The Rasterizer Stage

Texturing, Lighting, Testing and Blending

Triangle Setup, Triangle Traversal and Back Face Culling

From Primitives To Fragments

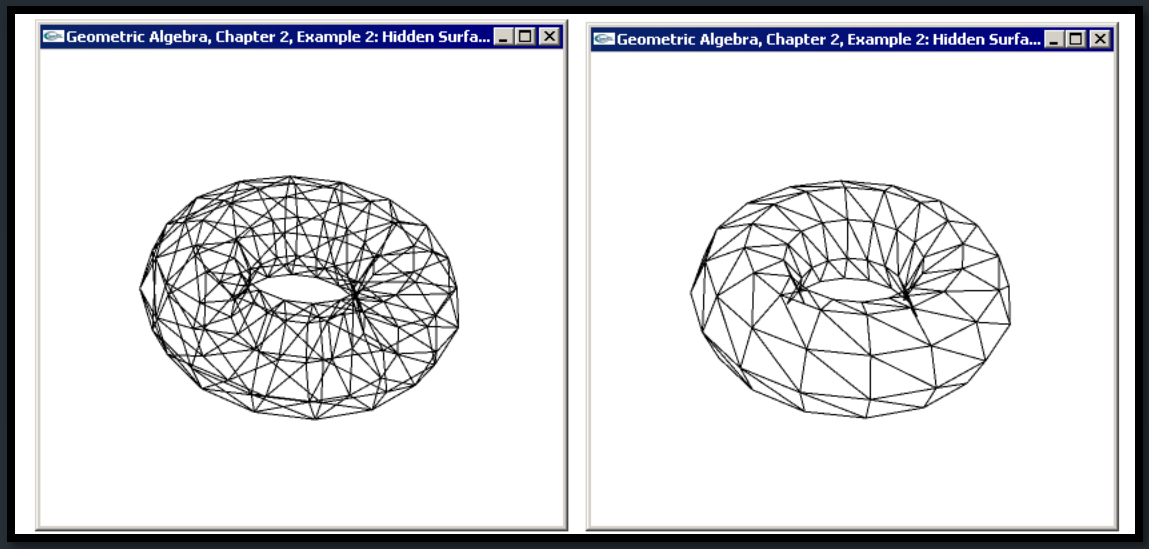
Post Clipping

- In the last stages of the geometry stage, vertices undergo the following:
  - They are projected into clip space by multiplication with the projection matrix
  - They are then clipped against the canonical view volume (unit cube), only vertices within the view volume are kept.
  - After clipping vertices are then converted into screen space coordinates. Due to the canonical view volume all vertices' depth (Z) values are between 0 and 1.
- It is these screen space vertices that are received by the triangle setup stage.

Triangle (primitive) Setup

- Triangle Setup links up the screen space vertices to form triangles (can form lines as well).
- This linking is done according to the topology specified for the set of vertices (triangle list, triangle strip, triangle fan, etc)
- Once the linking is complete, backface culling is applied (if enabled).
- Triangles that are not culled are sent to the triangle traversal stage.

Backface Culling



- Backface culling is a technique used to discard polygons facing away from the camera.
- This lessens the load on the rasterizer by discarding polygons that are hidden by front facing ones.
- It is sometimes necessary to disable backface culling to correctly render certain objects.

Image courtesy of http://www.geometricalgebra.net/screenshots.html

Backface Culling

- Triangle Winding order is extremely important for backface culling.
- Backface culling determines whether a polygon is front facing or not by the direction of the Z value of the primitive's normal vector (why the Z value?).
- This normal vector is calculated by taking the cross product of the two edge vectors (v_1-v_0) and (v_2-v_0).

Triangle Traversal

- Once the triangle has been constructed, the triangle traversal stage moves over (traverses) the triangle surface, and creates a fragment for each pixel covered by the triangle.
- Fragments contain vertex data that has been interpolated across the surface from the primitives vertices.
- Each Fragment contains at least the following:
  - The pixel location (X,Y) – Range: ([0,screenWidth], [0,screenHeight])
  - The depth value (Z) linearly interpolation– Range: [0,1]
  - Additional Vertex Data such as vertex color value, texture coordinates and normal vectors.

- These fragments are tested (covered later) and if valid are then sent to the pixel shader, which calculates the final fragment color!

The Pixel Shader

- The pixel shader is responsible for the calculation of the final color for a fragment.
- Pixel shaders may take a variety of input data structs.
- Pixel shaders only return a color value.
- Pixel Shaders are responsible for operations such as texturing and per pixel lighting.

The Texturing Pipeline and Texture Sampling Methods

Texturing

Texture Mapping

- Texture Mapping (Texturing) is a process where we modify a surface at every point by using an image or function.
- This allows us to use a color map (an image) to define the colors of a surface instead of simple vertex colors.
- In addition to using a texture to define color values, it can be used to store other surface information such as normal vectors.
- Texturing is performed in the Pixel Shader Program.

Textures

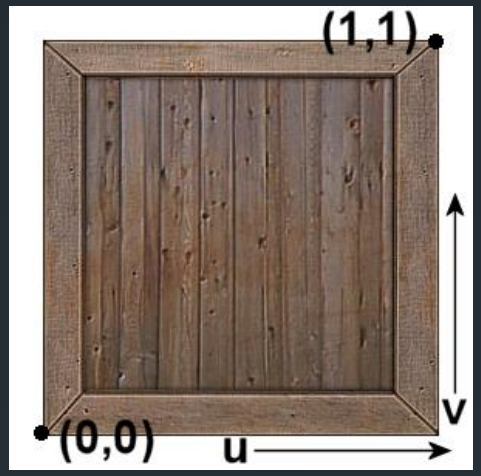
- A texture (or texture map) can be thought of as simple N dimensional data table.
- There are three common types of textures: 1D, 2D and 3D.
- 2D textures are represented by bitmapped images and are the most commonly encountered type and will be our focus.
- Depending on the bit depth of our texture map (8,16,24,32) we can store various data in our textures.

Textures

- Each data element in a texture is called a texture element (texel) and even though each texel is usually represented by a pixel in our texture map, we use the term texel to differentiate it from pixels on the screen.

- The most common type of texture is a color map (diffuse/albedo map), this type of texture changes the surface color of an object.

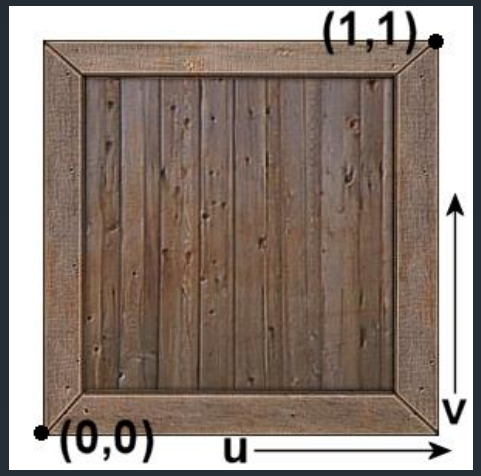
Texture Coordinates



- Each texel is referenced (indexed) by a texture coordinate pair: (u,v), defining the horizontal and vertical pixel positions in the image respectively.

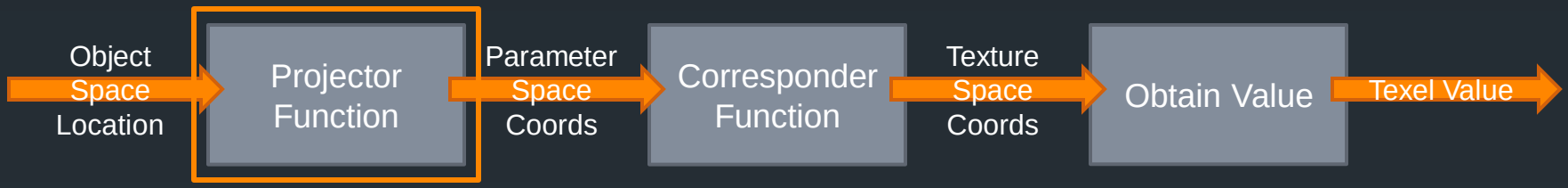
- These uv coordinates range from 0 (start of the texture) to 1 (end of the texture).

Texture Coordinates



- Just like vertex colors, texture coordinates are assigned at each model's vertices and are linearly interpolated across a surface during fragment generation.
- The texturing pipeline ensures that each fragment has a valid texture coordinate pair.

Texturing Pipeline: The Projector Function

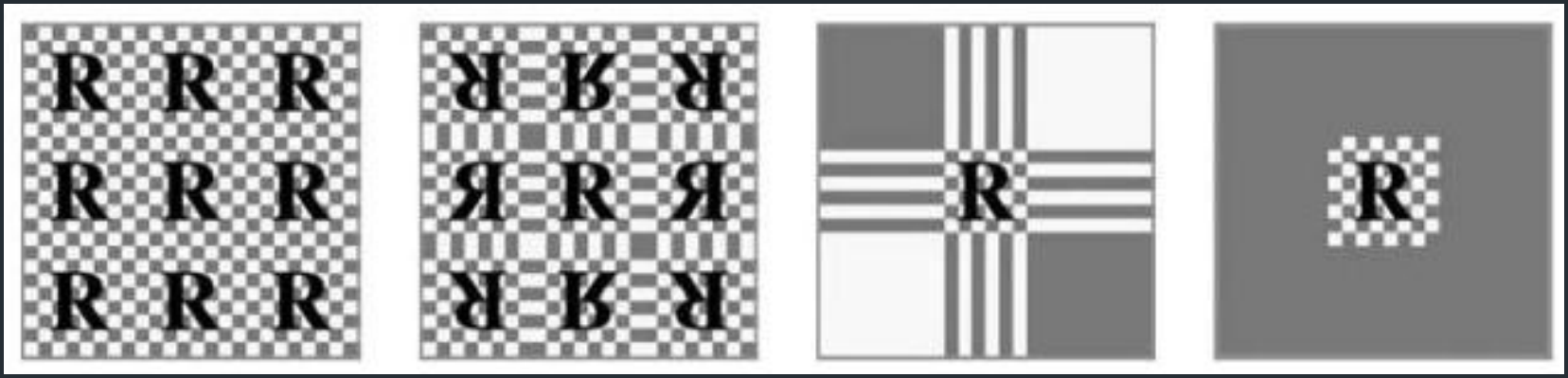


- The Projector Function converts a point from 3D object space to a (u,v) pair in parameter space by interpolating the vertex texture coordinates.
- Parameter space is texture space without the range limitations
- This means that each fragment generated from a triangle face during triangle traversal now has a (u,v) pair associated to it.
- These (u,v) pairs are usually fractions and outside of the [0,1] range for texture coordinates and so do not directly correspond to a valid texel location on the texture map.

Texturing Pipeline: The Corresponder Function

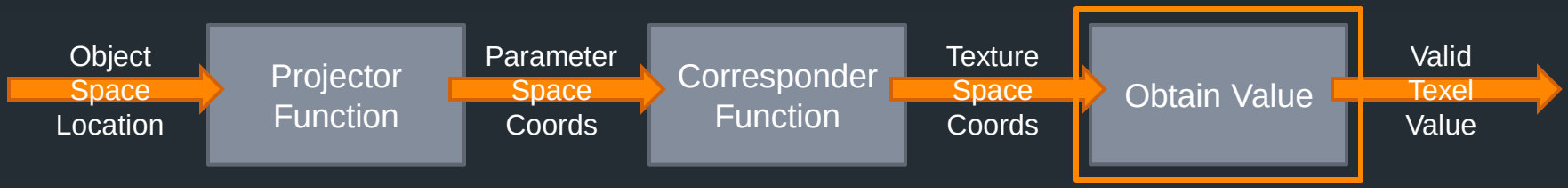
- The corresponder function converts the parameter space (u,v) coordinates to valid texture space coordinates.
- Parameter space coordinates often have u,v values that are invalid (>1 or <0).
- These values need to be converted to the valid texture space range [0,1].
- This clamping is performed using an addressing mode.

Addressing Modes



- Wrap: The texture repeats across the surface. The difference between the start and end u,v values define how many times the texture will be repeated horizontally and vertically.
- Mirror: Same as Wrap, except the texture is now mirrored for every other repetition.
- Clamp/Clamp to Edge: All values outside the range [0,1] are clamped to this range. This results in all coordinates outside texture space being set to the value of the nearest edge texel coordinates.
- Border/Clamp to Border: All values outside the range [0,1] are clamped to a user specified texel value.

Texturing Pipeline: Obtain Value Stage

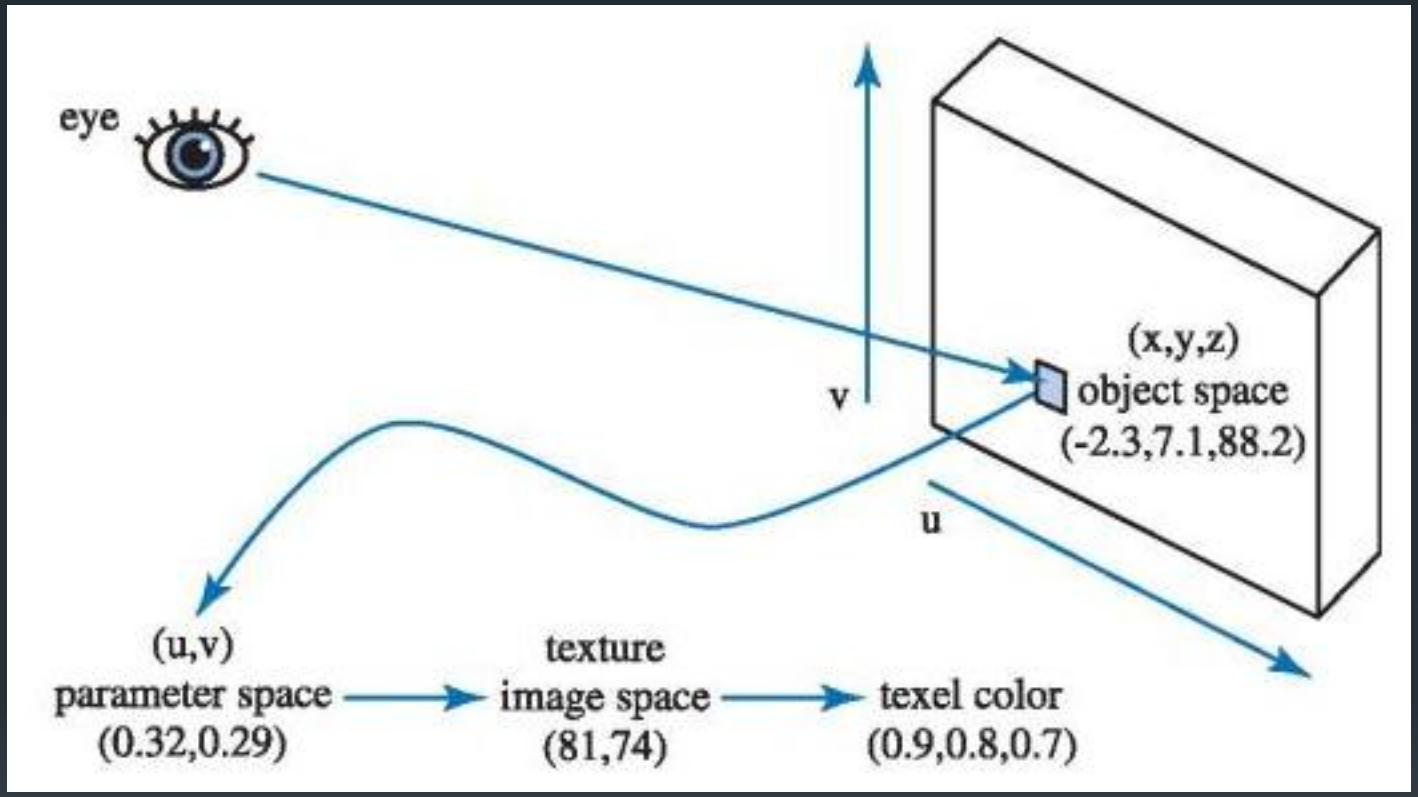


- Texture space coordinates from the corresponder function may be fractions and so not point directly to a texel location.
- The obtain value stage calculates the correct texel value to return for a texture coordinate pair.
- This technique is called sampling/filtering, the simplest solution is to select the texel whose center is nearest to the input texture coordinates. This is called point sampling/nearest neighbor filtering.
- This Stage returns the texel value to the pixel shader

Texturing Pipeline

- The texturing pipeline stage take places in several different stages in the modern GPU pipeline.
- The Projector Function occurs during the Triangle Traversal Stage.
- The Obtain Value Function take place in the Pixel Shader Stage and take the form of an HLSL sampler structure.

Texturing Pipeline Overview



UV Mapping

3D Model

Color Map

- UV Mapping is a technique whereby a 3D model in unfolded or unwrapped onto a flat texture.
- Each vertex in the model is given specific texture coordinates.
- Texturing each face will only use a small subset of the texture map.
- This enables 3D modelers to use a single texture map for extremely complex models.

A Texturing Example

Texture

Required Result

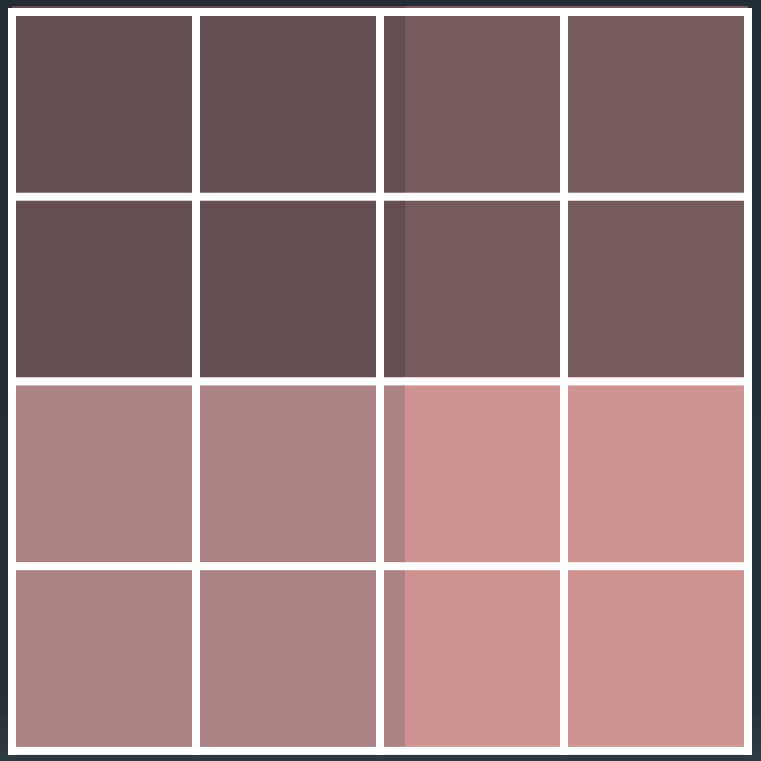
Texel Density, Minification, Magnification, Mipmapping

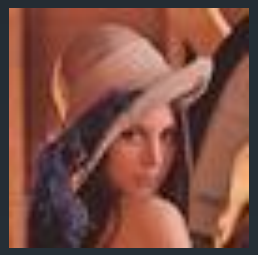
Texturing Artifacts and Aliasing

Texel Density

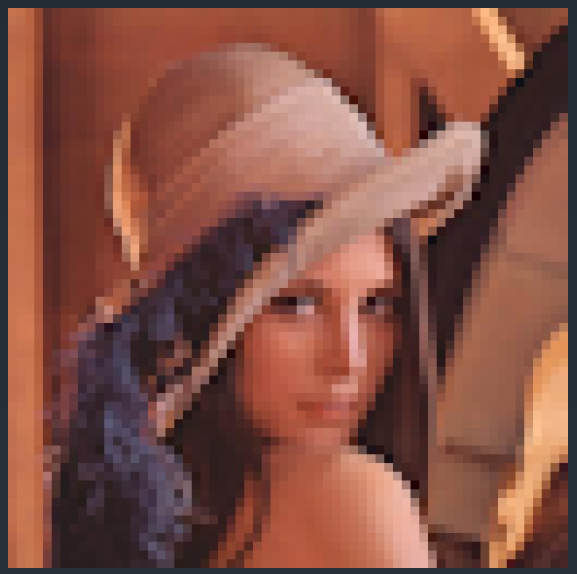
- Texel Density is the ratio of texels to fragments over a surface.
- If we draw a quad that is displayed by a 200x200px block and texture it using a 128x128px texture, then we will have a resulting texel density < 1
- If the texel density is < 1, the texture being used is too small and will effectively be magnified to cover the surface, this is known as magnification
- If the texel density is > 1, the texture being used is too large and will effectively be shrunk to cover the surface, this is known as minification.

Magnification





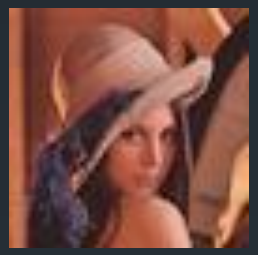
64x64



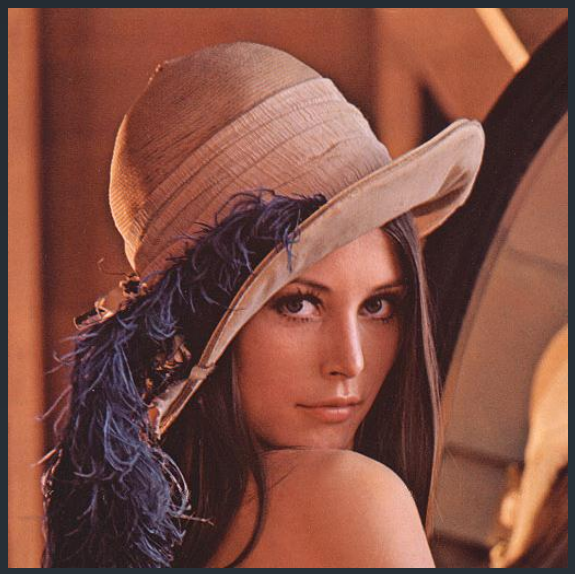
512x512

- Magnification occurs when the texel density is < 1
- This means that each pixel covers less than one texel, as a result texel values get repeated in the final image and this causes pixelation.

Minification



64x64



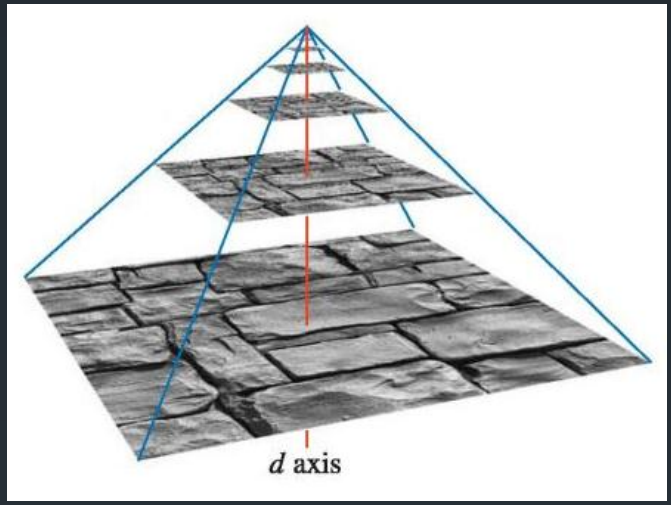
512x512

- Magnification occurs when the texel density is > 1
- This means that each pixel covers more than one texel. Using point sampling to select the texel value now causes severe artifacts known as pixel swimming.

Mipmapping

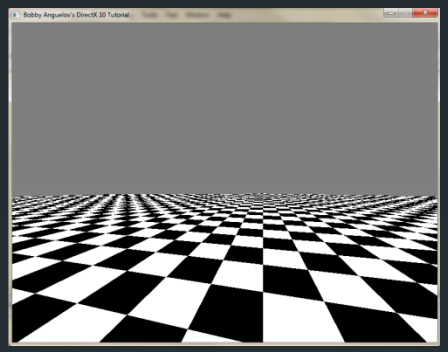
- To correct the artifacts seen due to mini/magnification, we need to ensure that the texel density is as close to 1 as possible.
- It is impossible to adjust the texel density at runtime with a fixed size texture.
- One solution is to create multiple sizes of the same texture and choose the texture size whose texel density is closest to 1. Doing so would greatly reduce the visual artifacts seen.
- This technique is called mipmapping.

Mipmapping



- Mipmapping is a technique that generates a mipmap chain (a set of the original texture in various size).
- The original texture is found at level 0.
- Each subsequent mipmap level (>0) in the chain consist of the previous mipmap downsampled to a quarter of the area.
- Selection of the appropriate mipmap to use is automatically done by the API/GPU.

Texturing Artifacts



Run Example Program!

- Minification causes moire patterns, pixel swimming and aliased lines . In addition it is highly inefficient to render high resolution textures when they will end up being displayed using a very low pixel count.
- Magnification causes blockiness/pixelation and aliasing of lines
- Mipmapping helps fix the above problems but textures pop in and out when swapping between miplevels.

Nearest Neighbor, Bilinear, Trilinear and Anisotropic filtering

Texture Sampling Methods

Texture Filtering

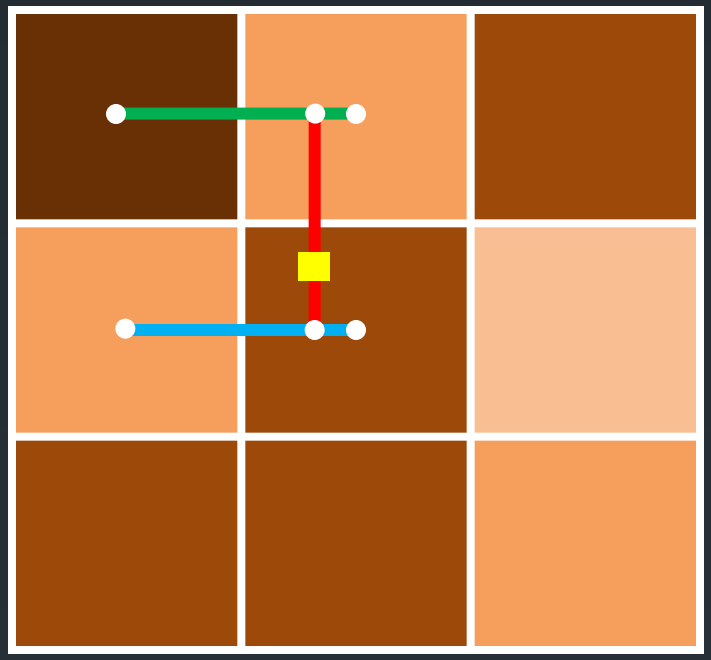
- In a perfect world, texel density will always be 1 (1 texel for each pixel) and so selection of the correct texel value for a fragment will be simple. This is almost never the case so we need methods to select a texel value for each pixel.

- Texture filtering deals with the selection of the texel value for a given fragment, during the obtain value stage. Remember that the texel coordinates provided to the obtain value stage are often fractions.

Nearest Neighbor Filtering

- Nearest neighbor filtering/ point sampling simply rounds the texture space (u,v) coordinates received to the nearest texel. As this is an estimate of the texel to be used, slight view changes may result in a neighboring texel being selected and so often surfaces will swap pixels rapidly as the view changes (this is term pixel swimming).

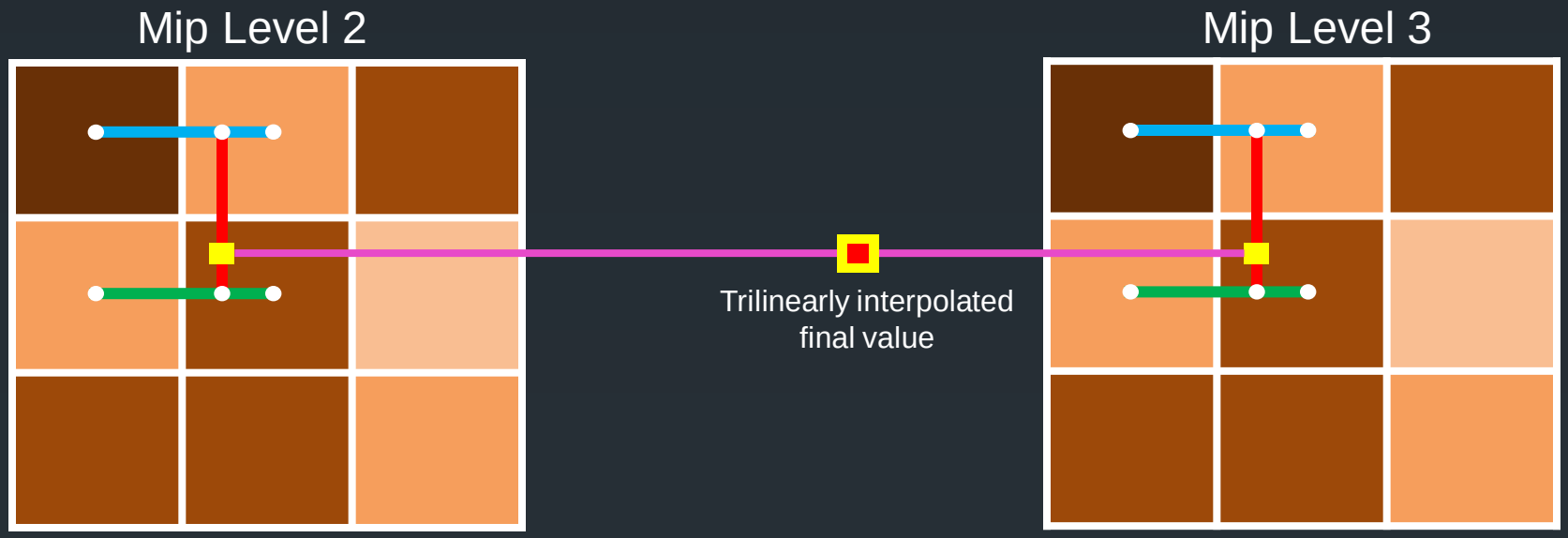
Bilinear Filtering



- Value interpolated across the top two texels
- Value interpolated across bottom two texels
- Final value is the interpolation between the the above values

- Bilinear filtering improves upon point sampling by returning the linear interpolation of the closest texel and its the four surrounding texels as well.
- This prevents pixel swimming and smoothens out jagged lines but causes the scene to be a bit blurry.

Trilinear Filtering

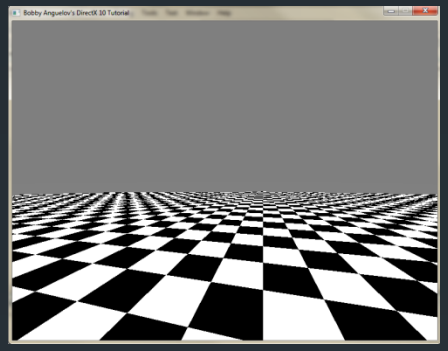


- Mipmapping has one major visual artifact, the swap between mipmap levels is noticeable.
- To prevent this abrupt visual change the closest two mipmap levels are selected and a texel value returned for each mipmap level using bilinear filtering.
- The final texel value returned is a linear interpolation between the two mip levels' bilinearly filtered texel values. The closer a mipmap level's texel density is to 1, the more weight it is given in the interpolation.

Anisotropic Filtering

- Isotropic means the same in all directions and so far all our filtering has been isotropic, anisotropic filtering means that filtering is not the same in all directions.
- Bilinear/Trilinear filtering both filter a texture using a uniform block of texels, which is okay when viewing the surface head on but when the surface is at an oblique angle to the viewer, this is not problematic (**why?**)
- With perspective when viewing a surface at an angle the surface gets stretched into the distance and so the texture filtering needs to adapt to this, anisotropic filtering samples texel using a trapezoidal region corresponding to the view angle.
- Anisotropic greatly improves visual quality at a distance!

Demonstration of Texture Filtering



Run Example Program!

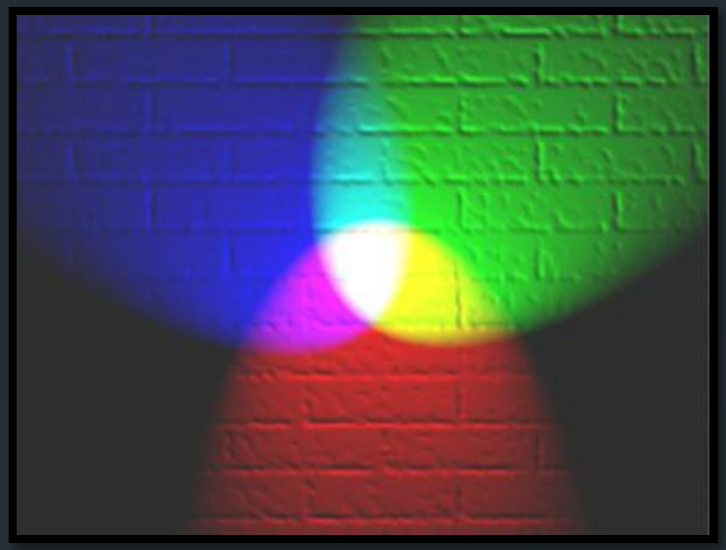
Lights, Materials, Reflection Models and Shading Models

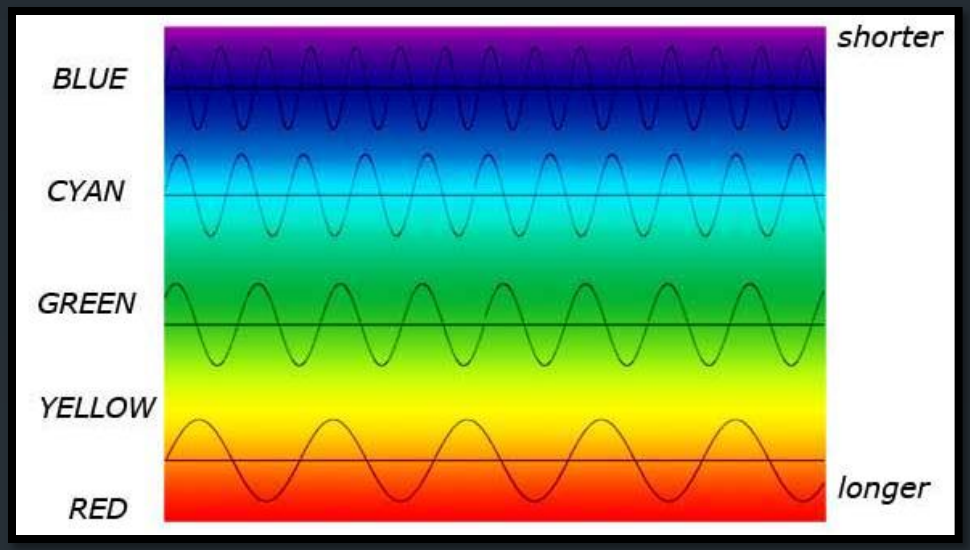
Lighting

Light

- Light is made up of electromagnetic waves that are picked up by the human eye (an electromagnetic sensor).
- There are various types of light sources around us, both natural and man-made. These light sources make it possible for use to see around us due to the reflection of the electromagnetic waves off of objects.
- Light waves are directional and several visual phenomena result from this e.g. shadows, lens flares, light volumes, etc.
- These visual phenomena are very important in the computer graphics fields as they add definition and realism to a scene.

Light Colors





- Light waves vary in wavelength and frequency. We see these variations as color.
- Human vision is tri-chromatic, which means that the eye has three color receptors. This is the reason why we generally model color as a combination of three primary colors (Red , Green and Blue).
- Since waves are additive so are light colors. This means that any color can be created by a combination of the three primary light colors.

Light Intensity and Attenuation

- A light source emits light waves with a certain amplitude (or energy), the greater the amplitude of the waves emitted the brighter that light source appears.
- The amplitude of the light wave is termed its intensity.
- As light travels, it loses energy and its amplitude decreases, this is termed attenuation. Think about how brightness decreases the further you are away from a light source.
- The amount of attenuation exhibited by a light wave is calculated by a distance falloff function, the simplest one being:

$$Attenuation(d) = \frac{1}{d^2}$$

Where d is the distance from the light source

Light in Computer Graphics

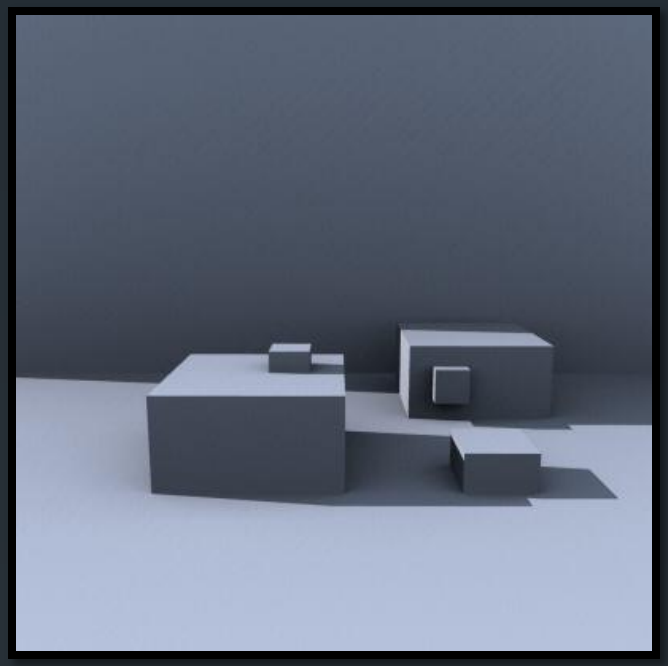
- Light intensity in computer graphics is defined in the same manner as a color: $L_{intensity}$ = [Red, Green, Blue]
- Each value represents the intensity of a color wave and has a range of [0,1]
- Attenuation is applied to a light by a scalar multiplication of the light with an attenuation value (A)
- Or by piecewise vector multiplication with a light wave attenuation triplet $A = [A_{red}, A_{green}, A_{blue}]$. This allows for 'per color' attenuation.

The importance of Lighting

Directional Lights, Point Light and Spotlights

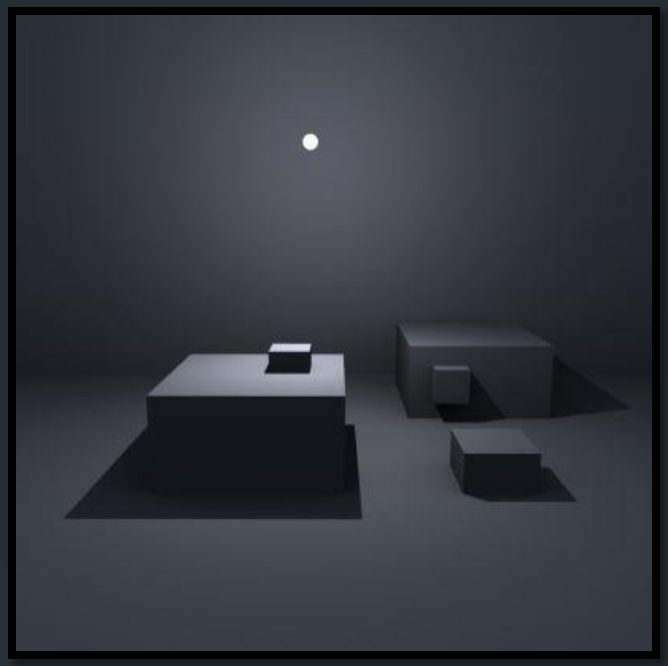
Light Sources

Directional Lights



- A directional light source is an infinitely distant light source (i.e. the sun).
- Directional lights are defined by:
  - A light intensity: $L_{intensity}$
  - A direction vector: $L_{direction}$
- Directional lights do not exhibit attenuation.

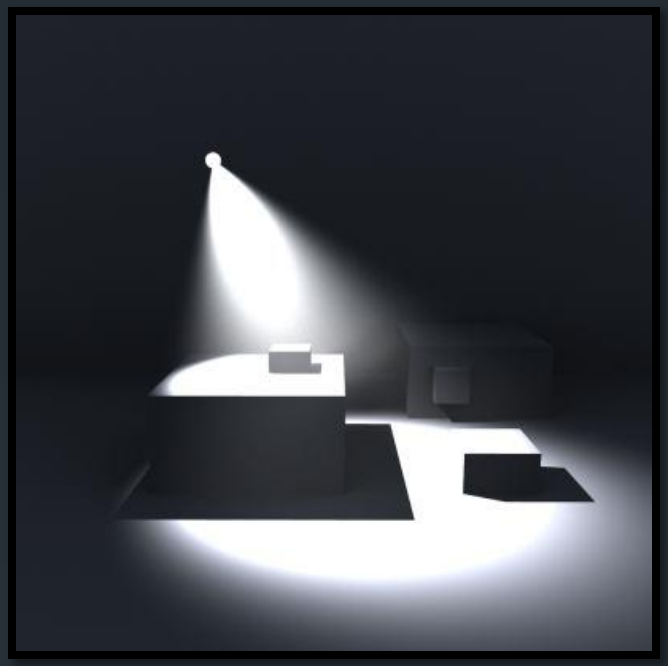
Point Lights



- A point light source is a light source at a specific location that emits light equally in all directions.
- Point lights are defined by:
  - A light intensity: $L_{intensity}$
  - A light position: $L_{position}$
- Point lights exhibit attenuation.

Images from the Blender Documentation - http://www.blender.org/documentation

Spotlights



- A spotlight source is a point light which is enclosed in a cone directing the light in a circular beam.
- Spotlights are defined by:
  - A light intensity: $L_{intensity}$
  - A light position: $L_{position}$
  - A direction vector: $L_{direction}$
  - A spotlight exponent: S_{exp}
- The spotlight exponent (S_{exp}) controls the tightness of the light beam
- Spotlights exhibit attenuation.

Images from the Blender Documentation - http://www.blender.org/documentation

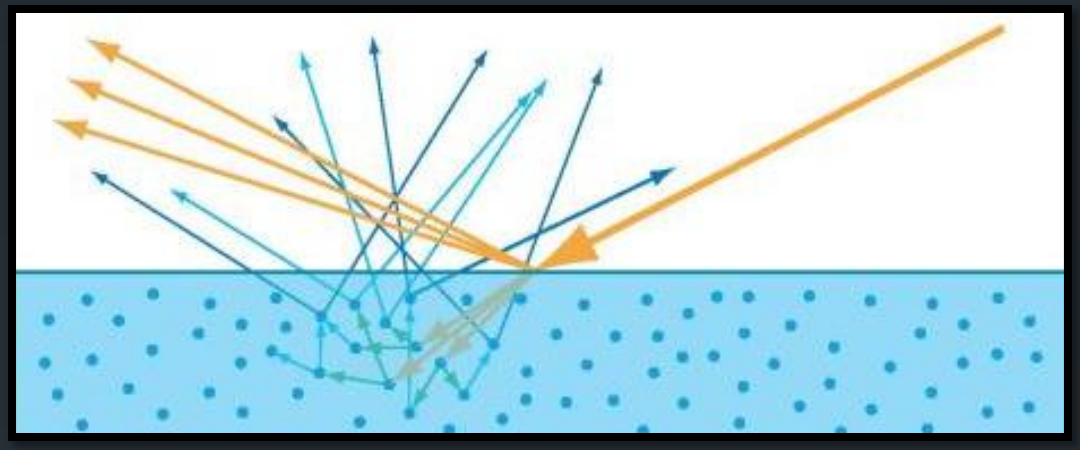
Ambient Light

- Ambient Light is the base light intensity applied to every object in the scene.
- This light source has no direction and is applied to all objects equally.
- Ambient light is defined by an intensity value: $I_{ambient}$

Diffuse and Specular Reflections, Surfaces

Light Surface Interactions

Surface Interaction

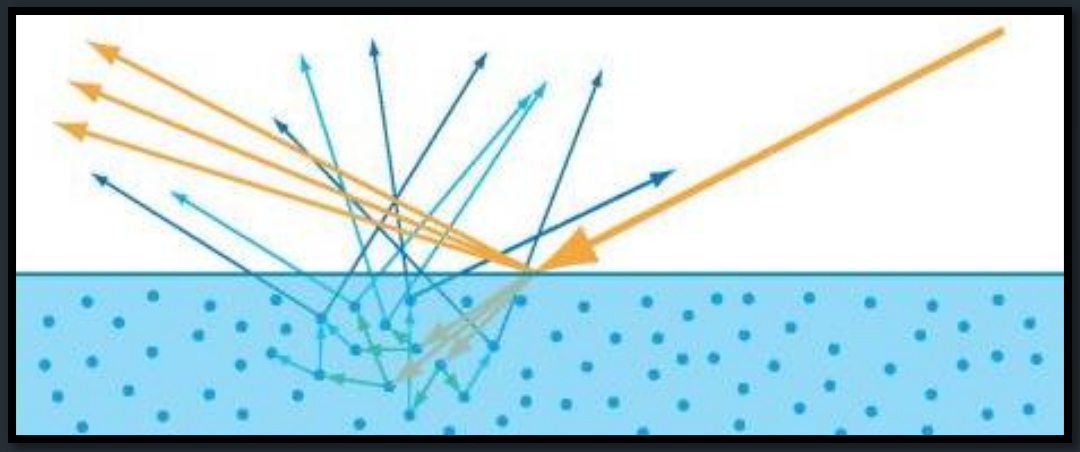


- The angle between the incoming light ray and the surface is called the angle of incidence.

- When a light wave collides with a surface, two reflections occur:

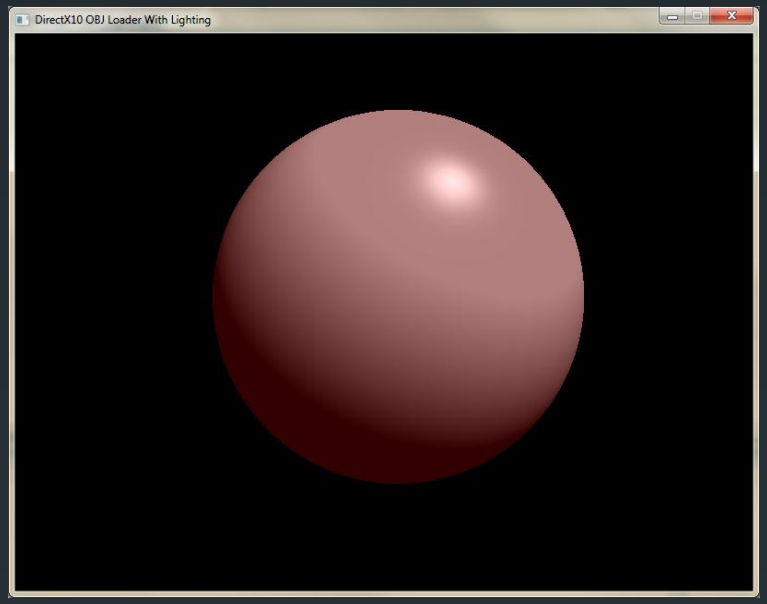
  - Specular Reflection: Some of the light is immediately reflected back at the surface (the orange reflected rays in the image above).
  - Diffuse Reflection: The rest of the light is refracted into the surface and then reflected back out (the blue reflected rays in the image above).

Surface Interaction: Specular Reflections



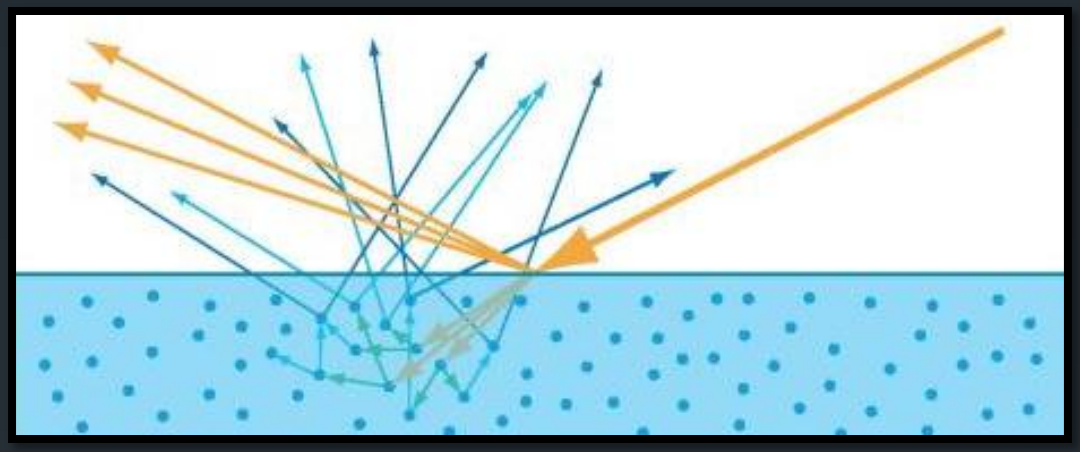
- Specular reflection occurs when the light ray hits a surface and is immediately reflected back without entering the surface.
- The reflected light rays leave the surface in an arc, at an angle roughly equal to the angle of incidence.
- The surface smoothness affects the width of the arc of the reflected specular light. Smooth surfaces will have a tighter spread while rougher surfaces will reflect light in a wider arc.

Surface Interaction: Specular Highlights



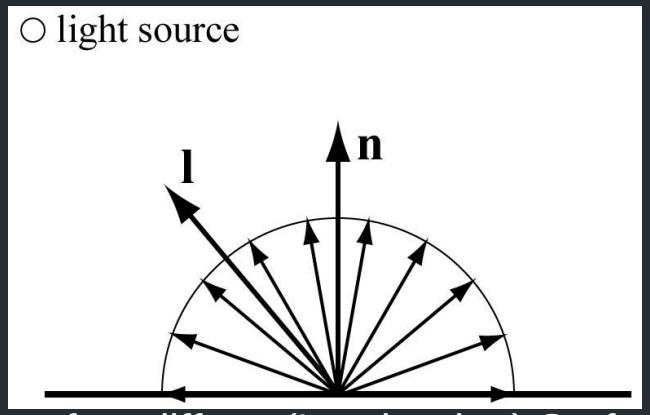
- Specular highlights occur when the view angle is very similar to the angle of the reflected specular light.
- This means that the viewer will see the almost perfect reflection of the incident light ray at that point.
- The closer the two vectors get the greater the intensity. This causes the specular highlight seen in the image to the left.
- The surface smoothness determines the size of the specular highlights, smooth surfaces have small specular highlights since the specular arc is small.

Surface Interaction: Diffuse Reflections



- Diffuse Reflection occurs when light rays enter a surface. The undergo transmission, absorption and scattering inside of the material before exiting the surface.
- A diffuse reflection will scatter the exiting light rays in random directions, and there is no dependence on the angle of incidence.
- Perfectly diffuse surfaces reflect exiting light equally in all directions, these surfaces are called Lambertian surfaces.

Diffuse Reflections: Lamberts Law



A perfect diffuse (Lambertian) Surface

- Lambert's Law states that the reflected light ray intensity at a perfectly diffuse surface is directly proportional to the cosine of the angle between the incoming light ray vector (from the surface to the light ($L = -L_{direction}$)) and the normal of the surface (θ).

$$I_{surface} = Lint_{ensity} \times \cos(\theta) \approx Lintens_{ity} \times n.l$$

- This is commonly used to model the diffuse reflection at a surface, and is the root of the standard N.L diffuse lighting model.

Phong and Blinn-Phong Reflection Models

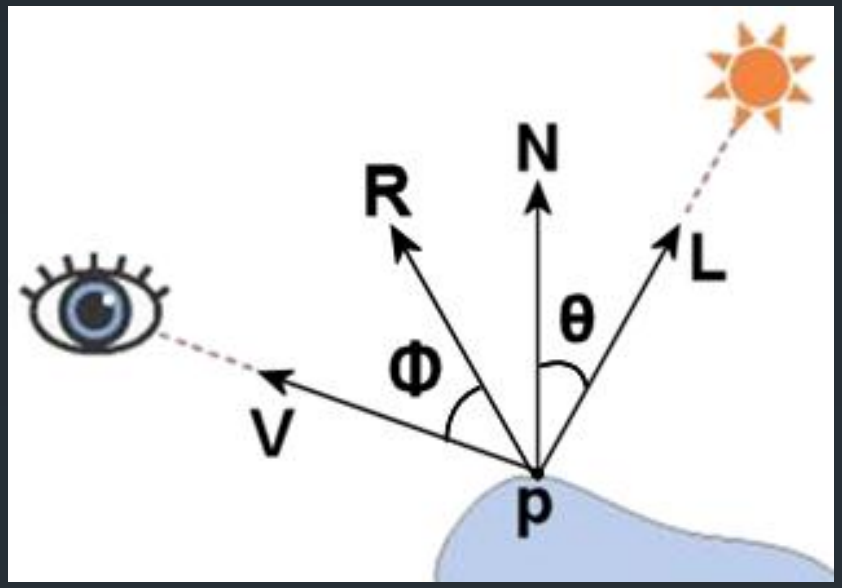
Local Reflection Models

The Phong Reflection Model

- In 1973, Bui Tuong Phong, a researcher at the University of Utah proposed a model for the reflection of light on surfaces and a model for the shading of these surfaces.

- This model is the basis of most modern lighting techniques found in computer graphics. It was so popular that a modified version (blinn-phong) is part of the fixed function pipeline in both openGL and DirectX <9 APIs.

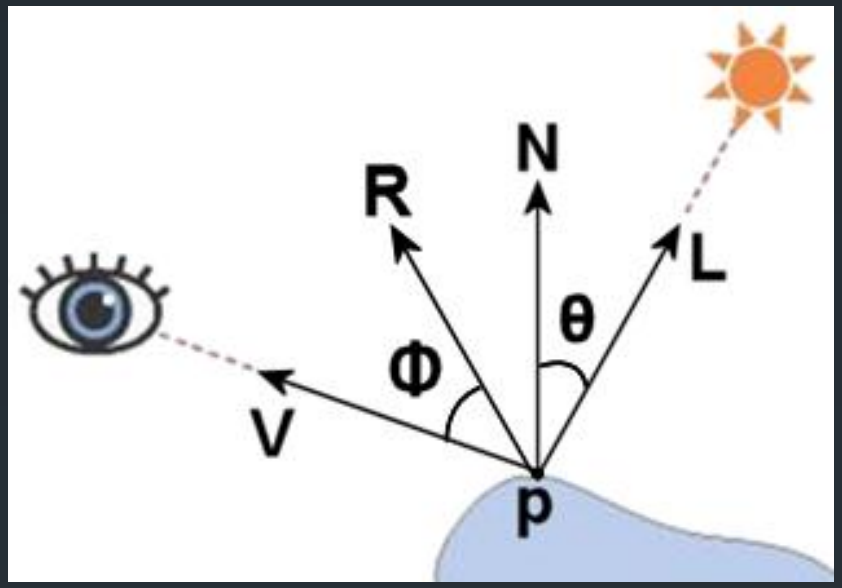
The Phong Reflection Model



- The Phong reflection model calculates the reflected light intensity at any point on a surface as the sum of the ambient, diffuse and specular reflection intensities:

$$I = Ia + Id + Is$$

The Phong Reflection Model



- To compute the total surface intensity, we need the following data at each point on a surface:
  - **N** – The Surface Normal Vector
  - **L** – A vector from the surface to the light (negative light direction vector)
  - **R** – The perfect reflection of L around N
  - **V** – The vector from the surface to camera (eye)
  - Surface Reflection Properties

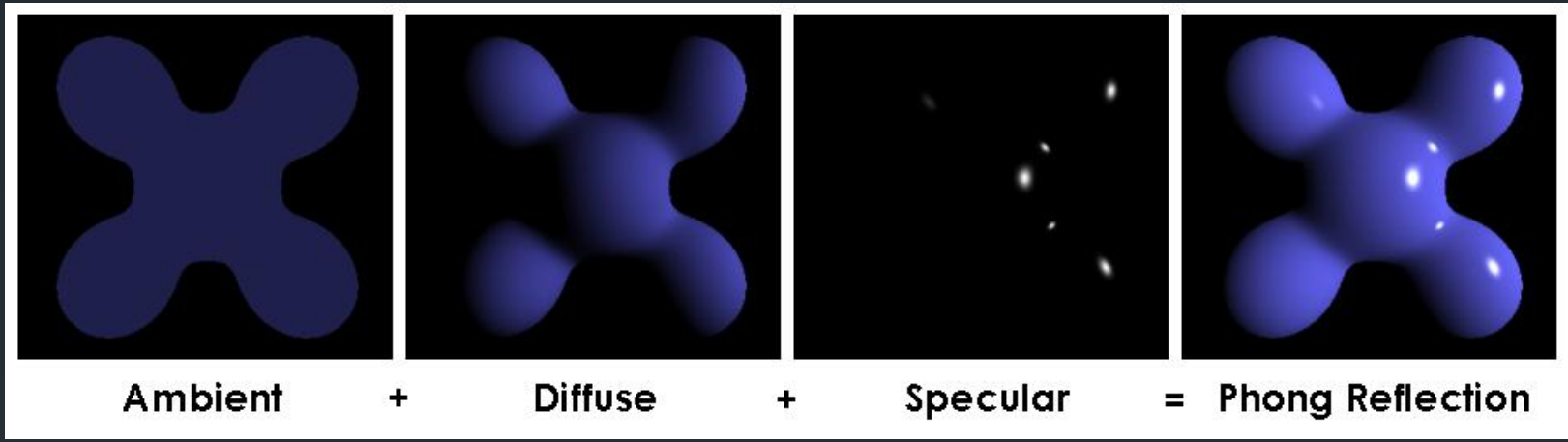
Basic Surface Materials

- Different materials tend to reflect light differently. Smooth materials like metals reflect more specular light than rough materials like wood.
- To be able to simulate different materials, we need information about how a surface reflects incoming light.

- Each material defines the following reflectivity constants:
  - Ambient Reflectivity Constant – K_a
  - Diffuse Reflectivity Constant – K_d
  - Specular Reflectivity Constant – K_s
  - Surface Smoothness Value – α (this value controls the size of the specular highlights)

- These reflectivity constants may be scalar values or triplets allowing for “per color” reflectivity constants.

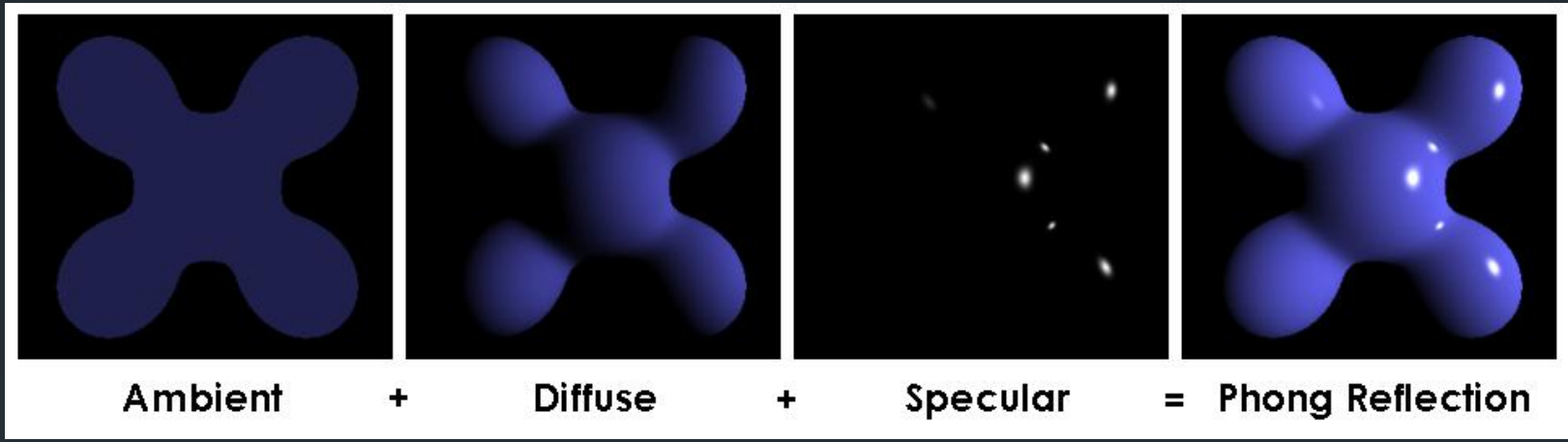
The Phong RM: Ambient Term



- This term defines the ambient light intensity at a point on the surface.

$$I_a = K_a \times I_{ambient}$$

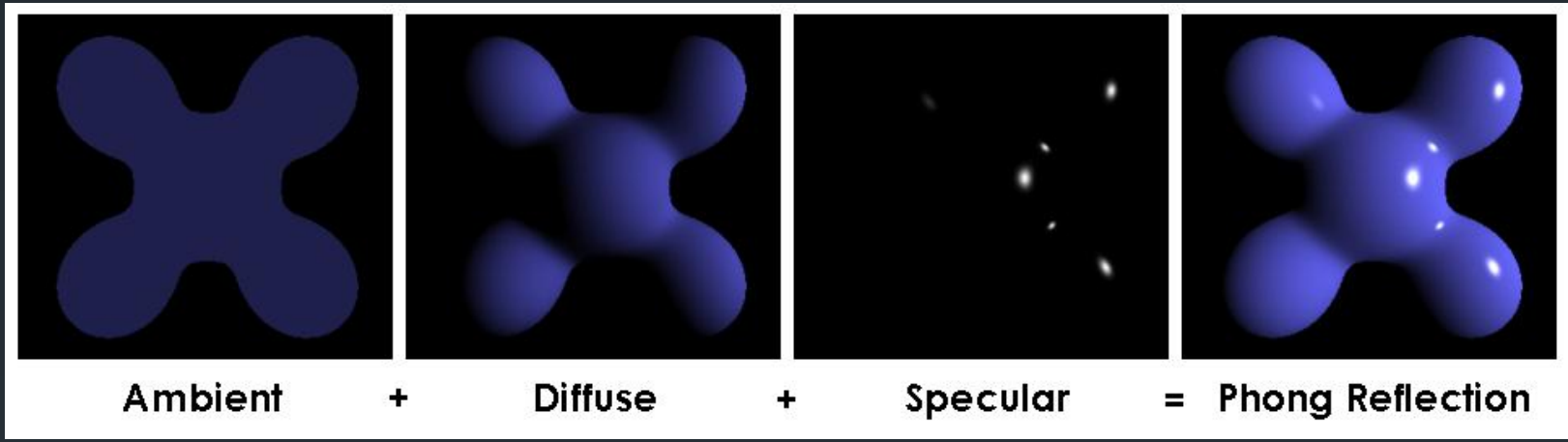
The Phong RM: Diffuse Term



- This term defines the diffuse intensity reflected at the surface, we assume all surfaces are perfectly diffuse and use Lambert's law.

$$I_d = K_d \times L_{intensity} \times N.L$$

The Phong RM: Specular Term



- This term defines the specular intensity reflected at the surface, the closer the view vector (V) is to the reflected vector (R) the stronger the intensity seen.

$$I_s = K_s \times L_{intensity} \times (R.V)^{\alpha}$$
$$where\ R = (2N \times N.L) - L$$

The Phong RM: Conclusion

- Once we have calculate the 3 intensities we combine them to produce:

$$I = K_a \times I_{ambient} + L_{intensity} \times (K_d(N.L) + K_s(R.V)^{\alpha})$$

- In the case of multiple lights, since lighting is additive and ambient lighting is only applied once:

$$I = K_a \times I_{ambient} + \sum_{n=0}^{n<lights} L_{intensity_n} \times (K_d(N.L_n) + K_s(R_n.V)^{\alpha})$$

The Blinn-Phong Reflection Model

- In 1977, Jim Blinn proposed a modification to the Phong reflection model aimed at improving the performance of the calculation of the specular term.
- He proposed replacing the R.V term with N.H where H is a halfway vector between the view direction vector(V) and the light direction vector (L).

$$H = \frac{L + V}{|L + V|}$$

- Since the halfway angle H is smaller than R this has the effect of creating smaller specular highlights using the α exponent.
- To counter this we use a new exponent a to bring the results of the two equations closer together.

The Blinn-Phong Reflection Model

- The specular term (Phong):

$$I_s = K_s \times L_{intensity} \times (R.V)^{\alpha}$$

- now becomes (Blinn-Phong):

$$I_s = K_s \times L_{intensity} \times (N.H)^{a}$$

Flat, Gouraud and Phong Shading Models

Shading Models

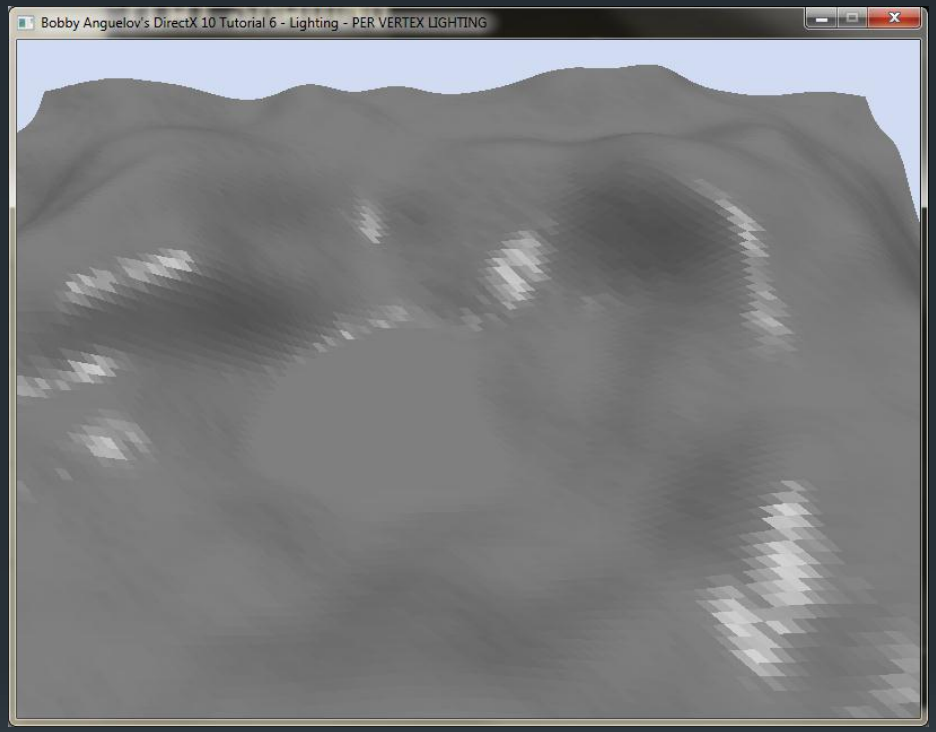
Shading Models

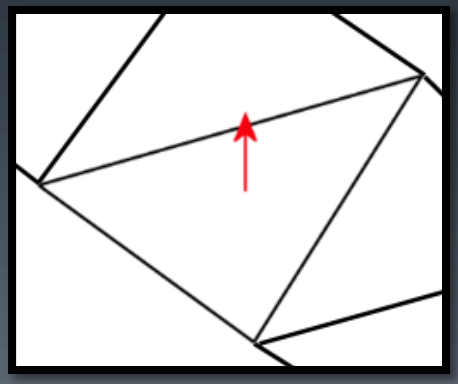
- Shading is the technique of applying a reflection model (phong, blinn, cook-torrance, etc) across an object's surfaces.

There are three main types of shading you will encounter:

- Flat Shading (Per Primitive)
- Gouraud Shading (Per Vertex)
- Phong Shading (Per Pixel)

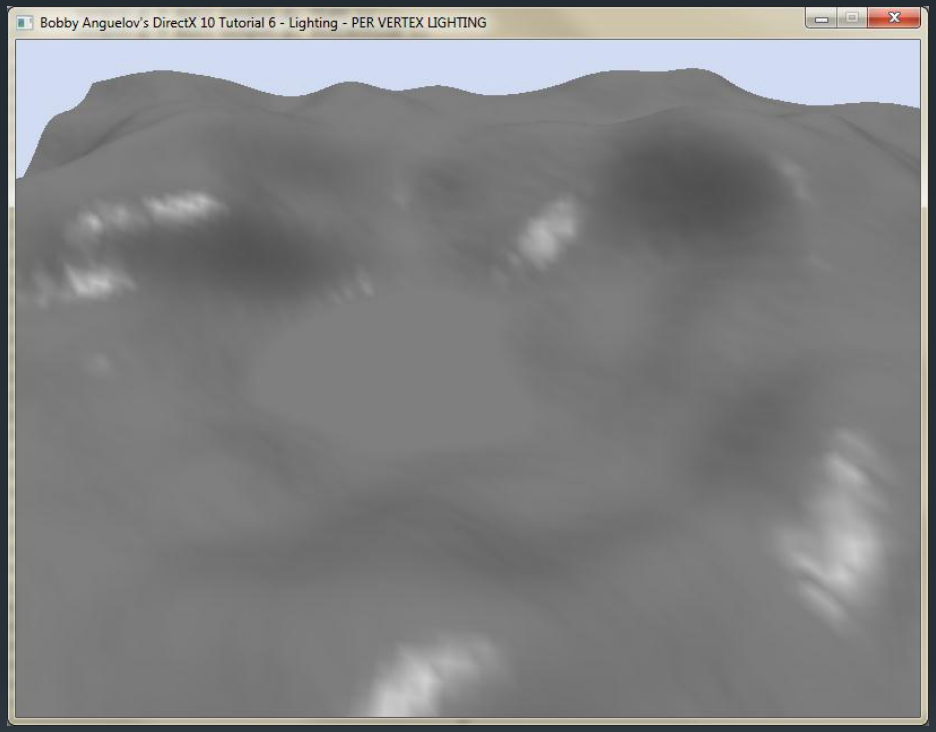
Flat Shading

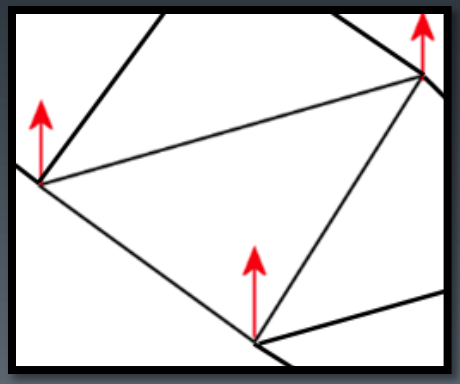




- The simplest form of shading is flat shading, in this technique, the reflection model is computed a single time per primitive, and the resulting intensity is used for the entire primitive's surface.
- This is often achieved by using the primitive's normal in the equation and results in obvious blockiness at the edges of primitives as the intensities differ for each primitive.

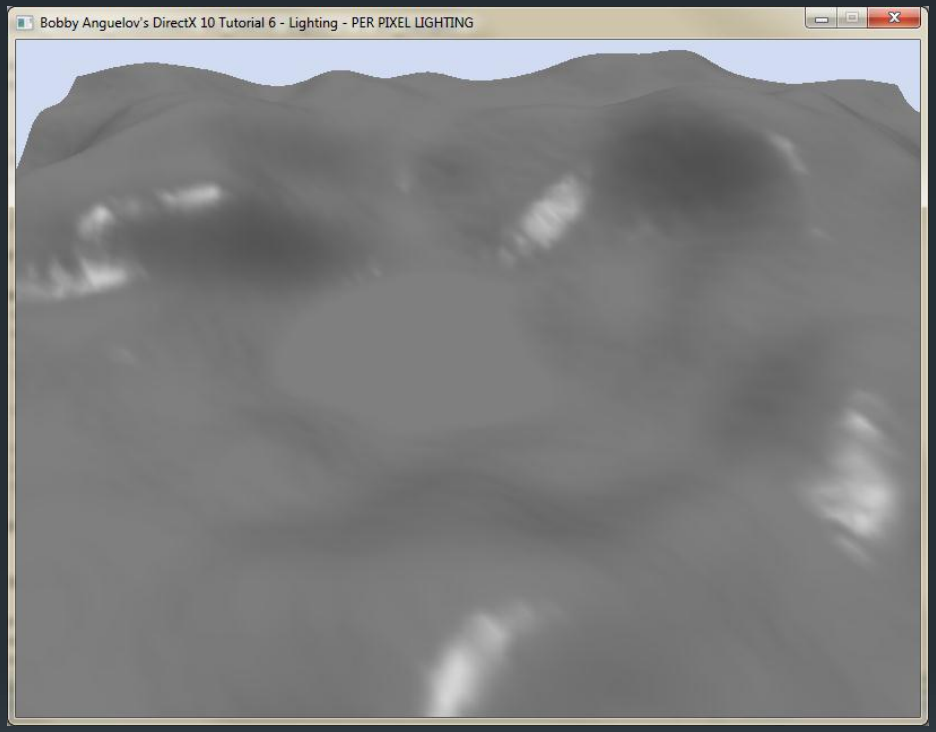
Gouraud Shading

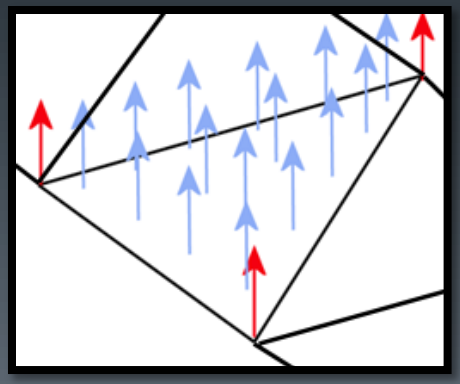




- Gouraud Shading evaluates the reflection model at each vertex in the primitive (per vertex).
- The resulting intensities are interpolated across the primitive during fragment generation (just like vertex colors).
- Instead of using a single normal per a primitive, we use the normal at each vertex.
- In a 3D mesh, vertices are often shared by multiple connected primitives, each one having its own normal. The vertex normal is the average of the normals of all primitives that share that vertex.

Phong Shading





- Phong Shading evaluates the reflection model at each pixel in the primitive (per pixel).
- Vertex positions and Normals are interpolated across the surface of the primitive.
- The reflection model is evaluated at each pixel using the new normal and position.
- Since the reflection model is evaluated at each pixel, this method has a much higher processing cost, but provides the best quality especially in regards to specular highlight quality.

Shading Models: Comparison

Flat Shading / Per Primitive

Gouraud Shading / Per Vertex

Phong Shading / Per Pixel

Depth Testing, Stencil Testing and Alpha Testing

Fragment Testing

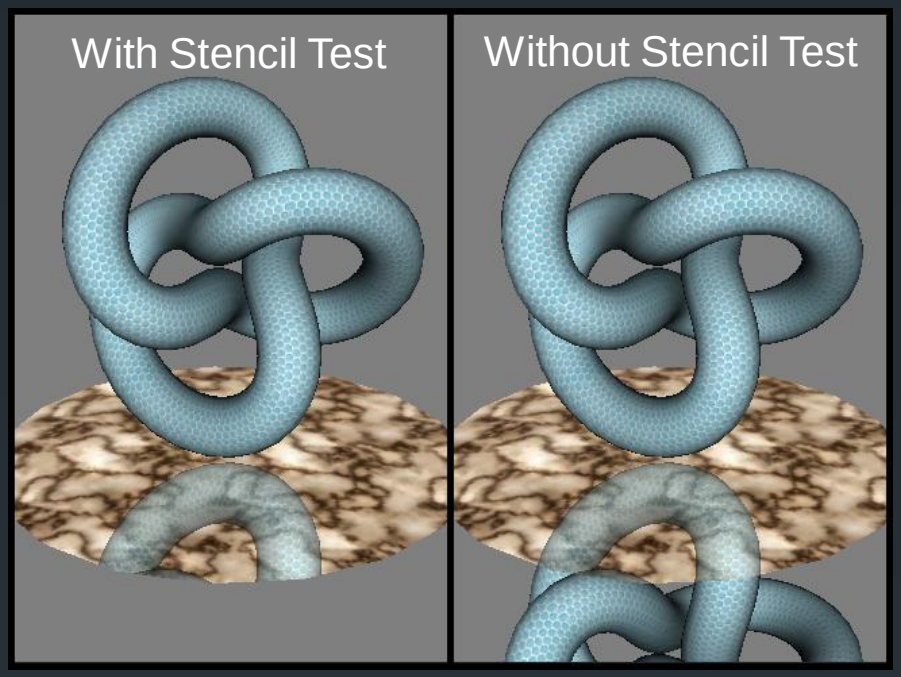
Fragment Testing

- Fragment Testing is a optimization technique to discard fragments that are not visible in the image saving an expensive write to the final render target.
- Fragment Testing occurs in or after the pixel shader stage (although a early depth test can performed prior to the pixel shader stage)
- Each fragment is tested against some parameters (often stored within a texture) and if the fragment fails the test it is discarded.

Alpha Testing

- Alpha testing is used for rendering that features transparency (not translucency). e.g. Chain link fence.
- Alpha testing uses the color map (diffuse) texture for an object and occurs within the pixel shader.
- When a color value is read from the diffuse texture, the alpha component is check and if its below a certain level, that fragment is discarded.
- This prevents the pixel shader from running any further expensive calculations on pixels that are invisible.

Stencil Testing

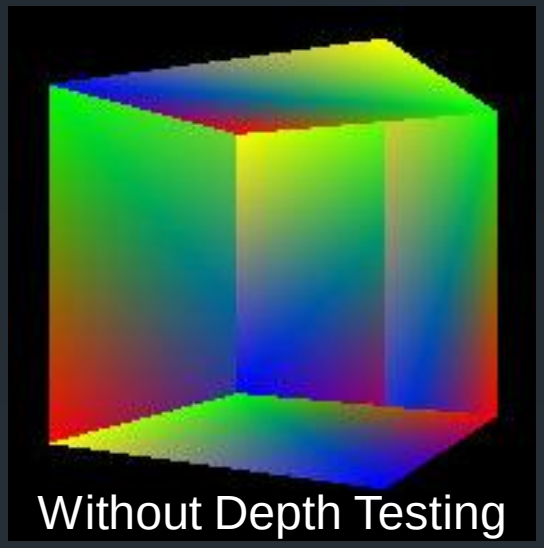


- Stencil testing occurs after the pixel shader stage.
- Stencil testing makes use of a stencil buffer texture to perform the fragment test.
- A stencil buffer is a texture with the same dimensions as the frame buffer.
- It acts as a mask allowing only certain areas of the frame buffer to be drawn to.

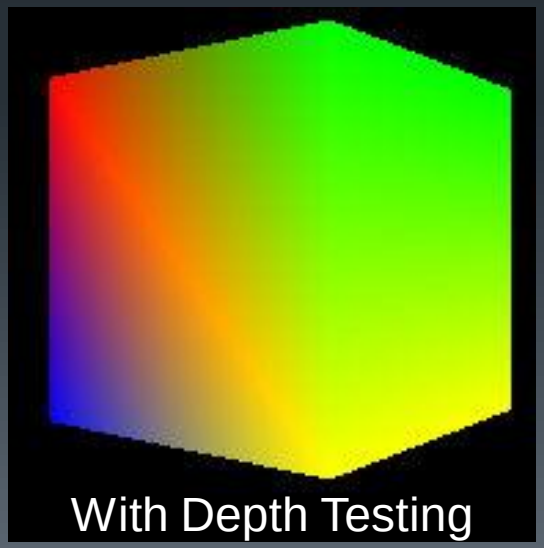
- After a fragment is processed by the pixel shader, it's position is checked in the stencil buffer, if the stencil buffer texel at that position allows writing then the fragment is written to the frame buffer otherwise it is discarded.

Image courtesy of O'Reilly IPhone 3D programming: http://iphone-3d-programming.labs.oreilly.com/ch06.html

Depth Testing



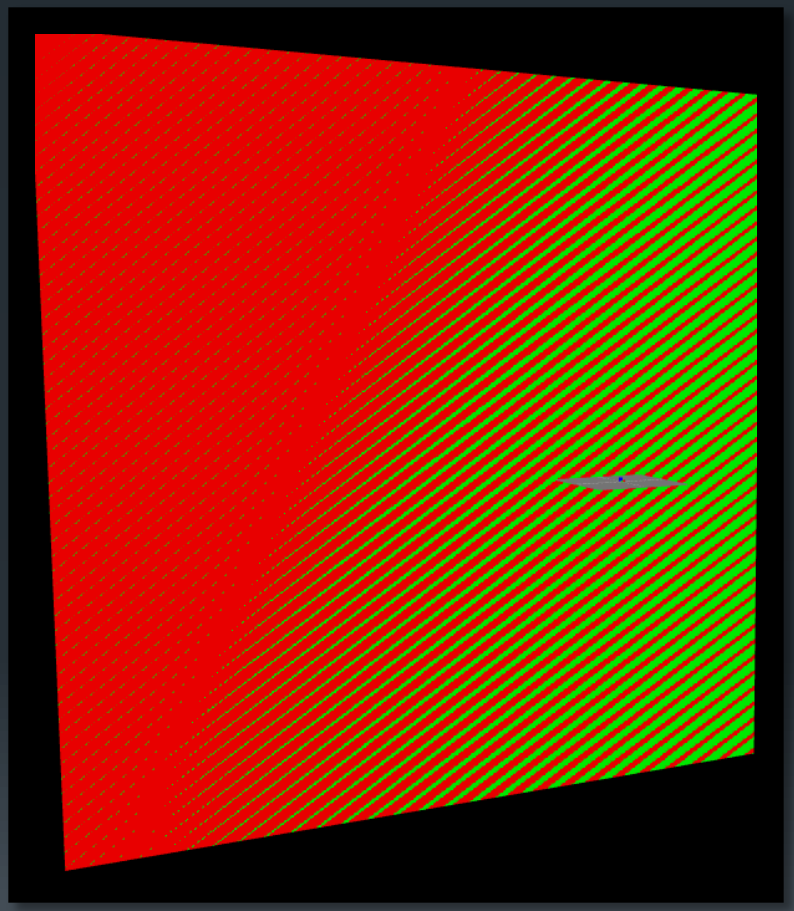
Without Depth Testing



With Depth Testing

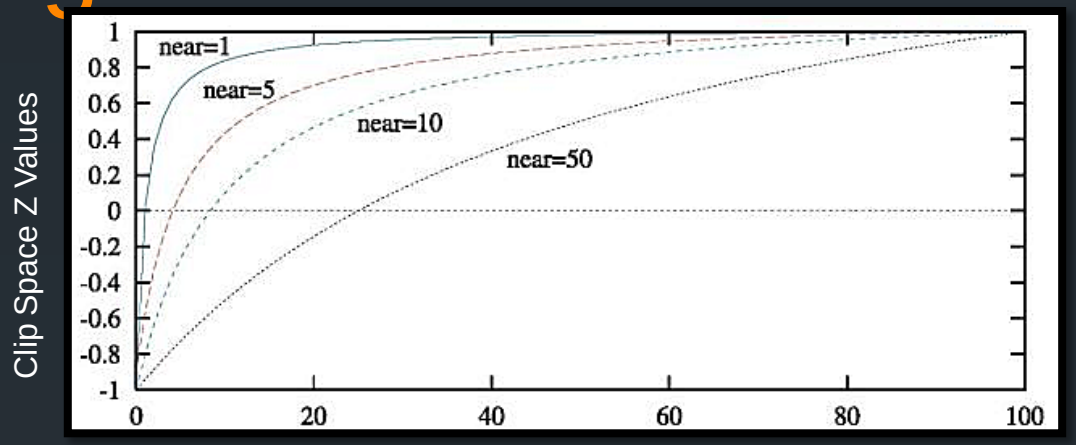
- Without depth testing, overdraw occurs depending on the order of the polygons drawn and results in severe visual anomalies due to far faces being drawn on top of near faces.
- To combat that, a depth buffer texture is used. Like the stencil buffer, the depth buffer is the same size as the frame buffer and each value in the buffer contains the depth value of each corresponding pixel in the frame buffer.
- When depth testing is enabled, prior to a fragment being drawn to the frame buffer, its depth value is compared to the current value in the depth buffer. If the fragment has a lower or equal value then it gets drawn to the frame (and its depth value gets recorded in the depth buffer) otherwise it gets discarded.
- Depth Testing is also known as Z-testing.

Z-Fighting



- Z-fighting occurs when we have two primitives that are coplanar and their depth values are very close to each other. Primitives appear to be merged together.
- The depth value stored in the depth buffer has limited precision and so depth test errors occur when a primitives interpolated high precision z-values are checked against the low precision depth buffer value.
- This only occurs with perspective projections due to transformation of the view frustum into a unit cube.

Z-Fighting



- During the projection of geometry into the view volume, the z value for each vertex gets divided by the w value, this occurs since the w value is not 1 after applying the perspective transform.
- This has the side effect that the precision of the clip-space depth values is not equally distributed across the view volume. The Range [0,1] is mapped to [0, 2^b -1] for a perspective projection where b is the depth buffer bit depth.
- Most of the precision occurs closer to the near plane and so distant objects (large z values) are defined in a much smaller range with lower precision.
- In the graph above the view frustum far plane is set at a depth of 100. Z values in the range [20,100] are defined in an interval of ~1.2 clip space units when the near plane is set to 50 and in an interval of only ~0.1 units with the near plane set to 1.
- There are various solutions to resolve z-fighting: the two most common ones are to add a z-offset to geometry or to simply increase the depth buffer's bit depth providing a higher degree of precision. Z-buffers historical only had 16bit precision but 24bit and even 32bit z-buffers are common today.

The Blending Equation and Alpha Blending

Blending

The Blending Stage

- This is the very last stage in the graphical pipeline and is the only stage that actually modifies the render target (frame buffer).
- Fragment that pass the testing stage (depth/stencil/etc) are valid and must be drawn to the render target.
- Drawing to the render target is performed by the blending stage using a blending equation to allow for a variety of visual effects like transparency and so on.
- The blending equation returns the final pixel color as a blend of the current fragment color and the current pixel color in the render target.

The Blending Equation

- The fragment is referred to as the source and the render target pixel is referred to as the destination.

- The blending equation is really simple:

$$Final\ Color = src\ color \times src\ blend\ factor \otimes dest\ color \times dest\ blend\ factor$$

$$where \otimes is\ the\ blending\ operator$$

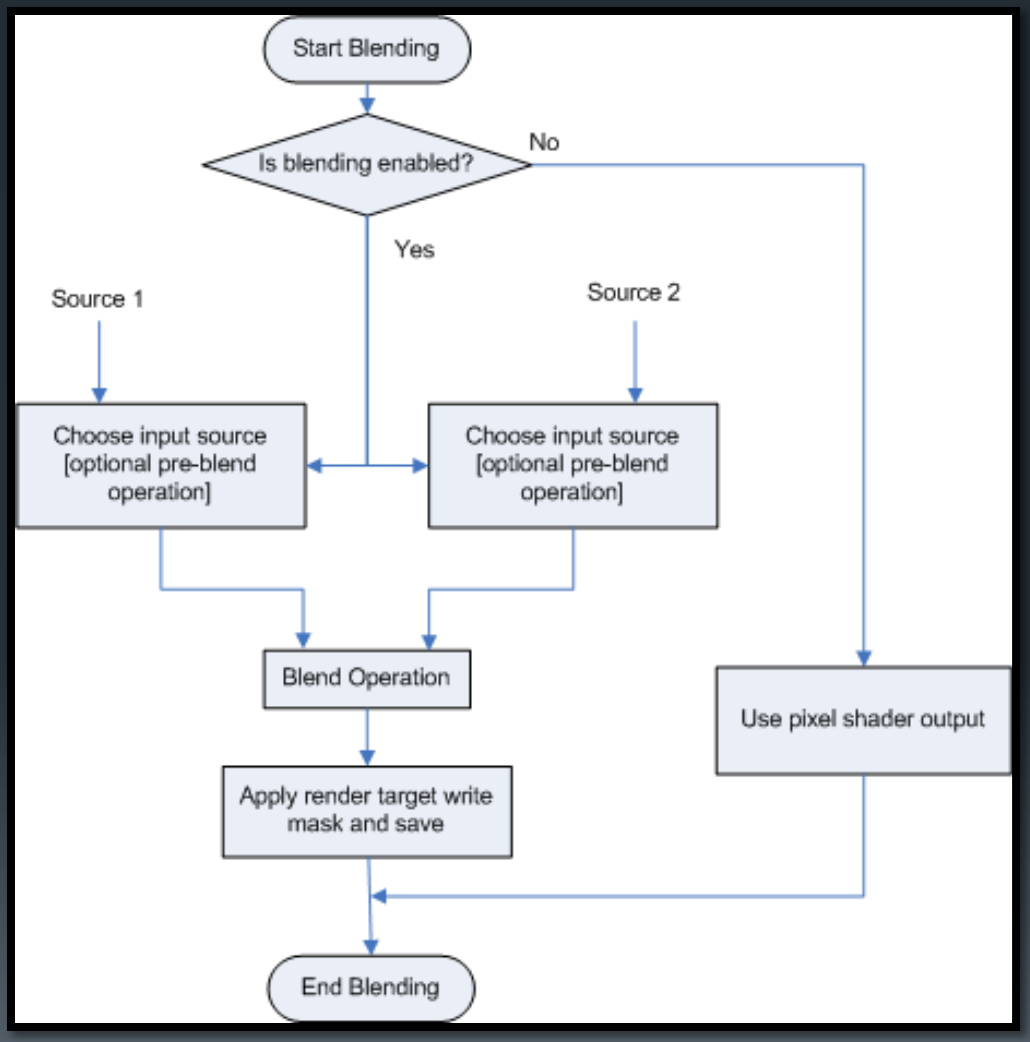
Blending Factors

- The common blending factors are:
  - 0
  - 1
  - Color Values (Source or Destination)
  - Alpha Values (Source or Destination)
- There are many more blending factors so check your API specs.

Blending Operators

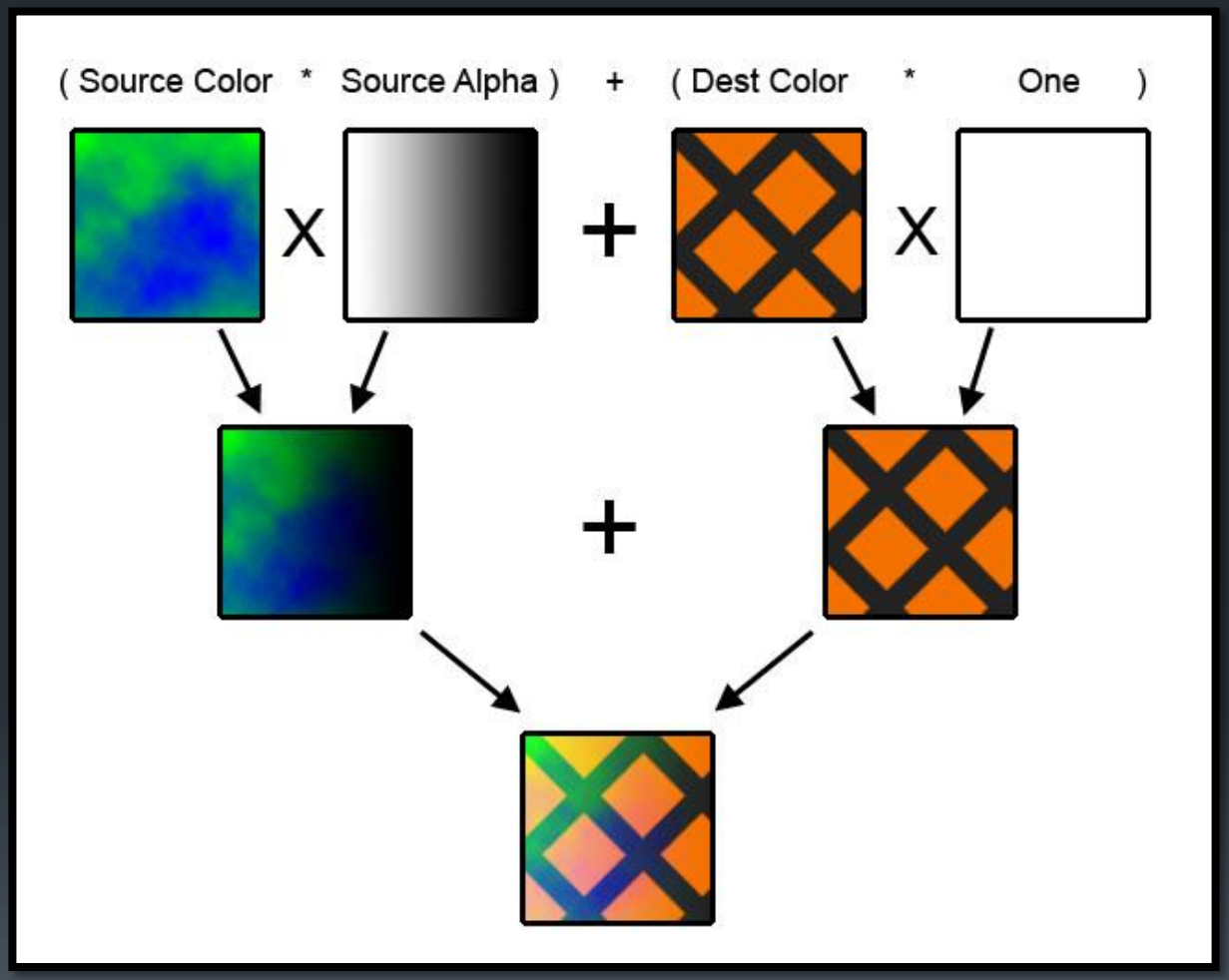
- The standard blending operators are:
  - Addition – Add the two values together
  - Subtraction – Subtract the values together (src – dest)
  - Reverse Subtraction – Subtract the values in reverse (dest – src)
  - Minimum – Minimum value from src and dest
  - Maximum – Maximum value from src and dest

DirectX10 Blending Stage



- NOTE: in DirectX, Blending is performed on the color values, and the input sources refer to the blend factors' data sources. I.E. use the source alpha as the source blend factor.
- Source 1 refers to source blend factor.
- Source 2 refers to the destination blend factor.
- The blending state is defined in the D3D10_BLEND_DESC structure.

Blending Example



- Texture blending example using addition and the source alpha blending factor.

Alpha Blending

- Alpha Blending gives the illusion of transparency in objects. Each color has an additional alpha value [0,1] that specifies the opaqueness of the color.
- An alpha of 1 is fully opaque while an alpha of 0 is fully transparent.
- To perform alpha blending, we set the blending equation as follows:
  - **Source Blending Factor**: src alpha
  - **Destination Blending Factor**: 1 – src alpha
  - **Blending Operation**: Addition

Alpha Blending Example