Vector Equations

Direction in angle: $\theta = \cos \frac{x_a}{||\vec{a}||}$

Parallelogram Rule: If two vectors share the same origin, the two diagonals in the formed parallelogram give $\vec{a} + \vec{b}$ and $\vec{b} - \vec{a}$

Dot Product: $\vec{a} \cdot \vec{b} = x_a x_b + y_a y_b = ||\vec{a}||||\vec{b}|| \cos\theta$

Angle Using Dot Product: $\theta = \arccos \frac{\vec{a} \cdot \vec{b}}{||\vec{a}||||\vec{b}||}$

Projection: $||\vec{a} \to \vec{b}|| = ||\vec{a}|| \cos\theta = \frac{\vec{a} \cdot \vec{b}}{||\vec{b}||}$

Length of Cross Product: $||\vec{c}|| = ||\vec{a} \times \vec{b}|| = ||\vec{a}||||\vec{b}|| \sin\theta$

Cross Product: $\vec{a} \times \vec{b} = (y_a z_b - z_a y_b)i + (z_a x_b - x_a z_b)j + (x_a y_b - y_a y_b)k$

Normalizing Vector $\hat{v} = \frac{\vec{v}}{\sqrt{v_x^2 + v_y^2 + v_z^2}}$

Normal to Surface $\vec{n} = (\vec{v_1} - \vec{v_0})X(\vec{v_2} - \vec{v_0})$

Normalize Perspective :Map to NDC with Scaling, Shear, and Transformation

$$\begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{-(right+left)}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{-(top+bottom)}{right-left} & 0 \\ 0 & 0 & \frac{far+near}{far-near} & \frac{-2(far*near)}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Viewing Transformation

Goals: 1. Define the camera/eye space: - Specify the position and orientation of the viewing camera 2. Establish mapping between the two coordinate system (a) World Space to Camera Space (b) Rotation and Translation

Define Camera Space:

Eye point: camera position

Look-at-point: center of image

Up vector: upwards orientation in the image

Rotation to view space: Must construct 3 vectors:

Look-at direction: $\hat{l} = normalize(lookat\vec{P}oint - eye\vec{P}oint)$

Right Vector: Vector that points to the right. Cross product of look at and up $\vec{r} = \vec{l} \times \vec{up}$

Third vector perpendicular to the look-at and right vectors also oriented in the up direction $\vec{u} = \vec{r} \times \vec{l}$

Rotation Matrix Component of Viewing transformation

$$\mathbf{R}_v = \begin{bmatrix} \hat{r}^T \\ \hat{u}^T \\ -\hat{l}^T \end{bmatrix}$$

Translation to eye-space: Move origin of world coordinate to eye position using $-eye_x$, $-eye_y$, $-eye_z$

Viewing Transformation

$$\mathbf{V} = \mathbf{R}_v \mathbf{T}_{-eye} = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & -\hat{r} \cdot e\vec{y}e \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & -\hat{u} \cdot e\vec{y}e \\ \hat{l}_x & \hat{l}_y & \hat{l}_z & \hat{l} \cdot e\vec{y}e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Visibility

Painter's Algorithm Draw Primitives from back to front

Sort by minimum depth and raesterize form farthest to nearest

Problems: 1. Wastes computation: Invisible parts have already been painted 2. Cyclical overlap and interpretation are problematic: Impossible to determine depth order

Binary Space Partitioning Trees Divide space into visibility regions : in 2d use lines in 3D: use planes

Basic idea: "spatial sorting" keeps track of which side of lines/planes primitives are on: Objects on the same side as the viewer can be drawn on top of objects on the opposite side, Objects on one side cannot intersect objects on the other side Pick oriented line segment from list as the root. Partition rest of lines according to which side they are on lines in "front" are moved to the left sub-tree. Lines to the "back" are moved to the right sub-tree.

Picking a partition line:

1. Randomly select a small number of candidate partitioning lines
2. Calculate number of lines that cross each candidate
3. Use candidate with least crossing as the next partition line

Traversal: Follow painters algorithm draw objects from farthest to nearest

If view location is on front side of a partitioning line: Lines on the back side are farther, lines on the front side are nearer

If the view location is on the back side of a partitioning line: Lines on the front side are farther, lines on the backside are nearer.

Determine side of partitioning line viewpoint is on via Line/Plane equation test

BSP Tree works best for moving viewpoints : just change traversal order of tree. Works well with static scenes.

Doesn't work great with dynamic scenes as primitives can cross partitioning lines. Dynamic adjustments are possible but slow things down.

Pixel-Level Visibility: Z-buffering: Consider visibility on a per pixel basis.

z-value: distance from scene point to the viewer(origin) related to depth values. Typically 24 bits, same as color buffer

Setup: Each pixel has a z-buffer to store the z value in addition to the frame buffer that stores the RGB values

Algorithm

Initialize zbuf to maximal value

for each pixel (i,j) obtained by scan conversion

```
if(znew(i,j) < zbuf(i,j)
   zbuf(i,j) = znew(i,j);
   write pixel(i,j);
```

Equation for z from projection matrix

$z' = \frac{z*(far+near)-2*far*near}{z*(far-near)}$

mapping of z values nonlinear

Problem: number of discrete discernible depths is greater closer to the near plane than near the far plane

Cons: Objects closer to the view are displayed with higher precision

Pros: Mat result in far away objects indiscernible

Linear interpolating interior z values from triangle vertices

$z = A_z x + B_z y + C_z$

Compute coefficients using edge parameters

z fighting: Objects closer to each other than minimum z discrimination mean interpenetration/improper display is possible

Can be minimized with high precision Z buffer pushing near clip plane out as far as possible and/or polygon offset (depth biasing)

Z Buffering Pros: Interpolation of pixel values from vertex values is easy to do and key idea in graphics, Nearly constant overhead: expensive in simple scenes but good for complex ones

Z Buffering Cons: Relatively late in pipeline (rasterization), Requires extra storage for z-buffer, Precision of depth buffer limits accuracy of object, depth ordering for large scale scenes (i.e. nearest to farthest objects), No perfect scheme for handling translucent objects

Illumination

Empirical Illumination in OpenGL

$I_{total} = I_{ambient} + I_{diffuse} + I_{specular}$

2 components of Illumination

1. Light Sources: Emittance Spectrum (color) Geometry (position and direction)

2. Surface Properties, Reflectance Spectrum (color), Geometry (position, orientation, and micro-structure)

Ambient Light Sources: indirect light from light sources Independent of objects position and orientation:

$I_{ambient} = surface_a * light\_source_a$

Directional light source: All rays from a directional light source have a common direction and no point of origin (example sunlight), Lighting direction is constant for every surface, A direction light source have a color

Point Light Source: Rays emitted from a point light radically diverge from the source (example lightbulb), The lighting direction to each point on a surface changes for a point light source

Computation p : surface point position, p_l : light source position

$\hat{l} = \frac{p - p_l}{|p - p_l|}$

Diffuse reflection: Proportional to cosine of angle between light direction and surface normal. Independent of viewing direction. For sphere only have to consider angles between 0 and 90

$I_{diffuse} = surface_d * light\_source_d * (\hat{n} \cdot \hat{l})$ **Specular Reflection**: Bright view-dependent highlight - Computing Reflection Vector: $\hat{r} = 2(\hat{n} \cdot \hat{l})\hat{n} - \hat{l}$

Empirical model of specular: **Phong Illumination**

$\hat{v}$: direction to the viewer n_{shiny} controls how quickly highlight falls off. Falls of faster at higher values $I_{specular} = surface_s * light\_source_s * (\hat{v} \cdot \hat{r})^{n_shiny}$

Empirical model of specular: **Blinn & Torrance Variation** -$\hat{H}$: halfway vector bisecting incoming light direction and viewing direction $\hat{H} = \frac{\hat{l} + \hat{v}}{|\hat{l} + \hat{v}|}$

$I_{specular} = surface_s * light\_source_s * (\hat{n} \cdot \hat{H})^{n_shiny}$

Difference between Phong and Blinn

-Angle between halfway vector and surface normal is likely to be smaller than angle between the reflection and viewing vector (unless surface is viewed at very steep angle)

-Can set larger exponent (shininess) for Blinn

Shading Methods

Flat Shading : simplest method applies only one illumination calculation for each primitive

Use centroid for illumination: $centroid = \frac{1}{vertices} \sum_{i=1}^{vertices} \frac{}{p_i}$

Problems: Polygonal shape is still apparent, For point light sources the direction to the light source varies over the facet, For specular reflections directions to the eye varies over the facet, These issues can be overcome when normals are introduced at each vertex

Gourard Shading :apply illumination model on a subset of surface points and interpolates the intensity of the remaining points on the surface

-Often illumination is applied at each vertex and interior is interpolated from these vertex values

-Can be accomplished using plane equation

-Drawback:Facets are still visible. Artifacts , Poor handling on specular highlights (can be lessened by using finer detailed geometry)

Phong Shading: Surface normal is linearly interpolated across polygonal facets and illumination is applied at every point

Pros: Better handling on specular highlights, Generally very smooth appearence

Cons: Slower than Gourard shading, not built into opengl

Texture

Mapping objects to texture

Each vertex (x,y,z in object coordinate), must associate 2D texture coordinates (s,t) So texture fits nicely over object

Planar mapping: Like projection take vertex coordinate (x,y,z) throw away one dimension Ex: drop z such that texture coord (u,v)= (x/W,y/H)

Cylindrical Mapping: Cylinder: r,θ, z with (u,v) = $\theta/(2\pi)$, z)

-Seams when wrapping around ($\theta = 0$ or 2π)

Spherical Mapping: Convert to spherical coordinates use latitude/longitude

-singularities at north and south poles

Cube mapping

Mapping Procedure: 1. Map square texture to basic map shape 2. Map basic map shape to object (or vice versa)

-Usually pretty straight forward: maps from square to cylinder, plane maps from object to these are simply coordinate transform

Decal Textures: For each triangle in model establish a corresponding region in the photo texture. During rasterization interpolate the coordinate indices into the texture map

Wrapping: behavior of texture coordinates outside of range [0,1] is determined by texture wrap options (ex: clamp, repeat)

Linear interpolation of u and v over a triangle leads to unexpected results: distortion when triangles don't have same depth, noticeable during animation

-Why? equal spacing in screen space(pixel) is not the same as in eye(texture) space in perspective projection

-Map values in screen space to values in eye space and interpolate

Magnification

Small texels mapped to a large region of pixels

Nearest neighbor: take color of closest texel

Bi-linear interpolation: Interpolate colors from the 4 closest texels

$\alpha = \frac{x - x_0}{x_1 - x_0}$ $\beta = \frac{y - y_0}{y_1 - y_0}$ $c = ((1-\alpha)c_0 + \alpha c_1)(1-\beta) + ((1-\alpha)c_2 + \alpha c_3)\beta$

Minification

Texels are smaller than pixels. More complicated than Magnification

-Several texels covering one pixel

-Multiple to one mapping

Aliasing Artifact : Popping (disappear and reappear) around details (parts of textures with high spatial frequency)

Spatial Filtering: Avoid aliasing by pre-filtering texture to remove high frequencies

Pre-filtering: essentially spatial integration over texture, done in pre-processing because too expensive to do on the fly

MIP Mapping

Pre-filtering technique. Idea: construct a pyramid of images that are pre-filtered and resampled at sampling frequencies that area a binary fractions of the original image's sampling.

MIP map has an additional 1/3 storage overhead

Finding MIP level: Idea: Use the projection of a pixel in screen into texture space to figure out which level to use

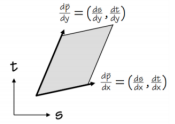
Finding MIP Level

- Idea: Use the projection of a pixel in screen into texture space to figure out which level to use

$u^*(x,y) = u/w = A_u x + B_u y + C_u$

$o^*(x,y) = 1/w = A_o x + B_o y + C_o$

$s = u \cdot textureWidth$



Applying chain rule: $\frac{ds}{dx} = \frac{ds}{du}\frac{du}{dx}$

$\frac{ds}{du} = textureWidth$

$\frac{du}{dx} = \frac{d(u^*(x,y)/o^*(x,y))}{dx} = \frac{A_u o^*(x,y) - A_o u^*(x,y)}{o^*(x,y)^2}$

Other derivatives can be computed in the same way.

Finding MIP Level

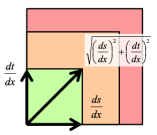
- Use the lengths of the projected pixel in texture space to define a measure of mismatch between sampling densities:

$m = \max(||\frac{d\vec{p}}{dx}||_s||\frac{d\vec{p}}{dy}||) = \max(\sqrt{(\frac{ds}{dx})^2 + (\frac{dt}{dx})^2}, \sqrt{(\frac{ds}{dy})^2 + (\frac{dt}{dy})^2})$

$\approx \max(\max(\frac{ds}{dx}, \frac{dt}{dx}), \max(\frac{ds}{dy}, \frac{dt}{dy}))$

- Now choose the appropriate level:

$level = \lfloor \log_2(m) \rfloor$



Problems: Overblurring due to Isotropic filtering: When a pixel covers many u texels but few v texels we always chose the largest pixel coverage to decide the level

Summed-Area Tables

Perform anisotropic filtering: can be used to compute average color for any arbitrary rectangular region in the texture space at a constant speed

2D array same size as texture: each entry stores sum of all texel colors above and to the left
-Compute color of pixel bounded by $(x_0, y_0), (x_1, y_1)$
1. Find sum of region contained in a box bounded x and y values:
$T(x_1, y_1) - T(x_0, y_1) - T(x_1, y_0) + T(x_0, y_0)$
2. Divide out area $(y_1 - y_0)(x_1 - x_0)$
-Less blurry than MIP MAPS
-Storage overhead is the same as texture size
-Less runtime work than MIP Maps more pre-processing work
Shadow Mapping
2 Types of shadows: 1. Hard shadows generated by point or directional lights 2. Soft shadows from area lights
Render image from light's point of view, render only z-buffer depth values
Store in shadow map: depth (z-buffer) values. Inverse camera and projection transform.
Rendering Shadow:
When lighting a point on a surface – For each light that has a shadow map transform to the shadow map's image space
- Get X,Y,Z values
- Compare Z to the depth value at X,Y in the shadow map - If the shadow map depth is less than Z some other object is closer to the light than this point this light is blocked, don't include it in the illumination If the shadow map is the same as Z this point is the one that's closest to the light illuminate with this light
Environmental Mapping: We can use transformed surface normals to compute indices into the texture map. These sorts of mapping can be used to simulate reflections, and other shading effects. This approach is not completely accurate. It assumes that all reflected rays begin from the same point, and that all objects in the scene are the same distance from that point.
Steps: 1. Create a 2D environment map 2. For each pixel on a reflective object, compute the normal 3. Compute the reflection vector based on the eye position and surface normal 4. Use the reflection vector to compute an index into the environment texture 5. Use the corresponding texel to color the pixel
Bump Mapping: Textures can be used to alter the surface normals of an object. This does not actual shape of the surface – we are only shading it as if it were a different shape. Changing normal changes angle in diffuse component of lighting
The texture map is treated as a single-valued height function. The value of the function is not actually used, just its partial derivatives. The partial derivatives tell how to alter the true surface normal at each point on the surface to make the object appear as if it were deformed by the height function.
Displacement Mapping: Like bump mapping except texture moves surface points

Ray Tracing

Forward Ray Tracing: Compute all rays from light source - wastes a ton of computation as not many make it to eyepoint
Backwards Ray tracing: consider rays that create the image, trace rays from pixels only
Ray casting: compute illumination at first intersected surface point only (takes care of hidden surface illumination)
-Send a ray from the focal point through each pixel and out into the scene and see if it intersects an object (use the background color if it doesn't) only shade closest intersection
-Shade a point by summing over all lights for each color channel
Testing for intersection: make sure it is in front of eye and nearest intersection
General formula: $p = o + td$
Ray-Sphere Intersection
o = origin of ray, d = direction of ray, r = radius of sphere, $\Delta p = p_c - o$
Solve for t $t = d \cdot \Delta \pm \sqrt{(d \cdot \Delta p)^2 - (mag(\Delta p)^2 - r^2)}$
Use smallest non-negative real solution
Ray-Polygon Intersection
Use plane equation $n \cdot x + m = 0$
Substitute x for $o + td$ solve for t plug t into get p
Ray-Triangle Intersection
put in later
Shadow Rays: Spawn new ray at intersection in light direction to make sure light is visible. If an intersection is found only use ambient component of light
Shadow ray dir = $normalize(lightPos - intersectionPoint)$
Shadow ray origin = $intersectionPoint$
If intersection is hit where $0 < t < magnitude(p_l - p)$ it is in shadow
Ray tracing needed for reflection and refraction. Apply ray tracing recursively to a maximum depth
Reflection Formula:$\hat{r} = \hat{l} - 2(\hat{n} \cdot \hat{l})\hat{n}$
Refraction Bending of light due to interface between media Index of Refraction (IOR) n speed of light in medium
Computing transition direction t
$\vec{n} = normal, n = \frac{n_1}{n_2}, c_1 = -\vec{v} \cdot \vec{n}, c_2 = \sqrt{1 - n^2(1 - c_1^2)}$
$\vec{t} = n\vec{v} + (nc_1 - c_2)\vec{n}$
Total internal reflection = $n^2(1 - c_1^2 \geq 1$
Issues with ray tracing: Aliasing, Shadows have sharp edges, no diffuse reflection from other objects, Intersection calculations are expensive
Acceleration Methods:
Bounding volumes: bound complex object in simple objects and test simple intersection before complex one want bounding volume as tight as possible.
Methods: Sphere, Axis-Aligned Bounding Box, Oriented Bounding Box, Slabs/K-dops
Hierarchical Bounding Volumes: Organize as tree each ray traverses through hierarchy
Spatial Subdivision: Divide space in to sub-regions place objects with in sub-regions into list only traverse lists of sub-regions that ray passes through
Distributed Ray Tracing: Replace single ray approximations with distribution of rays
Improvements to image: Anti-aliased edges, objects in/out of focus according to a lens, motion blur of fast moving objects, soft shadows glossy reflection glossy translucency
Anti-aliasing 2 solutions:
1. Super sampling: increases sampling rate but does not completely eliminate aliasing
2. Distribute samples randomly: converts aliasing energy to noise which is less objectionable to the eye
Depth of field: start with normal eye ray and find intersection with focal plane. Choose jittered point on lens and trace line from lens point to focal point. Average colors and assign to pixel
Soft shadows: Model each light as a sphere, send multiple jittered shadow rays toward a light sphere use fraction that reach to attenuate color

Image-Based Rendering

Doesn't require geometric calculations
Pros: Modest computation compared to classical C.G., Cost independent of scene complexity, Imagery from real or virtual scenes
Cons of pre-computation: fixed look-from or look-at point, static scene geometry, fixed lighting
Panoramas Created by stitching together multiple images
Light Field: Synthetic light fields can be created from sheared perspective views, an array of images
-Resampling: generate a ray, find closest rays in light field, return a combination of radiance of those rays
Improvement for aliasing: Add rough depth information to improve rendering quality/reduce size
Light stage images improve fixed lighting problem.
Edge Equation for interpolation (in this case color)
$A = y_0 - y_1$
$B = x_1 - x_0$
$C = -[A(x_0 + x_1) + B(y_0 + y_1]/2$ unless $x_1 \approx x_0$ and $y_1 \approx y_0$ then $C = 0$

$$\frac{1}{C_1 + C_2 + C_3} \begin{bmatrix} A_2 & A_3 & A_1 \\ B_2 & B_3 & B_1 \\ C_2 & C_3 & C_1 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$