CS3241

ILLUMINATION

Illumination Given a point on the surface, the light source, the view point, compute color

Shading Given a polygon and rasterization, compute color for each fragment of the polygon

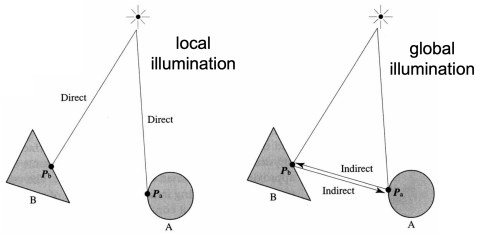
Local vs global

Local reflection

- Considers relationship between light source, single surface point, view point
- No interaction with other objects

Global illumination

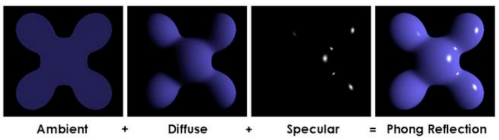
- Considers all light sources and surfaces
- Inter-reflections and shadows



PHONG ILLUMINATION EQN

$$I_{Phong} = I_a k_a + f_{att} I_p k_d (N \cdot L) + f_{att} I_p k_s (R \cdot V)^n$$

- Assumes N, L, R, V are unit vectors
- Assume that light source is a point



Ambient $I_a k_a$

- Produces a uniform lighting effect (color, intensity) on every point on every surface
- Each surface has its own ambient material, so it can appear as a different color

Math

- I_a is the luminance of the light
- k_a is the ambient material property

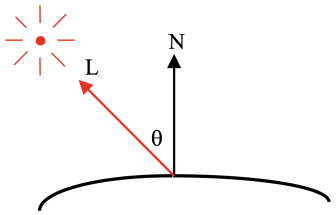
Diffuse $f_{att} I_p k_d (N \cdot L)$

- Gives color to the surface point according to light position and surface normal

Surface normal

- Triangle with vertices A, B, C: $N = \pm(B - A) \times (C - A)$, consider the sign appropriately using RHGR
- Curved surface: normal of the plane tangent to that point

Lambert's cosine law



States that diffuse reflection $\propto \cos\theta = N \cdot L$

Attenuation factor f_{att}

- As the distance d from the point to the light source increases, the light received will be weaker
- In physics, it is $\propto \frac{1}{d^2}$. If d changes slightly, the light intensity can change significantly
- In OpenGL, we use $f_{att} = \frac{1}{a + bd + cd^2}$ to allow more granular control

Math

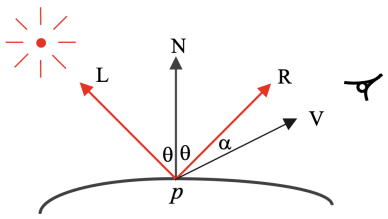
- I_p is the luminance of the light coming from a point p
- k_d is the diffuse material property
- $N \cdot L$ is for diffuse reflection
- $f_{att} = \frac{1}{a + bd + cd^2}$ is the attenuation factor, for some user-defined constants a, b, c

Specular $f_{att} I_p k_s (R \cdot V)^n$

- Adds highlights to shiny surface

Highlight

- Because we assume that the light source is a point, shininess is inversely propotional to the size of the highlight
- Highlight is dependent on V, because the highlight moves when the viewer moves



Math

- Assume N, L, R, V are unit vectors

$$\alpha = \cos^{-1}(R \cdot V)$$
$$R = 2(N \cdot L)N - L$$

- k_s is the specular material property
- n is the shininess coefficient - as n increases, highlights become smaller and sharper
- I_p and f_{att} are defined in the diffuse section

Material properties

- Modelled using k_a, k_d, k_s and n
- k_a, k_d, k_s are vectors of 3 RGB colors, taking values between 0 and 1
- Shininess coefficient n has value from 1 to 128 in OpenGL

Multiple light sources

$$I_{Phong} = I_a k_a + \sum_i f_{att} I_p k_d (N \cdot L_i) + f_{att} I_p k_s (R_i \cdot V)^n$$

Blinn-Phong model (T6)

$$I_{Blinn-Phong} = I_a k_a + I_p k_d (N \cdot L) + I_p k_s (N \cdot H)^n$$

where $H = \text{normalize}(L + V)$

- Produces larger specular highlights than Phong model
- More efficient if light source and viewer are far apart, because L and V will be roughly the same for all the surface points, so H also does not need to be recomputed

Retroreflection (T6)

- A retroreflector reflects most of the light back in the incident direction
- Set $R = L$ in Phong equation
- e.g. Cat's eye, road signs

ILLUMINATION IN OPENGL

- Vertex processing stage: model-view transformation, lighting computation, texture coordinates, then projection to clip space
- Since it is between the model view transformation and the projection matrix, it is performed in eye/camera space

Specify vertex normal

- Use `glNormal3f(x, y, z)` or `glNormal3fv(p)`
- Normal should have unit length. Use `glEnable(GL_NORMALIZE)` for auto-normalization, with a performance penalty

Enable lighting computation

- Use `glEnable(GL_LIGHTING)`. Thereafter, `glColor()` is ignored
- Use `glEnable(GL_LIGHTi)` to enable light source i, where $0 \le i \le 7$ (OpenGL supports 8 lights)
- Use `glLightModeli(param, GL_TRUE)` to activate certain light model parameters
  - `GL_LIGHT_MODEL_LOCAL_VIEWER` sets the view ray to be from each surface point to the origin of the eye coordinate system, for computing specular reflections. Otherwise, it is taken to be parallel to and in the direction of the $-z$ axis.
  - `GL_LIGHT_MODEL_TWO_SIDED` shades both sides of polygons independently

Defining a point light source

```
// RGBA values
GLfloat a0[] = {1.0, 0.0, 0.0, 1.0};
GLfloat d0[] = {1.0, 0.0, 0.0, 1.0};
GLfloat s0[] = {1.0, 0.0, 0.0, 1.0};
GLfloat p0[] = {1.0, 2.0, 3.0, 1.0};

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, p0);
glLightfv(GL_LIGHT0, GL_AMBIENT, a0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, d0);
glLightfv(GL_LIGHT0, GL_SPECULAR, s0);
```

Enable global ambient light

- `glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`

Moving light sources

- The positions and directions of light sources are affected by the model-view transformation

Setting material properties

```
// RGBA values
GLfloat a[] = {0.2, 0.2, 0.2, 1.0};
GLfloat d[] = {1.0, 0.8, 0.0, 1.0};
GLfloat s[] = {1.0, 1.0, 1.0, 1.0};
GLfloat shine = 100.0;

glMaterialfv(GL_FRONT, GL_AMBIENT, a);
glMaterialfv(GL_FRONT, GL_DIFFUSE, d);
glMaterialfv(GL_FRONT, GL_SPECULAR, s);
glMaterialfv(GL_FRONT, GL_SHININESS,
    shine);
```

Customizing front/back faces

- Default behaviour shades only front faces, which works correctly for convex objects
- If set two sided lighting (see Enable lighting computation), OpenGL shades both sides of a surface
- Use `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK` to set properties of each side

Emissive term

- Can simulate a light source by giving a material an emissive component
- Not affected by any sources or transformations

```
GLfloat e[] = {0.0, 0.3, 0.3, 1.0};
glMaterialfv(GL_FRONT, GL_EMISSION,
    emission);
```

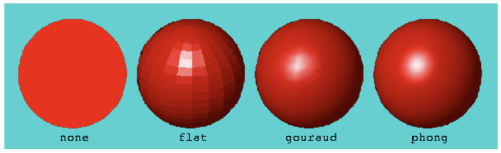
OpenGL lighting comp formula

$$I_{OpenGL} = k_e + I_{ga}k_a + \sum_i f_{att,i}\left[I_{a,i}k_a + I_{d,i}k_d(N \cdot L_i) + I_{s,i}k_s(R_i \cdot V)^n\right]$$

I_{ga} — global ambient light intensity
$I_{a,i}$ — Light i ambient intensity
$I_{d,i}$ — Light i diffuse intensity
$I_{s,i}$ — Light i specular intensity
$f_{att,i}$ — Light i attenuation factor

k_e — material emission intensity
k_a — material ambient reflectance
k_d — material diffuse reflectance
k_s — material specular reflectance
n — material shininess

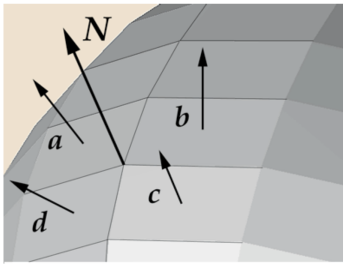
SHADING



Flat shading

- Choose any vertex of the polygon, compute its color with PIE, apply that color to the polygon
- Each polygon has one color
- Distinctive color difference between neighbouring polygons
- `glShadeModel(GL_FLAT)`

Gouraud shading



$N = \text{normalize}(a + b + c + d)$

- For each vertex, compute the average normal vector of polygons sharing that vertex (so we need to know how to find neighbouring polygons)
- Apply PIE at the vertex using the average normal vector
- Smoothly interpolate computed colors at vertices of the polygon to the interior of the polygon
- Per-vertex lighting computation
- `glShadeModel(GL_SMOOTH)`

Phong shading

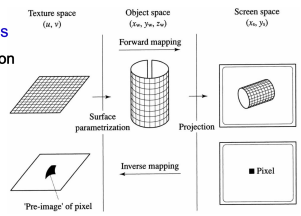
- For each fragment in the polygon, interpolate normal vectors from vertices
- Apply PIE at the fragment using the interpolated normal vector
- Per-fragment (pixel) lighting computation, hence generally more expensive than Gouraud shading
- Better at producing highlights, since each fragment has its own normal vector
- Not supported by OpenGL, but can be supported by reprogramming the rendering pipeline using shaders

TEXTURE MAPPING

Overview

- Consists of two mappings
  - Surface parameterization
  - Projection
- Implementation
  - Forward mapping
  - Inverse mapping

Which one is suitable for polygon-based raster graphics?

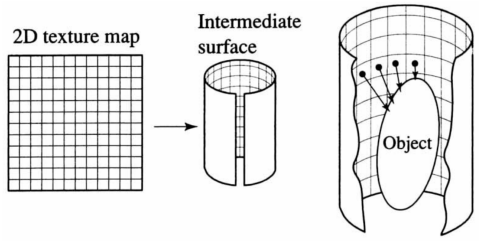


- Forward mapping only guarantees that each texel maps to some point in the screen space. Each pixel in screen space may have 0, 1, or many texels mapped to it, which is not ideal.
- Inverse mapping guarantees that each pixel has exactly 1 texel mapped to it.

Surface parameterization

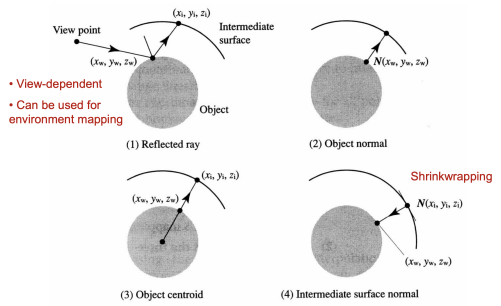
- Defines a mapping between 3D surface and 2D texture map
- $(x_w, y_w, z_w) \leftrightarrow (s, t)$, where $0 \le s, t \le 1$
- For polygonal models, texture coordinates are specified at vertices, and interpolated over the polygon
- Non-trivial surface topology can cause severe distortion of textures

How to parameterize



- First, apply S mapping - Texture map is projected onto a simple intermediate surface (e.g. plane, cylinder, sphere, cube). Intermediate surface chosen should be the most representative of the object
- Then, apply O mapping - 3D intermediate surface is then mapped onto the object surface

O mapping methods



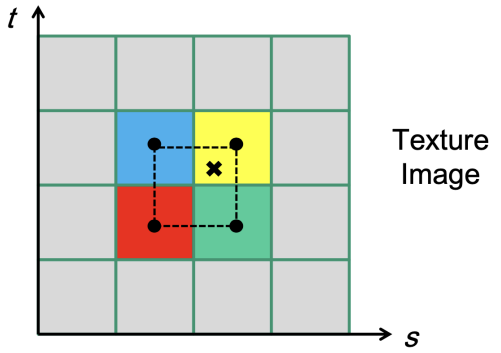
Examples

S mapping: plane, O mapping: shrinkwrapping
S mapping: cylinder, O mapping: shrinkwrapping
S mapping: sphere, O mapping: shrinkwrapping

Texture coordinates mapping

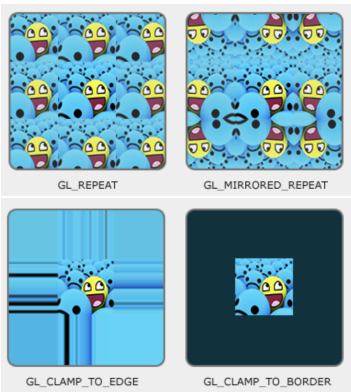
Texture filtering

- At each fragment, the interpolated texture coordinates may not correspond to a texel center, so we need bilinear interpolation of adjacent texels



Wrapping

- Although we defined $0 \le s, t \le 1$, OpenGL allows vertex coordinates outside of this range
- Behaviour is defined according to the wrapping mode
  - Clamp `GL_CLAMP_TO_EDGE` - clamped to $[0, 1]$
  - Repeat `GL_REPEAT` (default) - repeats the texture



Anti-aliasing

Aliasing

- Aliasing can happen if texture map is point-sampled at each fragment
- Occurs during texture minification (when a big part of texture image is mapped to a single texel)

Anti-aliasing

- To fix, texture map should be area-sampled at each fragment
- A fragment is mapped to a quadrilateral area (pre-image) in the texture space, and the average color of the texels is used

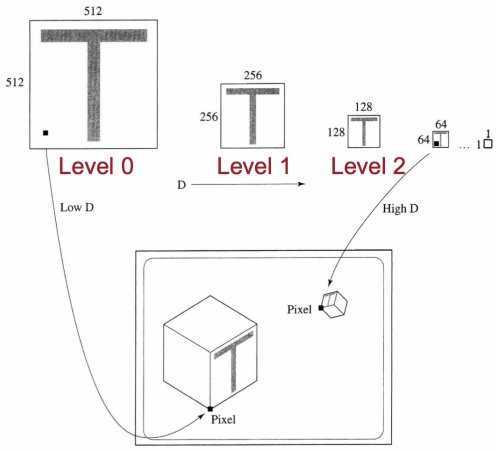
Problems

- Need to find pre-image
- Need to sum the colors of several texels quickly
- Solution: mipmapping

Mipmapping

- Create a set of pre-filtered texture maps by approximating each pre-image using a square
- Point sample the appropriate texture map for each fragment according to the degree of texture minification

Creating mipmaps



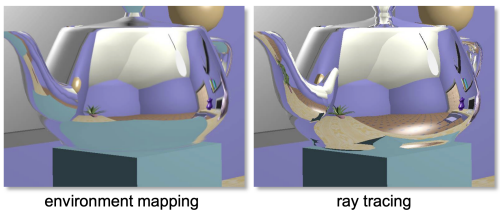
- Created by averaging down the original image successively by half the resolution
- Starts at level 0 (original image)
- Each level halves the width and the height

Choosing mipmap level

- Choose according to amount of texture minification, e.g.
  - If fragment pre-image corresponds to ≤ 1 texel, use mipmap level 0
  - If fragment pre-image corresponds to 2×2 texels, use mipmap level 1
- Ideal level may be non-integer
  - Perform linear interpolation of texels retrieved from two consecutive mipmap levels
  - Called trilinear texture map interpolation

Applications

Environment/Reflection mapping

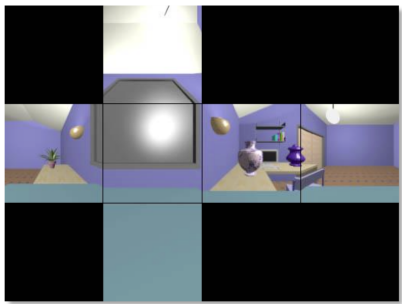


- Shortcut for rendering shiny objects
- Used in Lab 3
- Geometrically correct only when the object is a point, and/or the surrounding is infinitely far away
- Cannot produce self reflection

Implementation

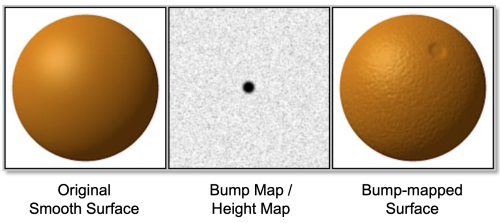
- Image of surrounding is first captured (from the position where the object is to be placed) and stored in a texture map
- When rendering the object, the reflected eye ray is used to reference the texture map

Cube map

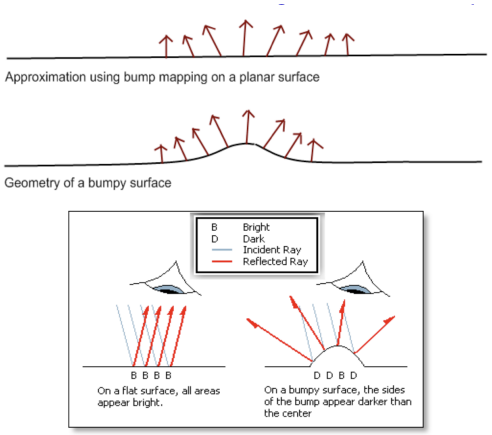


- Image of the environment can be stored in a cube map
- Consists of 6 separate images pieced together, situated in the $+x, -x, +y, -y, +z, -z$ directions

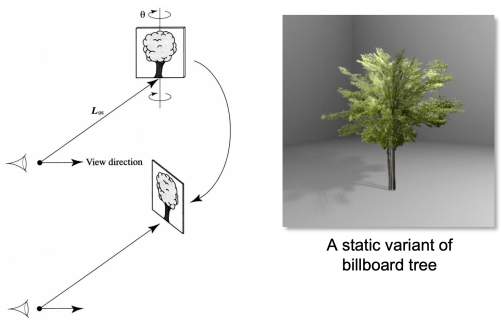
Bump mapping



- Simulates small complex geometric features on surfaces without needing to model them
- Height field is used to perturb the surface normals, and the perturbed normals are used in light reflection computation
- Bumps facing you directly look realistic, but bumps near the silhouette will look flat



Billboarding



- Image is dynamically rotated so it is always facing you (normal is parallel to view direction)

3D texture mapping

- Does not have the downsides of 2D texture mapping
  - Texture distortions
  - Hard to parameterize non-trivial surface topology
- 3D texture is defined everywhere in 3D space (usually procedurally generated to save memory)
- 3D function is evaluated at 3D surface point to obtain texture value
- Analogous to sculpting or carving an object out of a block of material

Texture mapping in OpenGL

- Occurs during fragment processing stage, after a color assigned to the fragment
- The fragment color can be either
  - Modified by texture access (using texture corodinates), or
  - Combined with the texture color by texture application
- For implementation, refer to external slides

RAY TRACING

Ray casting

```
For each pixel
    Construct a ray from the eye
    For each object in the scene
        Find intersection with the ray
        Keep if closest
    Shade // depending on light and
          normal vector (e.g. Phong
          reflection model)
```

- Achieves hidden surface removal naturally

Rasterization vs ray casting

- Rasterization: given a primitive in 3D space, determine which pixels are covered by the primitive
  - Hard to produce global illumination
- Ray casting: given a pixel, determine which primitive covers it
  - Entire scene data must be available when computing each pixel

Ray tracing

From the closest intersection point, secondary rays are shot out

- Reflection ray
- Refraction ray
- Shadow rays (to each light source)

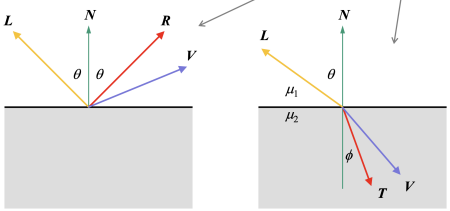
WHITTED RAY TRACING

- Also called recursive ray tracing

Equation

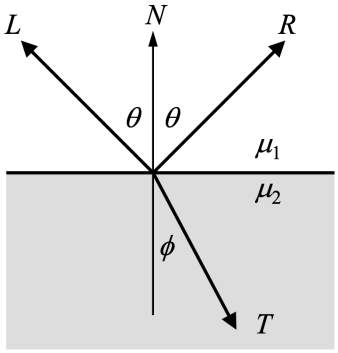
$$I = I_{local} + k_{rg} I_{reflected} + k_{tg} I_{transmitted}$$

where $I_{local} = I_a k_a + I_{source}[\, k_d(N \cdot L) + k_r(R \cdot V)^n + k_t(T \cdot V)^m \,]$



- Similar to the Phong illumination equation, except specular computation is split into the reflected part and the transmitted/refracted part
- $k_r(R \cdot V)^n$ refers to the specular reflection
- $k_t(T \cdot V)^m$ refers to the transmitted/refracted specularity
- k_{rg} stands for global reflection, because the effect is a global effect, coming from another object
- Similarly, k_{tg} stands for global transmission
- If surface is opaque, ignore refracted specularity term

Computing reflection/refraction ray



- Assume all vectors are unit vectors

Reflection ray Like in the Phong illumination model, $R = 2(N \cdot L)N - L$

Refraction ray

- Snell's law: $\mu_1 \sin\theta = \mu_2 \sin\phi$
- Define $\mu = \mu_1/\mu_2$, then

$$T = -\mu L + \left(\mu\cos\theta - \sqrt{1-\mu^2(1-\cos^2\theta)}\right)N$$
$$= -\mu L + \left(\mu(N\cdot L) - \sqrt{1-\mu^2(1-(N\cdot L)^2)}\right)N$$

Shadow rays

- Also called light rays or shadow feelers
- From each surface intersection point, a shadow ray is shot towards each light source, to determine if there is any occlusion between light source and surface point
- If occluder is opaque, then the object is occluded.

$$I_{Phong} = I_a k_a + k_{shadow} I_{source}\Big[k_d(N\cdot L) + k_r(R\cdot V)^n + k_t(T\cdot V)^m \Big]$$

- If translucent, then
  - Light is attenuated (reduced) by the k_{tg} of the occluder
  - However, it is also affected by the distance the light travels through the occluder (i.e. the size of the occluder), but this is not accounted for
  - Additionally, refraction of light ray from light source is ignored
  - Both are physically incorrect, but this approach is taken to save on computation, as it is expensive to compute the realistic scenario

Describing a scene

- Camera view and image resolution
  - Camera position, orientation
  - Field of view
  - Image resolution
- Point light sources
  - Position
  - Brightness, color
  - Global ambient light
- Object materials
  - $k_{rg}, k_{tg}, k_a, k_d, k_r, k_t$ (each is a RGB vector)
  - n, m
  - Refractive index μ if $k_{tg} \neq 0$ or $k_{rg} \neq 0$ (can use different μ for R, G, B)
- Objects

Recursive ray tracing

$$I(\boldsymbol{P}) = I_{local}(\boldsymbol{P}) + I_{global}(\boldsymbol{P})$$
$$= I_{local}(\boldsymbol{P}) + k_{rg}\, I(\boldsymbol{P}_r) + k_{tg} I(\boldsymbol{P}_t)$$

where:

$\boldsymbol{P}$ is the hit point
$\boldsymbol{P}_r$ is the hit point discovered by tracing the reflected ray from $\boldsymbol{P}$
$\boldsymbol{P}_t$ is the hit point discovered by tracing the transmitted ray from $\boldsymbol{P}$
k_{rg} is the global reflection coefficient
k_{tg} is the global transmitted coefficient

For each reflection/refraction ray spawned, we can trace it just like tracing the original ray

When to stop recursion

- When surface is totally diffuse (and opaque)
- When reflected/refracted ray hits nothing
- When recursion depth threshold is reached
- When the contribution of the reflected/refracted ray to the color is too small (less than a threshold)
  - Product of reflection/transmission constants that represents the path of the light ray
  - e.g. refracted twice, reflected once:

$$k_{tg1} \times k_{tg2} \times k_{rg3}$$

$$(k_{rg1}|k_{tg1}) \times (k_{rg(n-1)}|k_{tg(n-1)}) < \text{threshold}$$

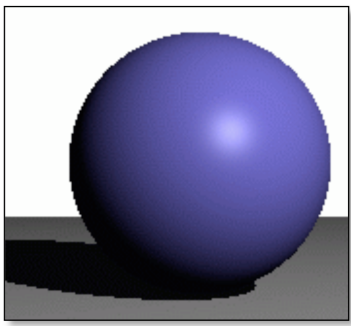
Limitations

Hard shadows

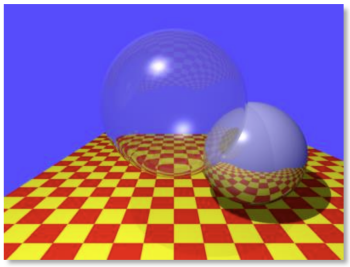
- A point is either in a shadow or not
- Light sources assumed to be point light sources

Aliasing (also known as jaggies)

The following picture shows jaggies and a hard shadow

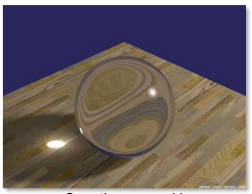


Inconsistency between highlights and reflections

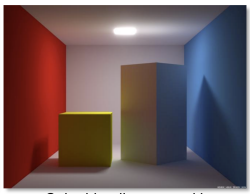


- We have sharp reflections (of the scene), but blurred highlights
- This is because they were computed via different methods: reflection is computed via reflection rays, while highlights are computed via Phong illumination model

Simulates only partial global illumination



Caustics caused by focusing of light



Color bleeding caused by diffuse-to-diffuse interactions

- Cannot simulate caustics
- Cannot simulate color bleeding, because it does not spawn secondary rays in all directions to collect potential reflect light coming from other diffuse surfaces

Counting number of rays (T8)

Assume the scene is enclosed, i.e. all rays will hit some object

- 1 primary ray
- 1 reflection ray per level of recursion
- 1 refraction ray per level of recursion (assuming there are non-opaque objects)
- 1 shadow ray per light source, per primary, reflection, or refraction ray

COMPUTING INTERSECTIONS

Ray representation

- Using two 3D vectors (origin, direction), or
- Using parametric form

$$P(t) = \text{origin} + t \times \text{direction}$$

Plane

- Plane is often in implicit form
  - $Ax + By + Cz + D = 0$, or
  - $N \cdot v + D = 0$, where $N = [A\ B\ C]^T$ and $v = [x\ y\ z]^T$

Intersection

- Substitute $v = P(t)$ and solve for t. The intersection is at $P(t)$.
- If no solution for t, then the ray is parallel to the plane and there is no intersection
- Verify that the intersection is not behind the ray origin, i.e. $t_0 > 0$

Normal

- Normal is either N or $-N$

Sphere

- Sphere (centered at origin) often in implicit form
  - $x^2 + y^2 + z^2 = r^2$, or
  - $v \cdot v - r^2 = 0$, where $v = [x\ y\ z]^T$

Intersection

- Let R_o be the ray origin, and R_d be the ray direction.

$$v \cdot v - r^2 = 0$$
$$(R_o + tR_d) \cdot (R_o + tR_d) - r^2 = 0$$
$$(R_d \cdot R_d)t^2 + 2(R_d \cdot R_o)t + (R_o \cdot R_o - r^2) = 0$$

- It is a quadratic equation in the form $at^2 + bt + c = 0$
- Discriminant $d = b^2 - 4ac$
  - $d < 0$: no solution
  - $d = 0$: 1 solution
  - $d > 0$: 2 solutions
- Solutions $t_\pm = \dfrac{-b \pm \sqrt{d}}{2a}$
- Smallest positive t value is chosen as t_0
- If sphere not centered at origin, transform the ray to sphere's local coordinate frame

Normal

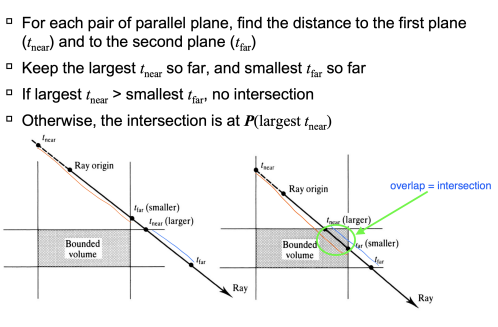
- Normal is $\dfrac{P(t_0)}{|P(t_0)|}$

Box

- A 3D box is defined by 3 pairs of parallel planes, where each pair is orthogonal to the other 2 pairs
- If it is axis-aligned, only need to specify the coordinates of the two diagonally opposite corners, as the planes can be deduced

Intersection

- In the 2D case, there are 2 intervals, which must overlap at a common region for there to be an intersection between the ray and box



- In the 3D case, there are 3 intervals, which must ALL overlap at a common region for there to be an intersection between the ray and box
- We are not interested in knowing the exact intersection point, because boxes are usually used as bounding volumes to enclose objects, for a quick test for intersection

Normal

Can be found by considering individual planes of the box

General polygon

- Finding intersection between a ray and a general polygon is difficult
  1. Compute ray-plane intersection (assumes polygon is planar)
  2. Determine whether intersection is within polygon (tedious for non-convex polygon)
  - In addition, interpolation of attributes at vertices are not well defined
- Since a polygon can be decomposed into triangles, easier to find ray-triangle intersection using barycentric coordinates method

Triangle

Barycentric coordinates

The barycentric coordinates of a point P on a triangle ABC is (α, β, γ) such that

$$P = \alpha A + \beta B + \gamma C$$

where $\alpha + \beta + \gamma = 1$ and $0 \le \alpha, \beta, \gamma \le 1$. This can be rewritten as

$$P = (1 - \beta - \gamma)A + \beta B + \gamma C$$
$$P = A + \beta(B - A) + \gamma(C - A)$$

to eliminate one variable.

Intersection

- Substitute $P(t) = P$

$$R_o + tR_d = A + \beta(B - A) + \gamma(C - A)$$

- Solve for t, β, γ
- Intersection if $\beta + \gamma < 1$ and $t, \beta, \gamma > 0$

Solving

- 3 equations, 3 unknowns: $Ax = B$
- Define A_i as the matrix A with the ith column replaced by B
- Cramer's rule states that the solutions are

$$\frac{|A_1|}{|A|}, \frac{|A_2|}{|A|}, \frac{|A_3|}{|A|}$$

Normal

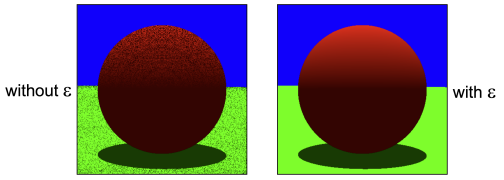
$$N_P = (1 - \beta - \gamma)N_A + \beta N_B + \gamma N_C$$

(remember to normalize)

Epsilon problem

Should not accept intersection for very small positive t

- Floating point inaccuracy might cause the ray origin to be slightly below the surface of the sphere
- When drawing shadow ray, the object seems to occlude itself
- Hence the pixel is considered occluded and a black dot is drawn



Solutions

- Method 1: Use an epsilon value $\varepsilon > 0$, and accept a intersection only if $t > \varepsilon$
- Method 2: When a new ray is spawned, advance the ray origin by an epsilon distance ε in the ray direction

CURVES AND SURFACES

Misc

Applications

- Drawing vector-based smooth curves
- Font design, representation, and rendering
- Smooth animation paths
- Designing smooth functions
- 3D model design, representation, and rendering
- Data fitting

Advantages for rendering

- More compact representation than a set of straight line segments / set of polygons
- Provide scalable geometric primitives
- Provide smoother and more continuous primitives than straight lines / planar polygons
- Animation and collision detection may become simpler and faster

Design criteria

- Local control of shape
- Smoothness and continuity
- Ability to evaluate derivatives
- Stability
- Ease of rendering

Representation

Explicit

2D curves $y = f(x)$

- Cannot represent vertical straight line, circle

3D curves $y = f(x)$ and $z = g(x)$

3D surfaces $z = f(x, y)$

- Cannot represent sphere

Implicit

2D curves $f(x, y) = 0$

- Straight line: $ax + by + c = 0$
- Circle: $x^2 + y^2 - r^2 = 0$

3D surfaces $f(x, y, z) = 0$

- Plane: $ax + by + cz + d = 0$
- Sphere: $x^2 + y^2 + z^2 - r^2 = 0$

3D curves

- Represented as the intersection of two 3D surfaces
- $f(x, y, z) = 0$ and $g(x, y, z) = 0$

Drawbacks Because the equations are basically membership tests,

- Difficult to obtain points on the curves and surfaces
- Rasterization is difficult (but ray tracing is easy)

Parametric

2D and 3D curves

- Each variable is expressed in terms of an independent variable u, the parameter

$$p(u) = \begin{bmatrix} x(u) \\ y(u) \end{bmatrix} \quad p(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix}$$

- Tangent vector is obtained by differentiating individual components

$$\frac{dp(u)}{du} = \begin{bmatrix} \frac{dx(u)}{du} & \frac{dy(u)}{du} & \frac{dz(u)}{du} \end{bmatrix}^T$$

3D surfaces

- Uses two parameters

$$p(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix}$$

- Normal vector

$$n = \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v}$$

Parametric polynomial curves and surfaces

- Parametric forms are not unique
- Parametric forms fulfill all 5 design criteria

Parametric polynomial curves

- A parametric polynomial curve of degree n or order $n+1$ is of the form

$$\begin{aligned} p(u) &= \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} \\ &= \begin{bmatrix} c_{x0} + c_{x1}u + c_{x2}u^2 + \cdots + c_{xn}u^n \\ c_{y0} + c_{y1}u + c_{y2}u^2 + \cdots + c_{yn}u^n \\ c_{z0} + c_{z1}u + c_{z2}u^2 + \cdots + c_{zn}u^n \end{bmatrix} \\ &= \sum_{k=0}^{n} u^k c_k \end{aligned}$$

where $c_k = \begin{bmatrix} c_{xk} & c_{yk} & c_{zk} \end{bmatrix}^T$.

- A curve segment is defined for $u_{\min} \le u \le u_{\max}$
- Can assume $0 \le u \le 1$

Parametric polynomial surfaces

- A parametric polynomial surface is of the form

$$p(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix} = \sum_{i=0}^{n} \sum_{j=0}^{m} c_{ij} u^i v^j$$

- $\{c_{ij}\}$ contains $3(n+1)(m+1)$ coefficients
- Normally, $n = m$
- A surface patch is defined for $0 \le u, v \le 1$

Parametric cubic polynomial curves

- Prefer to define long curve by joining multiple curve segments of lower degree
  - Local control of shape
  - Stability
- In particular, degree 3 / cubic parametric polynomial curve segments are preferred
- Can be written as

$$p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3 = \sum_{k=0}^{3} u^k c_k = u^T c$$

where

$$c = [c_0 \; c_1 \; c_2 \; c_3]^T$$
$$u = [1 \; u \; u^2 \; u^3]^T$$
$$c_k = [c_{xk} \; c_{yk} \; c_{zk}]^T$$

Specifying curve segments

- In practice, we don't want to specify a curve segment by directly providing the coefficients of $p(u)$
- We prefer to provide geometric data that can be used to derive the values of the coefficients of $p(u)$
- Like using control points (Bezier curves) or data points (cubic interpolating curves)

CUBIC INTERPOLATING CURVES

Definition

- Given control points p_0, p_1, p_2, p_3
- $p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3$ passes through all control points
- $p(0) = p_0$, $p\left(\frac{1}{3}\right) = p_1$, $p\left(\frac{2}{3}\right) = p_2$, $p(1) = p_3$

Solving for c

- From the properties above,

$$p_0 = p(0) = c_0$$

$$p_1 = p\left(\frac{1}{3}\right) = c_0 + \frac{1}{3}c_1 + \left(\frac{1}{3}\right)^2 c_2 + \left(\frac{1}{3}\right)^3 c_3$$

$$p_2 = p\left(\frac{2}{3}\right) = c_0 + \frac{2}{3}c_1 + \left(\frac{2}{3}\right)^2 c_2 + \left(\frac{2}{3}\right)^3 c_3$$

$$p_3 = p(1) = c_0 + c_1 + c_2 + c_3$$

- We can also write in matrix form as $p = Ac$, where

$$p = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & \left(\frac{1}{3}\right)^2 & \left(\frac{1}{3}\right)^3 \\ 1 & \frac{2}{3} & \left(\frac{2}{3}\right)^2 & \left(\frac{2}{3}\right)^3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

- We can solve for c by performing matrix inversion, so $c = A^{-1}p = M_I p$
- $A^{-1} = M_I$ is called the interpolation geometry matrix, and is the same for any 4 control points

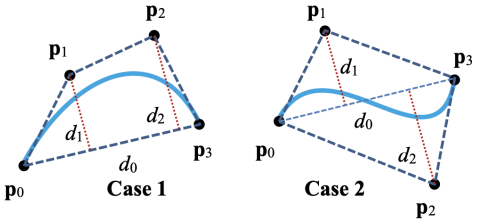
$$M_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix}$$

BEZIER CURVES

Bezier patches vs polygon mesh

- Bezier patches can be subdivided appropriately according to the size of the image, obtaining high rendering quality without making unnecessary computations
- Polygon mesh uses a fixed number of polygons. For a small image, there could be too many polygons, wasting computation. For a large image, there could be too few polygons, resulting in low rendering quality.

Flatness of a Bezier curve



- Let d_1, d_2 be the distance from p_1, p_2 to the line segment from p_0 to p_3. Then $d = \max(d_1, d_2)$ is a measure of flatness. The smaller the d, the flatter the curve
- Can also consider angles $\angle p_1 p_0 p_3$ and $\angle p_2 p_3 p_0$. As they get smaller, the curve gets flatter
- A flat Bezier curve can be drawn as a straight line segment instead

Flatness of a Bezier surface patch

- Define an average plane of the 4 corner control points $p_{00}, p_{03}, p_{30}, p_{33}$
- Find the maximum distance from the 16 control points to the average plane