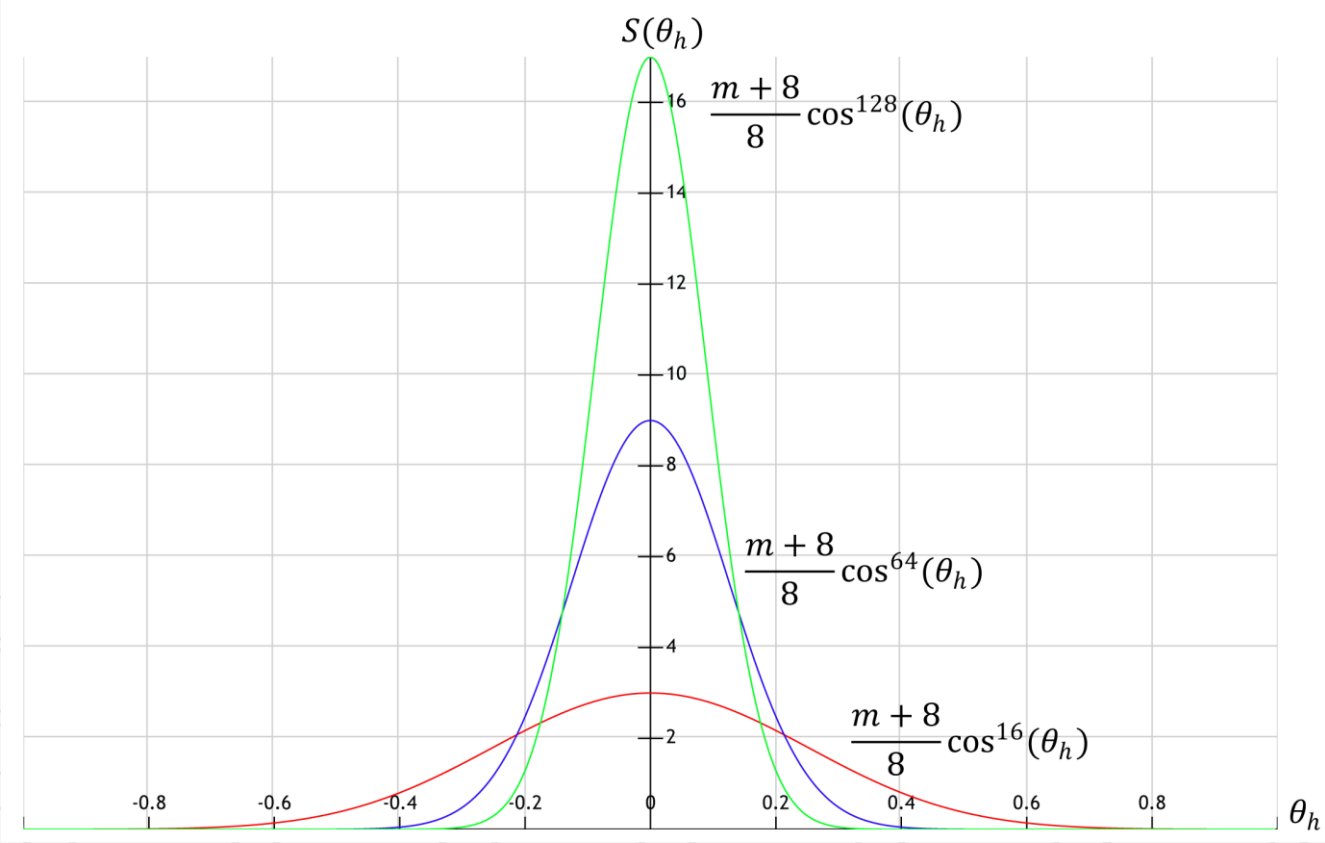
A function to model specular reflection

We can combine $\rho(\theta_h)$ with a normalization factor to obtain a new function that models the amount of specular reflection of light based on roughness.

we have added the $\frac{m+8}{8}$ normalization factor so that light energy is conserved; it is essentially controlling the height of the curve in this figure so that the overall light energy is conserved as the specular lobe widens or narrows with *m*.

For a smaller *m*, the surface is rougher and the specular lobe widens as light energy is more spread out; therefore, we expect the specular highlight to be dimmer since the energy has been spread out.

For a larger *m*, the surface is smoother and the specular lobe is narrower; therefore, we expect the specular highlight to be brighter since the energy has been concentrated.



Lighting Model Recap

- The total light reflected off a surface is a sum of ambient light reflectance, diffuse light reflectance and specular light reflectance

- This leads to the lighting equation our shaders implement:

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

- Ambient Light: c_a models the amount of light reflected off the surface due to indirect light

- Diffuse Light: c_d models light that enters the interior of a medium, scatters around under the surface where some of the light is absorbed and the remaining light scatters back out of the surface

- Specular Light: c_s models the light that is reflected off the surface due to the Fresnel effect and surface roughness

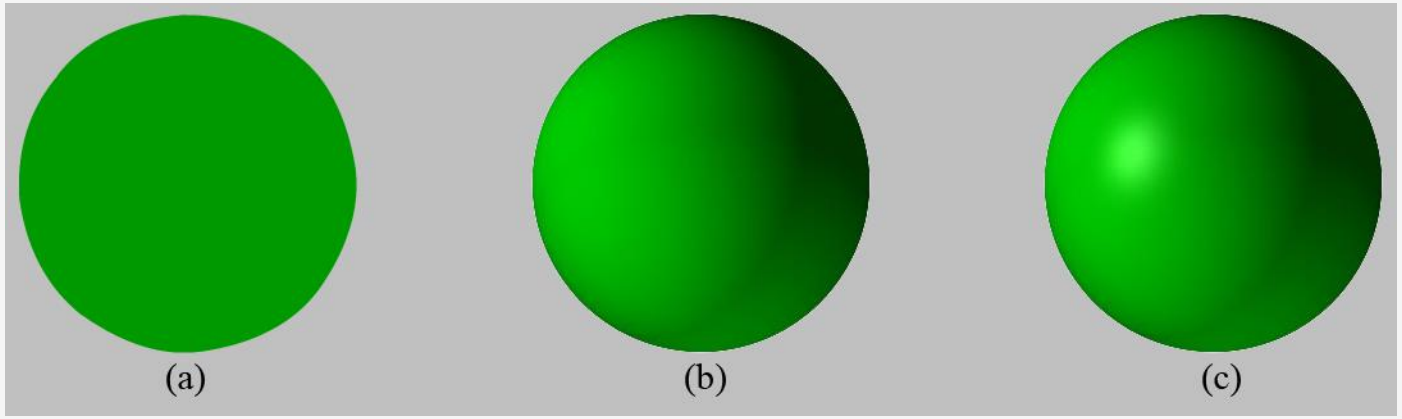
1. **L**: The light vector aims toward the light source.
2. **n**: The surface normal.
3. **h**: The halfway vector lies halfway between the light vector and view vector (vector from surface point being lit to the eye point).
4. $\mathbf{A}_L$: Represent the quantity of incoming ambient light.
5. $\mathbf{B}_L$: Represent the quantity of incoming direct light.
6. $\mathbf{m}_d$: Specifies the amount of incoming light that the surface reflects due to diffuse reflectance.
7. **L**·**n** : Lambert's Cosine Law.
8. α_h: Angle between the half vector **h** and light vector **L**.
9. $\mathbf{R}_F(\alpha_h)$: Specifies the amount of light reflected about **h** into the eye due to the Fresnel effect.
10. m: Controls the surface roughness.
11. $(\mathbf{n}\cdot\mathbf{h})_h$: Specifies the fraction of microfacets with normals **h** that make an angle θ_h with the macro-normal **n**.
12.: $\frac{m+8}{8}$ Normalization factor to model energy conservation in the specular reflection.

Sphere with ambient, diffuse, and specular lighting

(a) Sphere colored with ambient light only, which uniformly brightens it.

(b) Ambient and diffuse lighting combined. There is now a smooth transition from bright to dark due to Lambert's cosine law.

(c) Ambient, diffuse, and specular lighting. The specular lighting yields a specular highlight.



Modeling real-world materials will require a combination of setting realistic values for the DiffuseAlbedo and FresnelR0, and some artistic tweaking.

For example, metal conductors absorb refracted light that enters the interior of the metal, which means metals will not have diffuse reflection (i.e., the DiffuseAlbedo would be zero).

We are not doing 100% physical simulation of lighting, it may give better artistic results to give a low DiffuseAlbedo value rather than zero.

The point is: we will try to use physically realistic material values, but are free to tweak the values as we want if the end result looks better from an artistic point of view.

Materials

Our material structure looks like this, and is defined in *d3dUtil.h*:

```
// Simple struct to represent a material for our demos. A production 3D engine
// would likely create a class hierarchy of Materials.
struct Material
{
// Unique material name for lookup.
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;
// Dirty flag indicating the material has changed and we need to update the constant buffer.
// Because we have a material constant buffer for each FrameResource, we have to apply the
// update to each FrameResource. Thus, when we modify a material we should set
// NumFramesDirty = gNumFrameResources so that each frame resource gets the update.
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();
};
```

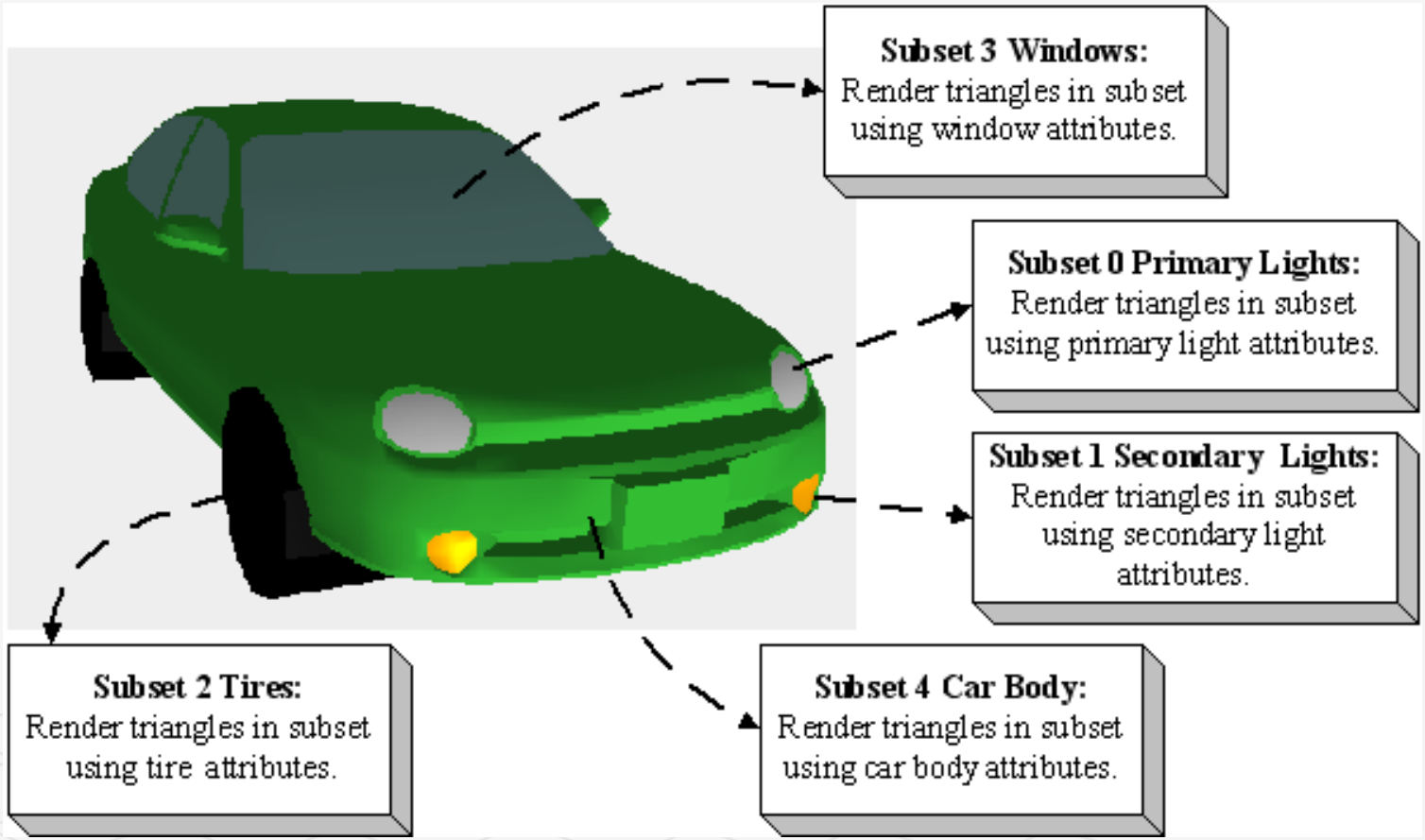
Materials

The material values may vary over the surface; that is, different points on the surface may have different material values.

For example, consider a car model as shown, where the frame, windows, lights, and tires reflect and absorb light differently, and so the material values would need to vary over the car surface.

To implement this variation, one solution might be to specify material values on a per vertex basis. These per vertex materials would then interpolated across the triangle during rasterization, giving us material values for each point on the surface of the triangle mesh.

Note that with our definition of roughness, the shininess of a surface is just the inverse of the roughness: *shininess* = 1 – *roughness* ∈ [0, 1].



LitWavesApp::BuildMaterials

For this chapter, we allow material changes at the draw call frequency. To do this, we define the properties of each unique material and put them in a table:

The above table stores the material data in system memory. In order for the GPU to access the material data in a shader, we need to mirror the relevant data in a constant buffer.

This is not a good water material definition, but we do not have all the rendering tools we need (transparency, environment reflection), so we fake it for now.

Just like we did with per-object constant buffers, we add a constant buffer to each FrameResource that will store the constants for each material (next slide)

```
std::unordered_map<std::string, std::unique_ptr<Material>> mMaterials;
void LitWavesApp::BuildMaterials()
{
auto grass = std::make_unique<Material>();
grass->Name = "grass";
grass->MatCBIndex = 0;
    grass->DiffuseAlbedo = XMFLOAT4(0.2f, 0.6f, 0.2f, 1.0f);
    grass->FresnelR0 = XMFLOAT3(0.01f, 0.01f, 0.01f);
    grass->Roughness = 0.125f;

auto water = std::make_unique<Material>();
water->Name = "water";
water->MatCBIndex = 1;
    water->DiffuseAlbedo = XMFLOAT4(0.0f, 0.2f, 0.6f, 1.0f);
    water->FresnelR0 = XMFLOAT3(0.1f, 0.1f, 0.1f);
    water->Roughness = 0.0f;

mMaterials["grass"] = std::move(grass);
mMaterials["water"] = std::move(water);
}
```

MaterialConstants & FrameResource structs

MaterialConstants structure contains a subset of the Material data that the shaders need for rendering.

```
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();
};
```

```
// Stores the resources needed for the CPU to build the command lists  for a frame.
struct FrameResource
{
public:

    FrameResource(ID3D12Device* device, UINT passCount, UINT objectCount, UINT
          materialCount, UINT waveVertCount);
    FrameResource(const FrameResource& rhs) = delete;
    FrameResource& operator=(const FrameResource& rhs) = delete;
    ~FrameResource();

     Microsoft::WRL::ComPtr<ID3D12CommandAllocator> CmdListAlloc;

     std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
    std::unique_ptr<UploadBuffer<MaterialConstants>> MaterialCB = nullptr;
    std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;

     std::unique_ptr<UploadBuffer<Vertex>> WavesVB = nullptr;

     UINT64 Fence = 0;
};
```

LitWavesApp::UpdateMaterialCBs

In the update function, the material data is then copied to a subregion of the constant buffer whenever it is changed (“dirty”) so that the GPU material constant buffer data is kept up to date with the system memory material data:

```
void LitWavesApp::UpdateMaterialCBs(const GameTimer& gt)
{
auto currMaterialCB = mCurrFrameResource->MaterialCB.get();
for(auto& e : mMaterials)
{
// Only update the cbuffer data if the constants have changed.  If the cbuffer
// data changes, it needs to be updated for each FrameResource.
Material* mat = e.second.get();
if(mat->NumFramesDirty > 0)
{
XMMATRIX matTransform = XMLoadFloat4x4(&mat->MatTransform);

MaterialConstants matConstants;
matConstants.DiffuseAlbedo = mat->DiffuseAlbedo;
matConstants.FresnelR0 = mat->FresnelR0;
matConstants.Roughness = mat->Roughness;

currMaterialCB->CopyData(mat->MatCBIndex, matConstants);

// Next FrameResource need to be updated too.
mat->NumFramesDirty--;}}}
```

LitWavesApp::DrawRenderItems

Each render item contains a pointer to a Material.

Note that multiple render items can refer to the same Material object; for example, multiple render items might use the same “brick” material.

Each Material object has an index that specifies where its constant data is in the material constant buffer.

From this, we can offset to the virtual address of the constant data needed for the render item we are drawing, and set it to the root descriptor that expects the material constant data.

We need normal vectors at each point on the surface of a triangle mesh so that we can determine the angle at which light strikes a point on the mesh surface (for Lambert’s cosine law).

we specify normals at the vertex level.

These vertex normals will be interpolated across the triangle during rasterization.

```
void LitWavesApp::DrawRenderItems(ID3D12GraphicsCommandList* cmdList, const
std::vector<RenderItem*>& ritems)
{
UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));
UINT matCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(MaterialConstants));

auto objectCB = mCurrFrameResource->ObjectCB->Resource();
auto matCB = mCurrFrameResource->MaterialCB->Resource();
for(size_t i = 0; i < ritems.size(); ++i)
{
auto ri = ritems[i];

cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB->GetGPUVirtualAddress() + ri-
>ObjCBIndex*objCBByteSize;
D3D12_GPU_VIRTUAL_ADDRESS matCBAddress = matCB->GetGPUVirtualAddress() + ri->Mat-
>MatCBIndex*matCBByteSize;

cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);
cmdList->SetGraphicsRootConstantBufferView(1, matCBAddress);

cmdList->DrawIndexedInstanced(ri->IndexCount, 1, ri->StartIndexLocation, ri-
>BaseVertexLocation, 0);
}
}
```

Parallel Lights

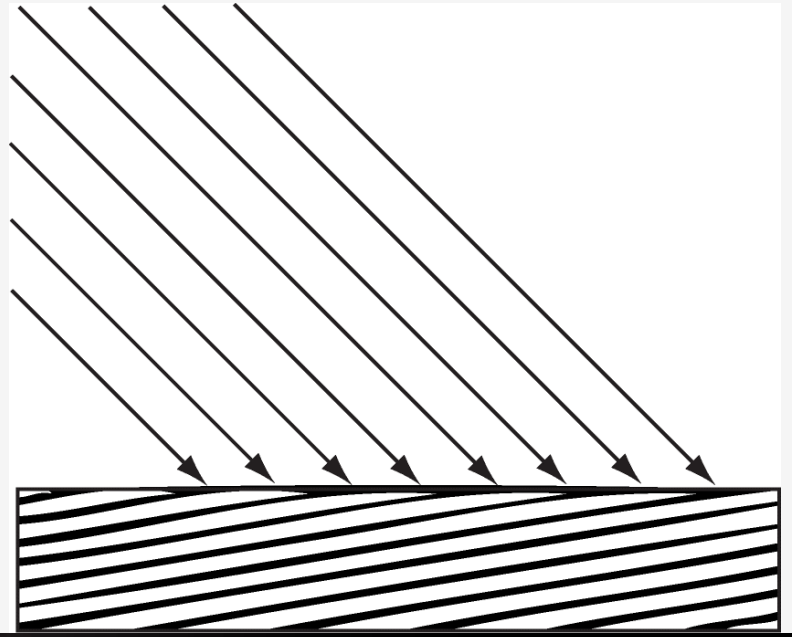
▪ A parallel light (or directional light) approximates a light source that is very far away

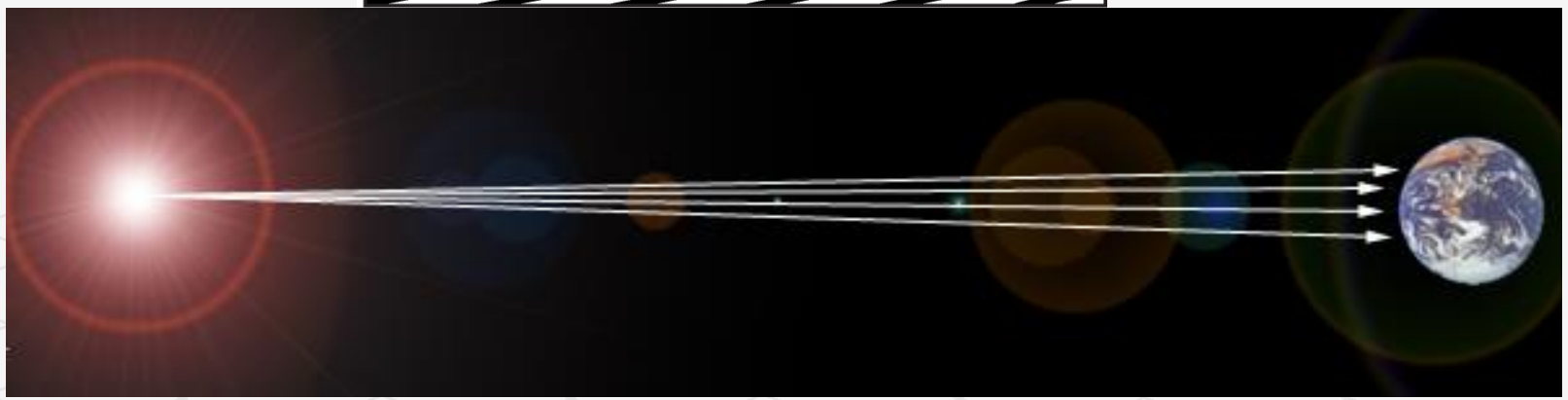
▪ We can approximate all incoming light rays as parallel to each other

▪ Because the light source is very far away, we can ignore the effects of distance and just specify the light intensity where the light strikes the scene

A parallel light source is defined by a vector, which specifies the direction the light rays travel. Because the light rays are parallel, they all use the same direction vector.

The light vector, aims in the opposite direction the light rays travel. A common example of a light source that can accurately be modeled as a directional light is the sun

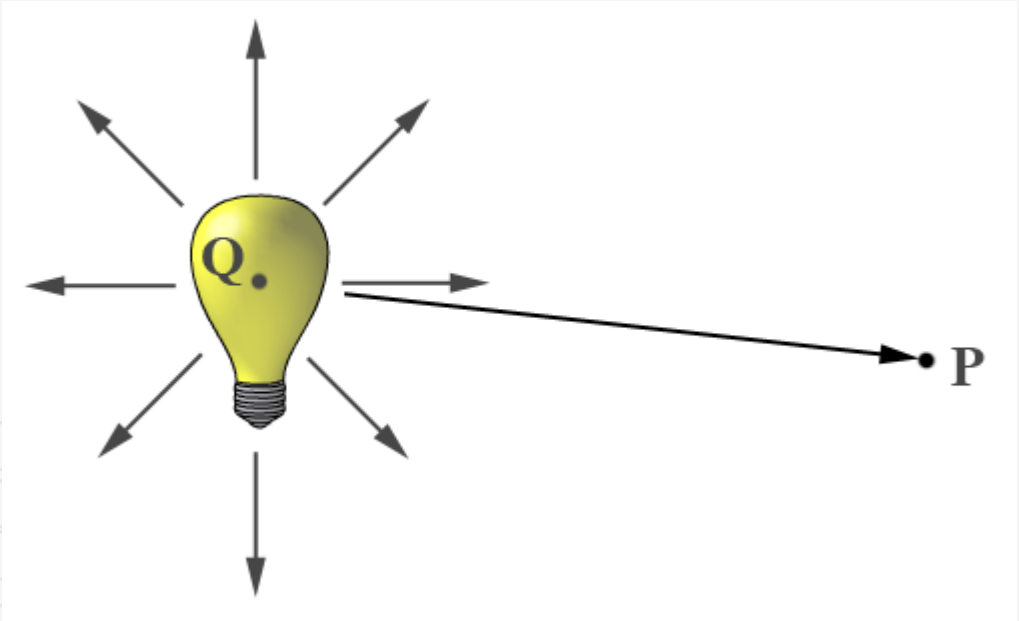




POINT LIGHTS

A good physical example of a point light is a lightbulb

- It radiates spherically in all directions
- The only difference between point lights and parallel lights is how the light vector is computed—it varies from point to point for point lights, but remains constant for parallel lights
- For an arbitrary point P, there exists a light ray originating from the point light position Q traveling toward the point
- As usual, we define the light vector to go in the opposite direction; that is, the direction from the point P to the point light source Q



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

Attenuation

Physically, light intensity weakens as a function of distance based on the inverse squared law. The light intensity at a point a distance *d* away from the light source is given by:

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

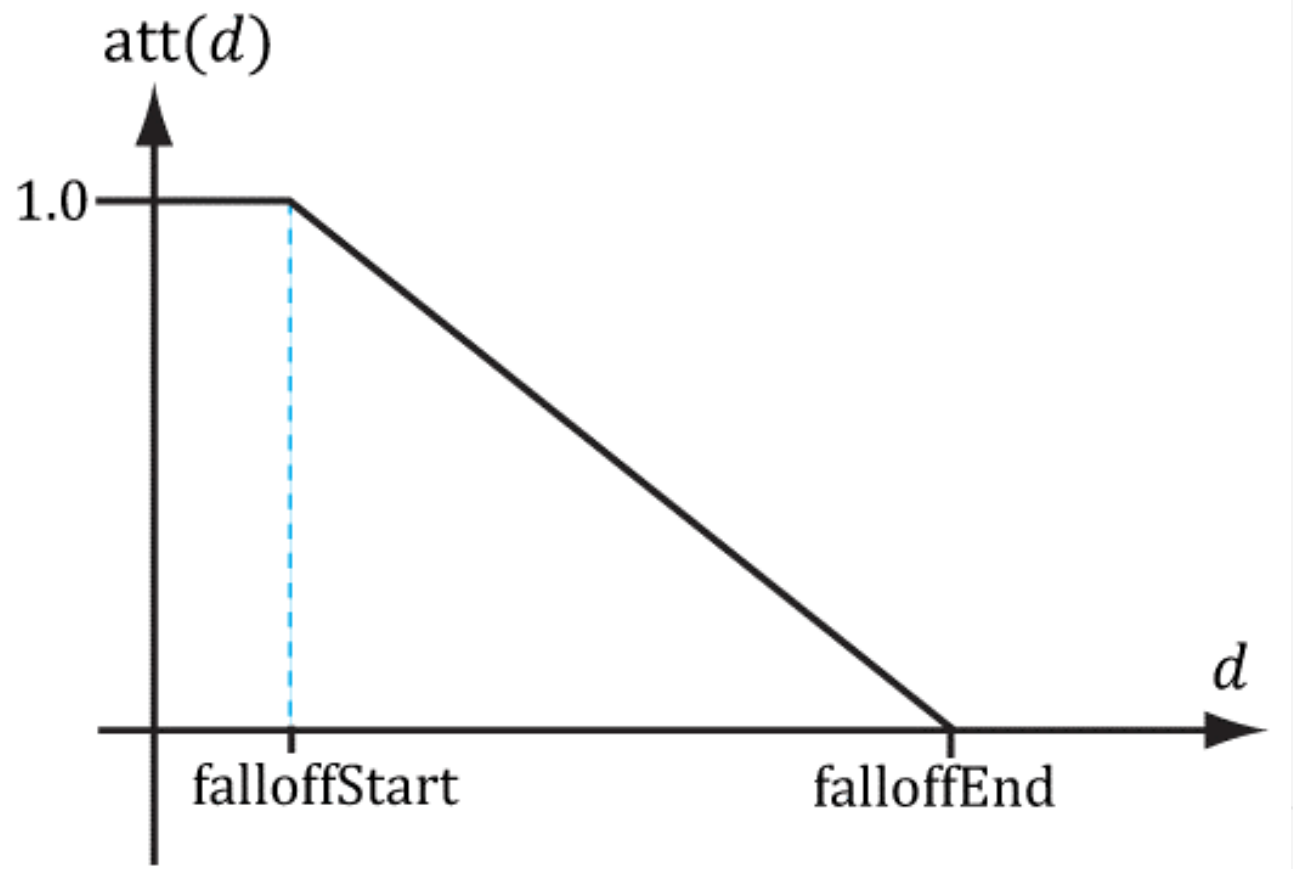
where I_0 is the light intensity at a distance $d = 1$ from the light source. This works well if you set up physically based light values and use HDR (high dynamic range) lighting and tone mapping.

An easier formula we will use in our demos, is a linear falloff function. The **saturate function** clamps the argument to the range [0, 1]:

$$\boldsymbol{Att(d) = saturate\left(\frac{falloffEnd - d}{falloffEnd - falloffStart}\right)}$$

The attenuation factor that scales the light value stays at full strength (1.0) until the distance d reaches

falloffStart, it then linearly decays to 0.0 as the distance reaches *falloffEnd*

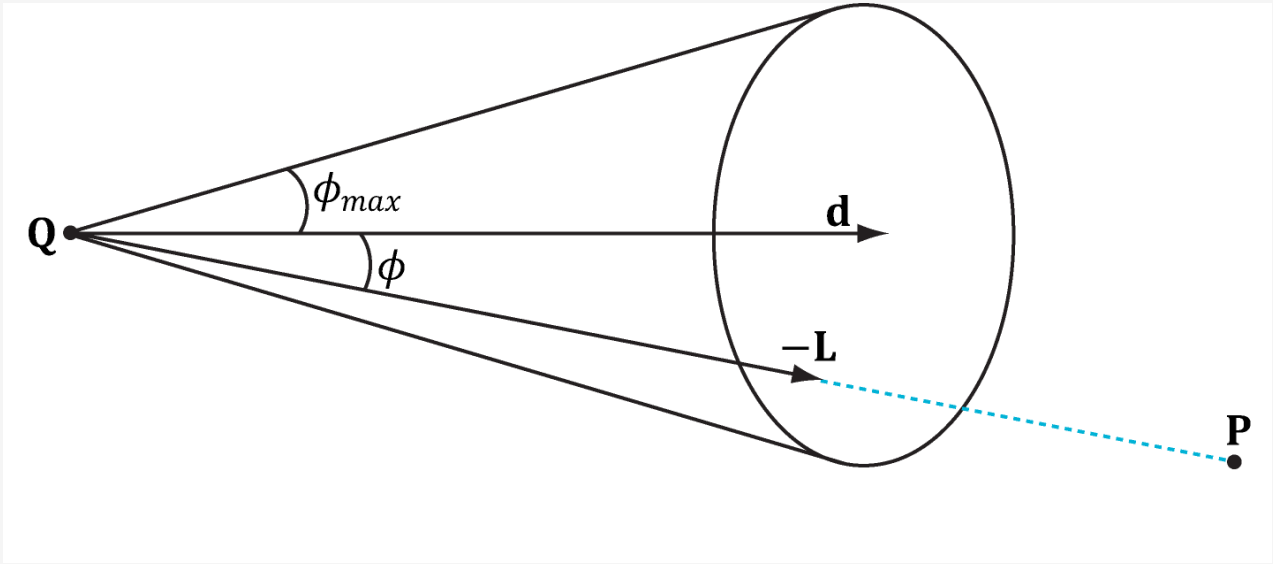


SPOTLIGHTS

A good physical example of a spotlight is a flashlight.

Essentially, a spotlight has a position **Q**, is aimed in a direction **d**, and radiates light through a cone.

Figure shows a spotlight that has a position **Q**, is aimed in a direction **d**, and radiates light through a cone with angle φ_{max}.



To implement a spotlight, we begin as we do with a point light: the light vector is given by.

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

P is inside the spotlight's cone (and therefore receives light) if and only if the angle ϕ between $-\mathbf{L}$ and **d** is smaller than the cone angle ϕ_{max}

All the light in the spotlight's cone should not be of equal intensity.

So how do we control the intensity falloff as a function of ϕ?

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

This gives us what we want: the intensity smoothly fades as ϕ increases; additionally, by altering the exponent *s*, we can indirectly control ϕ_{max} (the angle the intensity drops to 0); We can shrink or expand the spotlight cone by varying *s*. For example, if we set $s = 8$, the cone has approximately a 45° half angle.

Spot lights vs. Point lights vs. Directional lights

The spotlight equation is just like the following directional light equation, except that we multiply the light source value $\mathbf{B}_L$ by both the attenuation factor att(d) and the spotlight factor k_{spot} to scale the light intensity based on where the point is with respect to the spotlight cone.

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

A spotlight is more expensive than a point light because we need to compute the additional k_{spot} factor and multiply by it.

A point light is more expensive than a directional light because the distance d needs to be computed (this is actually pretty expensive because distance involves a square root operation), and we need to compute and multiply by the attenuation factor.

To summarize, directional lights are the least expensive light source, followed by point lights, followed by spotlights being the most expensive light source.

LIGHTING IMPLEMENTATION

In *d3dUtil.h*, we define the following structure to support lights.

This structure can represent directional, point, or spot lights.

Depending on the light type, some values will not be used; for example, a point light does not use the Direction data member.

```
struct Light

{

DirectX::XMFLOAT3 Strength = { 0.5f, 0.5f, 0.5f }; // Light color

float FalloffStart = 1.0f; // point/spot light only

// directional/spot light only

DirectX::XMFLOAT3 Direction = { 0.0f, -1.0f, 0.0f };

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

};
```

LightingUtils.hlsl

In *d3dUtil.h*, we define the following structure to support lights.

This structure can represent directional, point, or spot lights.

However, depending on the light type, some values will not be used; for example, a point light does not use the Direction data member.

```
struct Light {
float3 Strength; // Light color
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
};
```

Structure Packing

Packing rules dictate how tightly data can be arranged when it is stored. HLSL implements packing rules for VS output data, GS input and output data, and PS input and output data. (Data is not packed for VS inputs because the IA stage cannot unpack data.)

If the C++ and HLSL structure layouts do not match, then we will get rendering bugs when we upload data from the CPU to GPU constant buffers using memcpy.

The order of data members listed in the Light structure (and also the MaterialConstants structure) is not arbitrary. They are cognizant of the HLSL structure packing rules.

In HLSL, structure padding occurs so that elements are packed into 4D vectors.

This structure gets nicely packed into three 4D vectors like this:

```
vector 1: (Strength.x, Strength.y, Strength.z, FalloffStart)
vector 2: (Direction.x, Direction.y, Direction.z, FalloffEnd)
vector 3: (Position.x, Position.y, Position.z, SpotPower)
```

Common Helper Functions

The three functions defined in *LightingUtils.hlsl*, contain code that is common to more than one type of light, and therefore we define in helper functions.

1. CalcAttenuation: Implements a linear attenuation factor, which applies to point lights and spot lights.

`saturate :` Clamps the specified value within the range of 0 to 1.

2. SchlickFresnel: The Schlick approximation to the Fresnel equations;

It approximates the percentage of light reflected off a surface with normal **n** based on the angle between the light vector **L** and surface normal **n** due to the Fresnel effect.

R0 is the reflection coefficient for light incoming parallel to the normal

$$R0 = \left(\frac{n-1}{n+1}\right)^2$$

where n is the index of refraction.

```
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;
}
```

Blinn-Phong Reflection Model

3. BlinnPhong: Computes the amount of light reflected into the eye; it is the sum of diffuse reflectance and specular reflectance.

In LDR rendering, very bright light sources in a scene (such as the sun) are capped at 1.0. When this light is reflected the result must then be less than or equal to 1.0. However, in HDR rendering, very bright light sources can exceed the 1.0 brightness to simulate their actual values. This allows reflections off surfaces to maintain realistic brightness for bright light sources.

When two vectors are multiplied with operator*, the multiplication is done component-wise.

In Phong shading, you must continually recalculate the dot product R.V between a viewer (V) and the beam from a light-source (L) reflected (R) on a surface.

If, instead, you calculate a halfway vector between the viewer and light-source vectors, R.V can be replaced with N.H, where N is the normalized surface normal.

```
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;

    // Our spec formula goes outside [0,1] range, but we are
    // doing LDR rendering.  So scale it down a bit.
    specAlbedo = specAlbedo / (specAlbedo + 1.0f);

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

Implementing Directional Lights

Given the eye position **E** and given a point **p** on a surface visible to the eye with surface normal **n**, and material properties.

`ComputeDirectionalLight` is a HLSL function outputs the amount of light, from a directional light source, that reflects into the to-eye direction **v** =normalize (**E** – **p**).

This function will be called in a pixel shader to determine the color of the pixel based on lighting.

```
float4 directLight =
ComputeLighting(gLights, mat, pin.PosW,
pin.NormalW, toEyeW, shadowFactor);
```

```
float3 ComputeDirectionalLight(Light L, Material mat, float3 normal, float3 toEye)
{
    // The light vector aims opposite the direction the light rays travel.
    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);
}

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
```

Implementing Point Lights

Given the eye position **E** and given a point **p** on a surface visible to the eye with surface normal **n**, and material properties, the following HLSL function outputs the amount of light, from a point light source, that reflects into the to-eye direction

v = normalize (**E** − **p**) .

This function will be called in a pixel shader to determine the color of the pixel based on lighting.

```
float3 ComputePointLight(Light L, Material mat, float3 pos, float3 normal, float3 toEye)
{
    // The vector from the surface to the light.
    float3 lightVec = L.Position - pos;

    // The distance from surface to light.
    float d = length(lightVec);

    // Range test.
    if(d > L.FalloffEnd)
        return 0.0f;

    // Normalize the light vector.
    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

Given the eye position **E** and given a point **p** on a surface visible to the eye with surface normal **n**, and material properties, the following HLSL function outputs the amount of light, from a spot light source, that reflects into the to-eye direction

v = normalize (**E** − **p**).

This function will be called in a pixel shader to determine the color of the pixel based on lighting.

```
float3 ComputeSpotLight(Light L, Material mat, float3 pos, float3 normal, float3 toEye)
{
    // The vector from the surface to the light.
    float3 lightVec = L.Position - pos;

    // The distance from surface to light.
    float d = length(lightVec);

    // Range test.
    if(d > L.FalloffEnd)
        return 0.0f;

    // Normalize the light vector.
    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;

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

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

Accumulating Multiple Lights

Lighting is additive, so supporting multiple lights in a scene simply means we need to iterate over each light source and sum its contribution to the point/pixel we are evaluating the lighting of.

Our sample framework supports up to sixteen total lights.

```
#define MaxLights 16
```

Directional lights must come first in the light array, point lights come second, and spot lights come last.

The shadowFactor *parameter will not be used until later. So for now, we just set this to the vector (1, 1, 1), which makes the shadow factor have no effect in the equation.*

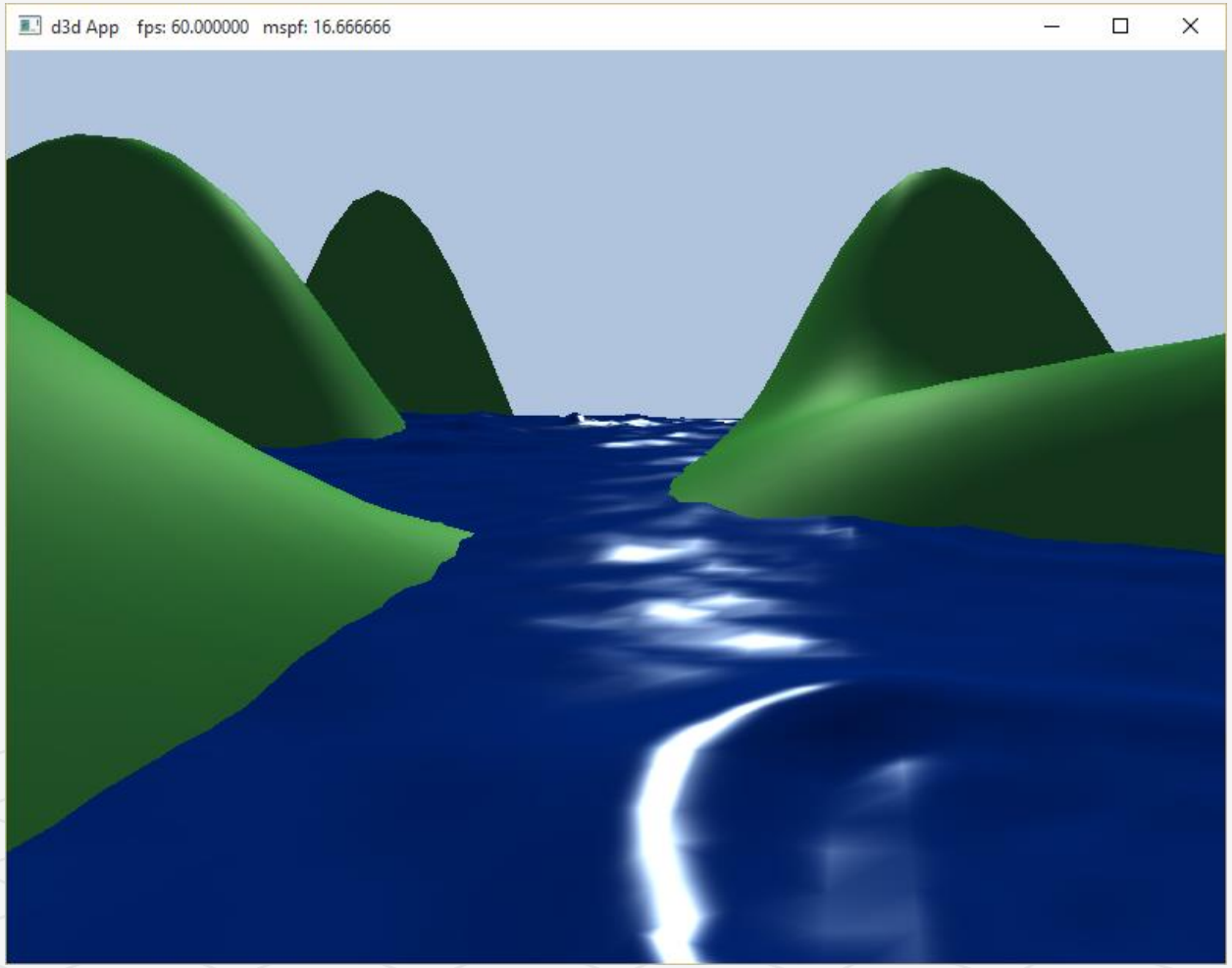
```
float4 ComputeLighting(Light gLights[MaxLights], Material mat,
                       float3 pos, float3 normal, float3 toEye, float3 shadowFactor)
{
    float3 result = 0.0f;

    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);
}
```

LIGHTING DEMO

The lighting demo builds off the “Waves” demo from the previous week.

It uses one directional light to represent the sun.

The user can rotate the sun position using the left, right, up, and down arrow keys.



Vertex Format

Lighting calculations require a surface normal. We define normals at the vertex level;

These normals are then interpolated across the pixels of a triangle so that we may do the lighting calculations per pixel.

We no longer specify a vertex color. Instead, pixel colors are generated by applying the lighting equation for each pixel.

To support vertex normals we modify our vertex structures like so:

When we add a new vertex format, we need to describe it with a new input layout description:

```
// FrameResource.h
struct Vertex
{
    DirectX::XMFLOAT3 Pos;
    DirectX::XMFLOAT3 Normal;
};

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

void LitWavesApp::BuildShadersAndInputLayout()
{
mShaders["standardVS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", nullptr,
"VS", "vs_5_0");
mShaders["opaquePS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", nullptr,
"PS", "ps_5_0");

    mInputLayout =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
        { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
    };
}
```

Normal Computation

The shape functions in GeometryGenerator already create data with vertex normals, so we are all set there. However, because we modify the heights of the grid in this demo to make it look like terrain, we need to generate the normal vectors for the terrain ourselves.

Because our terrain surface is given by a function $y = f(x, z)$, we can compute the normal vectors directly using calculus, rather than the normal averaging technique.

To do this, for each point on the surface, we form two tangent vectors in the $+x$- and $+z$- directions by taking the partial derivatives. These two vectors lie in the tangent plane of the surface point. Taking the cross product then gives the normal vector:

The function we used to generate the land mesh is: $\boldsymbol{f(x, z)}$

The surface normal at a surface point $(x, f(x, z), z)$ is given by: $\boldsymbol{n(x, z)}$

This surface normal is not of unit length, so it needs to be normalized before lighting calculations.

$$\mathbf{T}_x = \left(1, \frac{\partial f}{\partial x}, 0\right)$$

$$\mathbf{T}_z = \left(0, \frac{\partial f}{\partial z}, 1\right)$$

$$\mathbf{n} = \mathbf{T}_z \times \mathbf{T}_x = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 0 & \frac{\partial f}{\partial z} & 1 \\ 1 & \frac{\partial f}{\partial x} & 0 \end{vmatrix}$$

$$= \left(\begin{vmatrix} \frac{\partial f}{\partial z} & 1 \\ \frac{\partial f}{\partial x} & 0 \end{vmatrix}, -\begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix}, \begin{vmatrix} 0 & \frac{\partial f}{\partial z} \\ 1 & \frac{\partial f}{\partial x} \end{vmatrix} \right)$$

$$= \left(-\frac{\partial f}{\partial x}, 1, -\frac{\partial f}{\partial z} \right)$$

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

$$\frac{\partial f}{\partial x} = 0.03z \cdot \cos(0.1x) + 0.3\cos(0.1z)$$

$$\frac{\partial f}{\partial z} = 0.3\sin(0.1x) - 0.03x \cdot \sin(0.1z)$$

$$\mathbf{n}(x,z) = \left(-\frac{\partial f}{\partial x}, 1, -\frac{\partial f}{\partial z} \right) = \begin{bmatrix} -0.03z \cdot \cos(0.1x) - 0.3\cos(0.1z) \\ 1 \\ -0.3\sin(0.1x) + 0.03x \cdot \sin(0.1z) \end{bmatrix}^T$$

LitWavesApp::GetHillsNormal

The normal vectors for the water surface are done in a similar way, except that we do not have a formula for the water.

However, tangent vectors at each vertex point can be approximated using a finite difference scheme.

```
float LitWavesApp::GetHillsHeight(float x, float z)const
{
    return 0.3f*(z*sinf(0.1f*x) + x*cosf(0.1f*z));
}


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;
}
```

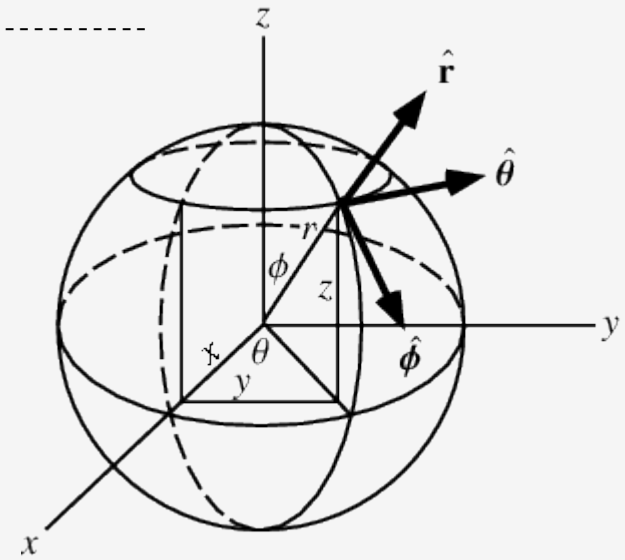
Updating the Light Direction

Our array of Lights is put in the per-pass constant buffer. So every frame, we need to calculate the new light direction from the sun and set it to the per-pass constant buffer.

We track the sun position in spherical coordinates (ρ, θ, ϕ) (radial, azimuthal, polar), but the radius ρ does not matter, because we assume the sun is infinitely far away. In particular, we just use ρ = 1 so that it lies on the unit sphere and interpret (1, θ, ϕ) as the direction towards the sun.

The direction of the light is just the negative of the direction towards the sun. Here is the relevant code for updating the sun.

```
cbuffer cbPass : register(b2)
{
….
Light gLights[MaxLights];
};
--------------------------------------------------------------------
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);
}
--------------------------------------------------------------------

void LitWavesApp::UpdateMainPassCB(const GameTimer& gt)
{
…...
XMVECTOR lightDir = -MathHelper::SphericalToCartesian(1.0f, mSunTheta, mSunPhi);

XMStoreFloat3(&mMainPassCB.Lights[0].Direction, lightDir);
mMainPassCB.Lights[0].Strength = { 1.0f, 1.0f, 0.9f };
-……
```



Update to Root Signature

Lighting introduces a new material constant buffer to our shader programs.

To support this, we need to update our root signature to support an additional constant buffer.

As with per-object constant buffers, we use a root descriptor for the material constant buffer to support binding a constant buffer directly rather than going through a descriptor heap.

```
void LitWavesApp::BuildRootSignature()
{
    // Root parameter can be a table, root descriptor or root constants.
    CD3DX12_ROOT_PARAMETER slotRootParameter[3];

    // Create root CBV.
    slotRootParameter[0].InitAsConstantBufferView(0);
    slotRootParameter[1].InitAsConstantBufferView(1);
    slotRootParameter[2].InitAsConstantBufferView(2);

    // A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(3, slotRootParameter, 0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

     ComPtr<ID3DBlob> serializedRootSig = nullptr;
    ComPtr<ID3DBlob> errorBlob = nullptr;
    HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc,
           D3D_ROOT_SIGNATURE_VERSION_1,
        serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());

    if(errorBlob != nullptr)
    {
        ::OutputDebugStringA((char*)errorBlob->GetBufferPointer());
    }
    ThrowIfFailed(hr);

    ThrowIfFailed(md3dDevice->CreateRootSignature( 0,
        serializedRootSig->GetBufferPointer(),
        serializedRootSig->GetBufferSize(),
        IID_PPV_ARGS(mRootSignature.GetAddressOf())));
}
```

Texture Mapping

1. To learn how to specify the part of a texture that gets mapped to a triangle.

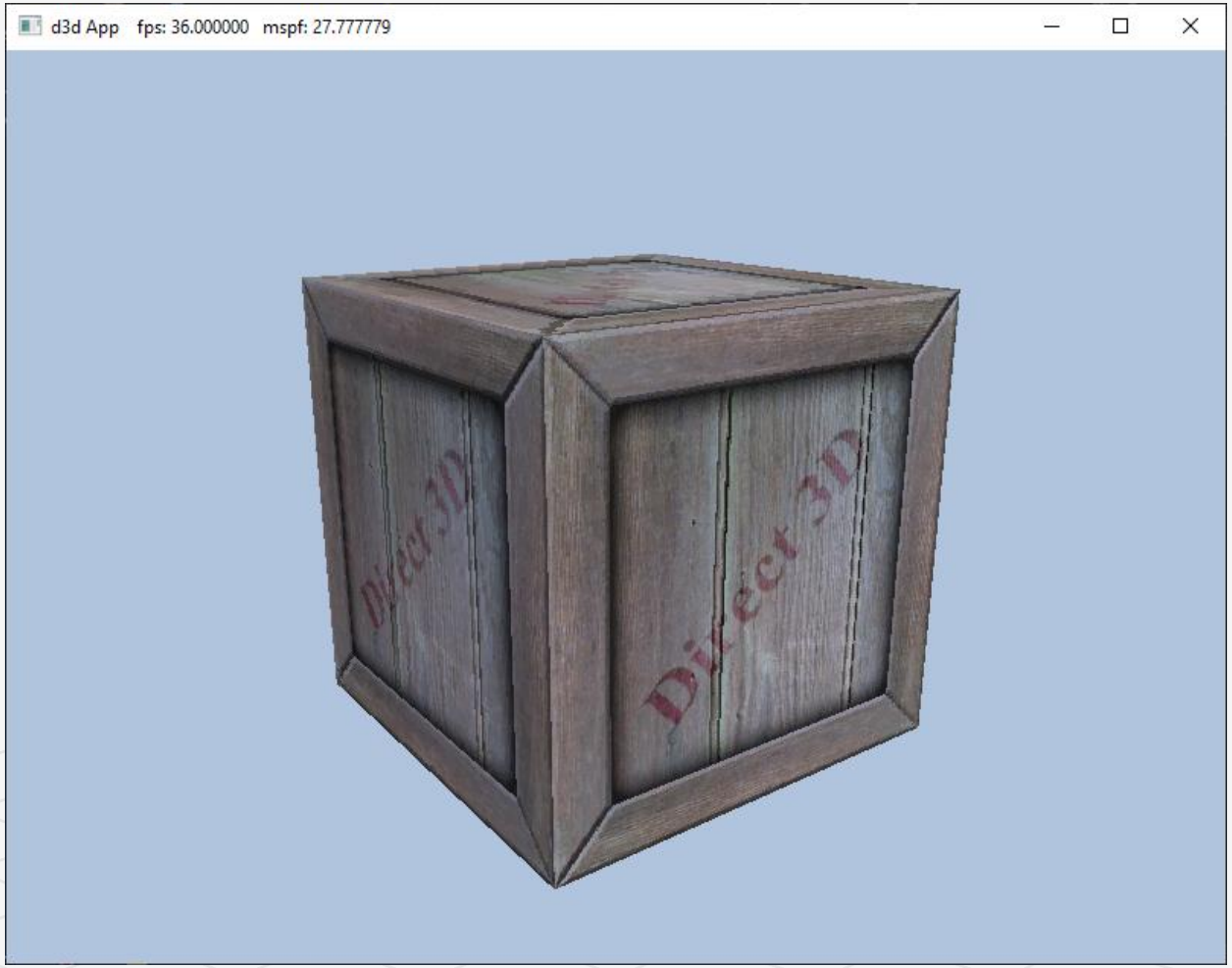
2. To find out how to create and enable textures.

3. To learn how textures can be filtered to create a smoother image.

4. To discover how to tile a texture several times with address modes.

5. To find out how multiple textures can be combined to create new textures and special effects.

6. To learn how to create some basic effects via texture animation.



Texture and Resource Recap

we have already been using textures:

The depth buffer and back buffer are 2D texture objects represented by the ID3D12Resource interface with the D3D12_RESOURCE_DESC::Dimension of D3D12_RESOURCE_DIMENSION_TEXTURE2D.

```
ID3D12Resource* CurrentBackBuffer()const;
 Microsoft::WRL::ComPtr<ID3D12Resource>
mDepthStencilBuffer;
// Create the depth/stencil buffer and view.
    D3D12_RESOURCE_DESC depthStencilDesc;
    depthStencilDesc.Dimension =
D3D12_RESOURCE_DIMENSION_TEXTURE2D;
```

A 1D texture (D3D12_RESOURCE_DIMENSION_TEXTURE1D) is like a 1D array of data elements, and a 3D texture (D3D12_RESOURCE_DIMENSION_TEXTURE3D) is like a 3D array of data elements. The 1D, 2D, and 3D texture interfaces are all represented by the generic ID3D12Resource.

Textures are different than buffer resources, which just store arrays of data; textures can have mipmap levels, and the GPU can do special operations on them, such as apply filters and multisampling. Because of these special operations that are supported for texture resources, they are limited to certain kind of data formats, whereas buffer resources can store arbitrary data. Example formats are:

- DXGI_FORMAT_R32G32B32_FLOAT: Each element has three 32-bit floating point components
- DXGI_FORMAT_R16G16B16A16_UNORM: Each element has four 16-bit components (0, 1 range)
- DXGI_FORMAT_R32G32_UINT: Each element has two 32-bit unsigned integer components
- DXGI_FORMAT_R8G8B8A8_UNORM: Each element has four 8-bit unsigned components (0, 1 range)
- DXGI_FORMAT_R8G8B8A8_SNORM: Each element has four 8-bit signed components (-1, 1 range)
- DXGI_FORMAT_R8G8B8A8_SINT: Each element has four 8-bit signed integer components (−128, 127 range)
- DXGI_FORMAT_R8G8B8A8_UINT: Each element has four 8-bit unsigned integer components (0, 255 range)

Binding a texture

A texture can be bound to different stages of the rendering pipeline;

Use a texture as a render target (i.e., Direct3D draws into the texture)

Use a texture as a shader resource (i.e., the texture will be sampled in a shader).

A texture can also be used as both a render target and as a shader resource, but not at the same time.

Rendering to a texture and then using it as a shader resource, a method called *render-to-texture*

For a texture to be used as both a render target and a shader resource, we would need to create two descriptors to that texture resource: 1) one that lives in a render target heap (i.e., D3D12_DESCRIPTOR_HEAP_TYPE_RTV) and 2) one that lives in a shader resource heap (i.e., D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV). Then the resource can be bound as a render target or bound as a shader input to a root parameter in the root signature.

```
// Bind as render target.
CD3DX12_CPU_DESCRIPTOR_HANDLE rtv = ...;
CD3DX12_CPU_DESCRIPTOR_HANDLE dsv = ...;
cmdList->OMSetRenderTargets(1, &rtv, true, &dsv);

// Bind as shader input to root parameter.
CD3DX12_GPU_DESCRIPTOR_HANDLE tex = ...;
cmdList->SetGraphicsRootDescriptorTable(rootParamIndex, tex);
```

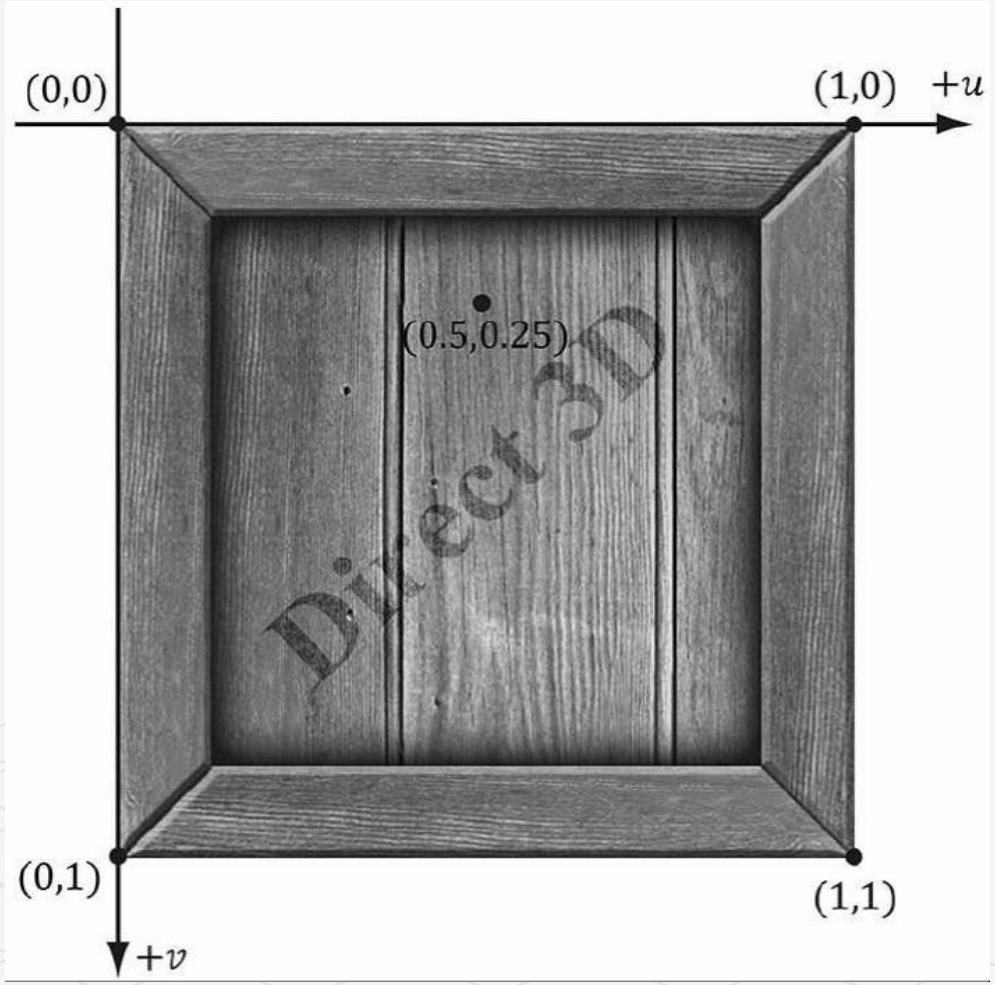
TEXTURE COORDINATES

Direct3D uses a texture coordinate system that consists of a u-axis that runs horizontally to the image and a v-axis that runs vertically to the image

▪ The coordinates, (u, v) such that $0 \leq u, v \leq 1$, identify an element on the texture called a texel

Notice that the *v*-axis is positive in the "down" direction

Also, notice the normalized coordinate interval, [0, 1], which is used because it gives Direct3D a dimension independent range to work with



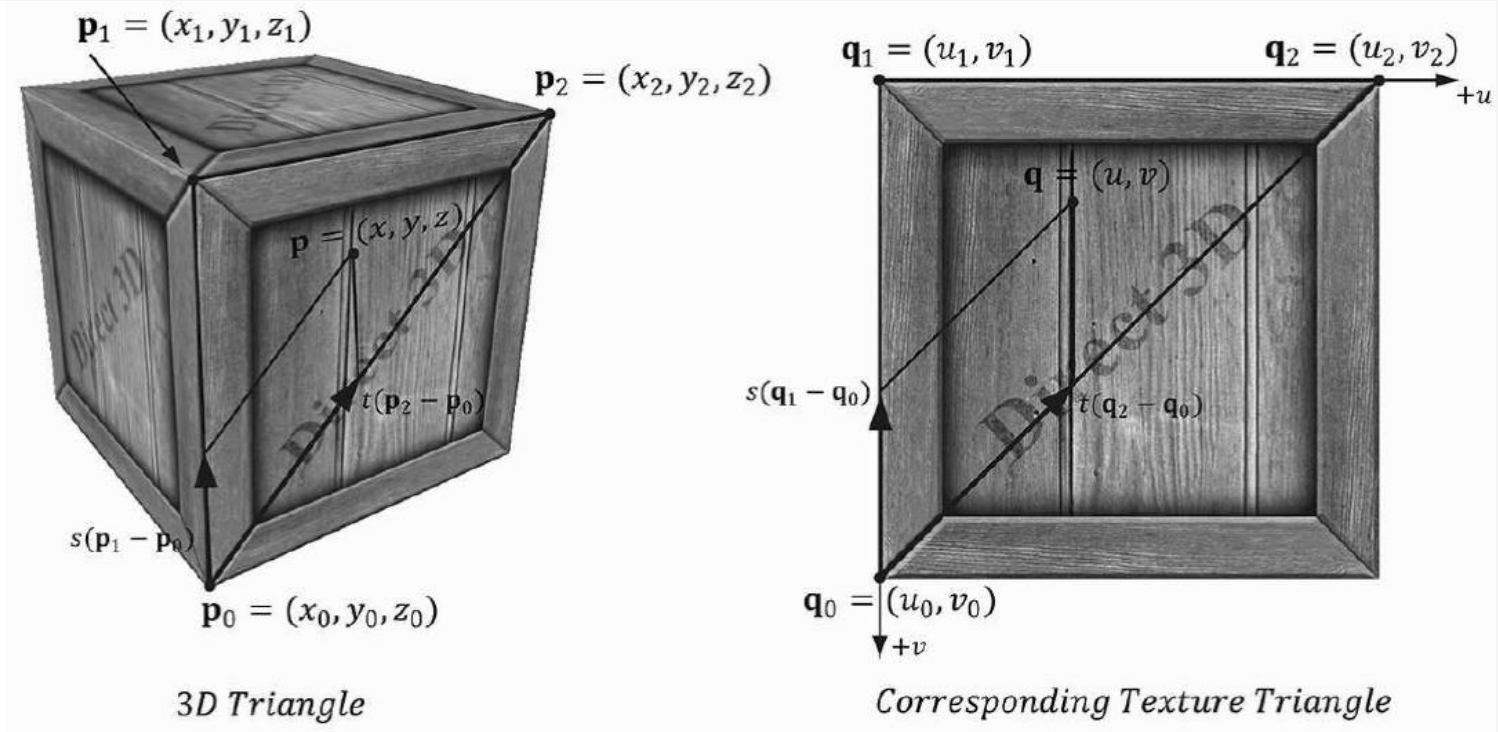
Texture Coordinates,

▪ On the left is a triangle in 3D space, and on the right we define a 2D triangle on the texture that is going to be mapped onto the 3D triangle

For each 3D triangle, we want to define a corresponding triangle on the texture that is to be mapped onto the 3D triangle

Let **p**0, **p**1, and **p**2 be the vertices of a 3D triangle with respective texture coordinates **q**0, **q**1, and **q**2. For an arbitrary point (*x*, *y*,*z*) on the 3D triangle, its texture coordinates (*u*, *v*) are found by linearly interpolating the vertex texture coordinates across the 3D triangle by the same *s*, *t* parameters; that is, if:

In this way, every point on the triangle has a corresponding texture coordinate.



$$(x, y, z) = \mathbf{p} = \mathbf{p}_0 + s(\mathbf{p}_1 - \mathbf{p}_0) + t(\mathbf{p}_2 - \mathbf{p}_0)$$

for $s \geq 0, t \geq 0, s + t \leq 1$ then,

$$(u, v) = \mathbf{q} = \mathbf{q}_0 + s(\mathbf{q}_1 - \mathbf{q}_0) + t(\mathbf{q}_2 - \mathbf{q}_0)$$

Implementation

```
----------FrameResource.h-----

struct Vertex

{

    DirectX::XMFLOAT3 Pos;

    DirectX::XMFLOAT3 Normal;

    DirectX::XMFLOAT2 TexC;

};
```

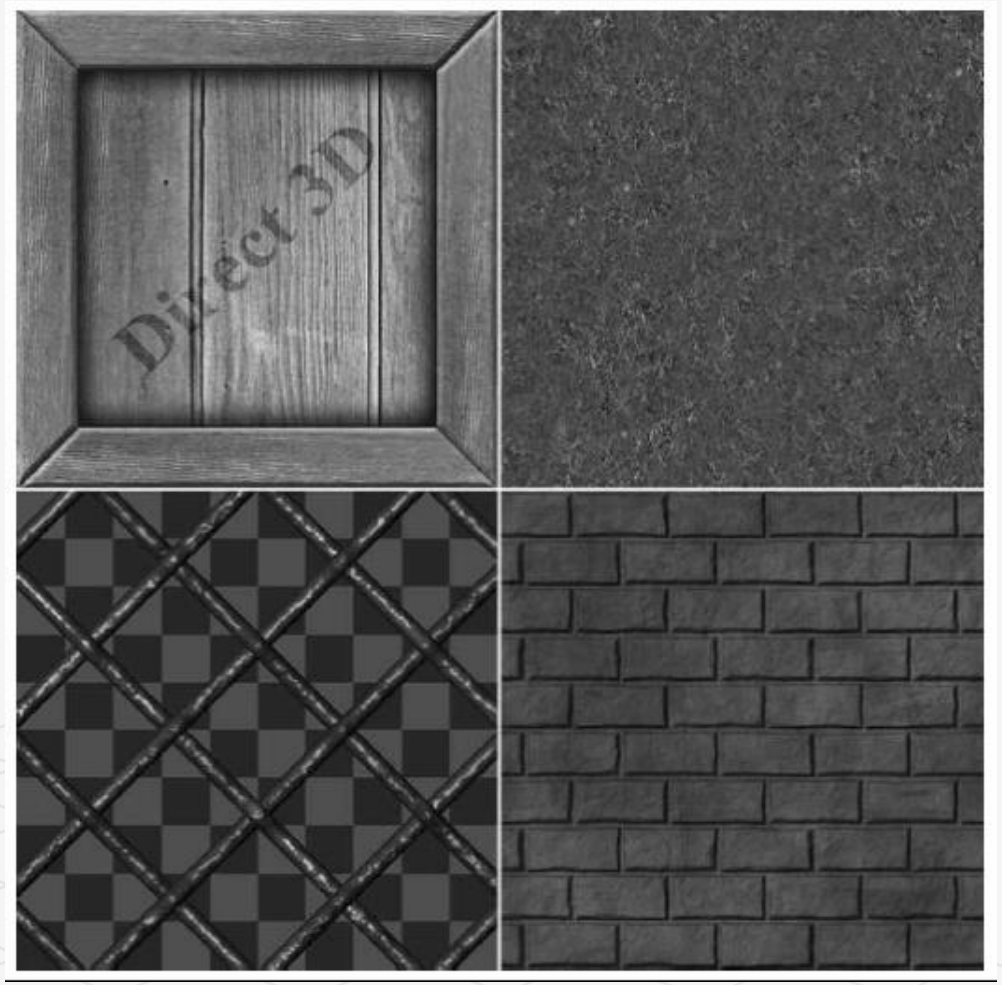
```
----------------Default.hlsl---------------------------------
struct VertexIn
{
     float3 PosL    : POSITION;
     float3 NormalL : NORMAL;
     float2 TexC    : TEXCOORD;
};
------------------------------CrateApp.cpp-----------------------------------------------------

void CrateApp::BuildShadersAndInputLayout()
{
mShaders["standardVS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", nullptr,
"VS", "vs_5_0");
mShaders["opaquePS"] = d3dUtil::CompileShader(L"Shaders\\Default.hlsl", nullptr,
"PS", "ps_5_0");

    mInputLayout =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
        { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
    };
}
```

Texture atlas

we can place several unrelated images on one big texture map and use it for several different objects

This is called a *texture atlas*

A texture atlas storing four subtextures on one large texture. The texture coordinates for each vertex are set so that the desired part of the texture gets mapped onto the geometry.



TEXTURE DATA SOURCES

For real-time graphics applications, the DDS (DirectDraw Surface format) image file format is preferred, as it supports a variety of image formats that are natively understood by the GPU; in particular, it supports compressed image formats that can be natively decompressed by the GPU.

Artists should not use the DDS format as a working image format. Instead they should use their preferred format for saving work. Then when the texture is complete, they export out to DDS for the game application.

DDS textures support the following used in 3D graphics development:

1. mipmaps

2. compressed formats that the GPU can natively decompress

3. texture arrays

4. cube maps

5. volume textures

The DDS format can support different pixel formats. The pixel format is described by a member of the `DXGI_FORMAT` enumerated type; however, not all formats apply to DDS textures. Typically, for uncompressed image data you will use the formats:

1. `DXGI_FORMAT_B8G8R8A8_UNORM` or `DXGI_FORMAT_B8G8R8X8_UNORM`: For low-dynamic-range images.

2. `DXGI_FORMAT_R16G16B16A16_FLOAT`: For high-dynamic-range images.

DDS Overview

1. NVIDIA supplies a plugin for Adobe Photoshop that can export images to the DDS format. The plugin is available at

https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop

Among other options, it allows you to specify the DXGI_FORMAT of the DDS file, and generate mipmaps.

2. Microsoft provides a command line tool called *texconv* that can be used to convert traditional image formats to DDS. In addition, the *texconv* program can be used for more such as resizing images, changing pixel formats, generating mipmaps and even more. You can find the documentation and download link at the following website https://walbourn.github.io/directxtex/

Direct3D supports compressed texture formats: BC1, BC2, BC3, BC4, BC5, BC6, and BC7:

1. BC1 (DXGI_FORMAT_BC1_UNORM): Use this format if you need to compress a format that supports three color channels, and only a 1-bit (on/off) alpha component.
2. BC2 (DXGI_FORMAT_BC2_UNORM): Use this format if you need to compress a format that supports three color channels, and only a 4-bit alpha component.
3. BC3 (DXGI_FORMAT_BC3_UNORM): Use this format if you need to compress a format that supports three color channels, and a 8-bit alpha component.
4. BC4 (DXGI_FORMAT_BC4_UNORM): Use this format if you need to compress a format that contains one color channel (e.g., a grayscale image).
5. BC5 (DXGI_FORMAT_BC5_UNORM): Use this format if you need to compress a format that supports two color channels.
6. BC6 (DXGI_FORMAT_BC6_UF16): Use this format for compressed HDR (high dynamic range) image data.
7. BC7 (DXGI_FORMAT_BC7_UNORM): Use this format for high quality RGBA compression. In particular, this format significantly reduces the errors caused by compressing normal maps.

Because the block compression algorithms work with 4 × 4 pixel blocks, the dimensions of the texture must be multiples of 4.

The advantage of these formats is that they can be stored compressed in GPU memory, and then decompressed on the fly by the GPU when needed

Texconv & *texassemble*

The following example inputs a BMP file *bricks.bmp* and outpts a DDS file *bricks.dds*

with format BC3_UNORM and generates a mipmaps chain with 10 mipmaps.

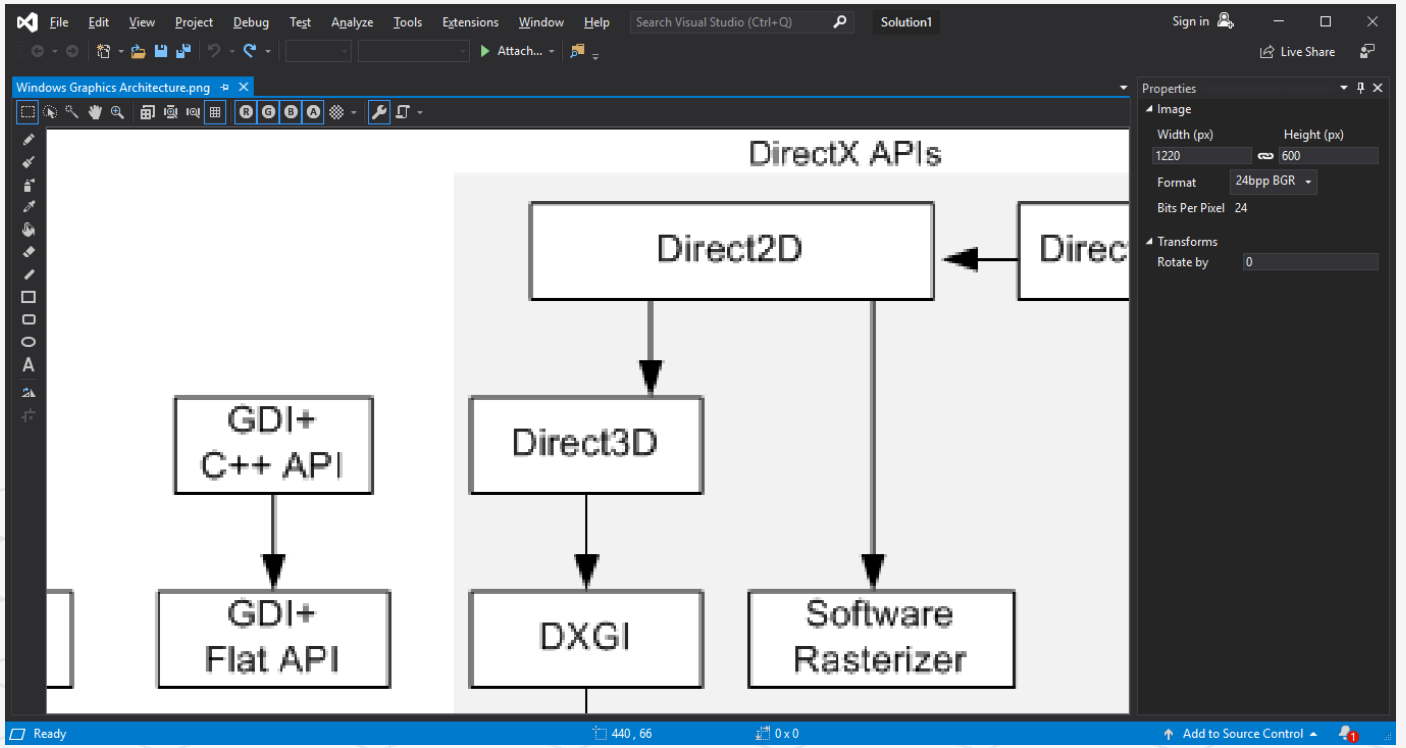
```
texconv -m 10 -f BC3_UNORM treeArray.dds
```

Microsoft provides an additional command line tool called texassemble, which is used to create DDS files that store texture arrays, volume maps, and cube maps.

The download link can also be found at:

https://github.com/Microsoft/DirectXTex

Visual Studio has a built-in image editor that supports DDS in addition to other popular formats. You can drag an image into Visual Studio and it should open it in the image editor. For DDS files, you can view the mipmap levels, change the DDS format, and view the various color channels.



Loading DDS Files

Microsoft provides lightweight source code to load DDS files at:

https://github.com/microsoft/DirectXTK12/tree/master/Src

1. device: Pointer to the D3D device to create the texture resources.

2. cmdList: Command list to submit GPU commands (e.g., copying texture data from an upload heap to a default heap).

3. szFileName: Filename of the image to load.

4. texture: Returns the texture resource with the loaded image data.

5. textureUploadHeap: Returns the texture resource that was used as an upload heap to copy the image data into the default heap texture resource. This resource cannot be destroyed until the GPU finished the copy command.

```
HRESULT DirectX::CreateDDSTextureFromFile12(

_In_ ID3D12Device* device,

_In_ ID3D12GraphicsCommandList* cmdList,

_In_z_ const wchar_t* szFileName,

_Out_ Microsoft::WRL::ComPtr<ID3D12Resource>&

texture,

_Out_ Microsoft::WRL::ComPtr<ID3D12Resource>&

textureUploadHeap);
```

Create a texture

To create a texture from an image called *WoodCreate01.dds*, we would write the following:

```
--------d3dutilh---------------------------------
struct Texture
{
// Unique material name for lookup.
std::string Name;

std::wstring Filename;

Microsoft::WRL::ComPtr<ID3D12Resource> Resource = nullptr;
Microsoft::WRL::ComPtr<ID3D12Resource> UploadHeap = nullptr;
};
```

```
void CrateApp::LoadTextures()
{
auto woodCrateTex = std::make_unique<Texture>();
woodCrateTex->Name = "woodCrateTex";
woodCrateTex->Filename = L"../../Textures/WoodCrate01.dds";
ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(md3dDevice.Get(),
mCommandList.Get(), woodCrateTex->Filename.c_str(),
woodCrateTex->Resource, woodCrateTex->UploadHeap));
mTextures[woodCrateTex->Name] = std::move(woodCrateTex);
}
```

SRV Heap

Once a texture resource is created, we need to create an SRV descriptor to it which we can set to a root signature parameter slot for use by the shader programs.

In order to do that, we first need to create a descriptor heap with ID3D12Device::CreateDescriptorHeap to store the SRV descriptors.

The following code builds a SRV heap with one descriptor that can store either CBV, SRV, or UAV descriptors, and is visible to shaders:

```
void CrateApp::BuildDescriptorHeaps()

{

// Create the SRV heap.

D3D12_DESCRIPTOR_HEAP_DESC srvHeapDesc = {};

srvHeapDesc.NumDescriptors = 1;

srvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;

srvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&srvHeapDesc,
IID_PPV_ARGS(&mSrvDescriptorHeap)));
```

Creating SRV Descriptors

Once we have an SRV heap, we need to create the actual descriptors. An SRV descriptor is described by filling out a `D3D12_SHADER_RESOURCE_VIEW_DESC` object, which describes how the resource is used and other information—its format, dimension, mipmaps count, etc.

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC
    {
    DXGI_FORMAT Format;
    D3D12_SRV_DIMENSION ViewDimension;
    UINT Shader4ComponentMapping;
    union
        {
        D3D12_BUFFER_SRV Buffer;
        D3D12_TEX1D_SRV Texture1D;
        D3D12_TEX1D_ARRAY_SRV Texture1DArray;
        D3D12_TEX2D_SRV Texture2D;
        D3D12_TEX2D_ARRAY_SRV Texture2DArray;
        D3D12_TEX2DMS_SRV Texture2DMS;
        D3D12_TEX2DMS_ARRAY_SRV Texture2DMSArray;
        D3D12_TEX3D_SRV Texture3D;
        D3D12_TEXCUBE_SRV TextureCube;
        D3D12_TEXCUBE_ARRAY_SRV TextureCubeArray;
        D3D12_RAYTRACING_ACCELERATION_STRUCTURE_SRV
RaytracingAccelerationStructure;
        } ;
    } D3D12_SHADER_RESOURCE_VIEW_DESC;
```

For 2D textures, we are only interested in the D3D12_TEX2D_SRV part of the union.

```
CD3DX12_CPU_DESCRIPTOR_HANDLE hDescriptor(

mSrvDescriptorHeap->GetCPUDescriptorHandleForHeapStart());

auto woodCrateTex = mTextures["woodCrateTex"]->Resource;

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};

srvDesc.Shader4ComponentMapping =
D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;

srvDesc.Format = woodCrateTex->GetDesc().Format;

srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;

srvDesc.Texture2D.MostDetailedMip = 0;

srvDesc.Texture2D.MipLevels = woodCrateTex->GetDesc().MipLevels;

srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;

md3dDevice->CreateShaderResourceView(woodCrateTex.Get(), &srvDesc,
hDescriptor);
```

Adding an Index into SRV heap for diffuse texture.

Right now we specify materials per draw call by changing the material constant buffer.

This means that all geometry in the draw call will have the same material values.

The idea of texturing mapping is to get the material data from texture maps instead of the material constant buffer.

```
float4 PS(VertexOut pin) : SV_Target
```

```
{  // Multiple texture sample with constant buffer albedo.
```

```
    float4 diffuseAlbedo = gDiffuseMap.Sample(gsamLinear, pin.TexC) * gDiffuseAlbedo;
```

We add an index to our material definition, which references an SRV in the descriptor heap specifying the texture associated with the material:

```
// Simple struct to represent a material for our demos.  A production 3D engine
// would likely create a class hierarchy of Materials.
struct Material
{
// Unique material name for lookup.
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;

// Dirty flag indicating the material has changed and we need to update the constant buffer.
// Because we have a material constant buffer for each FrameResource, we have to apply the
// update to each FrameResource.  Thus, when we modify a material we should set
// NumFramesDirty = gNumFrameResources so that each frame resource gets the update.
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();
};
```

Binding Textures to the Pipeline

Assuming the root signature has been defined to expect a table of shader resource views to be bound to the 0th slot parameter, we can draw our render items with texturing using the following code:

A texture resource can actually be used by any shader (vertex, geometry, or pixel shader). For now, we will just be using them in pixel shaders.

```
void CrateApp::DrawRenderItems(ID3D12GraphicsCommandList* cmdList, const std::vector<RenderItem*>& ritems)
{
    UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));
    UINT matCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(MaterialConstants));

auto objectCB = mCurrFrameResource->ObjectCB->Resource();
auto matCB = mCurrFrameResource->MaterialCB->Resource();

    // For each render item...
    for(size_t i = 0; i < ritems.size(); ++i)
    {
        auto ri = ritems[i];

        cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
        cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
        cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

CD3DX12_GPU_DESCRIPTOR_HANDLE tex(mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);

        D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB->GetGPUVirtualAddress() + ri-
>ObjCBIndex*objCBByteSize;
D3D12_GPU_VIRTUAL_ADDRESS matCBAddress = matCB->GetGPUVirtualAddress() + ri->Mat->MatCBIndex*matCBByteSize;

            cmdList->SetGraphicsRootDescriptorTable(0, tex);
        cmdList->SetGraphicsRootConstantBufferView(1, objCBAddress);
        cmdList->SetGraphicsRootConstantBufferView(3, matCBAddress);

        cmdList->DrawIndexedInstanced(ri->IndexCount, 1, ri->StartIndexLocation, ri->BaseVertexLocation, 0);
    }
}
```

FILTERS

suppose the monitor resolution is 1024 × 1024 and the wall's texture resolution is 256 × 256. This illustrates texture *magnification*—we are trying to cover many pixels with a few texels, meaning between every texel point lies four pixels.

If we have a 1D texture with 256 samples and an interpolated texture coordinate u = 0.126484375

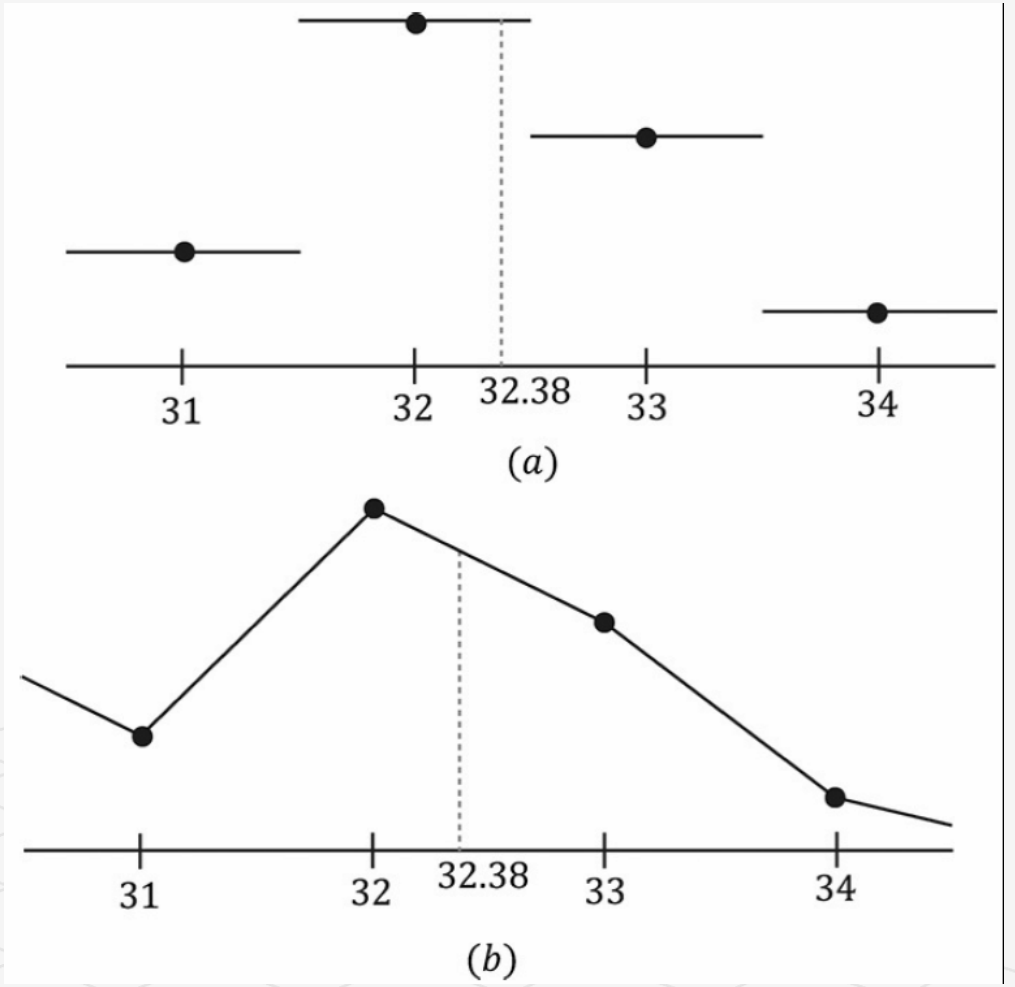
This normalized texture coordinate refers to the 0.126484375 × 256 = 32.38 texel.

Of course, this value lies between two of our texel samples, so we must use interpolation to approximate it.

- Given the texel points, we construct a piecewise constant function to approximate values between the texel points (a)

▪ This is sometimes called nearest neighbor point sampling, as the value of the nearest texel point is used

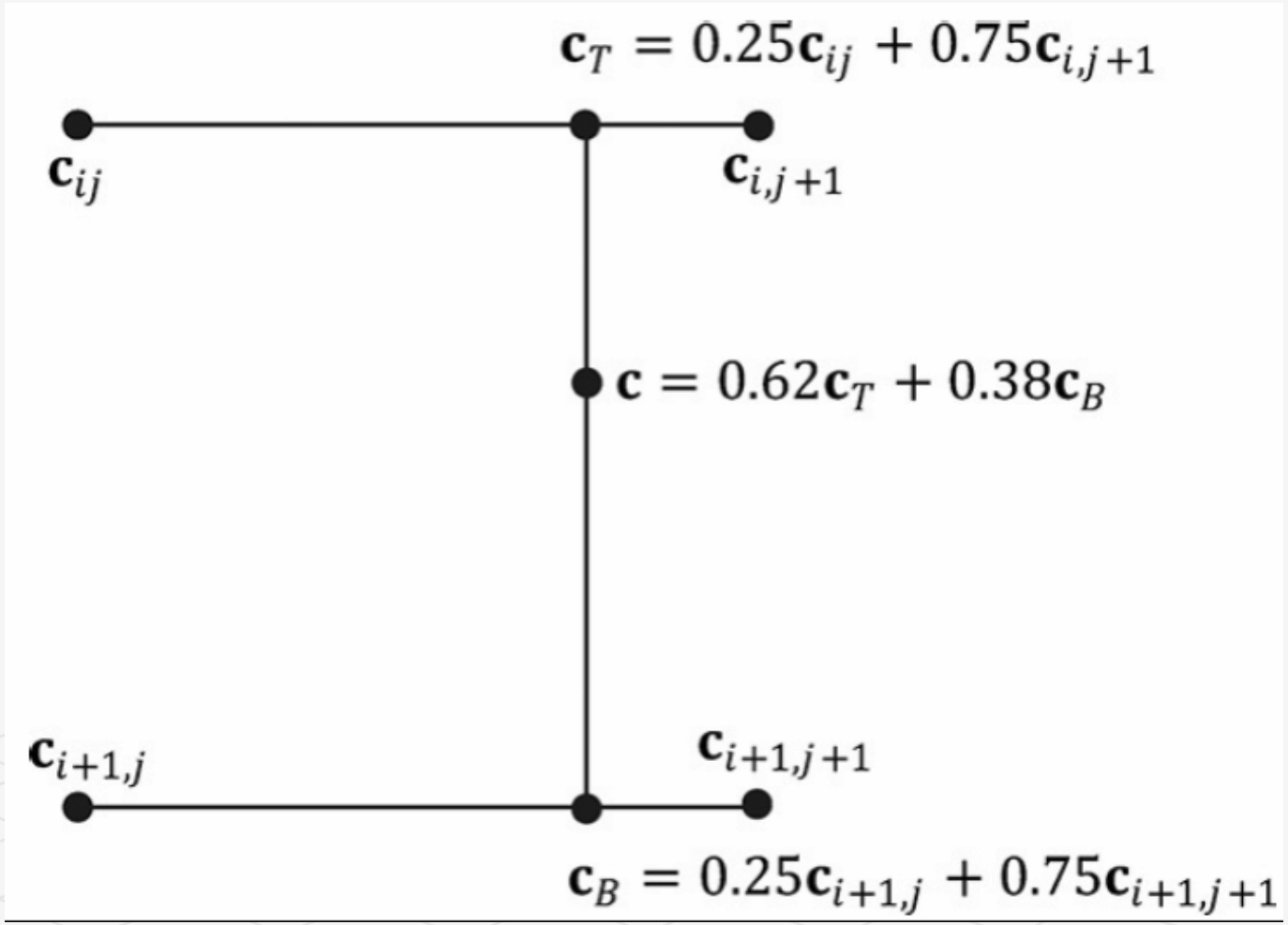
- Given the texel points, we construct a piecewise linear function to approximate values between texelpoints (b)



2D linear or bilinear interpolation

Given a pair of texture coordinates between four texels, we do two 1D linear interpolations in the *u*-direction, followed by one 1D interpolation in the *v*-direction.

- We have four texel points: cij, ci,j + 1, ci + 1, j, and ci + 1,j + 1
- We want to approximate the color of c, which lies between these four texel points, using interpolation
- In this example, c lies 0.75 units to the right of cij and 0.38 units below cij
- We first do a 1D linear interpolation between the top two colors to get cT
- Likewise, we do a 1D linear interpolate between the bottom two colors to get cB
- Finally, we do a 1D linear interpolation between cT and cB to get c

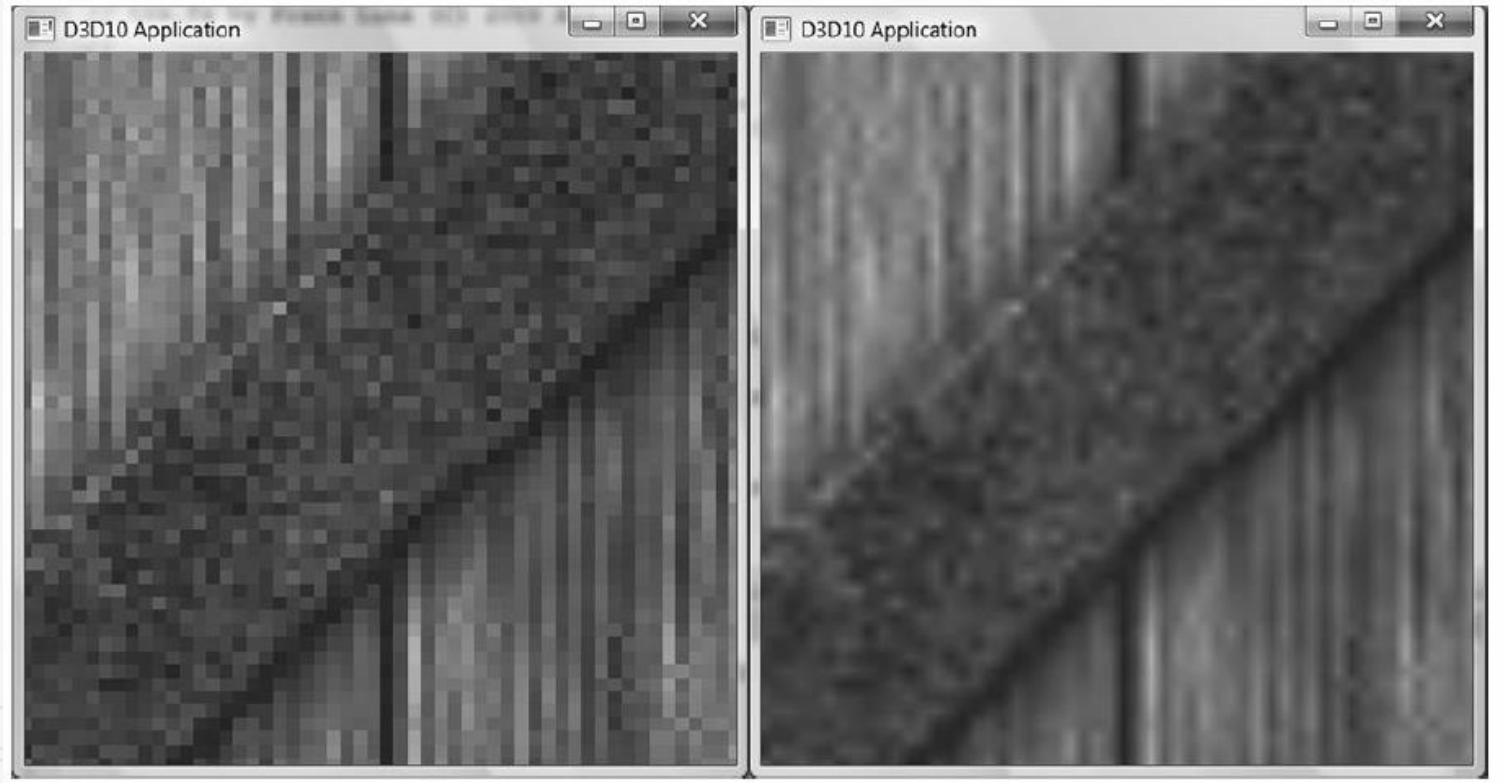


The difference between constant and linear interpolation

Constant interpolation has the characteristic of creating a blocky looking image.

Linear interpolation is smoother, but still will not look as good as if we had real data (e.g., a higher resolution texture) instead of derived data via interpolation.

We zoom in on a cube with a crate texture so that magnification occurs. On the left we use constant interpolation, which results in a blocky appearance; this makes sense because the interpolating function has discontinuities , which makes the changes abrupt rather than smooth. On the right we use linear filtering, which results in a smoother image due to the continuity of the interpolating function.



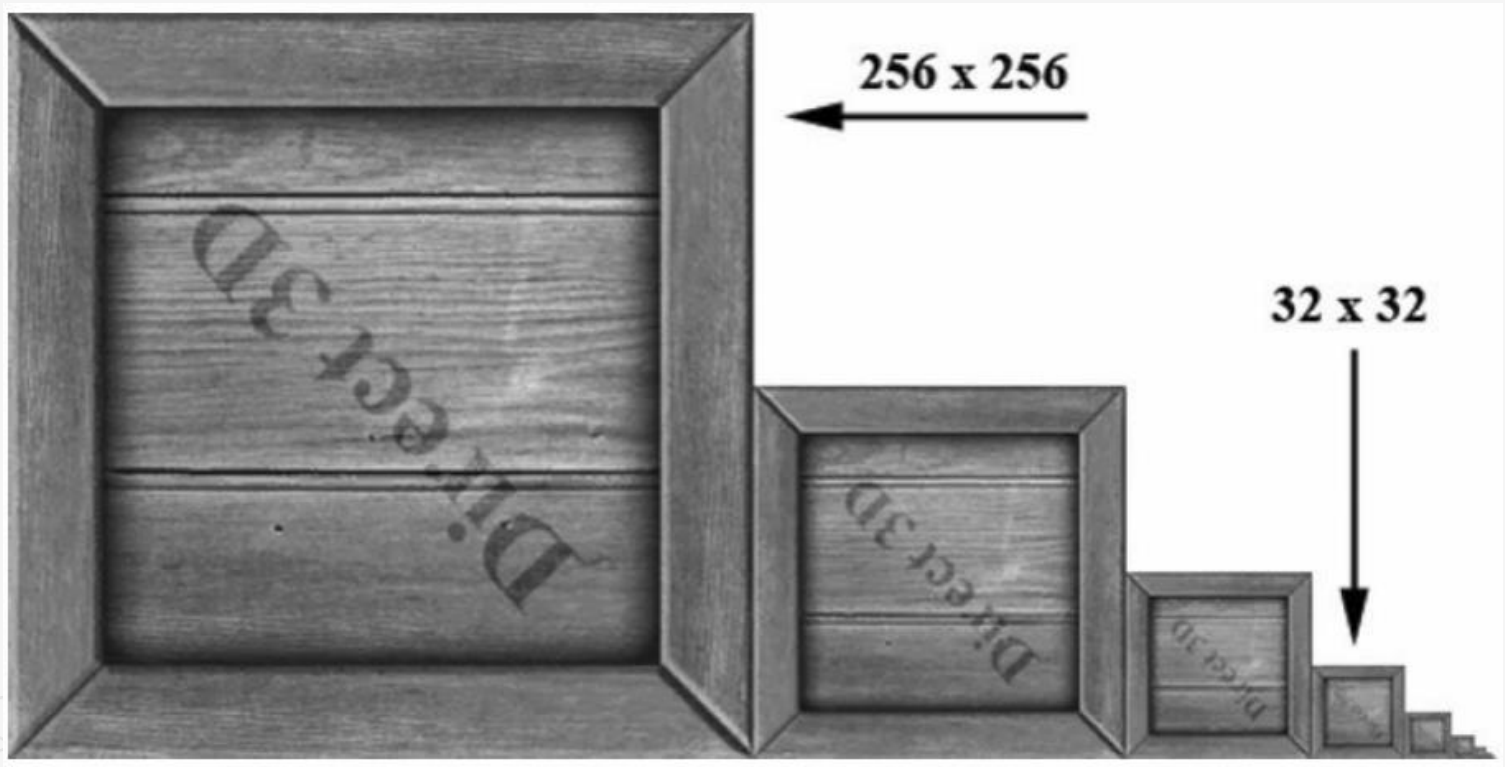
Minification

Minification is the opposite of magnification. In minification, too many texels are being mapped to too few pixels.

So now we have 256 × 256 texels getting mapped to 64 × 64 screen pixels.

The technique of mipmapping offers an efficient approximation for this at the expense of some extra memory.

At initialization time (or asset creation time), smaller versions of the texture are made by downsampling the image to create a mipmap chain



Mipmapping

At runtime, the graphics hardware will do two different things based on the mipmap settings specified by the programmer:

1. Pick and use the mipmap level that best matches the projected screen geometry resolution for texturing, applying constant or linear interpolation as needed. This is called *point filtering* for mipmaps because it is like constant interpolation—you just choose the nearest mipmap level and use that for texturing.

2. Pick the two nearest mipmap levels that best match the projected screen geometry resolution for texturing (one will be bigger and one will be smaller than the screen geometry resolution). Next, apply constant or linear filtering to both of these mipmap levels to produce a texture color for each one. Finally, interpolate between these two texture color results. This is called *linear filtering* for mipmaps because it is like linear interpolation—you linearly interpolate between the two nearest mipmap levels.

Anisotropic Filtering

This filter helps alleviate the distortion that occurs when the angle between a polygon's normal vector and camera's look vector is wide (e.g., when a polygon is orthogonal to the view window).

This filter is the most expensive, but can be worth the cost for correcting the distortion artifacts.

The top face of the crate is nearly orthogonal to the view window.

(Left) Using linear filtering the top of the crate is badly blurred.

(Right) Anisotropic filtering does a better job at rendering the top face of the crate from this angle.

ADDRESS MODES

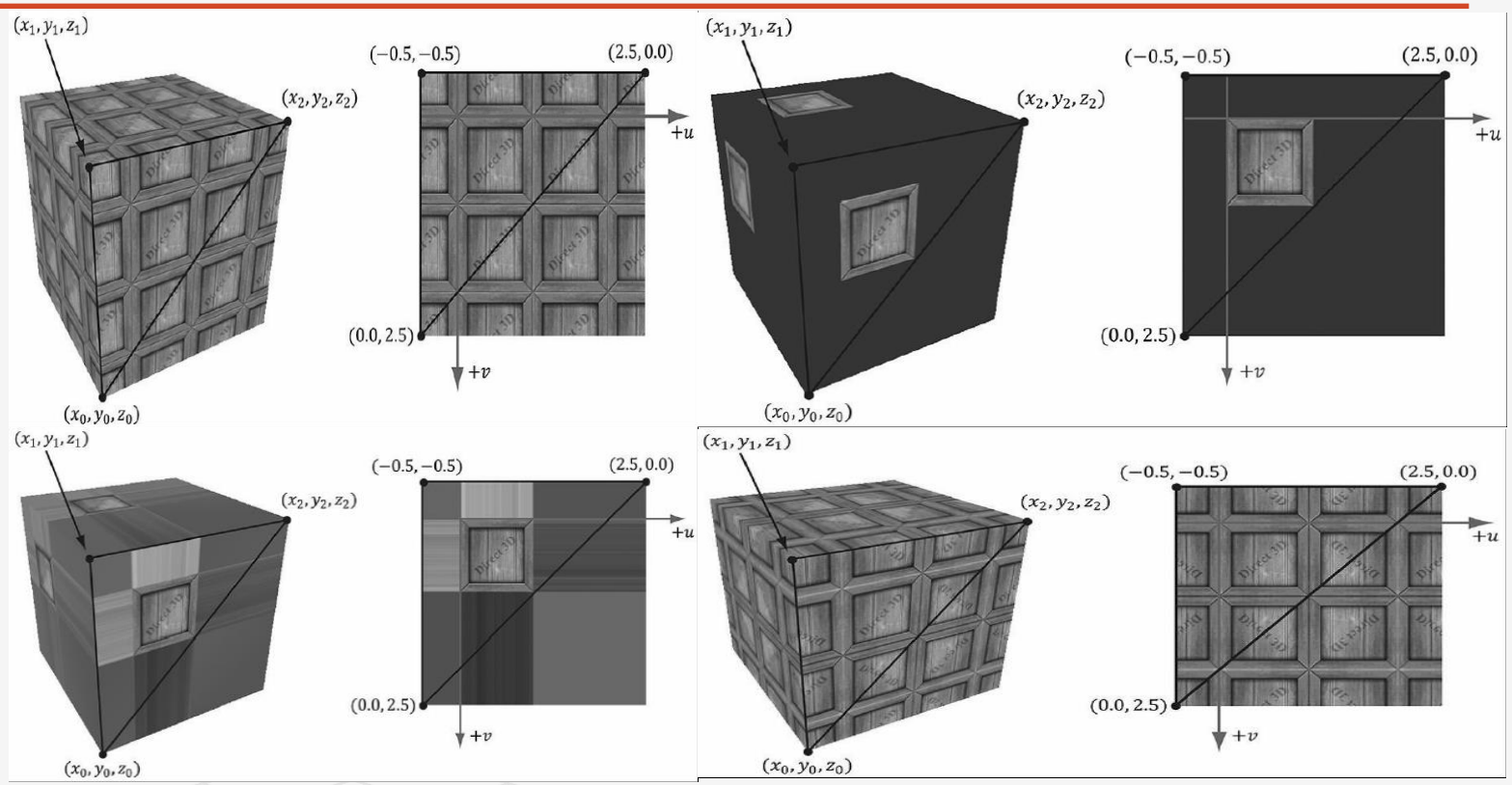
A texture, combined with constant or linear interpolation, defines a vector-valued function $T(u, v) = (r, g, b, a)$. That is, given the texture coordinates $(u, v) \in [0, 1]^2$ the texture function T returns a color (r, g, b, a). Direct3D allows us to extend the domain of this function in four different ways (called *address modes:*

1. *wrap* extends the texture function by repeating the image at every integer junction

2. *border color* extends the texture function by mapping each (u, v) not in $[0, 1]^2$ to some color specified by the programmer

3. clamp extends the texture function by mapping each (u, v) not in $[0, 1]^2$ to the color $T(u0, v0)$, where $(u0, v0)$ is the nearest point to (u, v) contained in $[0, 1]^2$

4. *mirror* extends the texture function by mirroring the image at every integer junction

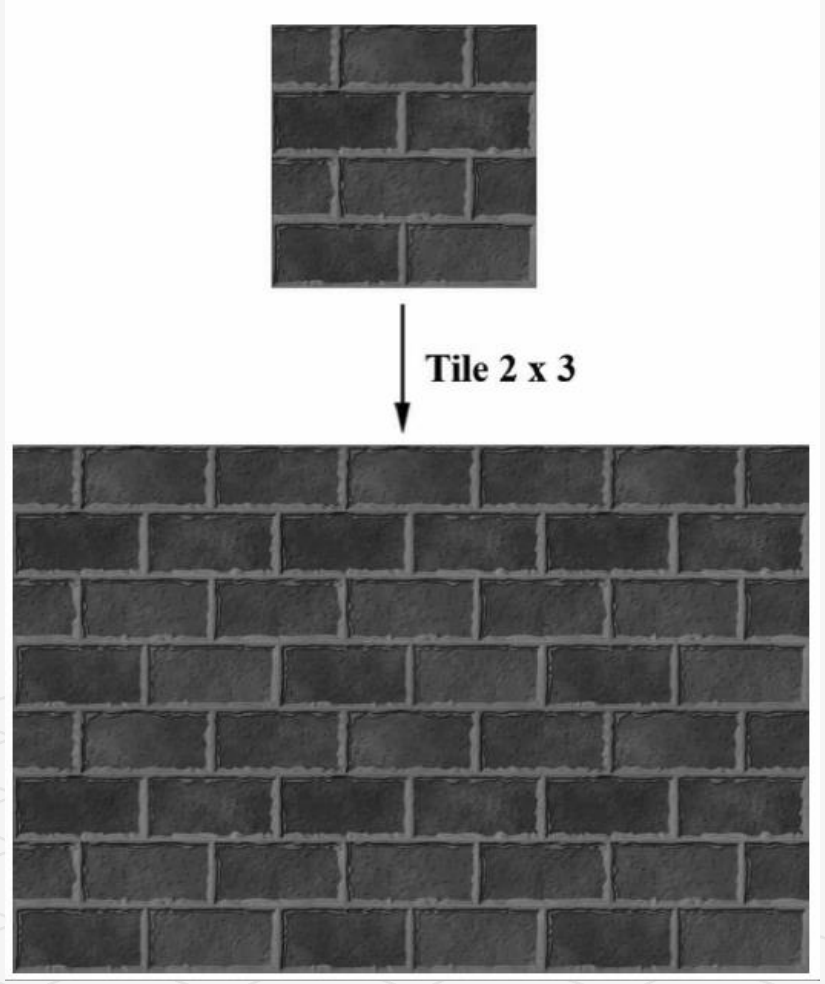


Address Mode

The wrap address mode is probably the most often employed; it allows us to tile a texture repeatedly over some surface. This effectively enables us to increase the texture resolution without supplying additional data

- Brick texture tiled 2 × 3 times

- Because the texture is seamless, the repetition pattern is harder to notice



Address modes

Address modes are described in Direct3D via the D3D12_TEXTURE_ADDRESS_MODE enumerated type:

```
enum D3D12_TEXTURE_ADDRESS_MODE
    {
        D3D12_TEXTURE_ADDRESS_MODE_WRAP= 1,
        D3D12_TEXTURE_ADDRESS_MODE_MIRROR= 2,
        D3D12_TEXTURE_ADDRESS_MODE_CLAMP= 3,
        D3D12_TEXTURE_ADDRESS_MODE_BORDER= 4,
        D3D12_TEXTURE_ADDRESS_MODE_MIRROR_ONCE= 5
    } D3D12_TEXTURE_ADDRESS_MODE;
```

Creating Samplers

An application will usually need several sampler objects to sample textures in different ways.

samplers are used in shaders. In order to bind samplers to shaders for use, we need to bind descriptors to sampler objects.

The following code shows an example root signature such that the second slot takes a table of one sampler descriptor bound to sampler register slot 0.

```
CD3DX12_DESCRIPTOR_RANGE descRange[3];

descRange[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);

descRange[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER, 1, 0);

descRange[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);
```

```
CD3DX12_ROOT_PARAMETER rootParameters[3];

rootParameters[0].InitAsDescriptorTable(1, &descRange[0], D3D12_SHADER_VISIBILITY_PIXEL);

rootParameters[1].InitAsDescriptorTable(1, &descRange[1], D3D12_SHADER_VISIBILITY_PIXEL);

rootParameters[2].InitAsDescriptorTable(1, &descRange[2], D3D12_SHADER_VISIBILITY_ALL);
```

```
CD3DX12_ROOT_SIGNATURE_DESC descRootSignature;

descRootSignature.Init(3, rootParameters, 0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT);
```

Sampler heap

If we are going to be setting sampler descriptors, we need a sampler heap. A sampler heap is created by filling out a `D3D12_DESCRIPTOR_HEAP_DESC` instance and specifying the heap type `D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER`:

```
D3D12_DESCRIPTOR_HEAP_DESC descHeapSampler = {};

descHeapSampler.NumDescriptors = 1;

descHeapSampler.Type = D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER;

descHeapSampler.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

ComPtr<ID3D12DescriptorHeap> mSamplerDescriptorHeap;

ThrowIfFailed(mDevice -> CreateDescriptorHeap(&descHeapSampler,

__uuidof(ID3D12DescriptorHeap),

(void**)& mSamplerDescriptorHeap));
```

D3D12_SAMPLER_DESC

Once we have a sampler heap, we can create sampler descriptors. It is here that we specify the address mode and filter type, as well as other parameters by filling out a `D3D12_SAMPLER_DESC` object:

```
typedef struct D3D12_SAMPLER_DESC
{
D3D12_FILTER Filter;
D3D12_TEXTURE_ADDRESS_MODE AddressU;
D3D12_TEXTURE_ADDRESS_MODE AddressV;
D3D12_TEXTURE_ADDRESS_MODE AddressW;
FLOAT MipLODBias;
UINT MaxAnisotropy;
D3D12_COMPARISON_FUNC ComparisonFunc;
FLOAT BorderColor[4];
FLOAT MinLOD;
FLOAT MaxLOD;
} D3D12_SAMPLER_DESC;
```

D3D12_FILTER types

1. D3D12_FILTER_MIN_MAG_MIP_POINT: Point filtering over a texture map, and point filtering across mipmap levels (i.e., the nearest mipmap level is used).

2. D3D12_FILTER_MIN_MAG_LINEAR_MIP_POINT: Bilinear filtering over a texture map, and point filtering across mipmap levels (i.e., the nearest mipmap level is used).

3. D3D12_FILTER_MIN_MAG_MIP_LINEAR: Bilinear filtering over a texture map, and linear filtering between the two nearest lower and upper mipmap levels. This is often called trilinear filtering.

4. D3D12_FILTER_ANISOTROPIC: Anisotropic filtering for minification, magnification, and mipmapping.

The following code shows how to bind a sampler descriptor to a root signature parameter slot for use by the shader programs:

The following example shows how to create a descriptor to a sampler in the heap that uses linear filtering, wrap address mode, and typical default values for the other parameters:

```
D3D12_SAMPLER_DESC samplerDesc = {};
samplerDesc.Filter = D3D12_FILTER_MIN_MAG_MIP_LINEAR;
samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 1;
samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_ALWAYS;
md3dDevice->CreateSampler(&samplerDesc,
mSamplerDescriptorHeap -> GetCPUDescriptorHandleForHeapStart());
```

```
commandList->SetGraphicsRootDescriptorTable(1,

samplerDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
```

Static Samplers

Direct3D provides a special shortcut to define an array of samplers and set them without going through the process of creating a sampler heap.

The Init function of the CD3DX12_ROOT_SIGNATURE_DESC class has two parameters that allow you to define an array of so-called static samplers your application can use.

Static samplers are described by the D3D12_STATIC_SAMPLER_DESC structure.

This structure is very similar to D3D12_SAMPLER_DESC, with the following exceptions:

1. There are some limitations on what the border color can be:

```
D3D12_STATIC_BORDER_COLOR borderColor = D3D12_STATIC_BORDER_COLOR_OPAQUE_WHITE,
```

2. It contains additional fields to specify the shader register, register space, and shader visibility, which would normally be specified as part of the sampler heap.

```
std::array<const CD3DX12_STATIC_SAMPLER_DESC, 6> GetStaticSamplers()
{
// Applications usually only need a handful of samplers.  So just define them all up front
// and keep them available as part of the root signature.

const CD3DX12_STATIC_SAMPLER_DESC pointWrap(
0, // shaderRegister
D3D12_FILTER_MIN_MAG_MIP_POINT, // filter
D3D12_TEXTURE_ADDRESS_MODE_WRAP,  // addressU
D3D12_TEXTURE_ADDRESS_MODE_WRAP,  // addressV
D3D12_TEXTURE_ADDRESS_MODE_WRAP); // addressW

const CD3DX12_STATIC_SAMPLER_DESC pointClamp(
1, // shaderRegister
D3D12_FILTER_MIN_MAG_MIP_POINT, // filter
D3D12_TEXTURE_ADDRESS_MODE_CLAMP,  // addressU
D3D12_TEXTURE_ADDRESS_MODE_CLAMP,  // addressV
D3D12_TEXTURE_ADDRESS_MODE_CLAMP); // addressW
```

SAMPLING TEXTURES IN A SHADER

A texture object is defined in HLSL and assigned to a texture register with the following syntax:

```
Texture2D    gDiffuseMap : register(t0);
```

tn: n is an integer identifying the texture register slot

The root signature definition specifies the mapping from slot parameter to shader register; this is how the application code can bind an SRV to a particular Texture2D object in a shader.

Sampler objects are defined HLSL and assigned to a sampler register with the following syntax. Note that texture registers use specified by sn where n is an integer identifying the sampler register slot.

```
SamplerState gsamPointWrap : register(s0);

SamplerState gsamPointClamp : register(s1);

SamplerState gsamLinearWrap : register(s2);

SamplerState gsamLinearClamp : register(s3);

SamplerState gsamAnisotropicWrap : register(s4);

SamplerState gsamAnisotropicClamp : register(s5);
```

Texture2D::Sample

Now, given a pair of texture coordinate (u, v) for a pixel in the pixel shader, we actually sample a texture using the Texture2D::Sample method.

We pass a SamplerState object for the first parameter indicating how the texture data will be sampled, and we pass in the pixel's (*u*, *v*) texture coordinates for the second parameter. This method returns the interpolated color from the texture map at the specified (*u*, *v*) point using the filtering methods specified by the SamplerState object.

```
Texture2D    gDiffuseMap : register(t0);
SamplerState gsamLinear  : register(s0);

struct VertexOut
{
      float4 PosH    : SV_POSITION;
     float3 PosW    : POSITION;
     float3 NormalW : NORMAL;
      float2 TexC    : TEXCOORD;
};
```

```
float4 PS(VertexOut pin) : SV_Target
{
    float4 diffuseAlbedo = gDiffuseMap.Sample(gsamLinear, pin.TexC) * gDiffuseAlbedo;

    // 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);

    // Light terms.
    float4 ambient = gAmbientLight*diffuseAlbedo;

    const float shininess = 1.0f - gRoughness;
    Material mat = { diffuseAlbedo, 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 = diffuseAlbedo.a;

    return litColor;
}
```

Specifying Texture Coordinates

The `GeometryGenerator::CreateBox` generates the texture coordinates for the box so that the entire texture image is mapped onto each face of the box.

```
GeometryGenerator::MeshData GeometryGenerator::CreateBox(float width, float
height, float depth, uint32 numSubdivisions)
{
    MeshData meshData;

// Create the vertices.

Vertex v[24];

float w2 = 0.5f*width;
float h2 = 0.5f*height;
float d2 = 0.5f*depth;

// Fill in the front face vertex data.
v[0] = Vertex(-w2, -h2, -d2, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f);
v[1] = Vertex(-w2, +h2, -d2, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f);
v[2] = Vertex(+w2, +h2, -d2, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f);
v[3] = Vertex(+w2, -h2, -d2, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f);

// Fill in the back face vertex data.
v[4] = Vertex(-w2, -h2, +d2, 0.0f, 0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 1.0f);
v[5] = Vertex(+w2, -h2, +d2, 0.0f, 0.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f);
```

Creating the Texture

We create the texture from file an initialization time as follows:

----------d3dUtil.h----------

```
struct Texture
{
std::string Name;
std::wstring Filename;

Microsoft::WRL::ComPtr<ID3D12Resource>
Resource = nullptr;

Microsoft::WRL::ComPtr<ID3D12Resource>
UploadHeap = nullptr;
};
```

We store all of our unique textures in an unordered map so that we can look them up by name.

```
std::unordered_map<std::string, std::unique_ptr<Texture>> mTextures;

void CrateApp::LoadTextures()
{
auto woodCrateTex = std::make_unique<Texture>();
woodCrateTex->Name = "woodCrateTex";
woodCrateTex->Filename = L"../../Textures/WoodCrate01.dds";
ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(md3dDevice.Get(),
mCommandList.Get(), woodCrateTex->Filename.c_str(),
woodCrateTex->Resource, woodCrateTex->UploadHeap));
mTextures[woodCrateTex->Name] = std::move(woodCrateTex);
}
```

Setting the Texture

Once a texture has been created and an SRV has been created for it in a descriptor heap, binding the texture to the pipeline so that it can be used in shader programs is simply a matter of setting it to the root signature parameter that expects the texture:

```
void CrateApp::DrawRenderItems(ID3D12GraphicsCommandList* cmdList, const
std::vector<RenderItem*>& ritems)
{
    UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));
    UINT matCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(MaterialConstants));

auto objectCB = mCurrFrameResource->ObjectCB->Resource();
auto matCB = mCurrFrameResource->MaterialCB->Resource();

    // For each render item...
    for(size_t i = 0; i < ritems.size(); ++i)
    {
        auto ri = ritems[i];

        cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
        cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
        cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

CD3DX12_GPU_DESCRIPTOR_HANDLE tex(mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);

        D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB->GetGPUVirtualAddress() + ri->ObjCBIndex*objCBByteSize;
D3D12_GPU_VIRTUAL_ADDRESS matCBAddress = matCB->GetGPUVirtualAddress() + ri->Mat->MatCBIndex*matCBByteSize;

        cmdList->SetGraphicsRootDescriptorTable(0, tex);
        cmdList->SetGraphicsRootConstantBufferView(1, objCBAddress);
        cmdList->SetGraphicsRootConstantBufferView(3, matCBAddress);

        cmdList->DrawIndexedInstanced(ri->IndexCount, 1, ri->StartIndexLocation, ri->BaseVertexLocation, 0);
    }
}
```

TRANSFORMING TEXTURES

gTexTransform and gMatTransform are variables used in the vertex shader to transform the input texture coordinates. we use an identity matrix transformation so that the input texture coordinates are left unmodified. Note that to transform the 2D texture coordinates by a 4 × 4 matrix, we augment it to a 4D vector. After the multiplication is done, the resulting 4D vector is cast back to a 2D vector by throwing away the z- and w-components.

```
// Constant data that varies per frame.
cbuffer cbPerObject : register(b0)
{
    float4x4 gWorld;
    float4x4 gTexTransform;
};
cbuffer cbMaterial : register(b2)
{
float4 gDiffuseAlbedo;
    float3 gFresnelR0;
    float  gRoughness;
    float4x4 gMatTransform;
};
```

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

    // Transform to homogeneous clip space.
    vout.PosH = mul(posW, gViewProj);

// Output vertex attributes for interpolation across triangle.
    float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
    vout.TexC = mul(texC, gMatTransform).xy;

    return vout;
}
```