SHADING MODELS FOR POLYGONS

Lights

Copyright @ 2008 by Jim X. Chen:
jchen@cs.gmu.edu
Vertex Color

- Vertex color can be
  - specified
  - calculated according to a lighting model
- The normal of each vertex can be
  - specified
  - calculated, for example, by averaging the normals of the polygons sharing the vertex
Constant Shading

• also know as flat shading
  `glShadeModel(GL_FLAT)`, only one color in a primitive (the last vertex in a primitive)

• Valid if the light source & the viewer are at infinity, and the polygon is NOT an approximation of a curved surface.
Polygon-Mesh Shading

• When approximate a curved surface by a polygonal mesh, using a finer mesh turns out to be ineffective

• When the light source is not at infinity, or the viewer is local, flat shading is not quite right

• Solution: interpolated shading
  – **Gouraud shading** (smooth shading)
  – **Phong shading** (normal interpolation)
Gouraud Shading

• Also called smooth shading, intensity interpolation shading or color interpolation shading, `glShadeModel(GL_SMOOTH)`

• each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line.

• The interpolation along edges can easily be integrated with the scan-line visible-surface algorithm.
Gouraud Shading

- Bilinearly Interpolate Colors at Vertices Down and Across Scan Lines

\[ A = \alpha I_1 + (1 - \alpha) I_2 \quad B = \beta I_1 + (1 - \beta) I_3 \]

\[ I = (1 - \gamma) A + \gamma B \]
\[ I = \frac{1 + (N \cdot L) + (\max(R \cdot V, 0))^2}{3} \]

\[ I_A = \frac{1 + \cos(60\text{degree}) + 0}{3} \]

\[ I_B = \frac{1 + \cos(60\text{degree}) + 0}{3} \]

\[ I_c = \frac{1 + 1 + 1}{3} \]
Phong Shading
(not provided in OpenGL)

• Also known as normal-vector interpolation shading
  – interpolates the surface normal vector rather than the color.
  – interpolation occurs between vertices, and then across a polygon span on a scan line
• Pixel color is calculated using the interpolated normal

Copyright @ 2008 by Jim X. Chen:
jchen@cs.gmu.edu
• One Lighting Calculation per Pixel
  – Approximate surface normals for points inside polygons by bilinear interpolation of normals from vertices
• Bilinearly Interpolate Normals at Vertices Down and Across Scan Lines

\[ A = \alpha N_1 + (1 - \alpha) N_2 \]
\[ B = \beta N_1 + (1 - \beta) N_3 \]

\[ N = (1 - \gamma)A + \gamma B \]
Gouraud shading v.s. Phong shading

• Gouraud shading
  • if a light source does not cover a vertex, the lighting is missing.
  • if a highlight fails to fall at a vertex, the specular reflection is missing.

• Phong shading
  • allows a light source covers a polygon’s interior.
  • allows specular reflection to be located in a polygon’s interior.
Shading Comparison

Flat  Gouraud  Phong

Wireframe  Flat

Gouraud  Phong
**RAY TRACING**

- *Ray tracing* is an advanced global lighting and rendering model that achieve better realism

- A ray is sent from the view point through a pixel on the projection plane to the scene

---

- **Light source**
- **View point**
- **N surface normal**
- **L feeler ray (light source direction)**
- **R reflected ray**
- **T transmitted ray**

---

Copyright @ 2008 by Jim X. Chen:

jchen@cs.gmu.edu
• *Ray tracing* is calculated in the device coordinates. So you can consider this is after viewport transformation.

• The easiest is to avoid any transformation and viewing, so the models are specified in the device coordinates.
- **Feeler ray** to light source for blocking (shadow ray)
- **Reflected ray** for reflection from other objects
- **Transmitted ray** for transparency and refraction
• Lighting is calculated at each point of intersection.

• The final pixel color is an accumulation of all fractions of intensity values from the bottom up.

• Rays (named feeler rays or shadow rays) are fired from the point under consideration to the light sources: \( L_i = 0 \) if light source blocked

• The reflection and transmission components at the point are calculated by recursive calls

\[
I_{\lambda} = I_{\lambda,e} + \sum_{i=0}^{K-1} f_{att}f_{spot}I_{\lambda,L_i} + I_{\lambda,r} + I_{\lambda,t}
\]

where \( I_{\lambda,r} \) accounts for the reflected light component, and \( I_{\lambda,t} \) accounts for the transmitted light component.
The reflected light component

• $I_{\lambda r}$ is a specular component calculated recursively by the above Equation.

• Here, the “view point” is the starting point of the reflected ray:

$$I_{\lambda r} = M_{\lambda s} I_{\lambda}$$

where $M_{\lambda s}$ is the “view point” material’s specular property.
The transmitted light component

- The transmission component $I_{\lambda t}$ is calculated similarly:

\[ I_{\lambda t} = M_{\lambda t} I_{\lambda} \]

where $M_{\lambda t}$ is the “view point” material’s transmission coefficient.

- The recursion terminates when a user defined depth is achieved or when the reflected and transmitted rays don’t hit objects.

- Computing the intersections of the ray with the objects and the normals is the major part of a ray tracing program, which may take hidden surface removal, refractive transparency, and shadows into its implementation considerations.
Ray/Sphere Intersection

• A sphere equation
  \[(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = r^2\]

• A 3D line equation
  \[x = x_0 + (x_1 - x_0); \quad y = y_0 + t(y_1 - y_0); \quad z = z_0 + t(z_1 - z_0);\]

• The intersection: solve for \( t \)
  \[- t^2 + Bt + C = 0; \quad t = \frac{-B \pm \sqrt{B^2 - 4C}}{2}\]
Basic Ray Tracing Algorithm

• For Each Pixel Ray
  – Primary ray
    • Test each surface if it is intersected
  – Intersected: Secondary ray
    • Reflection ray
    • Transparent – Refraction ray
Basic Ray Tracing Algorithm

• For Each Pixel Ray
  – Primary ray
    • Test each surface if it is intersected
  – Intersected: Secondary ray
    • Reflection ray
    • Transparent – Refraction ray
Ray Tracing Tree

• Ray tree represents illumination computation
  – One branch  $$\rightarrow$$ reflection
  – The other branch  $$\rightarrow$$ transmission
  – Terminated  $$\rightarrow$$ reach the preset maximum, no intersection (background), or strike a “light source”
Ray Tracing Tree

- Ray tree represents illumination computation
• Pixel intensity
  – Sum of intensities at root node
  – Start at terminal node
  – If no surfaces are intersected, the intensity of background

Ray Tree

\[ I_{\text{pixel}} \]

\[ I_{\text{back}} \]
Recursive Raytracing Example

1. access a pixel in the viewport

```java
public void reshape(GLAutoDrawable glDrawable, int x, int y, int w, int h) {

    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();

    gl.glOrtho(-WIDTH / 2, WIDTH / 2, -HEIGHT / 2, HEIGHT / 2, -4 * HEIGHT, 4 * HEIGHT);

    gl.glViewport(0, 0, WIDTH, HEIGHT);

    gl.glDisable(GL.GL_LIGHTING); // calculate color by ray tracing
    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
}
```
2. Initialize objects and light sources

// initialize 'ns' number of spheres
for (int i = 0; i < ns; i++) {
    sphere[i][0] = 10 + WIDTH * Math.random() / 10; // radius
for (int j = 1; j < 4; j++) { // center
    sphere[i][j] = -WIDTH / 4 + WIDTH * Math.random() / 2;
}
}

// initialize 'nl' light source locations
for (int i = 0; i < nl; i++) {
for (int j = 0; j < 3; j++) { // light source positions
    lightSrc[i][j] = -10 * WIDTH + 20 * WIDTH * Math.random();
}
}
3. Firing the ray

```java
public void display(GLAutoDrawable glDrawable) {

    // starting viewpoint on positive z axis
    viewpt[0] = 0;   viewpt[1] = 0;   viewpt[2] = 1.5*HEIGHT;

    // trace rays against the spheres and a plane
    for (double y = -HEIGHT / 2; y < HEIGHT / 2; y++) {
        for (double x = -WIDTH / 2; x < WIDTH / 2; x++) {

            // ray from viewpoint to a pixel on the screen
            raypt[0] = x;   raypt[1] = y;   raypt[2] = 0;

            // tracing ray (viewpt to raypt) for depth bounces
            rayTracing(color, viewpt, raypt, depth);

            glBegin(GL_POINTS);
            glColor3dv(color, 0);
            glVertex2d(x, y);
            glEnd();
        }
    }
}
```
4. Bounces

// recursive rayTracing from vpt to rpt for depth bounces, finding final color
public void rayTracing(double[] color, double[] vpt, double[] rpt, int depth) {
    for (int i = 0; i < 3; i++) { color[i] = 0; } // if (depth == 0) black color
    if (depth != 0) {// calculate color
        intersect(vpt, rpt, p, n, depth); // intersect at p with normal n
        // view direction vector for lighting and reflection
        for (int i = 0; i < 3; i++) { vD[i] = vpt[i] - rpt[i]; }
        phong(color, p, vD, n); // current point’s color without reflec. & trans.
        // reflected ray
        reflect(vD, n, rD);
        for (int i = 0; i < 3; i++) { rpoint[i] = rD[i] + p[i]; } // a point on the reflected ray starting from p
        tpoint[i] = -vD[i] + p[i] - n[i]*0.2; // transmitted ray
    }
    // recursion to find a bounce at lower level
    rayTracing(reflectClr, p, rpoint, depth - 1);  rayTracing(transmitClr, p, tpoint, depth - 1);
    for (int i = 0; i < 3; i++) {
        color[i] = color[i] + 0.4*reflectClr[i] + 0.3*transmitClr[i];
    }
}

Copyright @ 2008 by Jim X. Chen:
  jchen@cs.gmu.edu
5. Calculate Intersections

```java
public void intersect(double[] vpt, double[] rpt, double[] point,
                      double[] normal) {

    // calculate intersection of ray with the closest sphere
    // Ray equation:
    // x/y/z = vpt + t*(rpt - vpt);
    // Sphere equation:
    // (x-cx)^2 + (y-cy)^2 + (z-cz)^2 = r^2;
    // We can solve quadratic formula for t and find the intersection
    // t has to be > 0 to intersect with an object

    // return point and normal
    point[0] = vpt[0] + t * (rpt[0] - vpt[0]);
    normal[0] = point[0] - sphere[i][1]; // from the sphere center to the current point
    normal[1] = point[1] - sphere[i][2];
}
```
public void phong(double[] color, double[] point, double[] vD, double[] n) {

    for (int i = 0; i < nl; i++) {
        // if intersect objects between light source, point in shadow
        intersect(point, lightSrc[i], ipoint, inormal);

        for (int j = 0; j < 3; j++) {
            lgtsd[j] = lightSrc[i][j] - point[j];
            // light source direction
            } normalize(lgtsd);
        for (int j = 0; j < 3; j++) {
            s[j] = lgtsd[j] + vD[j]; // for specular term
            } normalize(s);

        double diffuse = dotprod(lgtsd, n);
        double specular = Math.pow(dotprod(s, n), 128);

        if (diffuse < 0) diffuse = 0;  
        if (diffuse==0 || specular < 0) specular = 0;
        color[0] = color[0] + diffuse + specular;
    }
    color[0] = color[0]/nl;
Distributed Ray Tracing

• averaging multiple rays distributed over an interval; samples randomly chosen points and averages the results
• also called stochastic ray tracing.
Adaptive Ray Tracing

• sample a pixel 4 times in a grid, then compare.
• If one sample is different from the others, then subdivide that square recursively
• All the samples are close to each other or a predefined depth is reached. The result is a (weighted) average of the samples

Copyright @ 2008 by Jim X. Chen: jchen@cs.gmu.edu
HW/PROJECT

• Implement a raytracing system that includes
  – Sphere, cube, cylinder, (cone, Optional)
  – “lookat” function for animation purpose
  – Antialiasing through distributed or adaptive raytracing
  – Optional: texture mapping, bump mapping, and environment mapping