

Computer Graphic Note

Jiechang Shi

Version: 1.1

Update: November 15, 2019

1 Class 1: Overview

1. Frame Buffer

- Pixel: One element of a frame buffer
- Pixel depth: Number of bytes per-pixel in the buffer
- Resolution: Width \times Height
- Buffer size: Total memory allocated for frame buffer
- **Exam Question:** Given Resolution and Pixel depth, Asked Buffer size
- Z Value: For solving the hidden-surface removal(HSR) problem

2 Class 2,3: Rasterization

1. For each pixel on screen, the sample point of that pixel is the center point of that pixel, which have **integer** coordinates. For examples, (3, 2).
2. A simple problem: Rasterizing Lines

Program Description: Given two endpoints, $P = (x_0, y_0)$, $R = (x_1, y_1)$ find the pixels that make up the line.

Note that: Lines are infinitely thin so they rarely fall on pixel sample point.

A Feasible Description: Rasterize lines as **closest** pixels to actual lines, with 2 requirement

- No Gap
- Minimize error(distance to true line)

To make this question simplify: Only consider situation that $|x_1 - x_0| \geq |y_1 - y_0| \geq 0 \wedge |x_1 - x_0| \neq 0$, which means the $-1 \leq slope \leq 1$. Otherwise we just exchange x and y .

A basic Algorithm: $k = \frac{y_1 - y_0}{x_1 - x_0}$, $d = y_0 - kx_0$, for each $x_0 \leq x \leq x_1$, $y = ROUND(kx + d)$. This method by brute force is inefficient because of the multiplication and the function $ROUND()$.

Basic Incremental Algorithm: for each $x_0 \leq x_i < x_{i+1} \leq x_1$, $y_{i+1} = y_i + k$, However, the successive addition of a real number can lead to a **cumulative error buildup!**

Midpoint Line Algorithm:

- For $0 \leq k \leq 1$
- For one approximate point $P = (x, y)$, we only have 2 choices for the next point $E = (x + 1, y)$ and $NE = (x + 1, y + 1)$, we should choose the one which is closer to $k(x + 1) + d$
- Calculate the middle point $M = (x + 1, y + \frac{1}{2})$
- If the **Intersection point** Q is below M , take E as next, otherwise take NE as next.
- Note that: we consider this equation:

$$f(x, y) = ax + by + c = (y_1 - y_0)x - (x_1 - x_0)y + (x_1y_0 - y_1x_0)$$

We assume $a > 0$

- For a point (x, y)
 - if $f(x, y) = 0$, (x, y) lies on the line.
 - if $f(x, y) < 0$, (x, y) lies upon the line.
 - if $f(x, y) > 0$, (x, y) lies below the line.
 - So we have to test $f(M) = a(x + 1) + b(y + \frac{1}{2}) + c = f(Former) + a + \frac{b}{2}$
 - Assume $a > 0$, if $f(M) > 0$ choose NE otherwise choose E
 - Update $f(Former)$:
 - If we choose E , $f(Former) = f(Former) + a$
 - If we choose NE , $f(Former) = f(Former) + a + b$
- Note that: a and b are **constant integer**, so here is no **cumulative error issue**

3. A harder problem: Triangles Rasterization

Why Triangle:

- Triangles (*tris*) are a simple explicit 3D surface representation.
- Convex and concave polygons (*polys*) can be decomposed into triangles.
- Tris are planar and unambiguously defined by three vertex(*verts*) coordinates (*coords*).

Definition: Find and draw **pixel** samples **inside** *tri* edges and interpolate parameters defined at *verts*

4. **Rasterization and Hidden Surface Removal(HSR) Algorithm Classes:**

- Image order rasterization: ray tracing/ ray casting
traverse pixel, process each in world-space

transform rays from image-space to world-space

- Object order rasterization: scan-line / LEE
traverse triangles, process each in image-space
transform objects from model-space to image-space

5. LEE Linear Expression Evaluation Algorithm:

- We already discussed in *Midpoint Line Algorithm* that how to determine a point is on the left(up) or right(below) the line, just *a quick review here*:
Assume the line have a positive *slope*
For an Edge Equation E , for point (x, y) : $E(x, y) = dY(x - X) - dX(y - Y)$
if $E(x, y) = 0$, (x, y) lies on the line.
if $E(x, y) < 0$, (x, y) lies right(below) the line.
if $E(x, y) > 0$, (x, y) lies left(up) the line.
- For **Rasterization**:
Compute LEE result for all three edges.
Pixels with **consistent sign** for all three edges are inside the *tri*.
Include **edge pixels** on left or right edges.
- LEE need to check every pixel in the bounding box.
- LEE is very good in parallel(SIMD) system.
- Furthermore: Given 3 random *verts* how to find CW edge cycle
Determine L/R and Top/Bot edges for edge-pixel ownership

6. Scan Line Rasterizer:

- Sort vets by Y
- Setup edge DDAs for edges
- Sort edges by L or R(The long edge on left or right)
- Start from Top Vertice, and switch DDA when hit the middle vertice.

7. Interpolate Z:

A general 3D plane equation has 4 terms: $Ax + By + Cz + D = 0$
 (A, B, C) is the normal of that plane, so $(X, Y, Z)_0 \times (X, Y, Z)_1 = (A, B, C)$
Then plug any vertex coord into equation and solve for D .
Given (A, B, C, D) and any point (x, y) can solve z

8. Used Z-buffer to remove hidden surfaces

Initial Z-buffer to MAXINT at the begining of every frame
Interpolate vertex Z values to get Z_{pix}
Only write new pixel to the buffer if $Z_{pix} < Z_{buffer}$
Notice that Z should always **bigger or equal to zero!**

9. Hidden Line Removal(HLR):

Simple z-buffer does not work when the render only draws edge(outlines of polygons).
Need edge-crossing and object sorting methods.

10. Painter's Algorithm: render in order front to back

A object is in front of another object means:
 Z of all verts of one object is less than the other.
This algorithm not work if Z -sort is ambiguous.

11. Warnock Algorithm:

Subdivide screen until a leaf region has a simple front/back relationship.
Leaf regions have one or zero surfaces visible, and the smallest region is usually a pixel

Usually use quad tree subdivision

12. BSP-Tree:

View-Independent binary tree(pre-calculated) allows a view-dependent front-to-back or back-to-front traversal of surfaces.

Use Painter Algorithm to do back-to-front traversal.

Useful for **transparency** - full depth-sort of all surface.

13. Culling:

- Culling with portals: pre-compute the invisible part.
- Culling by View Frustum: Skip a triangle iff all its vertices are beyond **the same screen edge!**
Pitfall: If the vertices are beyond different edge, some part of the *tri* might still in the screen. Image a giant *tri* that cover the whole screen.
- Backface Culling: For **closed(water-tight)** objects, surfaces with **oriented-normals facing away** from the camera are never visible.
Pitfall: BF Culling only work for water-tight object!
- Frustum: Only visible triangles are drawn into the frame buffer.

3 Class 4,5,6,7: Transformations

- Linear transformations (*Xforms*) define a mapping of coordinates (*coords*) in one coordinate frame to another.

$$V_b = X_{ba} V_a$$

- Homogeneous Vector** (V) is 4×1 columns $(x, y, z, w)^T$
- Homogeneous Transforms** (X) is 4×4 matrix
- From Homogeneous Vector to 3D Vector:

$$x = \frac{x}{w}, y = \frac{y}{w}, z = \frac{z}{w}$$

- Why** we use Homogeneous Vector?

We want to uniform the transform matrix including translation, scaling, rotation in the same form of matrix.

3.1 Transformation Matrix

- Translation:

$$T(t_x, t_y, t_z) \Rightarrow \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, T^{-1}(t_x, t_y, t_z) = T(-t_x, -t_y, -t_z)$$

- Scaling:

$$S(s_x, s_y, s_z) \Rightarrow \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, S^{-1}(s_x, s_y, s_z) = S\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$$

- Rotation, CCW:

$$R_x(\theta) \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_x^{-1}(\theta) = R_x^T(\theta)$$

$$R_y(\theta) \Rightarrow \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_y^{-1}(\theta) = R_y^T(\theta)$$

$$R_z(\theta) \Rightarrow \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_z^{-1}(\theta) = R_z^T(\theta)$$

- Pitfall: commutative property is for S,R only.

Here assume **uniform scaling** in all dimensions.

If S is not an uniform scaling matrix. S,R don't have commutative property.

3.2 Spaces Transformation

1. NDC to Output Device

$$X_{sp} \Rightarrow \begin{bmatrix} \frac{xs}{2} & 0 & 0 & \frac{xs}{2} \\ 0 & -\frac{ys}{2} & 0 & \frac{ys}{2} \\ 0 & 0 & MAXINT & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that:

Output Device is **RH coords** and origin in **upper left**. $X \in [0, xs), Y \in [0, ys), Z \in [0, MAXINT]$

NDC is **LH coords** and origin at screen center. $X, Y \in [-1, 1], Z \in [0, 1]$

2. Perspective Projection

$$X_{pi} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \\ 0 & 0 & \frac{1}{d} & 1 \end{bmatrix}$$

What is d And Why there are two $\frac{1}{d}$?

- Assume camera is on $(0, 0, -d)$, perspective(also image) plane is $z = 0$, a object in world space is (X, Y, Z)
- Defined FOV(field of view) as the angle the camera can see.
- Note: $X \in [-1, 1]$, so the distant(d) from Forcus point to view plane can be calculate by this equation:

$$\frac{1}{d} = \tan\left(\frac{FOV}{2}\right)$$

- Futher More: The object project to view plane can be calculate by these equations:

$$\frac{X}{Z+d} = \frac{x}{d} \Rightarrow x = \frac{X}{\frac{Z}{d}+1}$$

$$\frac{Y}{Z+d} = \frac{y}{d} \Rightarrow y = \frac{Y}{\frac{Z}{d}+1}$$

$$\frac{Z}{Z+d} = \frac{z}{d} \Rightarrow z = \frac{Z}{\frac{Z}{d}+1}$$

$$(x, y, z) = \left(\frac{X}{\frac{Z}{d}+1}, \frac{Y}{\frac{Z}{d}+1}, \frac{Z}{\frac{Z}{d}+1} \right)$$

- Write this 3D vector to Homogeneous Vector:

$$(x, y, z, w) = \left(X, Y, Z, \frac{Z}{d}+1 \right)$$

- Futher: We forcus on the **range** of Z now is $z \in (-\infty, d)$.

But in NDC we hope $z \in [0, 1)$ So:

We delete all vector that $Z < 0$, because they cannot project to the view plane.

For $z \geq 0$, we define $z' = \frac{z}{d}$

- $(x, y, z', w) = \left(X, Y, \frac{Z}{d}, \frac{Z}{d}+1 \right) = X_{pi} * (X, Y, Z, 1)$

Pitfall: Do Z interpolation in Perspective Plane!

Why we need the fareset plane?

Asymptotic curve of Z vs. z , that z increase slower when Z is large.

It might map different Z to the same z

3.3 Camera Matrix

1. Assume camera position is c , camera look-at point is l , here c and l are both in world coordinate. And the world up vector is \vec{up}
2. Camera Z-axis **in world coordinate** is

$$\vec{Z} = \frac{\vec{cl}}{\|\vec{cl}\|}$$

3. Camera Y-axis **in world coordinate** is the orthogonal(vertical) part of world-up vector to Z-axis which is

$$\vec{up}' = \vec{up} - (\vec{up} \cdot \vec{Z})\vec{Z}$$

$$\vec{Y} = \frac{\vec{up}'}{\|\vec{up}'\|}$$

4. Camera X-axis **in world coordinate** is orthogonal to both Y and Z axes. So:

$$\vec{X} = \vec{Y} \times \vec{Z}$$

5. Build the X_{wi} from camera space to world space:

X-axis vector $[1, 0, 0]$ in camera space should be \vec{X} in world space, also for Y,Z-axis vectors, So:

$$X_{wi} \Rightarrow \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6. Also we need to add the translation of the camera to the Matrix:

$$X_{wi} \Rightarrow \begin{bmatrix} X_x & Y_x & Z_x & c_x \\ X_y & Y_y & Z_y & c_y \\ X_z & Y_z & Z_z & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

7. Now we can get the invert matrix X_{iw} :

$$X_{iw} \Rightarrow \begin{bmatrix} X_x & Y_x & Z_x & -X \cdot c \\ Y_x & Y_y & Y_z & -Y \cdot c \\ Z_x & Z_y & Z_z & -Z \cdot c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

8. In this proof we know that: If we know the X,Y,Z-axis in world coordinate for a specific space, we can easily build and invert the translation from or to that space.

This method can also be used to **proof the general rotation matrix**.

9. Orbit a Model about a Point

The idea is the same as place camera.

Need to care about **which space** you current in!

4 Class8-10 Illumination and Shading

1. Global vs. Local Illumination

Global: Models indirect illumination and occlusions Local: Only models direct illumination

2. Irradiance

$$E = \int_{\Omega} I(x, \omega) \cos \theta dx$$

Note: $I(x, \omega)$ is the light intensity arriving from all directions and entering the hemisphere Ω over unit surface area.

Also we only care the vertical(normal) part of the light, we dismiss all lights parallel to the surface by using $\cos \theta$

3. Simplified lighting:

Assume all lights are distant-point light.

- Source have **uniform** intensity distribution
- Neglect distance fallout
- Direction to source is constant within scene
- Using 2 parameters to define a light:
direction(x, y, z) vector from surface to light source
intensity(r, g, b) of the light

4. specular reflection and diffuse reflection

- Color shift by attenuation of RGB components for all reflection
- Specular Reflection Model(View-Dependent):

$$L_j(V) = L_e \cdot K_s \cdot (V \cdot R)^{spec}$$

$$R = 2(N \cdot L)N - L$$

Note: V and R should be normalized.

Direction: Reflection occurs mainly in the "mirror" R direction, but there is some spread in similar directions V .

spec controls the distribution of intensity about R . Higher value of *spec* make the surface smoother

K_s controls the color attenuation of Surface.

- Diffuse Reflection Models(View-Independent):

$$L_j = L_e \cdot K_d(L \cdot N)$$

Direction: All **output** directions are the same. But we only care vertical **input** light.

L and N should be normalized.

K_d is the surface attenuation component.

- Ambient Light

$$L_j = L_a \cdot K_a$$

Direction: All input and output directions are the same.

Only one ambient light is needed and allowed.

- Complete Shading Equation:

$$Color = (K_s \sum L_e \cdot (V \cdot R)^{spec}) + (K_d \sum L_e \cdot (L \cdot N)) + (L_a \cdot K_a)$$

5. Detail about HW4(Lighting Implementation)

- \vec{L} denotes the direction to a infinity-far point-light source
- \vec{E} denotes the camera direction. If camera is far away, \vec{E} is constant(In HW4.)
- \vec{N} is specified at triangle vertices.
- \vec{R} must be computed for each lighting calculation (at **a point**).

Calculation of \vec{R} :

$$\vec{R} = 2(\vec{N} \cdot \vec{L})\vec{N} - \vec{L}$$

Avoiding sqrt-root in this calculation

- Choosing a Shading Space: We need all $\vec{L}, \vec{E}, \vec{N}, \vec{R}$ in some affine(pre-perspective) space
Suggest use **Image Space** for HW4.
Model space is also a reasonable choice since Normal vectors are already in that space. This is most **efficient**!
- **Image Space Lighting (ISL)**
Create a Transformation stack from model space to image space.
Need to **normalized the Scale and delete translation** for each matrix, **only maintain the rotation**, before push into this stack!
- **Check** the sign of $\vec{N} \cdot \vec{E}$ and $\vec{N} \cdot \vec{L}$:
Both positive: Compute lighting model.
Both negative: **Flip normal**(\vec{N}) and compute lighting model.
Different sign: Skip it.
- **Check** the sign of $\vec{R} \cdot \vec{E}$: If negative, set to 0.
- **Check** color overflow(> 1.0): Set to 1.
- **Compute Color** at all pixels:
Per Face - flat shading
Per Vertex - interpolate vertex colors, Gouraud Shading(specular highlights are undersampled, aliased).
Per Pixel - interpolate normals, Phong Shading (Expensive computation, but better sampling)
Set **Shading Modes Parameter** for different lighting calculation.
- **Pitfall in Phong** Interpolation:
Need to **normalize** the interpolation normal vector.

4.1 Class10: Something More About Shading

1. Non-Uniform Scaling:

A non-uniform scaling alters the relationship between the surface orientation and the Normal Vector.

So we **cannot** use the same matrix M for transformation of the Normals and the vertex coordinates.

We can fix this by using a different transformation $Q = f(M)$ for transforming the Normals.

2. How to create a matrix for Normals:

In HW4, We create a matrix **dismiss all** scale matrix.

For Detail:

As the definition of Normals:

$$\vec{N}^T \cdot \vec{P} = 0$$

After include the transform matrix:

$$(Q\vec{N})^T \cdot (M\vec{P}) = 0$$

By Definition of Matrix Multiplier:

$$\vec{N}^T \cdot Q^T \cdot M \cdot \vec{P} = 0$$

Since we already know $\vec{N}^T \cdot \vec{P} = 0$ we only need the inner part equal to identity matrix:

$$Q^T \cdot M = I, Q = (M^{-1})^T$$

Note that: If we only used uniform scaling: $S = I$ after normalization.

If we compute Q for each M pushed on the X_{im} transform stack, the resulting X_n stack has Q and therefore allows non-uniform scaling.

3. Model Space Lighting(MSL):

Only need to transform Global lighting parameters once per models.

Also need to transform Eye/camera direction into model space.

5 Class 11-13: Texture Mapping

5.1 Screen-Space Parameter Interpolation

1. In Z-buffer interpolation, we know that linear interpolation for z is **wrong in image space**, we need to interpolate in **perspective space**.
2. Accurate interpolation of RGB color or Normal vectors should also take perspective into account.
But we can ignore the color and normal interpolation error.
3. Interpolation for **Texture Function**: checkerboard Example: Using Linear Interpolation for u & v is also wrong!
4. How to compute perspective-correct interpolation of u, v at each pixel.

- For each parameter P , we used P^s to denote the value in perspective space.
- Note that: For Z interpolation $V_z^s = \frac{V_z}{\frac{V_z}{d} + 1} = \frac{V_z \cdot d}{V_z + d}$
- Rescale V_z^s to $V_z^s \in [0, Z_{max}]$

$$V_z^s = \frac{V_z \cdot d}{V_z + d} \cdot \left(\frac{Z_{max}}{d}\right) = \frac{V_z \cdot Z_{max}}{V_z + d}$$

- We can also get the invert equation:

$$V_z = \frac{V_z^s \cdot d}{Z_{max} - V_z^s}$$

- For parameter from image space to perspective space:

$$P^s = \frac{P}{\frac{V_z}{d} + 1} = \frac{Pd}{V_z + d}$$

- Also we can get inver equation:

$$P = \frac{P^s(V_z + d)}{d}$$

- We don't have V_z but we already calculated V_z^s in HW2, so we can used that:

$$P^s = \frac{P}{\left(\frac{V_z^s}{Z_{max} - V_z^s} + 1\right)}$$

$$P = P^s \cdot \frac{V_z^s}{Z_{max} - V_z^s} + 1$$

- Note that we only have V_z^s and Z_{max} in this equation that we already know the value, we don't need to care d and some other parameter.
- We used $V_z' = \frac{V_z^s}{Z_{max} - V_z^s}$ to simplify the equation:

$$P^s = \frac{P}{V_z' + 1}$$

$$P = P^s \cdot (V_z' + 1)$$

5. The Step for Parameter interpolation:

Get V_z^P for each vertex.

Transform P to perspective space P^s for each vertex.

Interpolate V_z^P for each pixel.

Interpolate P^s for each pixel.

Transform P^s back to P by using V_z^P for each pixel.

5.2 Texture

1. Scale u, v to Texture Image Size:

(u, v) coords range over $[0, 1]$

2D Image is a pixel array of $xs - 1, ys - 1$

But $u * (xs - 1)$ might not be Integer so we need to interpolate the color for non-Integer (u, v) coordinate from nearest 4 Integer point.

$$Color(p) = (1 - s)(1 - t)A + s(1 - t)B + stC + (1 - s)tD$$

2. For Phong Shading, using texture function $f(u, v)$ to replace k_d and k_a

3. For Gouraud Shading, using $f(u, v)$ to replace all k_s, k_d and k_a

4. Procedural Texture

5. Bump Texture:

- Alter normals at each pixel to create bump.
- Normal Perturbation(\vec{P}): $N' = N + P$, \vec{P} should be in the same space as N . But \vec{P} should not be in model space.
- Better spaces for \vec{P} are Surface Coordinates, Tangent Space.

6. Surface Coordinates:

7. Normal Space encodes the surface normal rather than perturbation.

8. Noise Texture:

- Perlin Noise:(Ref(Chinese): <https://www.cnblogs.com/leoin2012/p/7218033.html>)
- Input: (x, y, z) for 3D and (u, v) in 2D
- Output: double value between 0 and 1
- We have 2 Pseudo Random Grid for each Integer Point(x, y, z are integers):
 - Noise Matrix(d): The color of Point for noise
 - Gradient Matrix(g): A random unit vector for each Point
- For each input vector(u, v), if (u, v) isn't Integer, we found 4-corners Integer Point: $(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)$
- For each Integer Point, we use **dot product** of distant vector(from (u, v) to Integer Point) and gradient vector to get the noise value.
- In perlin noise every interpolation is in 1-D. So 2-D need first interpolate y-axis(twice) and then interpolate x-axis. 3-D need 7 interpolation. We used linear-interpolation in slides but we can use **Fade** function(easy curves) for better interpolation.
- Turbulence: Sum noise with diminishing amplitude:

$$turbulence(x) = \sum_k \frac{1}{2^i} |noise(2^i x)|$$

9. Environment(Reflection) Mapping:

- Basic Idea: During rendering, compute the reflection of Eye vector(not the light vector)
- **Ignore the position of surface point** in scene. We assume all points are on center point of the scene.
- Light and scenery are all merge into environment texture.
- No object inter-reflection or shadow
- Blur texture to simulate diffuse reflection
- Sharp texture to simulate specular reflection

10. Cube Map:

- Transform each Eye reflection vector R back to world space

- Find Max component: indicated which face of cube it would intersect
- Compute intersection of R with cube face:
Move all Reflection ray tail to center of cube. Rescale the max component(for example y) of vector R to 1.0.
The other 2 component(x, z) indicate the texture-pixel.

11. Refraction Map:

- Use Snell's law to compute refraction vector
- Color aberration simulated with $f(\lambda)$ refraction angle for multiple color bands

5.3 Implementation Of Textutre(HW5)

1. Step1: Texture coordinates: surface point $\rightarrow (u, v)$
Input: vertex in image space
Output: (u, v)
2. Step2: $(u, v) \rightarrow$ RGB color
Input: (u, v) Output: RGB color from image LUT
3. Interpolation of (u, v) need to be in perspective space.
4. Interpolation of 4-corner for non-Integer (u, v) is needed.

6 Class14-16 Antialiasing

6.1 The Source of Aliasing

1. Quantization error arise from insufficient accuracy of sample
2. Aliasing error arise from insufficient samples
3. Nyquist Theorem: Sample at least twice the rate of highest frequency present in the signal.
f(t) filtered for cutoff freq ω_F (Remove high frequencies before sampling)
Sample Rate $\frac{1}{T_0}$ is greater than $2\omega_F$
Reconstruct(interpolate) with *sinc* function
4. Solution: Band-limit the input signal before sampling.

6.2 Implement Antialiasing(HW6)

1. Antialiasing by jitter supersampling
2. Sample a pixel several with different center and weight

6.3 Texture Antialiasing

1. Sample Rate Mismatch: Texture sampling rate generally does not match screen pixel sample rate (Texel:Pixel ratio)
2. Projected texture in screen image should sample near same rate(1:1) to texture map.
1 Texel: Many Pixel: No aliasing problem, But blur. Fix by using higher resolution textures.
1 Pixel: Many Texel: Aliasing problem. Fix by sample rate is twice highest freq in texture.
3. Mip Map: Pre-compute filtered version of texture image at octave scale/size intervals.
Using average color of 2×2 texels.
The space cost is only 33% more.
Scale for each level of Mip Map:

$$Scale = \frac{dU}{dX} = \frac{dV}{dY}$$

Pixel Scale is more complex since it is non-axis-aligned(after rotation and projection):

$$PixelScale = (\frac{dU}{dX}, \frac{dU}{dY}, \frac{dV}{dX}, \frac{dV}{dY})$$

An approach to match Scale and PixelScale is choosing the highest PixelScale component.(Blur is better than aliasing!)

4. 3D Interpolation: If the Pixel Scale is between 2 Texel Scales, we need to interpolate between 2 texture samples.
5. Anisotropic Interpolation: Combine more than 2×2 pixel in each texture samples.
6. Summed-Area Table(SAT):Compute a texture table T so that each texel has sum of **all texels above and left**
SAT provide an approximation approach to get the avagerage color in $O(1)$.
Note that: the texels sample might not be axis-aligned since the pixel to texel projection. But the different is strictly less than $\frac{1}{2}$.

7 Final Exam Review

1. Shading Equation:

$$Color = (K_s \sum L_e \cdot (E \cdot R)^{spec}) + (K_d \sum L_e \cdot (L \cdot N)) + (L_a \cdot K_a)$$

Know the meaning for every terms:

- K_s, K_a, K_d
 - L_e, L_a
 - s
 - N, L, R, E
 - Equation1: $R = 2(N \cdot L)N - L$
- ### 2. Shading Mode:(Flat, Gouraud, Phong)
- ### 3. Texture
- ### 4. Calculate the normal: With Non-translate and Non-scale Matrix To Image Space
- ### 5. Other Topic:
- Enviroment Shading

8 BRDF

Reference for this section(In Chinese): <https://blog.csdn.net/yjr3426619/article/details/81098626>

8.1 What is BRDF

- Basic Idea of BRDF: The reflect rate for a surface is based on input vector and camera vector.

$$f(l, v) = \frac{dL_0(v)}{dE} (For NonPoint Light) = \frac{L_o(v)}{E_L \cos \theta_i} (For Point Light)$$

- l : input light vector
- v : camera vector
- L_0 : Output Radiance(Color)
- E_l : Input Irradiance(Color* π in Unity)
- θ_i : angle between input vector and surface normal
- Properties of $f(l, v)$
 - $f(l, v) \geq 0$
 - $f(l, v) = f(v, l)$
 - $R(l) = \int_{\phi} f(l, v) \cos \theta_0 d\omega \leq 1$, Discussed later
- How to calculate color based on BRDF:
 - In Point Light, Point Camera:

$$L_0(v) = \sum_k f(l_k) \times E_l \cos \theta_{ik}$$

- Directional-hemispherical reflectance:

- Definition:

$$R(l) = \int_{\phi} f(l, v) \cos \theta_0 d\omega$$

- ϕ is the hemispherical in surface normal half
- θ_0 is the angel between camera view and surface normal
- ω is the space angle
- Basically, $R(l)$ is **the sum of the (directional) energy** of all output light in hemisphere.
- The output energy **should be equal or less than** the input energy so:

$$R(l) \leq 1$$

8.2 How to calculate BRDF

8.2.1 From Lambertian Model

- **Lambertian Model**: Only Diffuse Color, No Specular Color. All output direction are the Same.
- In this model, we assume the output energy is part of input energy. So We can build this equation:

$$R(l) = C_{diff}$$

So:

$$C_{diff} = R(l) = \int_{\phi} f(l, v) \cos \theta_0 d\omega$$

We know $f(l, v)$ and C_{diff} are both constant number:

$$f(l, v) \cdot \int_{\phi} \cos\theta_0 d\omega = C_{diff}$$

$$f(l, v) = \frac{C_{diff}}{\pi}$$

8.2.2 A little bit harder: Phong Model

- We assume the ambient light equal to 0 then the shading equation is:

$$L_0 = (\cos\theta_i * C_{diff} + (\cos\alpha_r)^{spec} * C_{spec}) \times B_L$$

- first part is **Lambertian model**
- α_r is the angel between reflection vector and camera vector
- And we have shading equation in BRDF format:

$$L_0(v) = f(l, v) \times E_l \cos\theta_{ik}$$

- So we have a basic equation for $f(l, v)$ that:

$$f(l, v) = \frac{C_{diff}}{\pi} + \frac{(\cos\alpha_r)^{spec} * C_{spec}}{\pi * \cos\theta_i}$$

- Notices that: If we need to normalize this equation, let $R_{spec}(l) = C_{spec}$
- First: When $\theta_i = \frac{\pi}{2}$, $f(l, v) = +\inf$, which is implausible, so we times $\cos\theta_i$ and a constant k :

$$f_{spec}(l, v) = k * \frac{(\cos\alpha_r)^{spec} * C_{spec}}{\pi}$$

- Second:

$$C_{spec} = R(l) = \int_{\phi} k * f_{spec}(l, v) \cos\theta_0 d\omega = k * C_{spec} / \pi * \int_{\phi} (\cos\alpha_r)^{spec} \cos\theta_0 d\omega$$

- It's very hard to do this calculation, by experience we can get:

$$k = \frac{spec + 2}{2}$$

- So in Phong Model:

$$f(l, v) = \frac{C_{diff}}{\pi} + \frac{(spec + 2)C_{spec} * (\cos\alpha_r)^{spec}}{2\pi}$$

8.2.3 A little bit more harder: Blinn-Phong Model

- We define \vec{H} is the Halfway-Vector(Normalize the vector of the middle point) of input vector and camera vector.

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

- Shading equation:

$$L_0 = (\cos\theta_i * C_{diff} + (\cos\beta)^{spec} * C_{spec}) \times B_L$$

β is the angel between H and surface normal vector.

- Follow the step in Phong Model:

$$k = \frac{spec + 8}{8}$$

- And:

$$f(l, v) = \frac{C_{diff}}{\pi} + \frac{(spec + 8)C_{spec} * (\cos\beta)^{spec}}{8\pi}$$

8.2.4 Microfacet Model in Disney Paper

- Equation First:

$$f(l, v) = diffuse + \frac{D(\theta_h)F(\theta_d)G(\theta_l, \theta_v)}{4\cos\theta_l\cos\theta_v}$$

- Microfacet Model: The surface is an aggregate of many microfacet, each microfacet have different normal and you can only see part of the microfacet, and some microfacet might be blocked by others.
- How many microfacets you can see is define by $D(\theta_h)$
- How many microfacets is blocked by other microfacets is define by $G(\theta_l, \theta_v)$
- The fraction of refraction energy in the total input energy is define by $F(\theta_d)$
- $\frac{1}{4\cos\theta_l\cos\theta_v}$ is a normalize factor.

8.2.5 BRDF With Physics Based Render

- In Disney's paper Section 5.3 5.6, it present how **roughness** can affect diffuse function and G function.
- Diffuse Function:

$$diffuse = \frac{baseColor}{\pi} (1 + (F_{D90} - 1)(1 - \cos\theta_l)^5)(1 + (F_{D90} - 1)(1 - \cos\theta_v)^5)$$

$$F_{D90} = 0.5 + 2 * \cos\theta_d^2 * roughness$$

- D Function:

$$D_{GTR} = \frac{c}{(\alpha^2 \cos^2\theta_h + \sin^2\theta_h)^\gamma}$$
$$\alpha = roughness^2$$

- F Function:

$$F_{Schlick} = F_0 + (1 - F_0)(1 - \cos\theta_d)^5$$

- G Function: Using GGX Function

Reference: *Microfacet Models for Refraction through Rough Surfaces*

$$\alpha_g = (0.5 + roughness/2)^2$$

9 Next Step:

- Build the BRDF shading function in unity.
- Find a better diffuse,D,F,G Function
- Put some new variable into diffuse,D,F,G